MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 116

Deadlock Avoidance in a Data-Flow Architecture

by

David P. Misunas

A paper to be presented at the Milwaukee Symposium
on Automatic Computation and Control, April 1975.

February 1975

DEADLOCK AVOIDANCE IN A DATA-FLOW ARCHITECTURE

David P. Misunas
Project MAC
Massachusetts Institute of Technology

Abstract

The implementation of a data-flow program on a data-flow computer architecture can introduce a synchronization problem between successive stages of the program. When the program is being utilized for pipelined computation, this problem can cause a deadlock condition to arise. The prevention of deadlock involves establishing a form of communication between the stages of the program, allowing a value to proceed to the next stage in the computation only when that stage is empty. Although this solution does not affect the processing time for some computations, it increases the minimum delay between successive activations of a given instruction in the program, and, in general, causes an increase in the time necessary to perform the complete computation.

1. INTRODUCTION

Efforts to develop a model of computation which can effectively express parallel activity have yielded a new form of program representation, known as data flow. The attractiveness of data flow lies in the fact that it is data-driven; that is, an instruction is enabled for execution only after each required operand has been provided by the execution of a predecessor instruction. Data-flow representations for programs have been described by Karp and Miller [7], Rodriguez [11], Adams [1], Dennis and Fosseen [4], Bährs [2], Kosinski [8], and Dennis [3].

We have been conducting architectural studies to investigate the design of a processor which can efficiently execute data-flow programs by taking advantage of the parallelism inherent in the data-flow representation. The resulting architectures offer attractive solutions to some of the problems of parallel systems. The usual problems of processor switching and memory/processor interconnection are avoided by the use of interconnection networks which have a great deal of inherent parallelism. The structure of the processor allows a large number of instructions to be active simultaneously. These active instructions pass through the networks concurrently and form streams of instructions for each pipelined functional unit.

Two such architectures utilizing a data-flow base language have been described by Dennis and Misunas [5, 6]. The <u>elementary data-flow processor</u> has been designed to execute a simple class of programs which are well-suited for the representation of signal processing type computations [5]. This class of programs permits only elementary computation; no decision capability is provided. The <u>basic data-flow processor</u> adds conditional and iterative constructs to the language and architecture and incorporates a multi-level memory system in which the active memory is operated as a cache, and individual instructions are retrieved from the auxiliary memory as they are required for computation [6]. The extension of the architectural concepts to a general-purpose computer, incorporating procedures, recursive program activation, and data structures of arbitrary size and shape, is currently under study.

A close examination of the architectural basis of these machines immediately brings one face-to-face with one of the classic problems of parallel computation, that of deadlock [9]. This deadlock problem manifests itself in a manner which affects the fundamental operation of these machines, and hence, modifications to the architectures are necessary in order to prevent a deadlock condition from arising. In this paper, the nature of the deadlock problem and its solution are explored within the environment of the elementary data-flow processor. The extension of the solution to the basic machine is a straightforward matter.

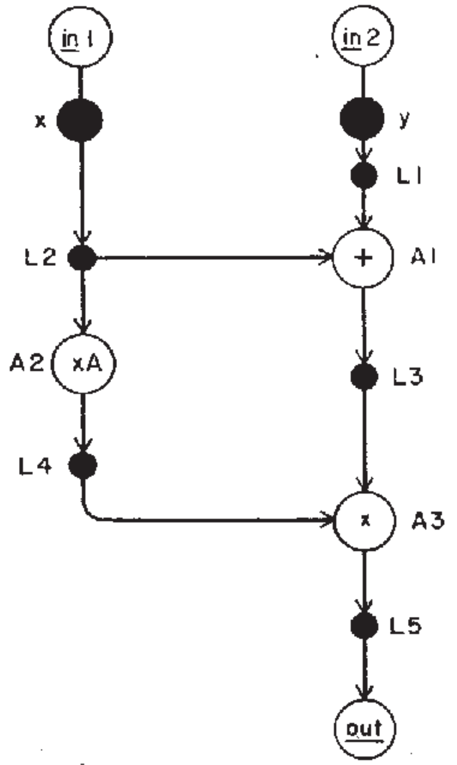


Figure 1. An elementary data-flow program.

2. THE ELEMENTARY PROCESSOR

The computational capability of the elementary data-flow processor is limited to programs expressed in the elementary data-flow language. A program in this language is constructed of two kinds of elements, called operators and links. An operator, designated by a circle, has a number of input arcs which supply values necessary for its execution, and one output arc. A small dot represents a link which has one input arc upon which it receives results from an operator and a number of output arcs over which it distributes copies of the result to other operators.

Tokens are represented by large solid dots and convey values over the arcs of the program. An operator with a token on each of its input arcs, and no token on its output arc, is enabled and sometime later will fire, removing the tokens from its input arcs, computing a result using the values carried by the input tokens, and associating the result with a token placed on its output arc. In a similar manner, a link is enabled when a token is present on its input arc, and no token is present on any of its output arcs. It fires by removing the token from its input arc and associating copies of the value carried by the input token with tokens placed on its output arcs.

The elementary data-flow program of Figure 1 has a token present on each input arc. Links L1 and L2 are enabled, and either one can fire -- suppose L2 does. Then operator A2 and link L1 are enabled, and once again, either can fire. In this manner, tokens travel through the program until a token appears on the output conveying the value Ax(x+y). Once operators A1 and A2 have fired, there are no tokens present on any of the arcs emanating from L1 and L2, and the links can fire as soon as the input operators deliver new values.

The structure of the elementary data-flow processor is presented in Figure 2. The Memory is a collection of Instruction Cells; one Instruction Cell must be associated with each operator of the program. An Instruction Cell contains three registers (Figure 3), one to hold an instruction which encodes the type of operator and its connections to other operators in the program, and two registers that receive values for use in the next execution of the instruction. When all three registers of a Cell are full, the Cell is said to be enabled and sends its contents to be operated upon by an appropriate Operation Unit.

The instruction format is shown in Figure 4. The first field contains the operation code which specifies the type of Operation Unit to be used and the function it is to perform. The second and third

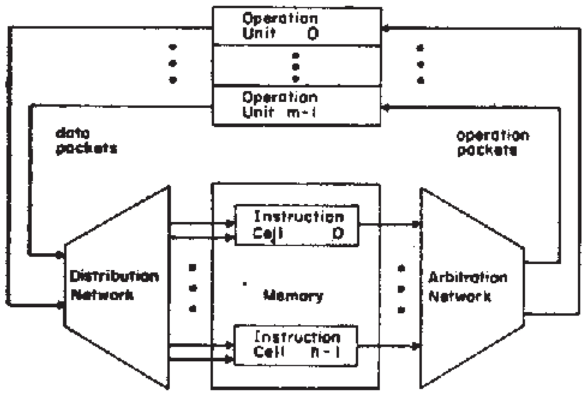


Figure 2. Structure of the elementary data-flow processor.

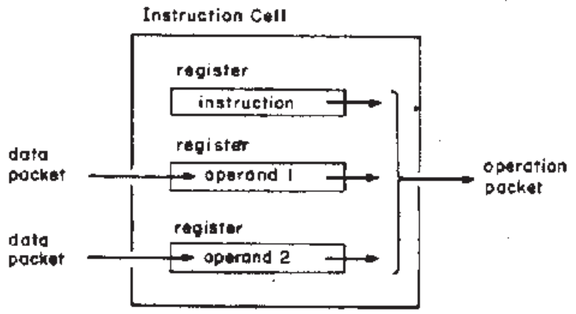


Figure 3. Operation of an Instruction Cell.

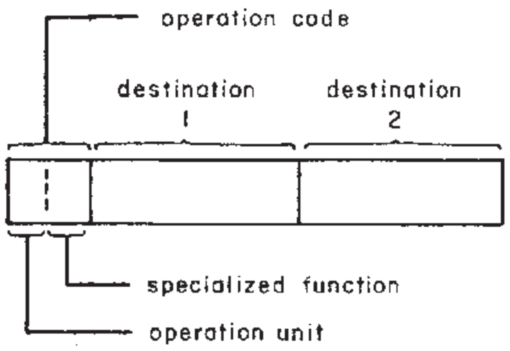


Figure 4. Instruction format.

fields hold the addresses of the registers which are to receive copies of the result.

In order to interconnect the Instruction Cells of the Memory and the Operation Units, a network, called the Arbitration Network, provides a path from each Instruction Cell to each Operation Unit. When an Instruction Cell is enabled, it passes its entire contents as an operation packet, consisting of the instruction and the two operands, into the Arbitration Network. The network is capable of accepting many operation packets simultaneously and will deliver each packet to the correct Operation Unit.

Upon receiving an operation packet, an Operation Unit performs the function specified by the operation code on the operands of the packet and produces a data packet, containing one copy of the result and a destination register address, for each destination specified in the instruction. A Distribution Network concurrently accepts data packets from the Operation Units and, utilizing the destination address of the packet, delivers each to the specified register. The Instruction Cell containing that register may then be enabled if an instruction and all operands are present.

A simplified structure of the Arbitration and Distribution Networks is presented in Figure 5. The networks are composed of three types of units. An arbitration unit passes packets arriving at its input ports one-at-a-time to its output port, using a round-robin discipline to resolve any conflicts. A switch unit passes a packet at its input to one of its outputs, controlled by some property of the packet. In the Arbitration Network this property is the operation code, whereas in the Distribution Network, the switch units are controlled by the destination address. A buffer unit stores a packet until the succeeding switch or arbitration unit is ready to accept it.

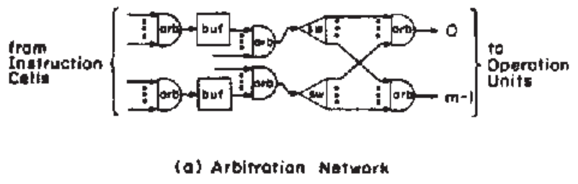
Due to the large number of inputs to the Arbitration Network, we wish to transfer data between the Memory Cells and the Arbitration Network in serial format to reduce the number of wires necessary. However, in order to maintain a high rate of packet flow at the output ports, we wish to transfer packets to the Operation Units in parallel format. For this reason, serial-to-parallel conversion is done gradually within the buffer units as a packet travels through the Arbitration Network. Parallel-to-serial conversion is performed in the Distribution Network for similar reasons.

3. THE DEADLOCK PROBLEM

The firing rule for an operator or link of a data-flow program states that the operator or link cannot become enabled unless there is no token on any output arc of that operator or link. However, the architecture, as described, provides no mechanism by which an instruction can check that the registers specified in its destination address fields are empty. Consequently, the firing rule can be violated. In illustration of what sort of problem this causes, consider the first input operator of Figure 1. The instruction representing this operator has no way of knowing whether or not the destination registers of the Instruction Cells representing operators A1 and A2 are empty and ready to receive new values, and the input instruction can send data packets to the Cells containing these destination registers before the Cells are ready to receive them. Such packets are stored in the buffer units of the Distribution Network, blocking access to succeeding switch units and preventing any other packet from being transferred to the portion of the Memory serviced by the succeeding switch units. A deadlock condition arises when one of the stored values blocks a data packet which is needed by the program in order to enable the Cell to which the stored packet is destined.

Assuming correct operation of the Operation Units, no deadlock condition can arise with the Arbitration Network. A packet in that network can only be blocked temporarily until the blocking packet moves into an Operation Unit.

The solution to the deadlock problem requires the



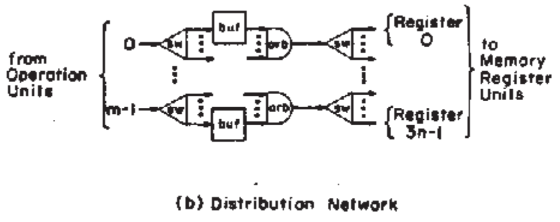


Figure 5. Structure of the Arbitration and Distribution Networks.

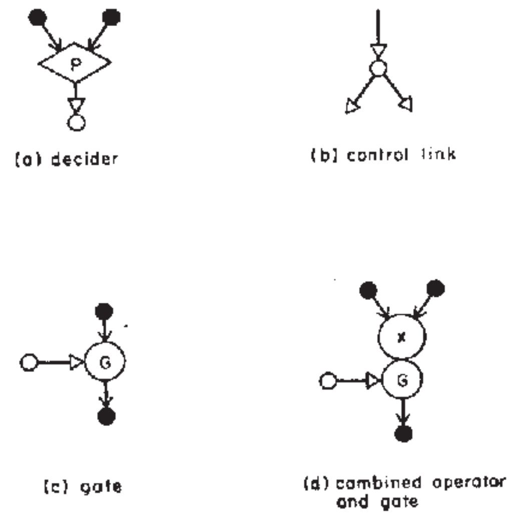


Figure 6. Additional constructs of the data-flow language.

addition of a form of feedback between operators of a program in order to force the program to observe the firing rule. In Figure 6 we present several new constructs which must be added to the data-flow language in order to establish the necessary feedback. The feedback is accomplished by the backward flow of control tokens. A control token is generated at a decider (Figure 6a), which, upon receiving a data token on each input arc, applies its predicate to the values carried by the input tokens and produces on its output arc a control token conveying either the value true or false or control. A control-valued token is produced by a decider with the nil predicate and is used to provide the necessary synchronization to avoid a deadlock. Control tokens are conveyed over control arcs and are transmitted by control links (Figure 6b).

The flow of data through the program is controlled by means of a gate (Figure 6c). A gate is enabled when it has a control-valued token on its control input and a data token on its data input. Upon firing, the gate removes the data token from its input arc and places an identical data token on its output arc, and the control token is absorbed. In order to simplify the diagrams, an operator and gate may be combined (Figure 6d). Such a joint operator is enabled when there is a data token present on each data input and a control-valued token present on the control input.

A deadlock-free version of a program is constructed from the original version by replacing each operator which could possibly place multiple tokens on its output arc by a joint operator and gate. The gate is controlled by the output of the immediately succeeding operator(s). When the link on the output of each of these succeeding operators receives a data token, it sends one copy to a decider with the nil predicate. This decider generates a control-valued token which is passed to the gate on the output of the first operator, allowing that operator to become reenabled when all necessary operands are present.

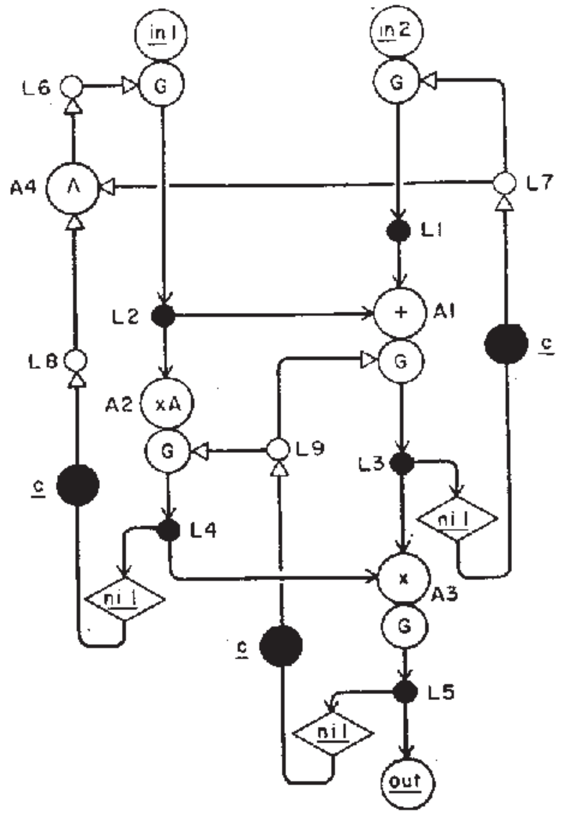


Figure 7. Deadlock-free version of the elementary data-flow program of Figure 1.

A deadlock-free version of the elementary data-flow program of Figure 1 is shown in Figure 7. The program contains an initial marking of control-valued tokens which permit the first set of data to enter. Link L2 provides a fan-out of two for the values produced by input operator 1, and hence, the control token for that operator is produced by ANDing two control-valued tokens from the succeeding operators A1 and A2.

Not all valid data-flow constructions can deadlock. For example, consider the program for a first order recursive digital filter shown in Figure 8. This program computes the value $y(t) = Ax(t) + By(t-1)$. Operators A3 and A4 cannot cause a deadlock within the loop since there can only be one token in the loop at a time. However, the possibility of a deadlock does arise with respect to the chain of operators: A1, A2, A3, A5.

If we introduce the possibility of more than one token being present in a loop, for example, by constructing a second order filter [5], we find that the possibility of a deadlock within the loop im-

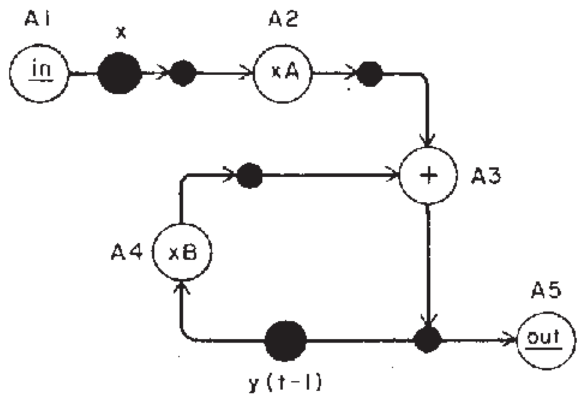


Figure 8. Data-flow program for a first order recursive digital filter.

mediately arises. The procedure which has been described to solve the deadlock problem is in line with this observation: in utilizing the procedure, a program is restructured so it consists of a series of adjacent loops, each of which can contain only one token at a time.

4. THE DEADLOCK-FREE ARCHITECTURE

Now that we have described the nature of the program modifications necessary to insure freedom from deadlock, we must consider the impact of these modifications on the architecture of the elementary data-flow processor. In order to convey control values to Instruction Cells, a special network, called the Control Network, is provided (Figure 9). This network is utilized, rather than the existing Distribution Network, due to the fact that a control packet,

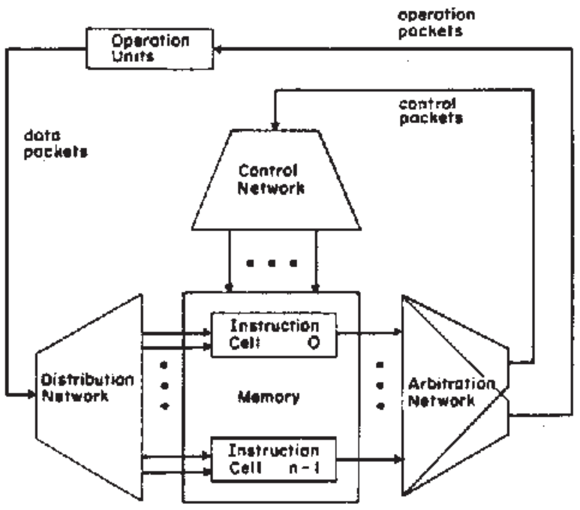


Figure 9. Revised version of the elementary data-flow processor.

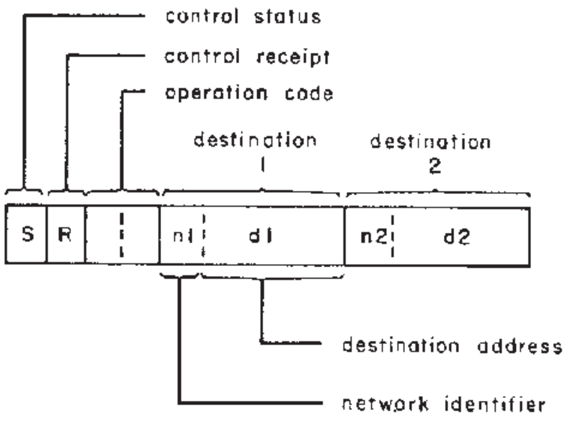


Figure 10. Revised instruction format.

conveying a control value, consists merely of an address to which the value is to be directed, and since there is no advantage to using serial format for its transmission, the parallel-to-serial conversion and buffering of the Distribution Network are not necessary. Also, the time between successive activations of a given operator is greatly affected by the necessity of waiting for a control token, and using a special high-speed network to convey the control tokens will increase utilization of the Cells of the machine.

For similar reasons, the Arbitration Network is divided into two parts -- one which conveys operation packets to the Operation Units, and one which transmits control packets in a parallel format to the Control Network.

The revised instruction format is presented in Figure 10. Each destination address has a network identifier (n1, n2) associated with it. This identifier specifies whether the address is to be used to direct a data packet through the Distribution Network or a control packet through the Control Network. If the Distribution Network is specified, operation proceeds as previously described. However, if the network identifier designates the Control Network, the associated address is removed from the operation packet at the first stage of the Arbitration Network and is used to form a control packet which is conveyed to the Control Network for delivery to the specified register.

The control status field of an instruction is equal to 1 if the output of the Instruction Cell is gated, that is, if the Cell must receive a control-valued packet before becoming enabled. A 1 in the control receipt field indicates the receipt of a control-valued packet. An Instruction Cell is enabled when it contains an instruction, all required operands, and the control status field is equal to the control receipt field. Upon being enabled and having its contents dispatched to the Arbitration Network, an Instruction Cell will reset its control receipt field to 0 to await the arrival of a new control packet.

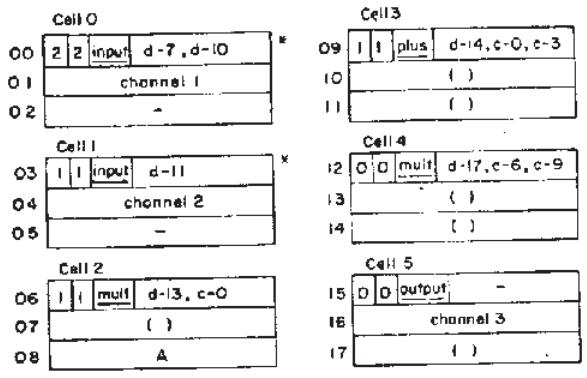


Figure 11. Initialization of Instruction Cells for the deadlock-free program of Figure 7.

The initial contents of the Cells for the deadlock-free elementary data-flow program of Figure 7 are shown in Figure 11. For the sake of simplicity, we have taken the liberty of writing any number of destination addresses in the destination address field of an instruction and have indicated the AND operation by setting the control status field of Cell 0 to two, requiring the receipt of two control-valued packets before the Cell can become enabled. The initial marking of control tokens is indicated by the initial values in the control receipt fields of the Cells.

Initially, Cells 0 and 1 are enabled and are directed to an input Operation Unit. Two input values are accepted over channels 1 and 2 and are sent as data packets through the Distribution Network to registers 7, 10, and 11. Upon transferring their contents as operation packets to the Arbitration Network, Cells 0 and 1 cannot be enabled again until each has received the specified number of control-valued packets. These control packets are provided to Cells 0 and 1 by Cells 2 and 3.

Although we have allowed ourselves to write an arbitrary number of destination addresses in the destination address field of an instruction, the instruction format, as described previously, only allows two destination addresses to be specified. This restriction necessitates the use of data distribution and control distribution Cells within the program to provide the desired fan-out. A data or control distribution Cell accepts a data or control packet and dispatches it through an identity operator to two destination registers. It should be noted that the ANDing of control packets from two succeeding operators can be implemented for free simply by requiring two operands in the designated control distribution Cell.

The architecture presented in Figure 8 greatly resembles the architecture utilized in the basic machine for the incorporation of conditional and iterative mechanisms [6]. However, a major difference exists in the fact that the elementary processor utilizes all operators of a program equally as often, whereas within the basic machine, only certain portions of a program will be active at any point in time, necessitating the utilization of a multi-level memory system so that only the active portions of a program are present in the Instruction Cells of the processor. This multi-level memory system of the basic processor is the main structural difference between the two architectures.

5. THE COST OF IMPLEMENTING THE DEADLOCK SOLUTION

The utilization of the Instruction Cells of the elementary processor can be measured in terms of the elapsed time between successive enablings of a given Cell. In the architecture originally described, the utilization of a Cell can be rather high, since data packets can be waiting in the Distribution Network for the Cell to output an operation packet and can move into the Cell immediately after the operation packet has been sent to the Arbitration Network.

In the deadlock-free version of the processor, there is a much greater delay between successive enablings of a Cell, since the Cell, upon sending out an operation packet, must wait for a data packet to arrive at each destination. At that time, assuming best case conditions, each destination Cell is enabled and returns a control packet to the original Cell, allowing it to be reenabled if new operands are present.

If D is the minimum delay encountered by an instruction in passing through the Arbitration Network, an Operation Unit, and the Distribution Network, and d is the minimum delay through the Arbitration Network and Control Network encountered by a control packet; then, if a Cell has one predecessor and one successor, the minimum delay between successive activations of the Cell is equal to D+d. If the Cell has two predecessors and two successors, a forwarding Cell of each type is required, and the minimum delay becomes 2(D+d).

The details of how these delays are computed are described in [10]. For the purposes of this discussion, we need merely note that this deadlock solution has a significant effect upon the utilization of a given Instruction Cell, necessitating, in general, more Instruction Cells to maintain a desired computational rate. However, there are specific algorithms, such as the fast Fourier transform, in which there exists a large amount of inherent parallelism and for which the deadlock solution does not cost anything in terms of throughput [10].

6. CONCLUSION

The problem of deadlock manifests itself in elementary pipelined computation. In higher level programs the problem is often already solved through the structure of procedures and loops which operate upon one set of data at a time. When the problem does arise, its solution is a straight-forward process, permitting a compiler to implement it easily, and freeing the programmer from the burden of establishing its absence.

7. REFERENCES

1. Adams, D. A. A Computation Model With Data Flow Sequencing. Technical Report CS 117, Computer Science Department, School of Humanities and Sciences, Stanford University, Stanford, Calif., December 1968.

2. Bährs, A. Operation patterns (An extensible model of an extensible language). Symposium on Theoretical Programming, Novosibirsk, USSR, August 1972 (preprint).

3. Dennis, J. B. First version of a data flow procedure language. Lecture Notes in Computer Science 19 (G. Goos and J. Hartmanis, Eds.), Springer-Verlag, New York, 1974, 362-376.

4. Dennis, J. B., and J. B. Fosseen. Introduction to Data Flow Schemas. November 1973 (submitted for publication).

5. Dennis, J. B., and D. P. Misunas. A computer architecture for highly parallel signal processing. Proceedings of the ACM 1974 National Conference, ACM, New York, November 1974.

6. Dennis, J. B., and D. P. Misunas. A preliminary architecture for a basic data-flow processor. Proceedings of the Second Annual Symposium on Computer Architecture, IEEE, New York, January 1975.

7. Karp, R. M., and R. E. Miller. Properties of a model for parallel computations: determinacy, termination, queueing. SIAM J. of Appl. Math. 14 (November 1966), 1390-1411.

8. Kosinski, P. R. A data flow language for operating systems programming Proceedings of ACM SIGPLAN-SIGOPS Interface Meeting, SIGPLAN Notices 8, 9 (September 1973), 89-94.

9. Habermann, A. N. Prevention of system deadlocks. Comm. of the ACM 12, 7 (July 1969), 373-377, 385.

10. Misunas, D. P. Performance of an Elementary Data-Flow Processor. Computation Structures Group Memo 115, Project MAC, M.I.T., Cambridge, Mass., February 1975.

11. Rodriguez, J. E. A Graph Model for Parallel Computation. Report TR-64, Project MAC, M.I.T., Cambridge, Mass., September 1969.