MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 129

Structure Processing in a Data-Flow Computer

by

David P. Misunas

(A paper to be published in the Proceedings of the 1975
Sagamore Computer Conference on Parallel Computation)

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0661.

August 1975

STRUCTURE PROCESSING IN A DATA-FLOW COMPUTER*

David P. Misunas
Project MAC
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract -- A data-flow computer uses a packet communication system to achieve highly parallel execution of programs expressed in data-flow form. The machine is composed of two sections which perform instruction processing and structure processing and share a common auxiliary memory. The structure processing section of the processor maintains data structures represented as acyclic directed graphs and is viewed as a functional unit by the instruction processing section; that is, instructions specifying structure operations are sent to the section, and the resulting values are returned to the instruction processing section. The organization of the structure processing section as a packet communication system permits the simultaneous processing of many structure operations, while avoiding the deadlock and synchronization problems often associated with systems that support concurrent memory transactions.

Introduction

The data-flow form of program representation has been developed as a method of expressing parallel activity [1, 2, 3, 5, 8, 9, 11]. The attractiveness of this form of representation lies in the fact that it is data-driven; that is, an instruction is enabled for execution when each required operand has been provided by the execution of a predecessor instruction.

The simplicity of this method of representation has led to the development of a number of computer architectures capable of executing programs expressed in data-flow form. Elementary forms of the data-flow language are utilized as the base language of a series of machines developed by Dennis and Misunas [6, 7]. The implementation of more complete data-flow languages, incorporating data structures and procedures, has been investigated by Misunas [10] and Rumbaugh [12, 13].

In the machines described by Dennis and Misunas [6, 7], the processing of instructions of a program is carried out in an instruction processing section which is structured as a packet communication system [4]. Sections of a machine are connected by interconnection networks which have a great deal of inherent parallelism, and the sections communicate by means of fixed size information packets. Each section is designed so that it never has to wait for a response to a packet it has transmitted if other packets are waiting for its attention. The extension of this concept to the organization of the structure processing section of a computer, described herein, proves very attractive, eliminating many of the deadlock and synchronization problems currently associated with systems that support concurrent memory transactions.

Data-Flow Structure Values

A program expressed in the data-flow language is constructed of two kinds of elements, called actors and links. An actor has a number of input arcs which supply values necessary for its execution and one output arc upon which results are placed. 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 results to other actors.

Values are conveyed over the arcs of a program by tokens, represented as large solid dots. An actor 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.

A value conveyed by a token is either an elementary value or a structure value. An elementary value is a single integer, real, string, or Boolean value. A structure value in a data-flow program is composed of a number of elementary values and is represented as an acyclic directed graph having one root node with the property that each node of the graph can be reached by a directed path from the root node. A node of the graph is either a structure node or an elementary node. A structure node serves as the root node for a substructure of the structure and represents a value which is a set of selector-value pairs

$$\{(s_1, v_1), \ldots, (s_n, v_n)\}$$

where

$$s_i \in \{\text{integers}\} \cup \{\text{strings}\}$$

$$v_i \in \{\text{elementary values}\} \cup \{\text{structure values}\} \cup \{\underline{\text{nil}}\}$$

and s_i is the selector of node v_i. An elementary node has no emanating arcs; rather, an elementary value is associated with the node. A node with no emanating arcs and no associated elementary value has value [nil].

*This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0661.

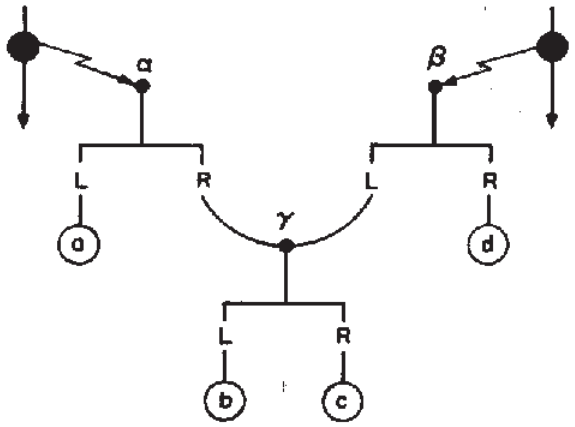

Figure 1. An example of two structures sharing a common substructure.

To illustrate the operation of the structure processing section of the processor, we shall limit our consideration to structures represented as binary trees. A selector of such a structure can have one of two values, L (left) and R (right), designating the left and right branches of the tree.

A structure value is represented by a data token carrying a pointer to the root node of the structure. In Figure 1 the structure α contains three elementary values a, b, and c, designated by the simple selector L and the compound selectors R·L and R·R respectively. Structure node γ of structure α is shared with structure β and is designated by a different selector in β than in α.

The data-flow program of Figure 2 transposes the elements of the four-element structure presented on its input. Initially, the input link of the program is enabled and, upon firing, creates four copies of the token conveying a pointer to structure α and places the copies on the inputs of the four select actors. Each select actor retrieves the value (either an elementary value or a structure value) at the end of the path specified as its argument. The resulting value is associated with a token placed on the output arc of the actor.

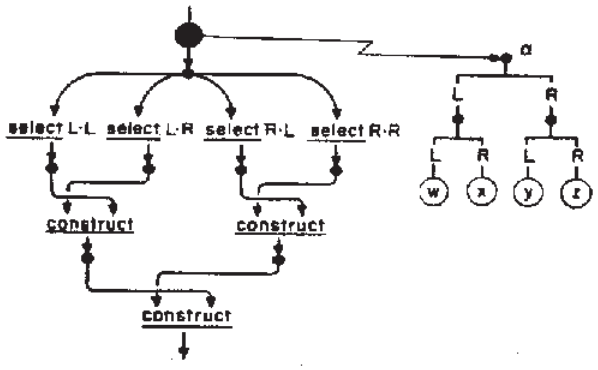

Figure 2. A simple data-flow program to transpose a four-element structure.

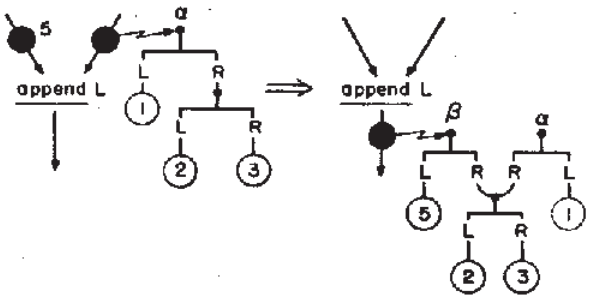

Figure 3. Operation of the append actor.

Each construct actor is enabled when it has a token on each input arc and, upon firing, creates a new structure of the values associated with the input tokens. In the program of Figure 2, the position of each input indicates the selector to be associated with the input in the resulting structure.

Structure values in a data-flow program are not modified; rather, new structure values are created which are modifications of the original values, while the original values are preserved. The append and delete actors provide the means of creating these new structure values.

The structure produced by the firing of an append actor is a version of the input structure which contains a new or modified component (Figure 3). If the specified node of the input structure has a selector corresponding to the selector argument of the actor, the value designated by that selector in the new structure is the input value. Otherwise the specified selector-value pair is added to the node of the new structure. Identical elements of the input and output structures are shared between the two structures.

In a similar manner, the structure appearing on the output arc of a delete actor is a version of the input structure in which the specified node in the new structure is missing the selector-value pair designated by the selector argument (Figure 4). As with the append actor, identical elements are shared between the input and output structures.

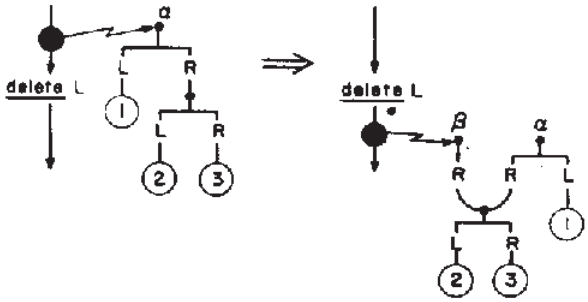

Figure 4. Operation of the delete actor.

Structure Representation

The storage of structures and the execution of instructions representing structure actors occurs in the structure processing section of a data-flow processor. The structure processing section consists of a Structure Operation Unit and a Structure Memory and attendant Arbitration and Distribution Networks. This section of the processor is viewed as a functional unit by the instruction processing section; that is, operation packets specifying structure operations are sent to the section, and data packets are returned. The organization of the structure processing section is shown in Figure 5.

Operation packets containing instructions representing structure actors are transmitted to the Structure Operation Unit by the instruction processing section. The Structure Operation Unit controls the execution of the instruction specified in each operation packet through instruction packets sent to the Structure Memory. The Structure Memory holds all structure values of the data-flow program, and all structure operations are performed in the Memory. Upon completion of a structure operation, the Structure Memory transmits a data packet containing the resulting elementary or structure value to the instruction processing section.

A node of a structure is contained in a two register Cell known as a Structure Cell and designated by a Cell identifier. The two registers of the Cell contain the left and right components of the structure, respectively; and hence no selector need be stored in a register. The first field of a register is a use code which indicates whether the item stored in the second field is the identifier of another Cell or an elementary value, or if the register is empty. A memory representation of the simple structure of Figure 1 is given in Figure 6.

The Structure Memory is composed of a number of Structure Cells. Each Structure Cell is capable of holding one node of a structure, and the identifier of the Cell specifies a path through the Distribution Network to the Cell. The Structure Memory receives instruction packets from the Structure Operation Unit commanding a specific Structure Cell to execute some structure operation upon the node located in the Cell. Upon completion of the operation specified in an instruction packet, a Structure Cell presents any result as a data packet to the Arbitration Network for conveyance to the instruction processing section. Any further structure operations are specified in instruction packets returned to the input of the Structure Memory.

A Structure Cell within the Structure Memory performs one of three operations upon the structure node contained in the Cell. The possible operations are:

1. select. Upon receipt of an instruction packet specifying a select operation

$$\left\{\begin{matrix}\underline{\text{select}}\ \text{dest} \\ s\end{matrix}\right\}$$

a Structure Cell follows one of two procedures, controlled by whether the selector s is a simple selector or a compound selector.

   a. If s is a simple selector, L or R, the content c of the Cell register designated by s is used to form a data packet

$$\left\{\begin{matrix}\text{dest} \\ c\end{matrix}\right\}$$

   which is presented to the Arbitration Network for transmission to the specified destination dest in the instruction processing section of the processor.

   b. If s is a compound selector $s_1 s_2 \ldots s_n$, $s_1 \in \{L, R\}$, the content β of the register designated by s_1 is the identifier of a Structure Cell and is used to form the instruction packet

$$\left\{\begin{matrix}\beta \\ \underline{\text{select}}\ \text{dest} \\ s_2 \ldots s_n\end{matrix}\right\}$$

   which is presented to the Arbitration Network for transmission to the input Distribution Network of the Structure Memory. The process is then repeated with the selector s_2 at Structure Cell β.

2. alter. The receipt of an alter instruction

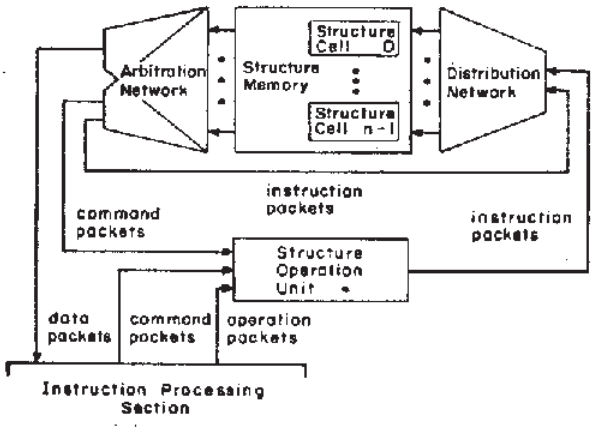

Figure 5. Organization of the structure processing section of the data-flow processor.

Cell α

elem	a
struc	γ

Cell β

struc	γ
elem	d

Cell γ

elem	b
struc	c

Figure 6. Memory representation of the structure of Figure 1.

$$\begin{Bmatrix} \underline{\text{alter}}\ \text{dest} \\ s \\ x \end{Bmatrix}$$

indicates that the contents of the Structure Cell are to be modified so the component designated by the selector s is set to x. Since structure values are not modified, a Structure Cell that receives an alter instruction, must receive two alter instructions, one for each register. When both have been received, a data packet containing the Cell identifier β is returned to the instruction processing section:

$$\begin{Bmatrix} \text{dest} \\ \beta \end{Bmatrix}$$

3. copy. A copy instruction

$$\begin{Bmatrix} \underline{\text{copy}}\ \beta \\ \text{dest} \\ s \end{Bmatrix}$$

specifies that the content of the register designated by s is to be transmitted to Structure Cell β. An instruction packet

$$\begin{Bmatrix} \beta \\ \underline{\text{alter}}\ \text{dest} \\ s \\ c \end{Bmatrix}$$

is formed of the register content c and is presented to the Arbitration Network for transmission to the Input Distribution Network.

Instructions are transmitted to the Structure Memory as instruction packets, each consisting of a Cell identifier and an instruction. The Cell identifier specifies a path through the Distribution Network to the Cell, and the packet received by a Cell consists of merely the instruction portion of the instruction packet.

The Structure Operation Unit maintains the reference count of each node in the Structure Memory, specifying the number of arcs terminating on the node and the number of references to the node existing in the instruction processing section of the processor. When a node becomes inaccessible due to the execution of some instruction of the program, the reference count of the node becomes zero, and the node is placed on a free node list which is used for the allocation of new nodes during program execution.

The processing of all structure operation packets by the Structure Operation Unit permits the unit to properly decrement reference counts as references to items are deleted through instruction execution. References to items are created in the Structure Memory by execution of a select instruction if the selected item is a structure value and in the instruction processing section through execution of an instruction representing a link of a data-flow program. We must require that in either case, command packets of the form

$$\begin{Bmatrix} \text{node identifier} \\ \underline{\text{up}} \end{Bmatrix}$$

are sent to the Structure Operation Unit, causing the reference count of the designated node to be properly incremented.

Now that we have considered the operation of a Structure Cell within the Structure Memory, we can describe the execution of each of the structure actors merely by listing the procedure followed by the Structure Operation Unit in processing the instruction. For the purposes of this discussion, it is assumed that all selectors are simple selectors.

The processing of a select instruction by the Structure Operation Unit merely causes the reference count of the designated node to be decremented. The content of the operation packet is then sent as an instruction packet to the specified node of the Structure Memory for execution of the select operation.

A construct instruction

$$\begin{Bmatrix} \underline{\text{construct}}\ \text{dest} \\ \text{L:}\ \alpha \\ \text{R:}\ \gamma \end{Bmatrix}$$

specifies that a new node is to be created with components α and γ, designated by the selectors L and R. The instruction is implemented by the Structure Operation Unit as two alter operations in the following manner:

1. Accept an identifier β from the free node list.
2. Transmit to the Structure Memory the instruction packets

$$\begin{Bmatrix} \beta \\ \underline{\text{alter}}\ \text{dest} \\ \text{L} \\ \alpha \end{Bmatrix} \quad \text{and} \quad \begin{Bmatrix} \beta \\ \underline{\text{alter}}\ \text{dest} \\ \text{R} \\ \gamma \end{Bmatrix}$$

transferring the values α and γ to the correct registers of β.

An operation packet containing an append instruction is of the following format

$$\begin{Bmatrix} \underline{\text{append}}\ \text{dest} \\ \text{L:}\ \alpha \\ x \end{Bmatrix}$$

where L is the selector of the element in Structure Cell α which is to be replaced by x in the new structure. The procedure followed by the Structure Operation Unit in execution of the instruction is as follows:

1. Accept an identifier β from the free node list.
2. Transmit the instruction packets

$$\begin{Bmatrix} \alpha \\ \underline{\text{copy}}\ \beta \\ \text{dest} \\ \text{R} \end{Bmatrix} \quad \text{and} \quad \begin{Bmatrix} \beta \\ \underline{\text{alter}}\ \text{dest} \\ \text{L} \\ x \end{Bmatrix}$$

to the Structure Memory to copy the register of node α designated by the selector R into Cell β and set the L-component of β to x.

An operation packet specifying a delete instruction is processed in a similar manner, causing the use code of the designated register to be set to empty.

To assure maximum use of the Structure Cells of the processor, the structure processing section utilizes a multi-level memory, so that only active structure nodes occupy the Structure Cells. The

Structure Memory acts as a cache for structure nodes; individual nodes are retrieved from the auxiliary memory as they become required for computation, and structure nodes are sent to the auxiliary memory upon creation through execution of an append, delete, or construct instruction. The structure of the auxiliary memory as a packet communication system is described by Dennis [4], and its use in conjunction with the structure processing section is presented in [10].

Conclusion

The described techniques for the implementation of data structures can be readily extended to larger and more complex structures. In order to implement structures with a fixed maximum number of arcs emanating from each node, the size of a Structure Cell is increased to accommodate the new node size. The use of arbitrary (to a fixed maximum size) integers or character strings as selectors is accommodated through the addition of a selector field to each register. A Structure Cell must then have the capability to choose from the node contained in the Cell an item whose selector matches a specified selector. These extensions allow the representation of a wide variety of structures, including the programs of the data-flow language [10].

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. Packet communication architecture. Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, New York, August 1975.

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

6. 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, 402-409.

7. 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, 126-132.

8. Karp, R. M., and R. E. Miller. Properties of a model for parallel computations: determinacy, termination, queueing. SIAM Journal of Applied Mathematics 14 (November 1966), 1390-1411.

9. 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.

10. Misunas, D. P. A Computer Architecture for Data-Flow Computation. SM Thesis, Department of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass., June 1975.

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

12. Rumbaugh, J. E. A Parallel Asynchronous Computer Architecture for Data Flow Programs. Report TR-150, Project MAC, M.I.T., Cambridge, Mass., May 1975.

13. Rumbaugh, J. E. A data flow multiprocessor. Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing, IEEE, New York, August 1975.