ok- a reprint

A VIEW OF DATAFLOW*

by

Kim P. Gostelow
Robert E. Thomas

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

June 1979

Technical Report 128

---

*This work was supported by NSF Grant MCS76-12460: The UCI Dataflow Architecture Project. To appear in 1979 AFIPS Conference Proceedings.

A view of dataflow*

by KIM P. GOSTELOW and ROBERT E. THOMAS
University of California
Irvine, California

INTRODUCTION

In 1946 John von Neumann outlined an organization for computers[1] that has dominated the languages and architecture of machines to this day—the familiar sequential, one-word-at-a-time instruction stream which modifies the contents of a memory. Although the von Neumann model has proved to be a viable and powerful approach to computation, we have chosen to explore other models of computation to determine if they offer advantages in ease of programming, exploitation of concurrency and performance. A primary motivation is new technology such as large scale integration (LSI) which has greatly expanded the range of choice in computer design.

Dataflow is an alternative model of computation which is particularly promising. The basic principles of dataflow are asynchrony and functionality, and thus are in distinct contrast to the von Neumann model. Readers familiar with look-ahead processors[2] such as the IBM 360/91 and the CDC 6600/7600 will find that the principles of dataflow are not new. However, our goals in exploiting the principles of dataflow are of a more fundamental nature than the goals of the above systems. Rather than using dataflow simply to improve the performance of von Neumann processors, we have adopted the semantics of dataflow as the base semantics of our system. A primary reason for this direction is a desire to explore the full generality of dataflow. Another reason, perhaps of greater importance, is our impression that the functional nature of dataflow simplifies the semantics of programming languages and thus may reduce the cost of software (especially in the case of multiprocessor systems). Our approach is first to design a fully-integrated system before attempting to construct hardware. This includes the design of a base machine language, a preliminary high-level language,[3] a user protection facility,[4] and a high-level exception handling facility,[5] all of which are based on the semantics of dataflow.

The following sections discuss details of the principles of dataflow with emphasis on a method of interpretation developed at Irvine. The general version of this interpreter is known as the *unfolding interpreter* and is described in the following section. (Details on a specific unfolding interpreter are available elsewhere.[3]) The third section presents implementation techniques for dataflow systems while the fourth section discusses some principles of multiprocessor design currently being developed.

* This work was supported by NSF Grant MCS76-12460: The UCI Dataflow Architecture Project.

BASIC PRINCIPLES OF DATAFLOW AND THE UNFOLDING INTERPRETER

The present section concentrates on the logical implications of dataflow semantics without regard to physical implementations, efficiency, etc. These latter topics are discussed in later sections.

Asynchrony and Functionality

The following introduces dataflow by showing the correspondence between constructs in the high-level dataflow language *Id* (for *I*rvine *d*ataflow) with schemata in a graphical dataflow machine language. The goal is twofold—first to show by example that programs need not be written in dataflow machine language, and second, to provide some intuition for understanding the execution of dataflow programs. We wish to emphasize that our purpose is to present the basis of dataflow and not to discuss the syntax of a particular dataflow language (Id), or the details of a particular machine language.[3]

Consider the following Id constructs:

```
s←(initial sum←0
    for i from 1 to n do
      new sum←sum+f(i)
    return sum);                                    (2.1)

procedure sum (i, k)
(return (if i > k then 0
                  else f(i)+sum(i+1, k)));
s←sum(1, n);                                        (2.2)
```

both of which can be expressed in mathematical terms as

$$s = \sum_{i=1}^{n} f(i)$$

Statement (2.1) is an assignment statement whose right-hand side is an Id loop expression. The statements in (2.2) are a

procedure definition followed by an application of that procedure. Each of these constructs has a number of inputs—a value for n, a definition for the function procedure f, and in the case of the sum procedure definition, the value of i. In addition, both (2.1) and (2.2) produce the same result. We abstract these two definitions of "sum" by considering each to be a "black box" as shown in Figure 1a. Each dark spot in the figure represents the presence of a data item referred to as a *token*. For now, we can consider a data item to be an instance of an integer, real, or boolean value.

The mechanics of computation within a black box can be ignored as long as three conditions are met: 1) a complete set of input values (i.e. tokens) is consumed, 2) the computation within the box has no effect on other computations except perhaps to compete for resources (i.e. there are no semantic side-effects), and 3) a complete set of result tokens is always produced if the computation terminates. A black box meeting these requirements is a *function*. The basis of dataflow is the definition and operation of interconnected functions. One way, for example, to interconnect functions is by composition (Figure 1b). Other dataflow interconnection schemes including cycles have been devised[3,6,7] but are not discussed here.

As opposed to the sequential, one-instruction-at-a-time memory cell semantics of the von Neumann computer, the basic principles of dataflow are:

1. Operations execute when and only when the required operands are available (asynchrony).
2. Operations are functions (there are no side-effects).

These principles imply that the order of execution of two functions, such as e and g in Figure 1b, is irrelevant since the computations internal to e and g cannot interact. Thus e and g can be computed concurrently. Such concurrency, present in the interconnection graph itself, is called *static parallelism*. A more interesting example of the asynchrony achievable in dataflow occurs when a function is executed more than once, either by iteration or by recursion (for example, function f in (2.1) and (2.2) previously). As shown in Figure 1c, suppose that a dataflow machine replicates the function f and its input and output lines for as many times as f is executed. Since f has no side-effects, each copy of f can be computed in any order or concurrently. This concurrency is called *dynamic parallelism* since the concurrency potential depends on the number of repetitions (determined at execution time) of the function f. Dynamic

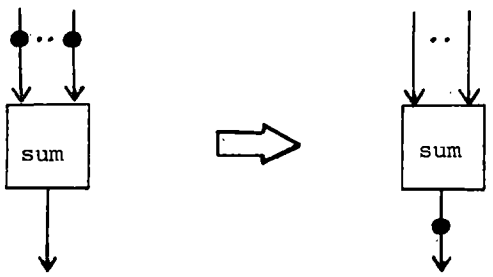

Figure 1a—Abstraction of the Id construct sum

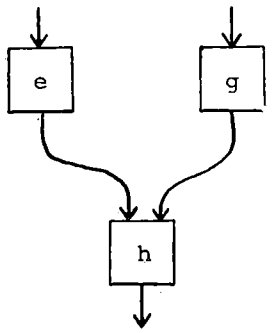

Figure 1b—Composition of functions

parallelism is of particular interest because it can affect the time complexity of an algorithm. For example, suppose the time complexity of function f in (2.1) and (2.2) is O(m) (i.e. assume that f has an additional parameter m). Then on a sequential machine the time complexity of either (2.1) or (2.2) would be O(nm). However, on a dataflow machine capable of dynamic parallelism, the processing time complexity would be O(n+m) because the time required is O(n) to generate the n instances of f, plus O(m) to simultaneously compute all instances of f (assuming O(n) processors are available), plus O(n) again to sum the resulting values. The total is O(n+m+n)=O(n+m), where for simplicity we have ignored some important considerations such as communication conflicts.

It is important to note that the input and output lines of a function are replicated for as many times as the function is executed. This implies that at most one token will ever travel on any given line instance, thus preserving functionality. In addition, the situation shown in Figure 2 is precluded by the "single-assignment rule" present in the high level language.

The replication of f and its input and output lines does not alone ensure dynamic parallelism since data dependencies in the program may inhibit it. For example, if the previous definition of sum is changed to

```
s←(initial sum←0; x←5
   for i from 1 to n do
     new x←f(x);
     new sum←sum+new x
   return sum)
```

that is

$$s = \sum_{i=1}^{n} x_i$$

where $x_i = f(x_{i-1})$ for $x_0 = 5$ and $1 \leq i \leq n$

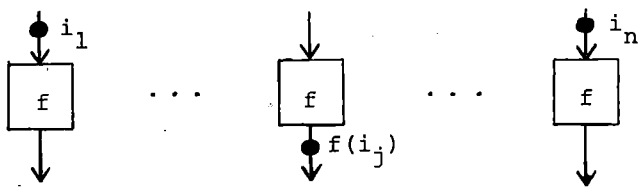

Figure 1c—Instances of function f

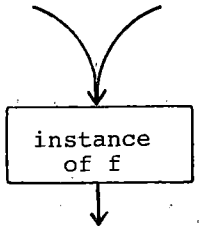

Figure 2—An illegal connection

then the input to f depends on the value computed by the previous instance of f. That is, the instances of f must be computed sequentially.

The Unfolding Interpreter

Using the simple notions of asynchrony and functionality discussed above, we present an interpreter which manages a *context* for each value produced and consumed in the system. The purpose of context management is to logically separate and direct the values to the proper instance of each function.

At this point we note that other dataflow interpreters have been defined[6,7] which rely on either fixed-size buffers and request/acknowledge communication between functions, or the assumption of an unbounded FIFO queue between each interconnected function. The unfolding interpreter is capable of far more asynchronous operation than these other interpreters because of the function copying it performs.

Each execution instance of a function is called an *activity* and is uniquely identified by an *activity name*. An activity name comprises two parts denoted u.*l* where u is the context part and *l* is a unique label referencing the description of the function to be computed by that activity. In addition, the referenced function description specifies the destination labels to be used for transmission of result tokens. A *dataflow object program* is a set of labeled function descriptions. The actions of the interpreter can now be stated:

1. Tokens generated by the execution of activities are grouped by activity name.
2. When the input tokens to an activity become available, the activity is executed according to its description.
3. Output tokens are produced by tagging the values resulting from the execution of an activity with the destination's activity name. The u part of the destination activity name is derived from the u part of the activity name of the producing activity according to a set of rules (some examples are given below). The *l* part is derived from the output destination information which is part of the description of the function executed by that activity. Note that the act of computing a destination activity name is equivalent to creating a "logical line instance" extending from the producing activity to the destination activity.

To illustrate, consider the dataflow object program in Figure 3a. Let the activity name of an instance of e be u.*l*. The rule for function composition says that the context of the output is identical to the context of the input, and the label of the output is specified by the description of the function being executed, i.e. the program code. This results in activity u.*l* producing an output token with destination activity name u.t. Now consider the case when an activity itself comprises (smaller) activities, for example a procedure call, which is provided for by the base machine language primitives A, BEGIN, END, and A^{-1} as shown in Figure 3b. This figure shows the creation of a new set of activities resulting from the application (call) of procedure f. Note that the description of procedure f is one of the input values to the procedure application box. (We will not be concerned here with the representation of procedure values.) Activity name generation for procedure call is as follows:

- *The A (activate) primitive*—Assume that the activity name of an instance of A is u.*l*. Since the procedure call represents a change in context, the A primitive "stacks" the context part u within the new activity name, thereby creating a unique context for the activities within f. The activity name produced is u′.beginf where u′=u.*l*. Also by convention, the A primitive groups into one vector value all of the input arguments so that exactly one input argument token is always delivered to the newly created instance of f.
- *The BEGIN primitive*—The purpose of BEGIN is to distribute the input arguments to the activities within f with no further change in the context u′.
- *The END primitive*—The END primitive "unstacks" the context u′ to reveal the outer context and the label *l*. It then constructs the activity name u.t (the activity to which the result of the procedure is to be returned) which can be accomplished in a number of ways. For example, t could be computed from *l* according to an agreed-upon rule. Also, by convention, the END primitive combines all output values onto one token for transmission to the A^{-1} activity.
- *The A^{-1} (terminate) primitive*—The purpose of the A^{-1} primitive is to distribute the results of f to activities in the outer context with no further change in the context u.

Though not presented here, other schemata for the un-

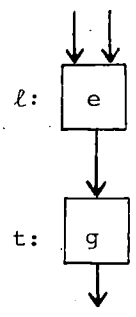

Figure 3a—a Composition of functions

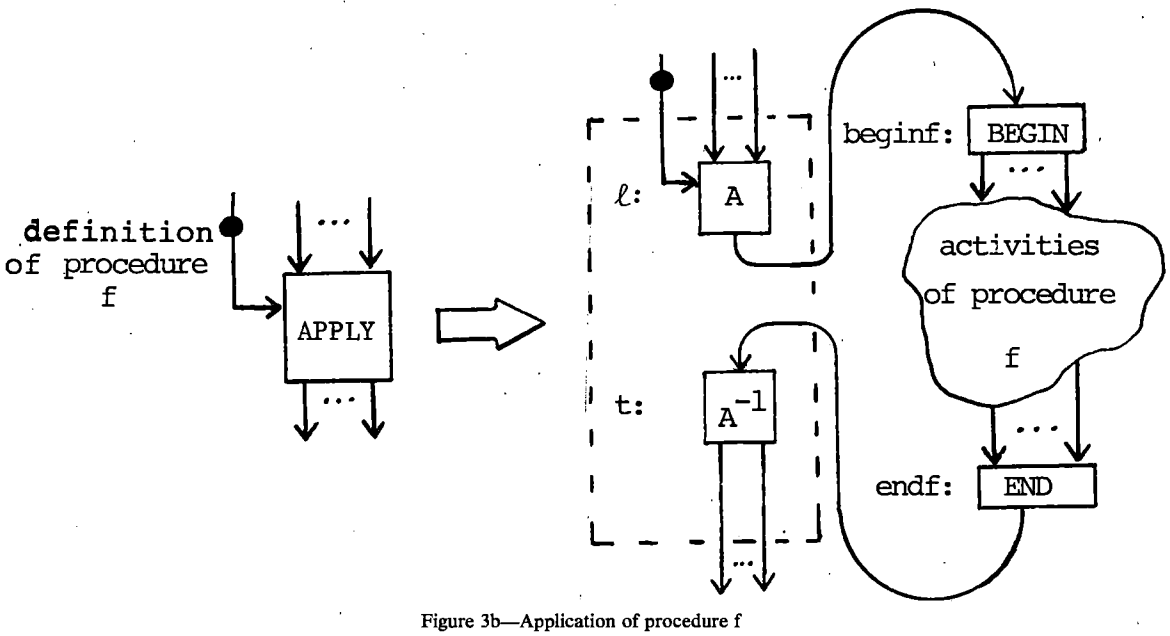

Figure 3b—Application of procedure f

folding interpreter have been devised.[3] In particular there is a *loop schema* which "unfolds" the loop body (including nested loops) to expose dynamic parallelism. (The unfolding interpreter gets its name from this capability.) The loop schema depends on adding a new field, i, to the context part of an activity name to yield (u.i).*l*. Like recursion, each iteration of a loop exists in a distinct context generated simply by incrementing the i field in the activity name.

IMPLEMENTATION SCHEMES

In this section we discuss techniques for efficient implementation of dataflow. Although von Neumann computers may be used to implement these techniques within a dataflow system, we rigidly maintain that the semantics of dataflow are the only semantics visible external to the system—a principle we consider vital to the success of dataflow.

Dataflow Structures and Memory

Operation of the unfolding interpreter requires many copies of program code. Logical copies are sufficient and can be created simply by copying the pointer to a physical copy since all code (and data) is read-only. (Note that the label *l* in an activity name is equivalent to a pointer.) Of course in a multiprocessor environment, having just one physical copy may imply a bottleneck. In this case, we consider it the responsibility of a particular implementation to selectively make physical copies (in distinct memories) to reduce the bottleneck.

Similiar remarks hold for the transmission and replication of values larger than simple integers, reals, booleans, etc. The need for logical copies is especially evident when a value, such as an entire matrix, is transmitted between two functions and the receiving function utilizes only a small part and discards the rest. Also, a common programming task is the production of a data object which differs in only small ways from another (perhaps large) input data object. Because dataflow values can never be modified, an entire new object must be created, making the straightforward copy-all approach quite expensive.

Dennis has shown[6] that the amount of copying can be reduced by properly defining a SELECT and APPEND operation on "structured" data residing in a conventional memory. A *dataflow structure* is a set of ⟨selector:value⟩ pairs where a *selector* is an integer or string and a *value* is any dataflow value (including another structure). A dataflow structure is always a tree (e.g. Figure 4a). The SELECT function (subscripting) has two arguments, a structure value and a selector, and it yields the value at the specified selector. The APPEND function has three arguments: a structure value, a selector, and a value to be appended to the given structure at the specified selector. APPEND does *not* modify the given structure but instead makes a logical copy of it with the new selector and value appropriately placed. This

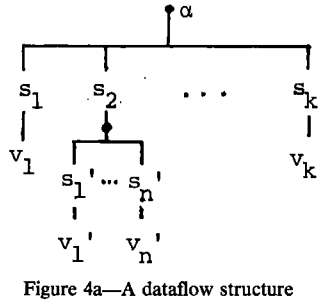

Figure 4a—A dataflow structure

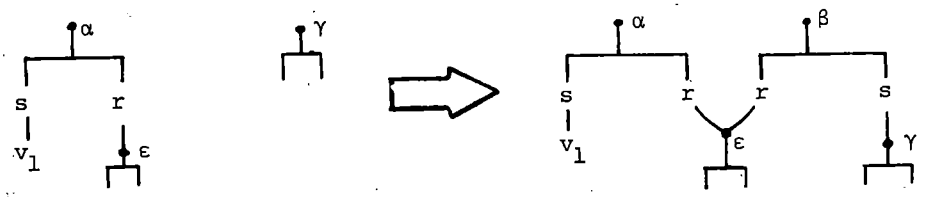

Figure 4b—APPEND (α, s, γ)→β

can be implemented (with pointers) such that a physical copy need be made only of the "top level" of the original structure value. Thus sub-structures can be shared between any number of structure values without violating dataflow semantics. A simple example is given in Figure 4b where both structures (logical trees) physically share the sub-structure at selector r.

Since the definition of dataflow structures precludes the construction of internal cycles, a simple reference count scheme can be used to reclaim structures no longer needed. The reference count method is also helpful in detecting the special case of an APPEND to a structure when there is only one logical copy of that structure (i.e. reference count equals one). In this case APPEND can quickly produce its output by simply updating the old structure in place.

In the following we consider several explicit representations for dataflow structures. When a dataflow structure has contiguous integer selectors, a vector of contiguous memory words may be used where one value (or a pointer to a value) is stored per memory word. This is termed *array* representation. It is easy to see that in array representation a (one level) SELECT can be done in constant time while APPEND requires O(n) time (for copying), where n is the number of words in the result memory vector.

A second representation, termed *selector vector*, is a fairly compact representation when string or sparse integer selectors appear in the dataflow structure. Again a contiguous memory vector is used but the (ordered) selectors are explicitly stored with the values. SELECT can then be done in O(log n) time using a binary search while APPEND requires O(n) time where n is the number of selectors.

A third representation of a dataflow structure is a modification to a "balanced" tree scheme such as an AVL tree, B-tree, or B*-tree.[8] In the following we have selected a specific B-tree, the 2-3 tree, to illustrate the concept. A 2-3 tree is a tree in which every vertex that is not a leaf has either two or three sons, and every path from the root to a leaf is of the same length.[9] A dataflow structure and its 2-3 tree representation is given in Figure 4c. Each internal vertex of a 2-3 tree contains the value of the largest selector appearing in its sub-tree. These values are used in the SELECT (and APPEND) operation to guide a modified binary search requiring O(log n) time, where n is the number of leaves in the tree. APPEND can also be done in O(log n) time,[10] where none of the vertices of the original tree are disturbed (except perhaps for reference counts) and most of the original 2-3 tree is shared between the argument and result structures without affecting functionality. The 2-3 tree representation also promotes efficient concatenation of dataflow structures (an operation quite useful in programs such as quicksort, fast Fourier transform, etc.) with constant time required in the best case and O(log n) time required in the worst case, given certain restrictions are met in the input structures.[10] However, a significant disadvantage with 2-3 tree representation is the extra memory required for the internal vertices of the tree; and while asymptotic behavior is good, O(log n), the constant factor in the equation also could be significant.

The reader may note that the design of an efficient dataflow memory system involves problems (i.e. memory management, garbage collection, choice of representations, etc.) which also occur on conventional systems. Currently these tasks are reprogrammed to some extent by each application program that is written. Our feeling is that by embedding these tasks within the system as close to hardware as is practical, a significant burden is removed from the application programmer, and if a good design is obtained, average performance will improve.

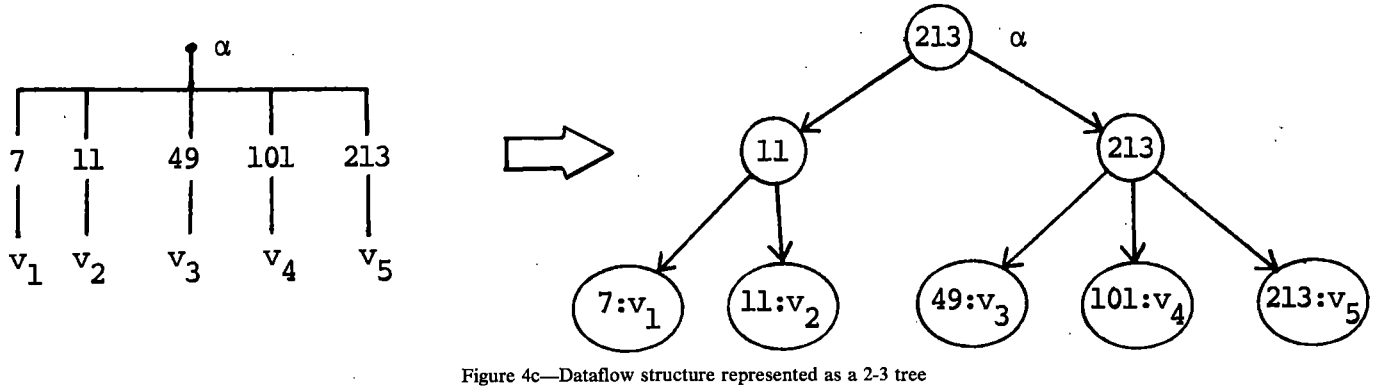

Figure 4c—Dataflow structure represented as a 2-3 tree

Implementation of the Unfolding Interpreter

One problem with the theoretical unfolding interpreter is the unbounded length of activity names, the primary purpose of which is to logically separate the tokens so that the inputs to each copy of each function are uniquely determined. For this purpose a unique number, N, for each context combined with the label *l* is sufficient. For example, starting with activity name N.*l* for the A activity in Figure 3b, the context is changed by obtaining a new unique number N′ to form activity name N′.beginf. In addition, an association is made in memory between N′ and the activity name N.t; alternatively a token carrying N.t can be sent from the A to the END activity. In either case, the return from the inner context is accomplished by the END activity which fetches the information N.t associated with the number N′ and uses it as a logical "return address" to the outer context.

The major question in this approach is the method of generating and managing the unique numbers. If enough bits (say, 60) are used to store N, uniqueness of N can be guaranteed to extend over the life of the system. However, if the system guarantees that none of the activities (and their associated tokens) exist from the previous use of N, a value N can be reused and significant savings achieved. This is generally not a problem in context management on sequential machines; however, due to the asynchrony of dataflow it is possible, for example, that an END activity finishes execution before all of the activities in the procedure have finished, even when the system guarantees that all such activities will eventually finish. One workable solution is to prevent END from finishing execution before all other activities and tokens of that procedure application have been consumed. With this restriction, a simple scheme such as a tree of stacks (cactus) can be used to implement unique number management with reusable names.

A second problem with the unfolding interpreter is the requirement for system-wide unique labels for object code. As is common in conventional systems, a tree-structured directory system with path specifications may be used to implement dataflow labels. If desired, path specifications can be included within activity names. For example, *l* can be split into two fields p.s where p is a pointer to a procedure and s is a function (statement) number in that procedure.

MULTIPROCESSOR DESIGN PRINCIPLES

Although dataflow principles can be advantageously applied to conventional systems, we believe that new concepts in computer architecture must be developed to take full advantage of the concurrency and functionality of the dataflow model. This final section is largely speculative because of the difficulty of accurately predicting the performance of proposed architectures. However, we have simulated variations of a specific dataflow architecture and the results are reported in detail elsewhere.[11] Moreover, since the design of this architecture contains much detail and changes rapidly, we summarize our experience with it in the form of tentative principles for the design of one possible form of dataflow machine. These principles are not new and have been applied to many systems. However, such a statement serves as a point of comparison with other views.

Our goal is to design a general purpose computing system which

1. Can effectively distribute small pieces of a computation over many processors in the machine;
2. Is modular enough so that additional blocks of processors can be easily added to increase the capacity of the machine;
3. Has a measure of fault tolerance so that hardware failures may decrease performance but will not necessarily halt the machine (i.e. fail-soft);
4. Does not require knowledge of the number and configuration of processors to write programs which effectively utilize these resources;
5. Does not depend on expensive interconnection schemes (e.g. crossbar switch) or extremely fast circuit speed for good performance;
6. Can support a number of simultaneous users.

The first principle of multiprocessor design (evidenced by simulation results) is the program-dependent tradeoff between distribution and localization of computation. Distribution may allow concurrent execution of a program, but it also tends to increase communication costs. Thus for any particular architecture and computation, there exists some optimal degree of distribution (perhaps as little as one processor) for which execution time is minimized. Locality (e.g. "the working set" in paging systems) is an established principle of conventional systems. Moreover, we believe that locality will be present to an even greater extent in dataflow due to the absence of side-effects and due to the high degree of structure imposed by our high level dataflow language. To take advantage of locality, we must consider two features of a dataflow system: 1) how the topology of the architecture allows reduced communication costs when physical locality is present and 2) how program locality is preserved in the mapping to physical hardware.

One topology which supports locality is a hierarchy of modules. For example, a primitive module could be a processor with memory which can execute any dataflow machine language instruction or group of instructions, including an entire dataflow program (the extreme case of locality). Primitive modules are connected to form larger modules which are then connected, etc., such that any module can be considered to consist of some processing power and some associated memory. This structure supports locality to the extent that communication within any given module can be made less costly than communication off the module.

A second aspect of locality is the mapping of program locality to physical locality in the machine. Assume each processor in the machine to have a distinct physical address. We have found that the activity names themselves constructed by the unfolding interpreter contain much of the locality information present in the source program. For example, activity names with the same context part belong to

the same instance of a procedure (or loop). In addition, the labels *l* in the object program can be assigned by the compiler such that numerically-close labels suggest close connection of functions. These pieces of information can be used by an activity *assignment function* which maps from logical activity names to physical processor addresses. The resulting physical address is placed on each output token to guide its transmission in the communication network. The selection of an appropriate assignment function is similiar to the problem of selecting an appropriate hash function for a scatter table but with the additional consideration of preserving locality where appropriate. Note that the assignment function and the communication network perform a partial sorting of activity names by physically directing tokens to their destinations. The final sorting is done by each processor on only those activity names which map to its physical address.

As a practical matter, designing a fairly good assignment function which distributes computation while preserving locality is not too difficult to do (at least for the small selection of programs we have executed on our simulator). However, the selection of an optimal assignment function is a difficult problem requiring further investigation. We also note that it is possible for the machine to tune its assignment function(s) at execution time for improved performance.

The second principle of multiprocessor design we have adopted is that the communication delay (ignoring conflicts) between any two processors should be no more than O(log n) where n is the total number of processors. If we rule out complete interconnection schemes, this principle also suggests a hierarchical interconnection of modules. However, a tree is not the only structure with the O(log n) property. Two other examples are the boolean n-cube[12] and the interconnection network of Wittie.[13] Both of these networks can be viewed as trees in which sufficient additional connections have been made such that the root node has become indistinguishable. (In other words, pick any node in the network; then appropriate connections can be deleted so that a tree remains.) Although much investigation remains to be done before selecting a particular interconnection network, we feel that a more highly connected structure than a tree is appropriate for two reasons: 1) the extra connections provide some measure of fault tolerance and 2) more flexibility is allowed to map the logical tree structure of many programs into the many physical trees present in the network. Currently we are favoring a modification of the Wittie network due to its lower implementation cost.

The last principle of multiprocessor design is recognition of the potential benefits of redundant copies of data and program code. Conventional systems have already developed this concept to some extent, primarily in the area of virtual memories and high speed caches. However, dataflow can take further advantage of redundant copies because values are never modified. We have three goals in pursuing the concept of redundancy: 1) to improve performance through concurrent access of data in distinct memories, 2) to improve performance through a caching scheme which localizes data to where it is most used, and 3) to identify each data copy so that if one copy is damaged, a search can be instituted to obtain another valid copy. For those readers interested in possible mechanisms to achieve some of these goals, Reference 11 should be of some help.

CONCLUSIONS

The decision to incorporate the full generality of dataflow is not without its costs. We have seen that the principles of dataflow sometimes suggest implementations which make "inefficient" use of memory. Of course better implementations may yet be found but we suggest that problems with memory be viewed in the context of the following points: 1) the full generality of dataflow is not always required—for example a program like matrix multiplication can be executed within the semantics of dataflow and still require little more memory than does a conventional machine, 2) duplication of data and code can have various benefits such as concurrent memory access and the possibility of recovering from hardware faults, and 3) the cost of hardware and memory is decreasing while the cost of software and system failures will probably continue to increase. Thus in a few years the "efficient" use of memory in many situations might be viewed quite differently.

An introduction to dataflow is not complete without mentioning other issues and capabilites. Dataflow *streams* and *managers* are available to program history-sensitive applications such as airline reservation systems, resource management, etc. directly in a high-level language;[14] in addition, abstract data types are available.

REFERENCES

1. Taub, A. H. (Ed.), *Collected Works of John von Neumann*, Vol. 5, The Macmillan Company, New York, 1963, pp. 34-79.
2. Keller, R. M., "Look-ahead processors," *ACM Computing Surveys*, Vol. 7, No. 4, December 1975, pp. 177-195.
3. Arvind., K. P. Gostelow and W. Plouffe, "An asynchronous programming language and computing machine," *Dept. of Information and Computer Science Technical Report 114A*, University of California, Irvine, CA, 1978.
4. Bic, L., "Protection and security in a dataflow system," *Dept. of Information and Computer Science Technical Report 126*, (Ph.D. Dissertation) University of California, Irvine, CA, 1978.
5. Plouffe, W., "Exception handling and recovery in a dataflow system," Ph.D. Dissertation, Dept. of Information and Computer Science, University of California, Irvine, CA, (in preparation).
6. Dennis, J. B., "First version of a data flow procedure language," *Symposium on Programming*, Institut de Programmation, University of Paris, Paris, France, April 1974, pp. 241-271 (also *MAC Technical Memorandum 61*, LCS/MIT, Cambridge, MA, May 1975).
7. Davis, A. L., "The architecture and system method of DDM1: a recursively structured data driven machine," *ACM/IEEE Fifth Annual Symposium on Computer Architecture*, Vol. 6, No. 7, April 1978, pp. 210-215.
8. Knuth, D. E., *The Art of Computer Programming*, Vol. 3, *Sorting and Searching*, Addison-Wesley, Reading, 1973, pp. 451-480.
9. Aho, A. V., J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, 1974, pp. 145-155.
10. Thomas, R. E., "A comparison of methods for implementing dataflow structures," *Dataflow Architecture Project Note 35*, Dept. of Information and Computer Science, University of California, Irvine, CA, May 1978.

11. Gostelow, K. P. and R. E. Thomas, "Performance of a dataflow computer," *Dept. of Information and Computer Science Technical Report 127*, University of California, Irvine, CA, 1979.
12. Sullivan, H. and T. R. Bashkow, "A large scale, homogeneous, fully distributed parallel machine, I," *ACM/IEEE Fourth Annual Symposium on Computer Architecture*, Vol. 5, No. 7, March 1977, pp. 105-117.
13. Wittie, L. D., "Efficient message routing in mega-micro-computer networks," *ACM/IEEE Third Annual Symposium on Computer Architecture*, Vol. 4, No. 4, January 1976, pp. 136-140.
14. Arvind., K. P. Gostelow, and W. E. Plouffe, "Indeterminacy, monitors, and dataflow," *Proc. Sixth ACM Symp. on Operating Systems Principles*, November 1977, pp. 159-169.