The Effect of Operation Scheduling on the Performance of a Data Flow Computer

MICHAEL GRANSKI, MEMBER, IEEE, ISRAEL KOREN, SENIOR MEMBER, IEEE, AND GABRIEL M. SILBERMAN, MEMBER, IEEE

Abstract—**The effect of incorporating a priority scheme into a data flow computer is studied in this paper. Specifically, we deal with the scheduling of instructions in a data flow program, and the mechanisms by which such scheduling may be implemented within a data flow computer.**

We show that the assignment of priorities to data flow operations is a special case of a problem in scheduling theory, and also belongs to the NP-complete class of problems. Therefore, we develop a heuristic approach, based on the well-known Critical Path algorithm, as a basis for determining instruction priorities.

Our conclusions, based on the simulation of programs executed in a modified data flow computer, show that adding a priority mechanism is not justifiable in the general case. This is due mostly to the inability to reach the potential improvement offered by scheduling operations, because of implementation restrictions. Nevertheless, certain algorithms (e.g., DFT) can still benefit from the proposed scheme, mainly because of their highly regular, static structure.

Index Terms—**Arbitration network, data flow, list scheduling, Modified Critical Path algorithm, performance.**

I. Introduction

OVER the past few years, several architectures have been proposed as the basis for data flow (DF) computers [1]–[4]. Even though these machines differ in their architecture and the DF language(s) they support, they all share the same computation model—the DF graph model (See Fig. 1).

The DF graph model represents a program by a directed graph $G(T, E)$ where each node T (called an *operator*) stands for either an individual program instruction or a group thereof, and the arcs E carry operands (tokens) from one operator to another. As opposed to the classical von Neumann model, there is no sequential control flow and, therefore, no program counter. Instead, each operator has its own firing rule which defines the conditions under which it is ready for execution. Consequently, at any moment there may be many operators (instructions) ready for execution and all of them may be executed in parallel. In the DF model there is no central memory with common variables on which the program's modules operate and, therefore, no side effects. Every operator operates only on its private operands and produces new operands to be consumed by other operators.

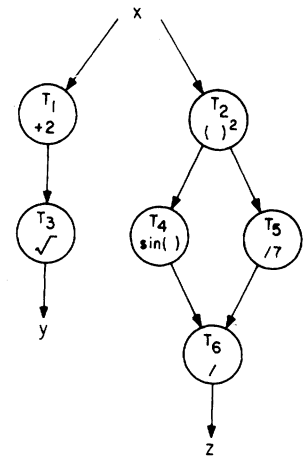


Fig. 1. An example of a DF graph representing the computation $y = \sqrt{x + 2}$, $z = (\sin x^2)/(x^2/7)$.

In the following we use the terms instruction and operator interchangeably, i.e., the material presented here applies equally to operators which represent single or multiple instructions. Whenever there is a difference between the two cases, it is noted in the text.

There are still many open questions concerning architectures and programming languages for DF computers. This paper concentrates on one of them—the scheduling of instructions in a DF computer (this subject was first mentioned in [3]). Conceptually, instructions in a DF program can be executed as soon as their operands become available. In practice, however, there are not enough processors to perform all these instructions and thus some of them are executed before others. The order in which these instructions are executed is neither defined in the compiler, nor supported by the DF architecture. Our purpose is to study the cost effectiveness of fixing, at compile time, the execution order of the instructions in a DF program, in conjunction with the introduction of a scheduling mechanism into the DF architecture to enforce this order.

Scheduling of operations in a DF computer has the potential of speeding up the execution of programs. However, a considerable overhead is expected when a scheduling mechanism is incorporated into the DF computer. It is our goal to investigate this question and determine the conditions under which scheduling of operations in a DF computer is beneficial.

In the next section the architecture of the MIT DF computer is briefly presented and the scheduling problem is described.

Manuscript received January 15, 1985; revised February 27, 1986.

M. Granski was with the Department of Electrical Engineering, Technion-Israel Institute of Technology, Haifa 32000, Israel. He is now with Zoran Microelectronics, Haifa, Israel.

I. Koren is with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003, on leave from the Department of Electrical Engineering, Technion-Israel Institute of Technology, Haifa 32000, Israel.

G. M. Silberman is with the Department of Computer Science, Technion-Israel Institute of Technology, Haifa 32000, Israel.

IEEE Log Number 8715298.

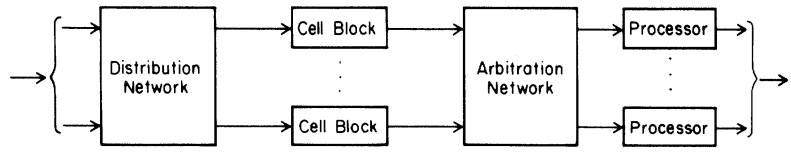


Fig. 2. The MIT data flow architecture.

In Section III a formalization of the scheduling problem for a DF graph is given and a heuristic algorithm to solve the problem, called *Modified Critical Path (MCP),* is described. Section IV is devoted to the architectural features which implement the scheduling. In Section V we present some simulation results and our conclusions are presented in Section VI.

II. DF Computers and the Scheduling Problem

Extensive research has been done in the area of DF computing and several suggested architectures for a DF computer have emerged [1]–[4]. In each of these architectures a scheduling mechanism can be incorporated. To illustrate a possible implementation of our suggested scheduling algorithm, we have selected the MIT DF architecture proposed by Dennis [1], [5] and shown in Fig. 2. A similar approach could be taken in implementing these features in other DF architectures, but their study is beyond the scope of this paper.

The MIT architecture provides for a large number (thousands) of processors, a distribution network, a number of cell blocks, and an arbitration network. The program instructions are stored in the cell blocks. Once an instruction becomes ready for execution, it is transferred with its operands to the arbitration network. The function of this network is to route ready for execution instructions to idle processors. After the instruction is executed, its results are transferred to the distribution network. This network passes the results back to the appropriate cell blocks.

Fig. 3 describes the implementation of a cell block. The update unit receives result packets from the distribution network. A result packet consists of a result operand and destination address(es) used by the update unit to pass the result operand to the proper target instruction(s). If an instruction needs no more operands and, therefore, it is ready for execution, the update unit moves its address to the instruction (FIFO) queue. The fetch unit takes the instruction addresses from this queue and generates operation packets for the arbitration network. An operation packet consists of the instruction opcode, operand(s), and result address(es).

Operation packets must then proceed through the arbitration network (refer to Fig. 2) to a processor for execution. Ideally, if there are any idle processors when an operation packet becomes available, that processor should receive the packet and act accordingly. But, in practice, the capacity of the arbitration network could cause a departure from this ideal behavior, because of conflicts between several processors requiring operation packets at the same time. This potential for a bottleneck, together with the realization that the arbitration network must take part in the enforcement of the schedule, lead us to search for an appropriate protocol to handle the request of operation packets by the processors, and their

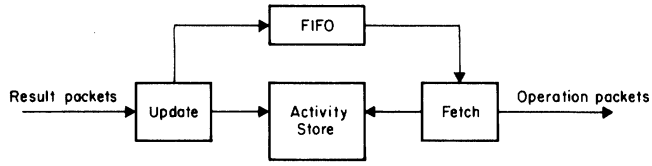


Fig. 3. Implementation of a cell block.

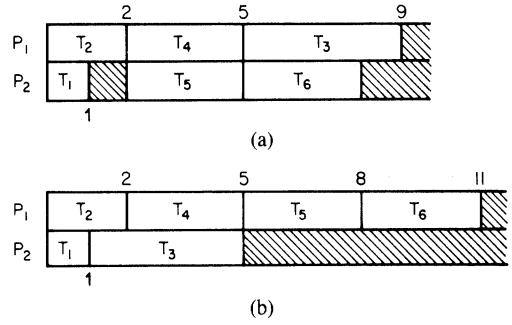


Fig. 4. Two possible schedules of the DF graph of Fig. 1 in a Gantt-chart form. The execution times of the instructions are $T_1 = 1$, $T_2 = 2$, $T_3 = 4$, $T_4 = T_5 = T_6 = 3$.

subsequent transfer by the arbitration network. Special attention is given in this protocol to the resolution of conflicts among requests from several processors.

To illustrate the effect of the order in which instructions are selected for execution on the total execution time of the program, Fig. 4 gives two possible schedules of the DF graph of Fig. 1. The schedules are given in a Gantt-chart form [6].

A solution to the scheduling problem must include a method for ordering the operations in a DF graph (at compile time) and appropriate modifications to the DF architecture to enable enforcement of this order during program execution. In the next section we describe in detail the scheduling problem for a DF graph and present some solutions. Then, in Section IV we describe the architectural mechanisms needed to implement these solutions within a DF computer.

III. Scheduling a DF Graph

A. Scheduling Problems

Scheduling theory includes a large number of models and problems [6]–[8]. One type of scheduling problem consists of

1) A set $P = \{P_1, \cdots, P_r\}$ of r identical processors.

2) A set $T = \{T_1, \cdots, T_n\}$ of n tasks.

3) A partial precedence order among the tasks. The notation $T_i < T_j$ means that T_j can be executed only after the execution of T_i has been completed. Usually the precedence order is given by a directed acyclic graph (DAG) $G(T, E)$ in which there is an arc from T_i to T_j if and only if $T_i < T_j$.

4) A function $t: T \rightarrow (0, \infty)$ which assigns an execution time to every task.

5) A performance measure. For example, the mean flow time, which is the average of the flow times (i.e., the finish times) of the tasks.

The objective is to find a schedule of the tasks T_i, $1 \leq i \leq n$, on the processors P_j, $1 \leq j \leq r$, which satisfies the partial precedence order and that optimizes the performance measure. Formally, a schedule should assign to every task the processor(s) on which it will be executed and indicate the exact time interval during which it will be performed. (In the case of identical processors, specifying the time interval for each task suffices to define the schedule.)

Another parameter of the scheduling problem determines whether to allow the interruption (and subsequent resumption) of a task before its completion. A schedule which allows such interruptions is referred to as a *preemptive schedule,* while *nonpreemptive scheduling* refers to systems without this capability. In the following we deal with a special case of nonpreemptive scheduling called *list scheduling*. For schedules in this category, the exact time interval and the processor assigned to a given task need not be specified. Tasks are ordered by priority, and whenever a processor becomes idle, it selects from the priority list the next task which is ready for execution. (The reason for choosing this type of schedule is its suitability for implementation within a DF architecture, as will become apparent in later sections.) The difference between list scheduling and the more general nonpreemptive scheduling is that in the former a processor cannot remain idle if there is a waiting task. Fig. 4(a) shows the best nonpreemptive schedule for the problem in Fig. 1, while Fig. 4(b) depicts the best list schedule with the priority list being $L = (T_2, T_1, T_3, T_4, T_5, T_6)$.

The solution space of a scheduling problem is enumerable and, therefore, it is possible to find an optimal solution through complete enumeration. Obviously, this is practical only for the smallest problems due to the large number of possible solutions. Unfortunately, most of the scheduling problems are NP-complete [6], [9] and there is no known polynomial algorithm for their solution. A more practical approach is to use heuristic algorithms which ensure polynomial search time, but do not always find an optimal schedule.

The best known heuristic algorithm for list scheduling is the *Critical-Path (CP)* algorithm [6], and most of the suggested heuristic algorithms for scheduling problems are variations of it. The basic idea is to assign to every node (task) in the precedence graph a weight that equals the maximum sum of execution times of nodes on any directed path from this node to a leaf in its corresponding subgraph. The tasks are given priority in descending order of weights. This way the algorithm finds the maximum weight critical paths in the graph and assigns higher priorities to the nodes in these paths. For example, T_2 in Fig. 4 has two equal weight critical paths: $T_2 \rightarrow T_4 \rightarrow T_6$ and $T_2 \rightarrow T_5 \rightarrow T_6$.

Several forms of the CP algorithm have been published. The one described below is based on a special case (for two processors) which appears in [6], and serves as the starting point for the proposed DF scheduling algorithm.

Definition: Let $l = (X_1, X_2, \cdots, X_k)$ and $l' = (X'_1, X'_2, \cdots X'_{k'})$ be sequences of positive numbers. We say that $l <$ l' *lexicographically* if 1) the two sequences agree up to j, for some j, but $X_j < X'_j$ (i.e., there exists a j, $1 \leq j \leq k$ and $1 \leq j \leq k'$, such that $\forall i$, $1 \leq i < j$, $X_i = X'_i$ and $X_j < X'_j$), or 2) the two sequences agree but list l is shorter (i.e., $\forall i$, $1 \leq i \leq k$, $X_i = X'_i$ but $k < k'$).

Given a scheduling problem with a precedence graph $G(T, E)$, let t_i be the execution time of node T_i, X_i be the node's weight. Let S_i denote the descendant group of T_i and $\langle S_i \rangle$ denote the index group of S_i, i.e.,

$$S_i \triangleq \{ T_j \mid T_i \rightarrow T_j \in G(T, E) \} \tag{3.1}$$

$$\langle S_i \rangle \triangleq \{ j \mid T_j \in S_i \}. \tag{3.2}$$

In the following CP algorithm, *Part A* computes the weight of every node in $G(T, E)$, while *Part B* builds the priority list L.

Algorithm CP (Critical Path):

Part A:

A.1 For every leaf node T_i (satisfying $S_i = \phi$) set $X_i = t_i$.

A.2 Let T_i be a node such that every T_j, $T_j \in S_i$, has been given a weight X_j, then

$$X_i = t_i + \max_{j \in \langle S_i \rangle} \{X_j\} \tag{3.3}$$

(i.e., the weight of a node equals its execution time plus the maximum weight of its descendants).

A.3 Repeat step A.2 until all the nodes in the precedence graph have been assigned weights. The successful completion of this process is assured because there are no cycles in the graph.

Part B:

B.1 For every node T_i in $G(T, E)$ list the weights of all its descendants in a decreasing order $l(T_i)$.

B.2 Let L' be the group of nodes with maximum weight in $G(T, E)$. Let T_i be a node in L' such that there is no other T_j in L' for which $l(T_i) < l(T_j)$ lexicographically. Add T_i to L (which is initially $L = \phi$) and delete it from $G(T, E)$.

B.3 Repeat step B.2 until all the nodes are in L and $G(T, E)$ is empty.

For the graph of Fig. 1 the CP algorithm yields the following weights: $X_1 = 5$, $X_2 = 8$, $X_3 = 4$, $X_4 = X_5 = 6$, $X_6 = 3$. The resulting priority list is $L = (T_2, T_1, T_3, T_4, T_5, T_6)$ and yields an optimal list schedule [see Fig. 4(b)].

Lemma 1: Let n be the number of nodes in the precedence graph. If the number of node descendants is bounded by s then the complexity of the CP algorithm is $O_{cp} = O(n^2 s)$, otherwise $O_{cp} = O(n^3)$.

Proof: See Appendix. ■

The polynomial CP algorithm does not always find the best priority list. However, some researchers [10], [11] have shown that it finds optimal or near optimal schedules most of the time. There are even some important special cases in which the CP algorithm has been proven to be optimal. Hu [12], in one of the first results in scheduling theory, proved that the CP algorithm is optimal if all the tasks have equal execution times and the precedence graph is a tree (or forest).

Coffman and Graham [13] have shown that the CP algorithm is optimal if all the tasks have equal execution times and there are only two processors. (But if we allow tasks of two kinds with execution time of one or two time units the problem is already NP-complete [9].)

B. The Scheduling Problem for a DF Graph

Scheduling for a DF graph is an instance of the more general scheduling problem described in the previous subsection, where

1) The operators (nodes) of the DF graph are the tasks to be executed.

2) The set of edges in the DF graph convey tokens from one operator to another, and together with the firing rules, replace the partial precedence order. (Notice that a DAG is a special case of a DF graph since the latter might have conditional nodes and loops.)

3) The performance measure is the total execution time of the DF graph.

4) Assignment of execution times to tasks is based on an estimate of how long it takes for a DF operator to execute. (For the case where operators represent single DF instructions, this is trivally done at compile time.)

5) The scheduling is nonpreemptive because it is not possible to interrupt the execution of a basic DF operator.

The approach we take to the solution of this problem, which is also amenable for implementation within a DF architecture (as we shall see in Section IV below), is based on list scheduling. Another advantage with this type of schedule is that there is no need to specify in advance the actual time interval during which an operator will be executed. This enables the production of schedules which can handle the case of DF graphs with conditional nodes and loops which are repeated a variable number of times.

Our approach to list scheduling is to extend the CP algorithm (as defined on DAG's) to handle conditional nodes and loops. This is the subject of the next two subsections.

C. Scheduling a DF Graph with Conditional Nodes

A conditional node is a node with two or more output arcs. Its firing rule directs the result operand to one and only one output arc. The decision is made at run time depending on the input operands, and, therefore, the critical path through a conditional node cannot be evaluated before execution.

The approach taken here follows that of Martin and Estrin in [14]. Assume that for every output arc of a conditional node we are given the probability that the node will fire and produce an output token on that arc. This probability is called the *fire probability* of the arc. Before the actual execution of the DF graph, it is only possible to evaluate the expected value of the critical path. This value is called the *mean critical path length*. In what follows we present a modification of *Part A* of the CP algorithm which finds the mean critical path length for every node.

Let $G(T, E)$ be an acyclic DF graph. Let p_{ij} denote the fire probability of an arc $e(T_i, T_j)$. If T_i is a conditional node then p_{ij} is the probability that T_i will produce a token on $e(T_i, T_j)$. If T_i is not a conditional node then $p_{ij} = 1$. Clearly, for every conditional node we have

$$\sum_{j \in \langle S_i \rangle} p_{ij} = 1. \tag{3.4}$$

Let $D_k(T^k, E^k)$ denote a deterministic subgraph of G obtained by the following procedure.

For every conditional node in G choose one output arc and delete all the others. (As a result there may be some nodes and arcs which do not have a directed path from any root of G to them; delete all these nodes and arcs.) The resulting subgraph is a deterministic subgraph of G. Executing G really means executing one of its deterministic subgraphs.

Let $p(D_k)$ denote the probability that the deterministic subgraph D_k will be executed. If the conditional nodes are independent, then $p(D_k)$ is equal to the product of the fire probabilities of all arcs in D_k,

$$p(D_k) = \prod_{e(T_i, T_j) \in E^k} p_{ij}. \tag{3.5}$$

The assumption that all the conditional nodes are independent is unrealistic since conditional nodes often check dependent (or even the same) conditions. Nevertheless, this assumption is necessary to make the analysis of the graph tractable. The conclusion is that the calculation of the mean critical path length presented here is only an approximation.

The probability that a node T_i will be executed equals the sum of the probabilities of all deterministic subgraphs which include T_i. Formally, let $\langle D_k \rangle_i$ denote the index group of all the subgraphs which contain T_i,

$$\langle D_k \rangle_i \triangleq \{k \mid T_i \in D_k\}. \tag{3.6}$$

Let $p(T_i)$ denote the probability that T_i will be executed. Then

$$p(T_i) = \sum_{l \in \langle D_k \rangle_i} p(D_l). \tag{3.7}$$

For every deterministic subgraph D_k we will use the CP algorithm to find the critical path length of its nodes. Let X_i^k be the critical path length of T_i in D_k. The mean critical path of T_i is the average weight of the critical path lengths of T_i in all the subgraphs and is given by

$$\overline{X_i} \triangleq \frac{1}{p(T_i)} \sum_{l \in \langle D_k \rangle_i} X_i^l p(D_l). \tag{3.8}$$

Example 3.1: A simple DF graph with one conditional node (T_3) is shown in Fig. 5. The fire probabilities of its output arcs are

$$p_{34} = p(e(T_3, T_4)) = \frac{2}{3} \quad p_{35} = p(e(T_3, T_5)) = \frac{1}{3}.$$

The graph has two deterministic subgraphs.

$$D_1(T^1, E^1): \; T^1 = \{T_1, T_2, T_3, T_4, T_6, T_7\}$$
$$E^1 = \{e(T_1, T_2), e(T_1, T_3), e(T_2, T_7), e(T_3, T_4), e(T_4, T_6), e(T_6, T_7)\}$$

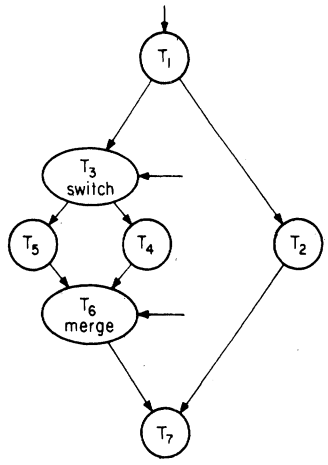


Fig. 5. A DF graph with a conditional node.

TABLE I
RESULTS OF EXAMPLE 3.1

T_i	t_i	$p(T_i)$	X_i^1	X_i^2	$\overline{X_i}$
T_1	2	1	13	10	12
T_2	5	1	8	8	8
T_3	1	1	11	7	29/3
T_4	5	2/3	10	–	10
T_5	1	1/3	–	6	6
T_6	2	1	5	5	5
T_7	3	1	3	3	3

$$D_2(T^2, E^2): T^2 = \{T_1, T_2, T_3, T_5, T_6, T_7\}$$
$$E^2 = \{e(T_1, T_2), e(T_1, T_3), e(T_2, T_7), e(T_3, T_5), e(T_5, T_6), e(T_6, T_7)\}.$$

The probability that a subgraph is executed is calculated using (3.5).

$$p(D_1) = \frac{2}{3}, \quad p(D_2) = \frac{1}{3}.$$

The execution probabilities of the nodes are computed from (3.7). The CP algorithm yields the critical path length of the nodes in D_1 and D_2 (X_i^1 and X_i^2, respectively). The mean critical path length is found using (3.8). The results are summarized in Table I.

The computation of the mean critical path length according to the definition in (3.8) is very costly. If a DF graph has d conditional nodes with two output arcs each, then the number of deterministic subgraphs is of the order of 2^d. The calculation of the mean critical path length requires $2^d \times O_{cp}$ steps and might render this method impractical.

Consequently, Algorithm ACP below replaces *Part A* of the CP algorithm, to calculate an approximated value for the mean critical path length. The new algorithm has the same complexity as the original CP algorithm, regardless of the number of conditional nodes. After assigning each node a weight that is an approximation of its mean critical path length, (unchanged) *Part B* of the CP algorithm is executed as before.

TABLE II
RESULTS OF EXAMPLE 3.2

T_i	t_i	S_i	α_i
T_1	2	$\{T_2, T_3\}$	35/3
T_2	5	$\{T_7\}$	8
T_3	1	$\{T_4, T_5\}$	29/3
T_4	5	$\{T_6\}$	10
T_5	1	$\{T_6\}$	6
T_6	2	$\{T_7\}$	5
T_7	3	ϕ	3

Algorithm ACP (Approximate Critical Path):

Given a DF graph $G(T, E)$, let α_i denote the approximation of the mean critical path length of node T_i.

1) For every leaf node T_i (satisfying $S_i = \phi$) set $\alpha_i = t_i$ (where t_i is the execution time of T_i).
2) Let T_i be a node such that every T_j, $T_j \in S_i$, has been assigned a weight α_j, then

$$\alpha_i = \begin{cases} t_i + \sum_{j \in \langle S_i \rangle} p_{ij}\alpha_j, & \text{if } T_i \text{ is a conditional node;} \\ t_i + \max_{j \in \langle S_i \rangle} \{\alpha_j\}, & \text{otherwise.} \end{cases} \tag{3.9}$$

3) Repeat step 2) until all the nodes in the precedence graph have been given weights.

Example 3.2: The results of applying Algorithm ACP to the DF graph of Fig. 5 are given in Table II. Comparing these to Table I we find that the approximations equal the mean critical path lengths for all the nodes except T_1, for which $\overline{X_1} = 12 > (35/3) = \alpha_1$.

It can be proven [15] that for every node in the graph $\alpha_i \leq X_i$.

D. Scheduling a DF Graph with Loops

The CP algorithm cannot be applied to a DF graph with loops since every node in the loop requires that all the others be given a weight before a weight can be assigned to it. Each node in a loop is executed several times, every time with a different path length from it to the nearest leaf. The ideal solution would be to assign each node a dynamic weight, but the list scheduling scheme forces a fixed priority.

A common approach in similar problems with loops is to unfold the body of the loop according to the number of iterations. This approach is inappropriate here because the number of iterations is not always known in advance. The solution we present here is again based on work by Martin and Estrin [16]. We first transform the original graph into an acyclic graph. (The purpose of the transformation is only to aid in assigning priorities; once a priority list has been found, the original graph is executed.) For simplicity, we discuss only loops with one input node and one output node. An analysis of more complicated cases can be found in [16].

The input and output nodes are called the entry and control nodes of the loop, respectively. The mean critical path length from the entry node to the control node is not affected by the cyclic to acyclic transformation (performed by Algorithm CA,

presented below). Other properties of the loop, for example the variance of the critical path length, are not preserved [16].

We will distinguish between two types of loops, *deterministic* and *random*. In the former, the loop is executed a fixed number of times—N, while in the latter, execution of the loop continues with probability p. If for a random loop we assume different iterations to be independent, the number of actual iterations can be represented by a random variable $\tilde{N}$ with expected value $\bar{N} = 1/(1 - p)$.

Algorithm CA (Cyclic to Acyclic Transformation)

For every loop in a given DF graph $G(T, E)$ take the following steps.

Let T_e be the entry node and T_c be the control node. Let L be the set of loop nodes, which includes the entry and control nodes and every other node which has a directed path either from the entry node to it or from it to the control node. Let N be the number of iterations.

1) Erase the feedback arc $e(T_c, T_e)$.
2) Multiply the execution time of every node in L by N (or by $\bar{N}$ in the case of a random loop).

In summary, for a given DF graph a list schedule can be found by the following steps. First, use Algorithm CA to eliminate all loops in the graph and adjust execution times of loop nodes. Then, use the CP algorithm to find the priority list. If the graph has conditional nodes, replace *Part A* of the CP algorithm with Algorithm ACP. We shall refer to this process as the *Modified Critical Path (MCP)* algorithm.

IV. Realizing the Scheduling Mechanism in a DF Architecture

A method for finding a heuristic scheduling of a DF graph has been presented in the previous section. The result of applying this algorithm (during compilation) to a DF graph is the priority list which defines the list schedule for that graph. In the following we describe the mechanisms used to enforce the above priorities while executing the DF program on the MIT DF architecture.

A. Overview

First, the relative priority of the instructions (the priority list) must be present in the activity store along with the instructions themselves. The straightforward solution is to add a priority tag to each instruction in memory. The disadvantages of this solution are the waste of memory space and the bounded number of priority levels. A better solution is to use the instruction's address as its priority. Then, when the program is loaded into the activity store, its instructions will be arranged in descending order of priority.

Next, we must be sure that the update unit, which transfers results from the distribution network to the target instructions in the activity store, implements the desired schedule. This is done by passing result tokens with lower addresses first, which guarantees their earlier transfer to the instruction queue.

The main components of our implementation are the communication networks present in the MIT DF computer, namely the arbitration and distribution networks.

Dennis [1], [5] proposed to implement the communication networks of a DF computer as a special type of packet-switching network called a *delta network*. A $N \times N$ delta network with N input ports and N output ports is built of basic units called routers. Each router has p inputs and p outputs and the routers are arranged in n stages such that $N = p^n$. The interconnections between consecutive stages in the network enable a communication path between every input and every output port. The routing of a packet in the network is determined by an n digit number in base p which heads the packet and is called the packet address. The packet is routed through the stages, passing one router in every stage. Every router chooses to transfer the packet to one of its outputs according to one digit in the packet address. This way, the n digits of the packet address correspond to the n stages in the delta network and every digit controls the passage through a router in one of the stages. Dennis [1] suggested the use of a delta network with 2×2 routers ($p = 2$), and $n = \log_2 N$ stages with $N/2$ routers each.

Comparing the delta network to a crossbar we find that in a crossbar the transfer time is constant and does not depend on N, but the number of connections increases as $O(N^2)$. On the other hand, in a delta network the number of connections increases only as $O(N \log_2 N)$, but the transfer time also increases with N as $O(\log_2 N)$. Another problem in delta networks is that a packet being transferred might temporarily block other packets. An accurate analysis [17] shows that the cost–performance ratio of the delta network is better than that of the crossbar.

The distribution network can be implemented by a delta network as described above. However, the implementation of the arbitration network raises some problems. First, assuming that all the processors are identical, an operation packet does not have a unique address because it can be executed by any of the processors. Second, the load must be balanced among the processors, and idle ones should be identified and new instructions directed to them. Third, if scheduling is to be implemented, the routing of instructions by the arbitration network must obey the priority list. These problems are discussed below.

B. Implementation of the Arbitration Network

The arbitration network in the MIT DF computer can be implemented as a delta network with certain changes in its management strategy and a special interconnection structure. The input side of the network consists of N *send units* which receive operation packets from the fetch unit and transmit them, one at a time, to N identical processors at the output side of the network.

There are many forms of a delta network that provide a path from every input to every output. We have chosen the network defined by the recursive construction shown in Fig. 6. Its advantage stems from the following property [15].

Property A: Whenever a router receives two operation packets at its inputs, the packet received through the upper input port has originated at a send unit with a index lower than that of the send unit which transmitted the other packet.

The communication protocol proposed for the arbitration network is based on the following general strategy. The transfer of an operation packet through the network is initiated

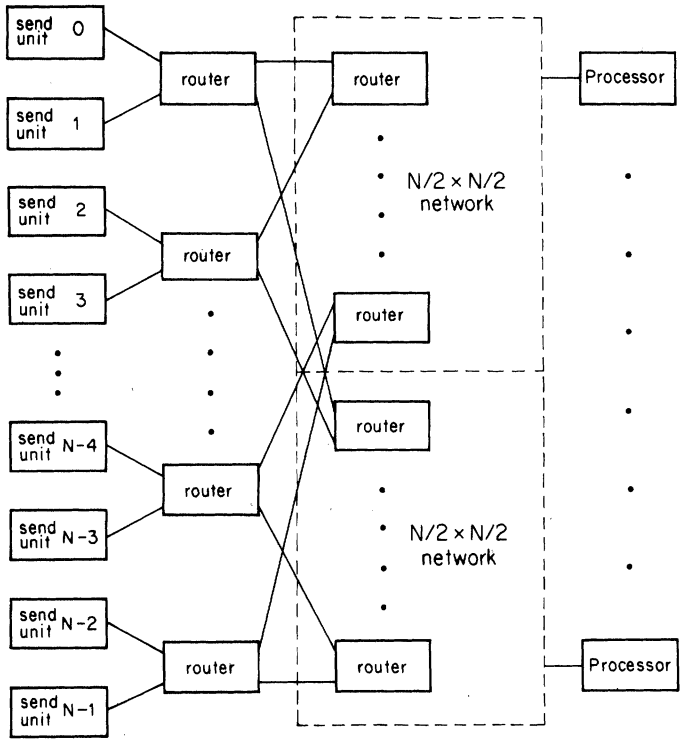


Fig. 6. Recursive construction of the arbitration network.

by an idle processor, by sending a request towards the send units. The network responds by transferring an operation packet (from some send unit) to the requesting processor. This strategy of operation eliminates the need for prior knowledge about the processor in which the instruction will be executed. Another advantage is that the load is automatically balanced between the processors.

In what follows, the phases of the complete communication protocol are presented.

Phase 1: When a processor becomes idle it sends a request signal to the router to which it is connected. This request is transferred (through dedicated lines in the arbitration network) to a maximum number of send units in the form of a spanning tree, with the requesting processor as its root. Every router which receives a request signal from one of its right ports (refer to Fig. 6) passes it to any of its left ports which is not engaged in a packet transfer. If a second request (from another idle processor) is received by a router, it is *not* passed on. At this point, the router has to remember only which right port(s) had a request.

Phase 2: When a send unit receives a request and has an operation packet, it starts to transmit the packet and waits for either an acknowledge or cancel signal. It may happen that several send units received, at the same time, requests originating at the same processor and start to transmit their packets simultaneously. However, only one packet will be transferred to the processor. A router receiving two packets as a response to a request from only one of its right ports, will pass on the packet received at its upper left port and send a cancel signal back through its lower left port. According to *Property A,* the uppermost send unit which receives the request transfers its packet successfully to the processor, while all the other send units receive a cancel signal. If the router which received two packets had requests from both its right ports (not necessarily simultaneous requests), it will pass on both packets, each one to a different requesting processor.

Phase 3: When a processor which issued a request receives an operation packet, it issues an acknowledge signal along the transfer path to the send unit and thus terminates the communication.

The above protocol enables multiple packet transfers through the network. The scheme of distributing the request generated by an idle processor to a maximum number of send units proved (in simulations) to be very effective. Every idle processor which had a free path to a nonempty send unit received a packet in minimum time. In this scheme each router has to handle at any time instant, at most two requests, thus avoiding bottlenecks. Also, if there are several idle processors, multiple packets sent in response to a single processor's request will be generally absorbed by these processors. Notice that operation packets are transmitted by send units using otherwise (presently) unutilized paths in the arbitration network, and are either passed to idle processors or canceled by some router along the way.

The protocol may introduce some changes to the priority list. If at a given time only a single packet is being transferred in the network, then the above protocol and *Property A* guarantee that the highest priority instruction will be transferred. However, if packets are being transferred in parallel (which is the usual case), then there might be some high priority send units which will not receive the request at all (due to blockages by other transfer paths). Among the send units which do receive the request, the uppermost is selected. This is a necessary compromise between the requirements of the scheduling algorithm and the network throughput.

A large delay between the execution of the current instruction and that of the next instruction by a processor can be somewhat reduced by the following scheme. After the current instruction (or the last in a group of instructions, for the case of multiple instructions per operation packet) is decoded, its execution time is known and the processor can issue a request t_α time units before the end of execution (t_α can be set, for example, to equal the average transfer time through the network).

Also to be considered is the mechanism by which operation packets move from the queue being filled by the fetch unit with instructions ready for execution into the send units. The possibility of organizing the send units themselves as a queue, with packets entering at its tail (see Fig. 7) and being sent by the first unit which receives a request, proved to be inadequate. Fig. 7 shows the steady state to which (as verified by simulation) this solution brings the system, i.e., the last send unit is engaged in a packet transfer and blocks the queue. All other send units are empty and most of the processors are idle. When the blocking send unit ends its transmission and is filled with a new packet, it will immediately receive another request and start transmitting again.

In order to overcome this problem, an *operation packet queue* is added in parallel to the send units, as shown in Fig. 8. In this scheme, operation packets move up in the queue unless the send unit to their right becomes empty, in which case the packet is passed to that send unit. Notice that this protocol

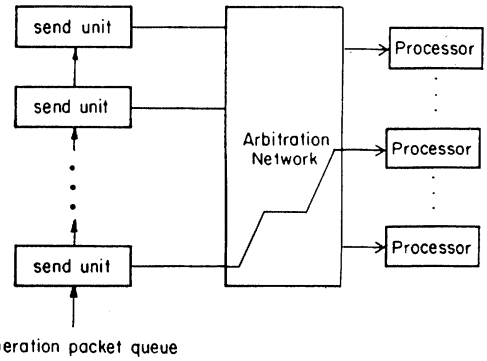


Fig. 7. An undesirable situation in the operation packet queue.

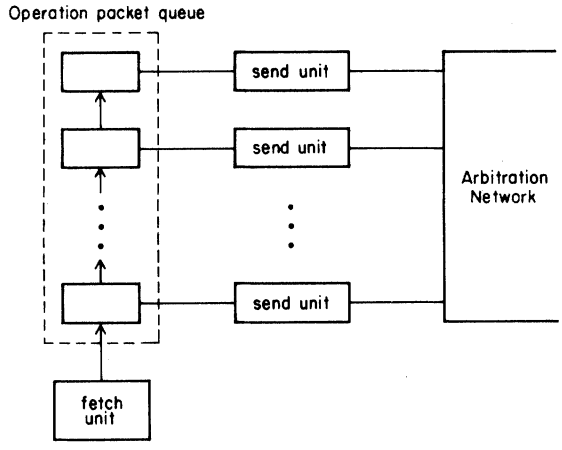


Fig. 8. The operation packet queue.

does not necessarily preserve the order of the operation packets in the queue, but rather constitutes another compromise between the scheduling mechanism and the goal of high throughput of the arbitration network.

V. Simulation Results

The implementation of the scheduling mechanism was tested by simulation [15], using a benchmark which included three types of DF graphs.

1) Graphs without conditional nodes and without loops. This group included some instances of the discrete Fourier transform for 3*3, 5*5, 8*8, 30*30, and 32*32 points.

2) Graphs with conditional nodes but without loops. As a representative of this type of graph, an 82-vertex graph used by Martin and Estrin [18] was chosen.

3) Graphs with loops. Some graphs representing numerical calculations such as the Newton–Raphson approximation and the Runge–Kutta method for solving differential equations were selected.

The experiments were carried out using three programs. The first program finds a priority list for a given DF graph according to the Modified Critical Path (MCP) algorithm presented in Section III. The second program finds a Gantt chart for a DF graph with a given priority list and a given number of processors. The third program simulates a complete DF computer with its scheduling mechanism as described in Section IV.

The Gantt-chart program is essentially a simulation of a DF computer with zero delays in the communication networks, and its purpose is to estimate the potential for performance improvement. The quality of the priority lists generated by the MCP algorithm was tested by comparing the results of the benchmark to the results of randomly generated priority lists. The Gantt chart of every graph in the benchmark was found for the priority list of the MCP algorithm and for five randomly generated lists. This was repeated for an increasing number of processors until the minimum finish time (limited by the longest path in the graph) was reached.

A typical result for the 32*32 DFT graph is shown in Fig. 9. The curve marked MCP is the finish time when the MCP algorithm is used. The differences between the results of the five random lists were small and their average finish time is shown by the curve marked R. For a single processor, the finish time equals the sum of the execution times of all the nodes in the graph, regardless of the priority list. The finish time that corresponds to the MCP algorithm decreases until it reaches the minimum of 33 time units with 16 processors. Increasing the number of processors beyond that lowers the utilization of the processors but does not shorten the finish time. The finish time for the random lists decreases more slowly and reaches its minimum with 33 processors. The advantage of the MCP algorithm for the 32*32 DFT can be clearly seen in Fig. 10(a). It depicts the speedup due to the use of the MCP algorithm which is defined by

$$\text{Speedup} = \frac{\text{average finish time for a random list}}{\text{finish time for the MCP list}}. \quad (5.1)$$

The maximum speedup due to the use of the MCP algorithm for the 32*32 DFT is achieved when there are 16 processors. Increasing the number of processors beyond 16 causes the finish time of the random lists to decrease slowly, but the finish time of the MCP algorithm does not improve. In general, when there are more processors than the inherent parallelism in the graph (so that every list may be executed in the minimum finish time), the speedup decreases to one. Clearly, the maximum speedup is not a constant, but depends on the structure of the DF graph.

The advantage of the MCP algorithm over random schedules can be seen when the number of processors is "tuned" to the parallelism of the problem at hand. Table III shows the number of processors needed to execute various DFT graphs in their minimum finish time using the MCP algorithm or random schedules. For large problems, the ratio of needed processors might be as high as 1:2 in favor of the MCP algorithm. We have also calculated lower bounds on the number of processors required to execute the same DFT graphs in minimum time. These calculations were done using the expressions presented in [19]. In all five cases the number of processors needed when using the MCP algorithm proved to equal the corresponding lower bound.

The speedup for graphs with conditional nodes and loops are less "well behaved." For the 82-vertex graph [Fig. 10(c)], the MCP algorithm can be even worse than a random list. This can be explained by the approximations of Algorithm ACP. The Newton–Raphson graph [Fig. 10(d)] shows an advantage for the MCP algorithm, but not as evident as for the DFT graphs. Clearly, the MCP algorithm has a greater potential for graphs without conditional nodes or loops, and with possibly a regular structure like the DFT graphs.

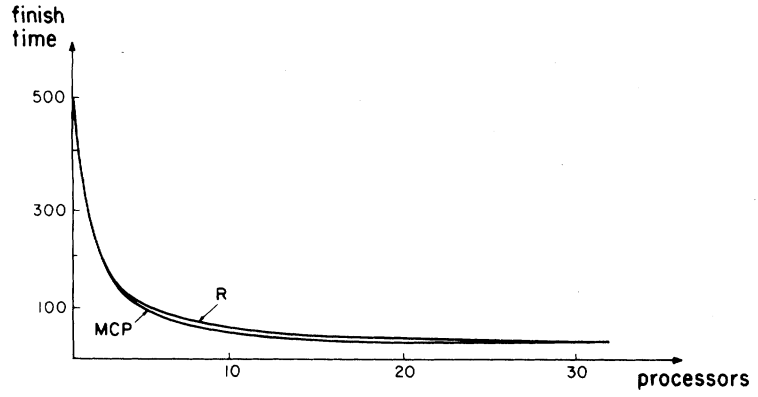


Fig. 9. The finish time of the 32*32 DFT graph.

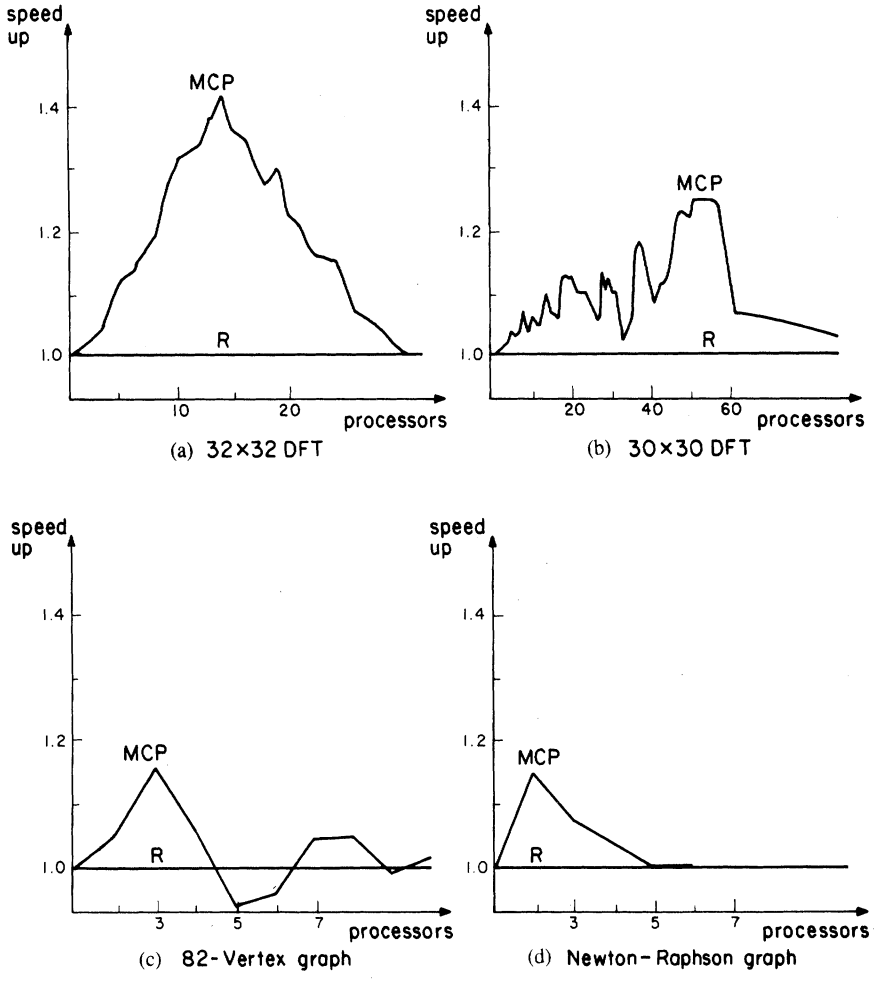


Fig. 10. Speedup factors for the Gantt-chart program.

TABLE III
NUMBER OF PROCESSORS FOR MINIMUM FINISH TIME

Problem	MCP Schedule	Random Schedules	Proc. Ratio = $\frac{\text{MCP schedule}}{\text{Random schedules}}$
3*3 DFT	6	7	0.86
5*5 DFT	20	21	0.95
8*8 DFT	4	8	0.5
30*30 DFT	54	98	0.55
32*32 DFT	16	32	0.5

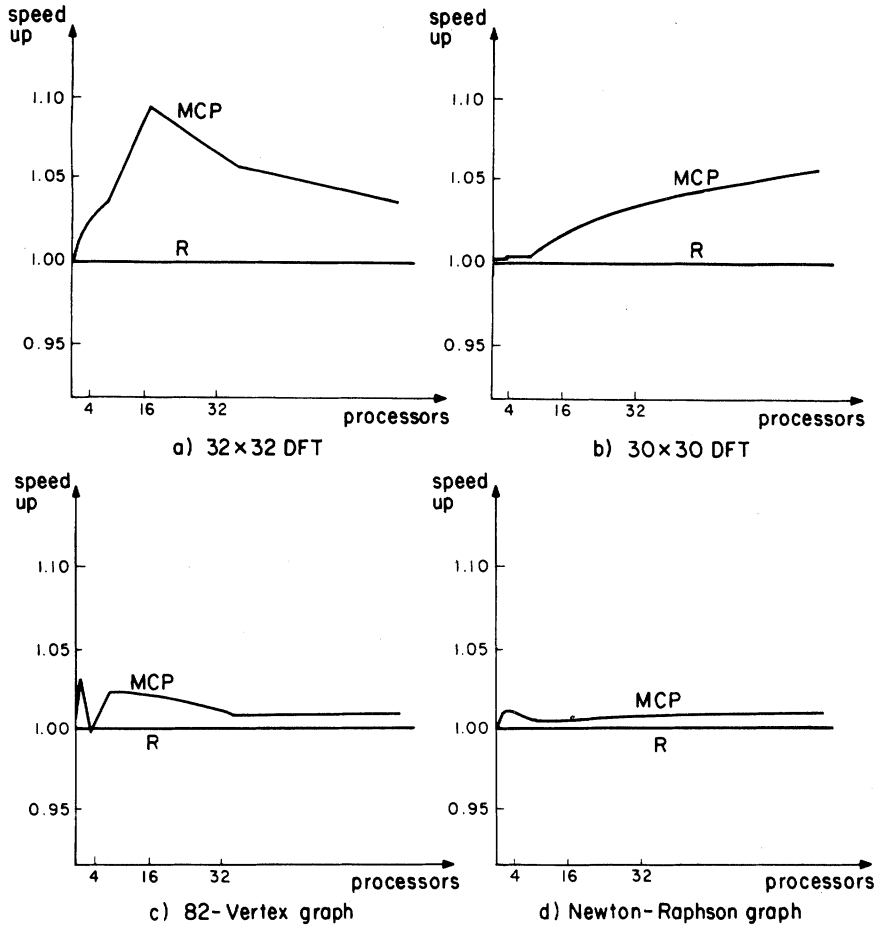


Fig. 11. Speedup factors for the simulation program.

The results of the Gantt-chart program showed the potential improvement of the MCP algorithm over random schedules. The simulation program was used to find to what extent the implementation presented in Section IV achieved this potential. This program performs a complete simulation of a DF computer with its communication networks. A fixed delay of 2 + $\log_2$ (number of processors) in the distribution network was assumed. (This delay is equal to the minimum transfer time in a delta network plus an overhead of two time units.)

The ratio between the transfer time through the arbitration network and the average execution time of instructions poses another problem. The DFT graphs, for example, have two types of operations—complex addition and complex multiplication. Let T_A and T_M denote the execution times of these operators, respectively. Assuming that a real addition takes one time unit and a real multiplication takes three time units we get, $T_A = 2$ and $T_M = 14$. The Gantt-chart program was run with the ratio $T_A/T_M = 1/7$. Let T_N denote the time required to transfer one word between two consecutive stages in the arbitration network. When the simulation program was run with the ratio $T_N/T_A/T_M = 1/1/7$ the utilization of the processors was near zero. Most of the time was spent in the arbitration network and all the priority lists gave equally bad results. This supports the idea that highly complex instructions, or blocks of instructions rather than single instructions, should be transferred and executed (e.g., [3], [20], [21]), so that the transfer time in the network will be sufficiently short relative to the execution time. For this reason, the ratio $T_N/T_A/T_M = 1/10/70$ was chosen in the simulation.

Fig. 11 shows the speedup factors obtained by the simulation program. Note that the structure of the delta network forces a number of processors which is a power of two. The results resemble those of Fig. 10 but the maximum speedup is significantly smaller (5–10 percent as opposed to 10–40 percent for the Gantt-chart program). Moreover, in graphs with conditional nodes or loops there is no significant difference between the MCP algorithm and random schedules. The reason for the speedup loss lies in the implementation of the arbitration network, which does not allow the exact execution of the MCP schedule. For example, a blockage in the network alters the execution order dictated by the priority list.

These observations seem to suggest that the scheduling algorithm and its implementation, as presented here, are not suitable for a general-purpose DF computer. However, consider DF computers intended for usage in special-purpose applications, such as numeric programs with a few conditional nodes and regular structures of computation. It is this environment, with its particular cost–performance tradeoffs, which makes our approach, and its potential for performance improvement, appealing.

VI. Conclusions

We showed in this paper a heuristic procedure, the Modified Critical Path Algorithm, which can be used to schedule the execution of instructions in a DF computer. Also presented, was a possible implementation of a priority mechanism to enforce the above schedule. Results from

simulation runs were presented to show both the potential for improvement by using the schedule, and the measure to which the implementation achieved this potential.

We conclude, based on the results obtained, that the hardware overhead required in the implementation of the arbitration network outweighs any real benefit in performance, for a general-purpose DF computer. Furthermore, it is the presence of conditional nodes and loops which precludes us from achieving better performance. Therefore, we feel that the value of this approach may be realized in special-purpose environments, in which the structure of the computational algorithms involved better suit our scheme.

Appendix

Proof of Lemma 1

Suppose there is a maximum of s descendants to every node in the graph. It is easy to see that step A.1 of the CP algorithm takes $O(n)$. Also, each execution of step A.2 requires $O(ns)$ operations since we must find all the nodes for which every descendant has already been assigned a weight and then find the maximum weight of their descendants. Since step A.2 is repeated up to n times, the complexity of *Part A* of the CP algorithm is $O(n^2s)$.

In step B.1 we obtain $l(T_i)$ in $O(s \log s)$ operations for every node, and thus require a total of $O(ns \log s)$ operations. Step B.2 takes $O(ns)$ operations and is repeated n times. Therefore, the complexity of *Part B* is also $O(n^2s)$ and consequently, $O_{cp} = O(n^2s)$. Now, if the number of descendants for each node is not bounded (by s), we replace s by n and obtain $O_{cp} = O(n^3)$. ■

References

[1] J. B. Dennis, "Data flow supercomputers," *Computer,* vol. 13, pp. 48–56, Nov. 1980.

[2] A. L. Davis, "The architecture and system method of DDM1: A recursively structured data driven machine," in *Proc. 5th Conf. Comput. Architecture,* 1978, pp. 210–215.

[3] K. P. Gostelow and R. E. Thomas, "Performance of a simulated data flow computer," *IEEE Trans. Comput.,* vol. C-29, pp. 905–919, Oct. 1980.

[4] I. Watson and J. Gurd, "A prototype data flow computer with token labeling," in *Proc. Nat. Comput. Conf.,* vol. 48, 1979, pp. 623–628.

[5] J. B. Dennis, G. A. Boughton, and C. K. Leung, "Building blocks for data flow prototypes," in *Proc. Caltech Conf. VLSI,* 1979, pp. 1–8.

[6] E. G. Coffman, Ed., *Computer and Job-Shop Scheduling Theory.* New York: Wiley, 1976.

[7] R. Kan, *Machine Scheduling Problems.* The Hague, The Netherlands: Nijhoff, 1976.

[8] M. J. Gonzalez, "Review on deterministic scheduling theory," *ACM Comput. Surveys,* vol. 9, pp. 174–204, Mar. 1977.

[9] J. D. Ullman, "NP-complete scheduling problems," *J. Comput. Syst. Sci.,* vol. 10, pp. 384–393, June 1975.

[10] W. H. Kohler, "A preliminary evaluation of the CP method for scheduling tasks on multiprocessor systems," *IEEE Trans. Comput.,* vol. C-24, pp. 1235–1238, Dec. 1975.

[11] C. V. Ramamoorthy, K. M. Chandy, and M. J. Gonzales, "Optimal scheduling strategies in a multiprocessor system," *IEEE Trans. Comput.,* vol. C-21, pp. 137–146, Feb. 1972.

[12] T. C. Hu, "Parallel sequencing and assembly line problems," *Oper. Res.,* vol. 9, pp. 841–848, June 1961.

[13] E. G. Coffman and R. L. Graham, "Optimal scheduling on two-processor systems," *Acta Informatica,* vol. 1, pp. 200–213, Jan. 1972.

[14] D. F. Martin and G. Estrin, "Path length computations on graph models of computations," *IEEE Trans. Comput.,* vol. C-18, pp. 530–536, June 1969.

[15] M. Granski, "Scheduling in a data flow computer," M.Sc. thesis Dep. Elec. Eng., Technion 1983 (in Hebrew).

[16] D. F. Martin and G. Estrin, "Models of computations and systems—Cyclic to acyclic graph transformations," *IEEE Trans. Electron. Comput.,* vol. EC-16, pp. 70–79, Feb. 1967.

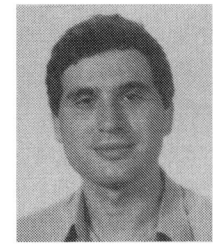
[17] J. H. Patel, "Performance of processor—Memory interconnection for multiprocessors," *IEEE Trans. Comput.,* vol. C-30, pp. 771–780, Oct. 1981.

[18] D. F. Martin and G. Estrin, "Experiments on models of computations and systems," *IEEE Trans. Electron. Comput.,* vol. EC-16, pp. 59–69, Feb. 1967.

[19] E. B. Fernandez and T. Lang, "Computation of lower bounds for multiprocessor schedules," *IBM J. Res. Develop.,* vol. 19, pp. 435–444, Sept. 1975.

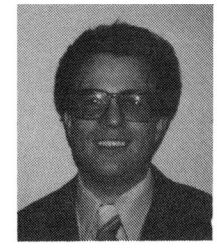
[20] J. E. Requa and J. R. McGraw, "The piecewise data flow architecture: Architectural concepts," *IEEE Trans. Comput.,* vol. C-32, pp. 425–438, May 1983.

[21] F. J. Burkowski, "Instruction set design issues relating to a static dataflow computer," in *Proc. 9th Conf. Comput. Architecture,* 1982, pp. 101–111.



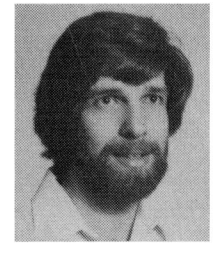
Michael Granski (S'82–M'84) received the B.Sc. (cum laude) and M.Sc. degrees in electrical engineering from the Technion–Israel Institute of Technology, Haifa, in 1981 and 1984 respectively.

His current research interests include computer architecture, VLSI CAD tools, and artificial intelligence. In 1984 he joined Zoran Microelectronics, Haifa, Israel, where he participated in several VLSI design projects.



Israel Koren (S'72–M'76–SM'87) received the B.Sc. (cum laude), M.Sc., and D.Sc. degrees in electrical engineering from the Technion–Israel Institute of Technology, Haifa, in 1967, 1970, and 1975, respectively.

Since 1979 he has been with the Departments of Electrical Engineering and Computer Science at the Technion–Israel Institute of Technology, where he became the Head of the VLSI Systems Research Center in 1985. Previously he has held positions with the University of California, Santa Barbara, and the University of Southern California, Los Angeles. In 1982 he was on sabbatical leave with the University of California, Berkeley. He was a Consultant to National Semiconductor, Israel, in architecture of microprocessors and high-speed algorithms for arithmetic operations, from 1984 to 1986, to Tolerant Systems, San Jose, CA, in architecture of fault-tolerant distributed computer systems in 1983, and to ELTA, Electronics Industries, Israel, in architecture of parallel signal processors from 1981 to 1982. Currently he is a Visiting Professor at the University of Massachusetts, Amherst. His current research interests are VLSI and WSI architectures, models for yield and performance, reliability evaluation of computer systems, and computer arithmetic.



Gabriel M. Silberman (M'87) received the B.Sc. (cum laude) and M.Sc. degrees in computer science from the Technion–Israel Institute of Technology, Haifa, in 1975 and 1976, respectively and the Ph.D. degree in computer science from the State University of New York, Buffalo, in 1980.

Since 1980, he has been with the Department of Computer Science and Electrical Engineering at the Technion, as an Assistant Professor. In 1983 he was a Visiting Scientist at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY. Since 1984 he has been a Research Fellow at the IBM Haifa Scientific Center, Haifa, Israel. His current research interests include computer architecture, operating systems, VLSI design verification, and fault simulation.

Dr. Silberman is a member of the Association for Computing Machinery and the IEEE Computer Society.