Ann. Rev. Comput. Sci. 1986. 1:225–53
Copyright © 1986 by Annual Reviews Inc. All rights reserved

DATAFLOW ARCHITECTURES

Arvind and David E. Culler

Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, Massachusetts 02139

ABSTRACT

Dataflow graphs are described as a machine language for parallel machines. *Static* and *dynamic* dataflow architectures are presented as two implementations of the abstract dataflow model. Static dataflow allows at most one token per arc in dataflow graphs and thus only approximates the abstract model where unbounded token storage per arc is assumed. Dynamic architectures tag each token and keep them in a common pool of storage, thus permitting a better approximation of the abstract model. The relative merits of the two approaches are discussed. Functional data structures and I-structures are presented as two views of data structures that are both compatible with the dataflow model. These views are contrasted and compared in regard to efficiency and exploitation of potential parallelism in programs. A discussion of major dataflow projects and a prognosis for dataflow architectures are also presented.

1. DATAFLOW MODEL

The dataflow model of computation offers a simple, yet powerful, formalism for describing parallel computation. However, a number of subtle issues arise in developing a practical computer based on this model, and dataflow architectures exhibit substantial variation, reflecting different standpoints taken on certain aspects of the model. For example, in the abstract dataflow model, data values are carried on *tokens,* which travel along the arcs connecting various instructions in the program graph, and it is assumed that the arcs are first-in-first-out (FIFO) queues of unbounded capacity (Kahn 1974). This

8756-7016/86/1115-0225$02.00

gives rise to two serious, pragmatic concerns: (*a*) How should the tokens on arcs be managed? (*b*) How should data structures, which are essentially composites of many tokens, be represented? The manner in which these concerns are resolved has major impact not only on the machine organization but also on the amount of parallelism that can be exploited in programs. In this review, we examine the major variations in dataflow architectures with regard to token storage mechanisms and data structure storage.

The paper is organized as follows. Section 1 introduces dataflow program graphs and the rules that determine when and how operations are performed. Also, it explains why data structures cannot be viewed as they are in conventional programming languages without seriously compromising the suitability of the dataflow approach for parallel processing. Section 2 examines the two token storage mechanisms adopted in current dataflow architectures. The *static dataflow* approach allows only one token to reside on an arc at any time, while the *tagged-token dataflow* approach allows essentially unbounded queues on the arcs with *no* ordering, but with each token carrying a tag to identify its role in the computation. Section 3 presents two alternatives to the view of data structures embodied in conventional languages. The first alternative treats a data structure as a value that conceptually is carried on a token. "Functional" structure operations, such as *cons*, are provided to create new structures out of old ones. This approach is elegant, but expensive to implement (even if the data structure is actually left behind in storage so that the token carries only a pointer) and restricts parallelism. The second alternative treats a data structure as a collection of slots, each of which can be written only once. Any attempt to read a slot before it is written is deferred until the corresponding write occurs. Section 4 gives an overview of the major dataflow projects. Finally, Section 5 gives our views of future developments in dataflow computers.

1.1 *Acyclic, Conditional, and Loop Program Graphs*

A dataflow program is described by a directed graph where the nodes denote operations, e.g. addition and multiplication, and the arcs denote data dependencies between operations (Dennis 1974). As an example, Figure 1 shows the acyclic dataflow program graph for the following expression.

$$\begin{aligned} \text{Let}\quad & x = a * b; \\ & y = 4 * c \\ \text{in}\quad & (x + y) * (x - y)/c. \end{aligned}$$

Any arithmetic or logical expression can be translated into an acyclic dataflow graph in a straightforward manner. Data values are carried on *tokens*, which flow along the arcs. A *node* may *execute* (or fire) when a token is available on

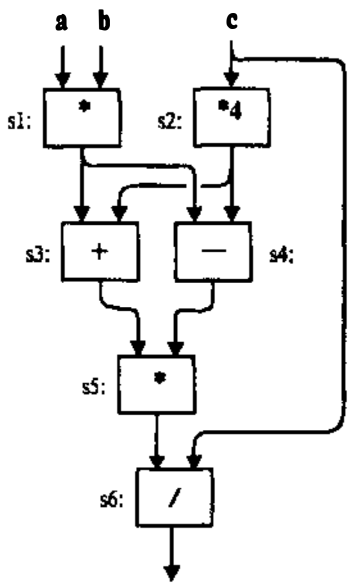


Figure 1 Acyclic dataflow graph.

each input arc. When it fires, a data token is removed from each input arc, a result is computed using these data values, and a token containing the result is produced on each output arc.

Nodes s1 and s2 in Figure 1 are both enabled for execution as soon as tokens are placed on the input arcs a, b, and c. They may fire simultaneously, or one may fire before the other; the results are the same in either case. The result of an operation is purely a function of the input values; there are no implicit interactions between nodes via side effects, for example, through shared memory. This example illustrates two key properties of the dataflow approach: (*a*) *parallelism,* i.e. nodes may potentially execute in parallel unless there is an explicit data dependence between them, and (*b*) *determinacy,* i.e. results do not depend on the relative order in which potentially parallel nodes execute.[1] Furthermore, notice that by supplying several sets of input tokens, distinct computations can be pipelined through the graph. In this example, a single wave of tokens on the input arcs produces a single wave of tokens on the output arcs. Graphs that have this property are called *well-*

[1]The unbounded FIFO queue model presented in this paper is a generalization of the dataflow model originally formulated by Dennis. His model (Dennis 1974) requires that the output arcs of a node be empty before it fires, implying that at most one token can reside on any arc. Kahn's paper (Kahn 1974) implies that the determinacy of dataflow graphs is preserved even without this restriction. Kahn's result also permits nodes to have internal state, but we do not consider this generalization.

behaved. All acyclic graphs for arithmetic and logical expressions are well-behaved.

In order to build *conditional* and *loop* program graphs, we introduce two control operators: *switch* and *merge*. Unlike the *plus* operator, *switch* and *merge* are not well-behaved in isolation, but yield well-behaved graphs when used in conditional and loop schemas (Dennis et al 1972). Consider first the conditional graph in Figure 2*a* that represents the expression if $x < y$ then $x + y$ else $x - y$. The initial tokens provide the data input to the *switches* as well as input to the predicate graph. The predicate graph yields a single boolean value that supplies the control input to all the *switches* and *merges*. A *switch* routes its data input to the output arc on the True side or False side, according to the value of the control input. Thus, the wave of input tokens is directed to the True or the False arm of the conditional. As long as the arms of the conditional are well-behaved graphs, a single wave of tokens will eventually arrive at the data input of the appropriate side of the *merge*. The *merge* selects an input token from the True or the False side input arc, according to the value of the control input, and reproduces the data input token on the output arc. To see that the conditional behaves appropriately when waves of inputs are presented to it, consider the tricky case in which the first wave of input tokens is switched to the True side, the second wave to the False side, and the tokens on the False side of the *merge* arrive before the tokens on the True side. The sequence of control tokens at the *merge* restores the proper order among the tokens on the output arcs.

The loop graph shown in Figure 2*b* computes $\sum_{i=1}^{N} F(i)$. The figure is somewhat stylized in that the dots are used to indicate that the output of the predicate is connected to each of the *switches* and *merges*, and the graph corresponding to function F is indicated by the "blob" containing F. The initial values of i and *sum* enter the loop from the False sides of the *merges* and provide data to the predicate and *switches*. If the predicate evaluates to True, the data values are routed to the loop body. Assuming the body is a well-behaved graph, eventually a single wave of results is produced that provides tokens on the True side of the *merges*. In this way, values circulate through the loop until the predicate turns to False, which causes the final values to be routed out of the loop and restores the initial False values on the control inputs to the *merges*. Note that if many waves of inputs are provided, only one wave at a time is allowed to enter the loop; the second wave enters the loop as soon as the first completes, and so on. Also note that loop values need not circulate in clearly defined waves. Suppose F is a very complicated graph or simply does not fire for a long time. The index variable i may continue to circulate, causing many computations of F to be initiated. This behavior is informally referred to as *dynamic unfolding* of a loop.

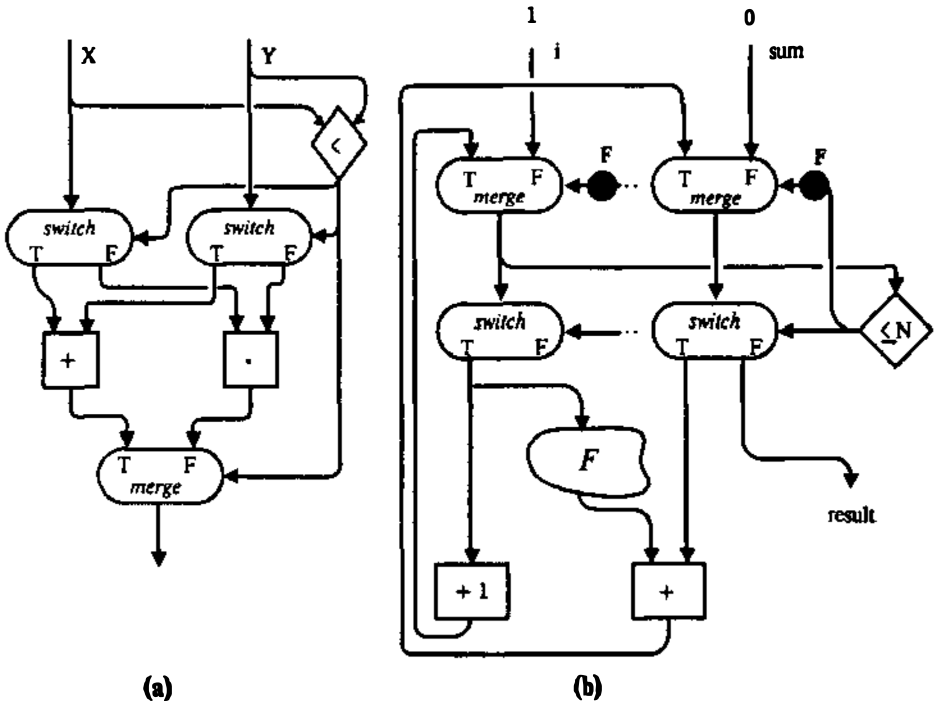


Figure 2 Conditional and loop graphs.

1.2 *Data Structures*

The dataflow model introduced thus far is fully general in a formal computational sense (Jaffe 1979) but has limited practical utility because of the absence of data structures. Suppose we introduce a data structure constructor *cons,* which "glues together" two data values to produce a new value. The components of a pair thus constructed are selected using functions *first* and *rest*. Since these new operators are functions, they fit easily in the dataflow model, provided we assume tokens can carry composite data values. Note that a component of the pair might be a pair, and so on; thus we must allow arbitrarily large structures to be carried on a token. Only in the abstract model do we think of structures as being carried on tokens; in practice, tokens carry pointers to structures that are left behind in storage. The *cons* operation can be extended to a general array operation *append,* which takes an array x, an index i, and an element v, and produces a new array y such that $y[j]$, i.e. the j^{th} element of y, is the same as $x[j]$ for all j not equal to i, and such that $y[i]$ is v.

Even though data structures sit aside in storage, we must be careful not to treat them as we do arrays or records in a conventional language such as Pascal or Fortran. Consider the effect of a conventional store operation that modifies an element of a data structure. In general, there may be many tokens carrying pointers to the structure. Suppose one is destined for a modify operation and another is destined for a *select* operation with the same index. The two operations can potentially execute in parallel because there is no explicit data dependency from one to the other. However, the value produced by the select operation depends upon which operation happens to execute first. This defeats the determinancy of the model: It is no longer true that instructions can execute in any order consistent with the data dependencies and that the results remain unaffected by the order. *Append,* however, does not change the data structure; it produces a new structure that is similar to the old one. Consider the earlier scenario in which a token is destined for a *select* and another carrying a pointer to the same structure is destined for an *append*. The *select* operates on the old structure and hence is not affected by the *append*.

These observations raise a tough question. Is it possible to support data structures efficiently and still maintain the elegance and simplicity of the dataflow model? We return to this question in Section 3.

1.3 *User-Defined Functions*

Another highly desirable property of a computation model is the ability to support user-defined functions. Each of our examples represents a function that, given a set of input values, produces a set of results. Any good high-level language provides a way of *abstracting* variables so that an expression can be turned into a procedure or a function. At the dataflow graph level, a user-defined function is no more than an encapsulation of a graph that allows arguments and results to be transmitted properly. Nonrecursive functions can be handled by graph expansion at compile time. However, to support user-defined functions more generally, we need an *apply* operator that takes as inputs a function value (i.e. the description of an encapsulated dataflow graph) and a set of arguments and that invokes the function on the specified arguments. There are subtle issues involved in the implementation of *apply*. For example, when should the graph corresponding to the function actually be created? After all the arguments have arrived? As soon as a particular argument has arrived? Often the semantics of function application in high-level languages requires the *apply* to be implemented in a particular way. However, all implementations must support dynamic expansion of graphs and a method to route tokens to input arcs of the newly created graph. If a copy of the function graph is to be reused, then a mechanism is required to distinguish tokens belonging to different invocations. In this latter case, the

FIFO queueing of tokens on arcs will not suffice. A mechanism for user-defined functions develops naturally out of the tagged-token approach, so we will return to this topic after discussing various implementations.

1.4 *Dataflow Graphs as a Parallel Machine Language*

We can view dataflow graphs as a machine language for a parallel machine where a node in a dataflow graph represents a machine instruction. The instruction format for a dataflow machine is essentially an adjacency list representation of the program graph: Each instruction contains an op-code and a list of destination instruction addresses. Recall that an instruction or node may execute whenever a token is available on each of its input arcs and that when it fires, the input tokens are consumed, a result value is computed, and a result token is produced on each output arc. This dictates the following basic instruction cycle: (*a*) detect when an operation is enabled (this is tantamount to collecting operand values); (*b*) determine the operation to be performed, i.e. fetch the instruction; (*c*) compute results; and (*d*) generate result tokens. This is *the* basic instruction cycle of any dataflow machine; however, there remains tremendous flexibility in the details of how this cycle is performed.

It is interesting to contrast dataflow instructions with those of conventional machines. In a von Neumann machine, instructions specify the addresses of the operands explicitly and the next instruction implicitly via the program counter (except for branch instructions). In a dataflow machine, operands (tokens) carry the address of the instruction for which they are destined, and instructions contain the addresses of the destination instructions. Since the execution of an instruction is dependent upon the arrival of operands, instruction scheduling and management of token storage are intimately related in any dataflow computer.

Dataflow graphs exhibit two kinds of parallelism in instruction execution. The first we might call *spatial* parallelism: Any two nodes can potentially execute concurrently if there is no data dependence between them. The second form of parallelism results from pipelining independent waves of computation through the graph. In the next section we show that it is possible to execute several instances of the same node concurrently, thereby exploiting this *temporal* parallelism.

2. TOKEN STORAGE MECHANISMS

The essential point to keep in mind in considering ways to implement the dataflow model is that tokens imply storage. The token storage mechanism is the key feature of a dataflow architecture. While the dataflow model assumes unbounded FIFO queues on the arcs and FIFO behavior at the nodes, it turns out to be very difficult to implement this model exactly. Two alternative approaches have been researched extensively. The first we call *static dataflow;* it provides a fixed amount of storage per arc. The other approach we call *dynamic* or *tagged-token dataflow;* it provides dynamic allocation of token storage out of a common pool and assumes that tokens carry tags to indicate their logical position on the arcs.

2.1 *Static Dataflow Machine*

The one-token-per-arc restriction can be incorporated in the model by extending the firing rule to require that all output arcs of a node be empty before that node is enabled. With this restriction, storage for tokens can be allocated prior to execution, since the number of arcs is fixed for a given graph. The basic instruction format is expanded to include a slot for each operand. Distributing tokens to destination instructions involves little more than storing data values in the appropriate slots. The slots have *presence flags* to indicate whether or not a value has been stored. Thus, when a token is stored, it can easily be determined if the other inputs are all present. This idea underlies the static dataflow machines proposed by Dennis and his co-workers (Dennis & Misunas 1974; Dennis 1980; Dennis et al 1984a), described in Figure 3.

Instruction templates reside in the *activity store*, and addresses of enabled instructions reside in the *instruction queue*. The *fetch unit* removes the first entry in the instruction queue, fetches the corresponding op-code, data, and destination list from the activity store, forms them into an *operation packet,* forwards the operation packet to an available *operation unit,* and finally clears the operand slots in the template. The operation unit computes a result, generates a *result packet* for each destination, and sends the result packets to the *update unit*. Instructions are identified by their address in the activity store, so the update unit stores each result and checks the presence bits to determine if the corresponding activity is enabled. If that is the case, the address of the instruction is placed in the instruction queue. These units operate concurrently, so instructions are processed in a pipelined fashion.

It is possible to connect many such processors together via a packet communication network. The activity store of each processor can be loaded with a part of a dataflow graph. Notice that large delays in the communication network do not affect the performance, i.e. the number of operations performed per second, as long as enough enabled nodes are present in each processor. This is an important characteristic of dataflow machines; they can use parallelism in programs to hide communication latency between processors.

2.1.1 ENFORCING THE ONE-TOKEN-PER-ARC RESTRICTION

The above description of the static machine skips over a very important and rather subtle point: the one-token-per-arc restriction of Dennis's model. Suppose the units

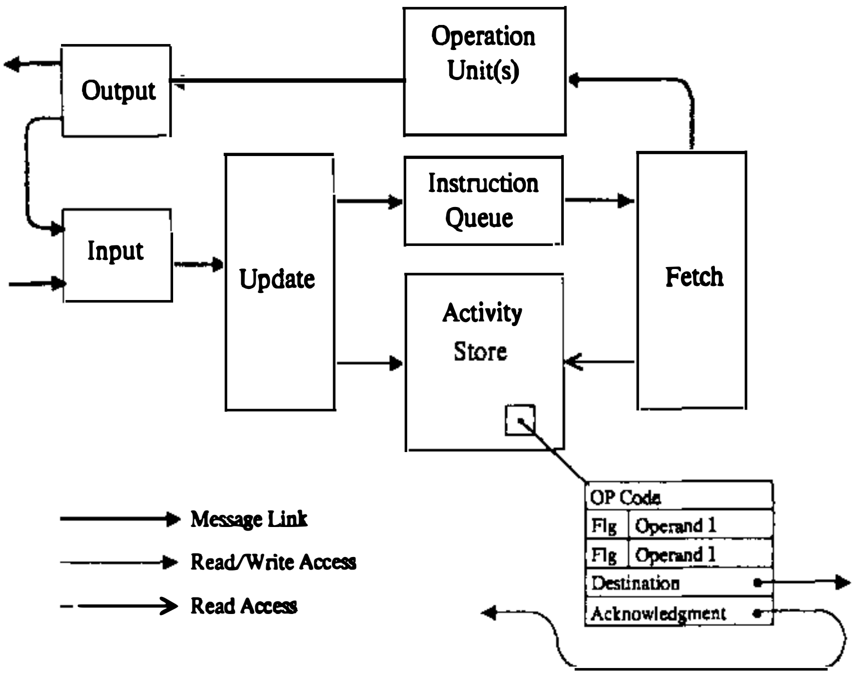


Figure 3 Static dataflow architecture.

communicate with a full send-acknowledge protocol, i.e. a token moves to the next unit only after that unit has signalled that it can accept the token, and the update unit writes into an operand slot only if the slot is empty. Even with these assumptions, multiple tokens belonging to the same arc may coexist in the machine, since there may be buffering in the units and communication network. It is infeasible for the update or fetch units to determine that there is no token in the system for a particular arc. If multiple tokens can coexist on an arc, then the FIFO assumption may be violated because two firings of a node may execute on different operation units within a processing element (PE), and the one that is logically second in the queue may finish first. The communication system will ultimately direct these result tokens to the same destination node, but in the wrong order. To see how the dataflow model malfunctions if tokens on an arc get out of order, consider the example in Figure 2*b* with the *plus* operator replaced by *minus*. The results of F(1) and F(2) can potentially reside on the left input to the *minus* concurrently, but if F(2) is processed before F(1) the answer will be wrong.[2]

If the one-token-per-arc restriction can be enforced, then the problems due

[2]Misunas shows (Misunas 1975) that multiple tokens per arc can also cause the machine to deadlock.

to reordering of tokens will not arise. The restriction cannot be enforced at the hardware level, but its effect can be achieved by executing only graphs that have the property whereby no more than one token can reside on any arc at any stage of execution. It is possible to transform any dataflow graph into a dataflow graph with this property. In the simplest transformation, for each arc in the graph, an *acknowledgment arc* is added in the opposite direction. A token on an acknowledgment arc indicates that the corresponding data arc is empty. Initially, a token is placed on each acknowledgment arc. A node is enabled to fire when a token is present on each input arc and each incoming acknowledgment arc. At the hardware level, the only difference between the two kinds of arcs is that the value of a token on an acknowledgment arc is ignored. Instead of the presence bits for operands, a counter is associated with each instruction. The counter is initialized to the number of operands plus the number of incoming acknowledgment arcs and is decremented by the update unit whenever an operand or acknowledgment arrives. The node is enabled when the counter reaches zero. Notice that the generation of acknowledgments must be delayed enough after the operation packet is formed so that there is no way for results of the second firing to overtake the first.

The one-token-per-arc restriction is not entirely satisfactory. Even though many of the acknowledgment arcs in a program graph can be eliminated (Montz 1980), the amount of token traffic increases by a factor of 1.5 to 2, the time between successive firings of a node increases drastically, and most importantly, the amount of parallelism that can be exploited in a program is reduced. In particular, the dynamic unfolding of loops is severely constrained, as shown by the following example. Suppose F in Figure 2*b* is replaced by the acyclic graph in Figure 1 (perhaps we take the inputs a, b, and c to be i). It should be possible to pipeline four distinct computations through this graph, but, unfortunately, with the static approach the second initiation must wait until the *divide* node fires, clearing the input arc for c. This problem has received substantial attention (Dennis & Gao 1983) and can be partially overcome by introducing extra identity operators to balance the path lengths in a graph. For example, if three identity nodes are added on the right input to the *divide* in Figure 1, the path lengths would be perfectly balanced. The balancing approach assumes that execution times for all operators are the same and that communication delays between operators are constant. Neither assumption is realistic, and balancing becomes computationally intractable without these assumptions.

We note in passing that modeling unbounded-FIFO dataflow graphs by fixed storage dataflow graphs (introduction of acknowledgment arcs is one example of such modeling) changes the "meaning" of a dataflow graph in a subtle way. A graph may be deadlock free in the unbounded case, but its corresponding graph with acknowledgment arcs may deadlock under certain circumstances. These shortcomings, in addition to the inability to handle user-defined functions, motivated work on the more general dynamic dataflow approach discussed below.

2.2 *Dynamic or Tagged-Token Dataflow*

Each token in a static dataflow machine must carry the address of the instruction for which it is destined. This is already a *tag*. Suppose, in addition to specifying the destination node, the tag also specifies a particular firing of the node. Then, two tokens participate in the same firing of a node if and only if their tags are the same. Another way of looking at tags is simply as a means of maintaining the logical FIFO order of each arc, regardless of the physical arrival order of tokens. The token that is supposed to be the i^{th} value to flow along a given arc carries i in its tag. The goal is to give simple tag generation rules for the control operators *switch* and *merge*. Arvind & Gostelow (1977a) have given such rules for Dennis's operators (Dennis 1974). However, if only well-behaved graphs are considered, then it is possible to develop even simpler tag manipulation rules (Arvind & Gostelow 1982). We briefly explain these latter rules as well as the effect of tagging on the dataflow model presented in Section 1.

2.2.1 TAGGING RULES We associate graphs, called *code blocks,* with each user-defined function. Each graph is either acyclic or a single loop. Thus a function containing nested loops is treated as several code blocks—one corresponding to each loop. A node is identified by a pair: code-block, instruction address. Tags have four parts: invocation ID, iteration ID, code block, and instruction address. The latter two identify the destination instruction and the former two identify a particular firing of that instruction. The iteration ID distinguishes between different iterations of a particular invocation of a loop code-block, while the invocation ID distinguishes between different invocations. All the tokens for one firing of an instruction must have identical tags, and enabled instructions are detected by finding sets of tokens with identical tags. Tokens also carry a port number that specifies the input arc of the destination node on which the token resides; this is not part of the tag and thus does not participate in matching.

Consider first the execution of an acyclic graph such as in Figure 1. A set of tokens whose tags differ only in the instruction address part is placed on the input arcs. When an instruction fires, it generates tags for each result token by using the destination address in the instruction as the instruction address part and copying the rest from the input tag. For conditionals the scenario is similar, but there are two destination lists. A single wave of inputs is steered through one arm or the other. We will ensure, however, that no two waves of inputs carry the same invocation and iteration IDs in their tags. Thus, for any

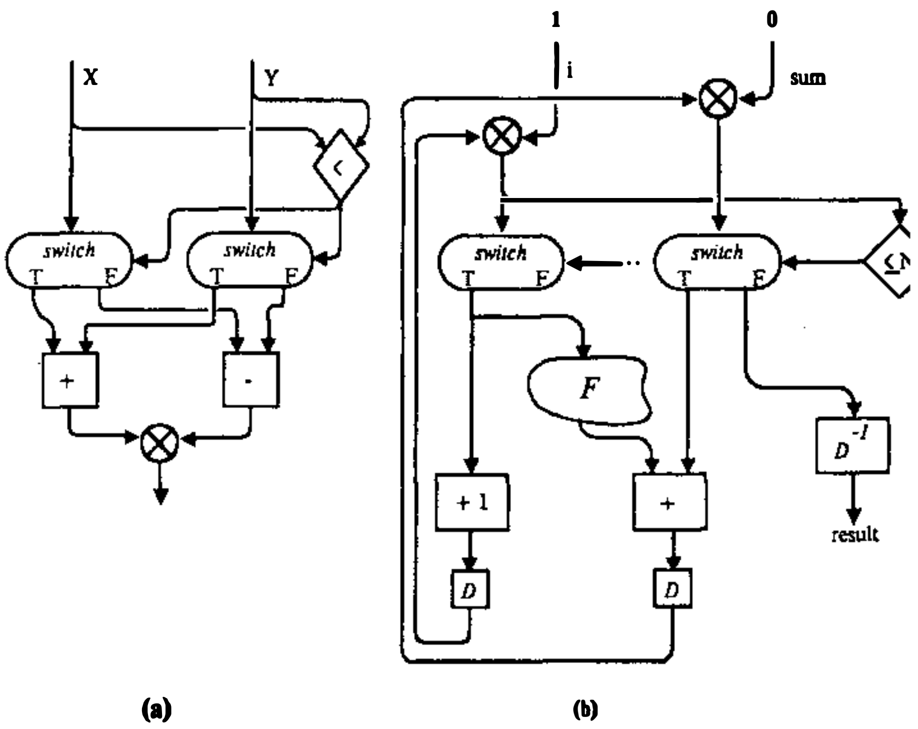


Figure 4 Conditional and loop graphs for tagged approach.

given tag, a data item carrying that tag will arrive at most on one side of the *merge*. Since the order of tokens on the arcs is immaterial, there is no need to orchestrate the *merge* via the output of the predicate as in the FIFO model; the streams of tokens produced by the two arms can be merged in an arbitrary fashion. This modified conditional schema is shown in Figure 4*a*. The ⊗ is not an operator; it merely denotes that two arcs converge on the same port.

The loop requires a control operator, named D, to increment the iteration ID portion of the tag (see Figure 4*b*). The iteration ID of each initial input to the loop is zero. Like the conditional schema, the *merges* can be eliminated from the loop schema because the tags on the tokens on the True and False sides of a *merge* will be disjoint. The D^{-1} operator is used to reset the iteration ID to zero. To implement nested loops and user-defined functions, an additional operator is required to assign unique invocation IDs. The *apply* operator takes a code-block name and an argument as input and forwards the argument to the designated code-block after assigning it a new invocation ID and setting its iteration ID to zero. The tag for the output arc of the *apply* node is also sent to the invoked graph so that the result can be returned to the destination of the *apply* node, as if it were generated by the *apply* node itself.

One may visualize the action of an *apply* as coloring input tokens in such a manner that they do not mix with tokens belonging to other invocations of the same code block. Of course, there must be a complementary operator to restore the original color for the result tokens. The interested reader is referred to Arvind et al (1978) for more detail.

The tagged-token approach eliminates the need to maintain FIFO queues on the arcs (though unbounded storage is still assumed) and consequently offers more parallelism than the abstract model presented in Section 1. In fact, it has been shown that no interpreter can offer more parallelism than the tagged-token approach (Arvind & Gostelow 1977b).

2.2.2 TAGGED-TOKEN DATAFLOW MACHINE A machine proposed by Arvind et al (Arvind et al 1983a) is depicted in Figure 5. It comprises a collection of PEs connected via a packet communications network. Each PE is a complete dataflow computer. The *waiting-matching* store is a key component of this architecture. When a token enters the waiting-matching stage, its tag is compared against the tags of the tokens resident in the store. If a match is found, the matched token is purged from the store and is forwarded to the *instruction fetch* stage, along with the entering token. Otherwise, the incoming token is added to the matching store to await its partner. (Instructions are restricted at most to two operands, so a single match enables an activity.) Tokens that require no partner, i.e. are destined for a monadic operator, bypass the waiting-matching stage.

Once an activity is enabled, it is processed in a pipelined fashion without further delay. The invocation ID in the tag designates a triple of registers (CBR, DBR, and MAP) that contain all the information associated with the invocation. CBR contains the base address of the code block in program memory; DBR contains the base address of a data area that holds values of loop variables that behave as constants, and MAP contains mapping information describing how activities of the invocation are to be distributed over a collection of PEs. The *instruction fetch* stage is thus able to locate the instruction and any required constants. The op-code and data values are passed to the *arithmetic logic unit* (ALU) for processing. In parallel with the ALU, the *compute tag* stage accesses the destination list of the instruction and prepares result tags using the mapping information. Result values and tags are merged into tokens and passed to the network, whereupon they are routed to the appropriate waiting-matching store.

It is important to realize that if the waiting-matching store ever gets full the machine will immediately deadlock; tokens can leave the waiting-matching section only by matching up with incoming tokens. A similar argument can be made to show that if the total storage between the output of the waiting-matching section and the paths leading to its input is bounded, a deadlock can

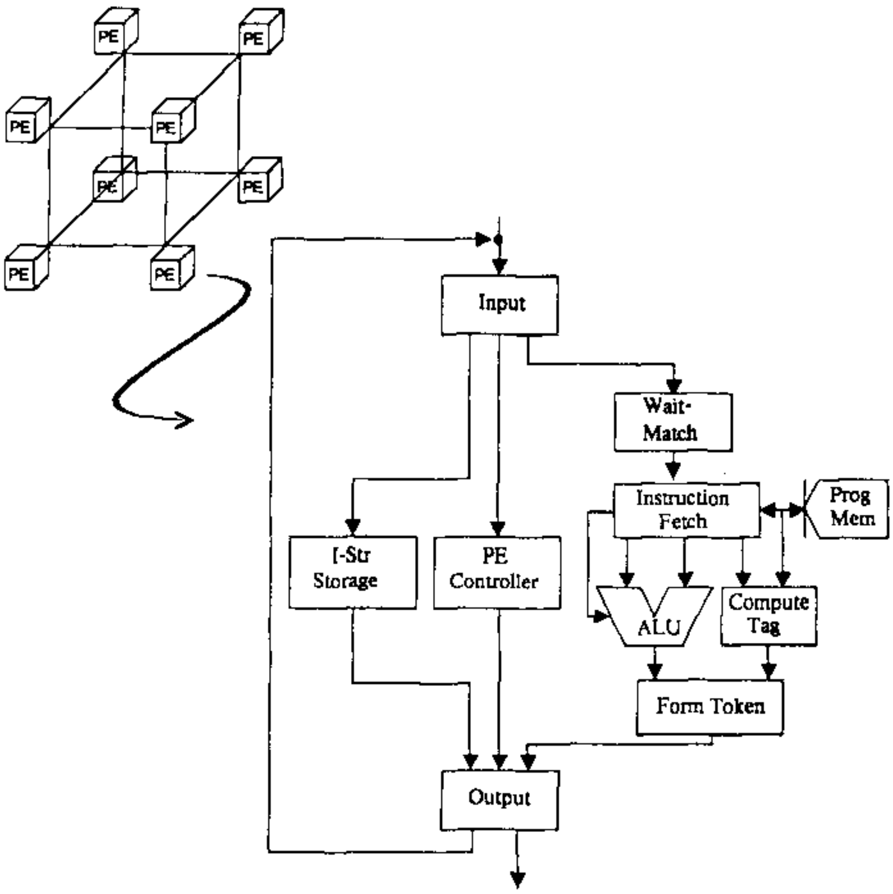


Figure 5 Processing element of the MIT tagged token dataflow machine.

occur (Culler 1985). Therefore, in addition to the functional units described in Figure 5, each PE must have a *token buffer*. This buffer can be placed at a variety of points, including the output stage or the input stage, depending on the relative speeds of the various stages. Both the waiting-matching store and the token buffer have to be large enough to make the probability of overflow acceptably small.

The *apply* operator is implemented as a small graph. The invocation request is passed to a system-wide resource manager so that resources such as a new invocation ID, program memory, etc, can be allocated for the new invocation. A code-block invocation can be placed on essentially any collection of processors. Various instances, i.e. firings, of instructions are assigned to PEs within a collection by "hashing" the tags. A variety of mapping schemes have been developed to distribute efficiently the most frequently encountered program structures. The MAP register assigned to a code-block invocation uses the hashing function for mapping activities of the code block.

Efficient handling of "loop constants" is a fairly low-level optimization, but important enough to deserve mention. In the abstract model, variables that are invariant over all iterations for a particular invocation of a loop, but that vary for different invocations, must be circulated. *N* in Figure 2*b* is an example of such a variable. Values of such variables cannot be placed in the instructions without making the graph non-reentrant. To avoid this overhead, most dataflow machines provide a mechanism for efficient handling of loop constants. As an example of the importance of this optimization, note that the inner loop of a straightforward matrix multiply program has seven loop variables, five of which are loop constants. In the MIT tagged-token machine, storage for such constants is allocated in program memory when a loop code-block is invoked; DBR points to this area and allows these constants to be fetched along with the instruction. The constant area is deallocated when the invocation terminates. If the loop invocation is spread over multiple PEs, setting up constant areas is a little tricky, since an image must be made in each PE before the first iteration is allowed to begin.

The tagged-token architecture circumvents the shortcomings identified in the static architecture, but it also presents some difficult issues. In the static machine, the storage has to be allocated for all arcs of a program graph, though tokens may coexist only on a small fraction of them. In contrast, token storage is used more efficiently in the tagged-token approach, because storage requirement is determined by the number of tokens that can coexist. However, programs exhibit much more parallelism under the tagged-token approach (actually even more so than the unbounded-FIFO model) and, consequently, can drive the token storage requirement so high that the machine may deadlock (Culler 1985). This has become a serious enough problem in practice that we now generate only those graphs in which the parallelism is bounded. In the dynamic machine, the mechanism for detecting enabled activities appears more complex, since matching is required as opposed to decrementing a counter. Further, tokens carry more tagging information though no acknowledgment tokens are needed. If tags are to be kept relatively small, there must be facilities for reusing tags. This, in turn, requires detecting the completion of code-block invocations, an action that generally involves a nontrivial amount of computation. This task would be virtually impossible if the graphs were not "self-cleaning," which is a consequence of graphs being well-behaved. Finally, an efficient mechanism is required for allocating resources to new code-block invocations.

2.3 *Tags as Memory Addresses and vice versa*

The performance of a tagged-token machine is crucially dependent upon the rate at which the waiting-matching section can process tokens. Though the size of the waiting-matching store depends upon many factors, based on our preliminary studies we expect that it will be in the range of 10K to 100K tokens. In this size range, a completely associative memory is ruled out, but a hash table possibly augmented with a small associative memory is viable, and the waiting-matching sections of the machines discussed in Section 4 are organized as such. Hashing basically involves calculating the address of a slot in the hash table by applying some "hash" function to the tag of the token. Examples of the hashing functions used in a tagged-machine are given in the references (Hiraki et al 1984).

Gino Maa, a member of our group, has suggested that tags should be viewed as addresses for a virtual memory in which the primitive operation is *store-extract*. Given a data and an address, the *store-extract* operation stores the data in the slot specified by the address if the slot is empty; otherwise, the contents of the slot are read and the slot is considered empty. A page of virtual memory may contain, for example, tokens with identical contexts. It is clear that only a tiny fraction of the virtual address space will be occupied at any given time, and physical storage is required only for this fraction. Thus, the problem of the design of the waiting-matching section becomes the problem of implementing a very large virtual memory (40-bit addresses or larger), where a nonexistent page is allocated automatically upon an attempt to access it and deallocated when all its entries are empty. Caches may be effective in organizing such a memory; there is evidence to suggest that when an incoming token finds its partner, the partner is usually among the most recently arrived tokens (Brobst 1986). The difference between the implementation of a large virtual address space and the hashing approach discussed earlier may be minimal; however, viewing tags as addresses allows us to place many variations of static and dynamic machines on a continuum, in which the address on a token in the static machine becomes the tag on a token in the dynamic machine.

Consider extending the static machine by operators to allocate activity store dynamically, thus allowing procedure calls to be implemented. In all such implementations, a part of the address serves the purpose of the "context" part of the tag in the dynamic machine, and the task of allocating a new context is subsumed by the task of allocating activity storage. A common optimization in such schemes is to separate the operand slots of an instruction from the rest and to allocate a new template containing operand slots for a code block at the time of invocation. To achieve sharing of a code block among several invocations requires relocation registers like CBR, DBR, etc, of the MIT tagged-token machine. Another variation discussed in the literature eliminates the need for acknowledgment arcs by allowing only acyclic graphs (Dennis et al 1984b; Preiss & Hamacher 1985). Since a loop can be modeled as a recursive procedure, this offers a trade-off between the cost of extra procedure calls and the savings gained from the elimination of acknowledgments. As discussed earlier, there are subtle issues associated with the implementation of the *apply* operator, e.g. the time of storage allocation affects the amount of parallelism that can be exploited by the machine.

A variation of the tagged-token machine that has been proposed by David Culler and Gregory Papadopoulos (also of our group) is to replace the waiting-matching section of the tagged-token machine by a token storage that is explicitly allocated at the time of procedure invocation. It is possible to do so if the storage requirement of a code block can be determined prior to invoking it. The type of bounded-loop graphs that we propose to run on the machine have this property.

After examining some of the variations discussed here, the distinction between static and dynamic dataflow becomes somewhat fuzzy. Choosing a good design among the ones proposed (or one yet to be proposed) is an active research topic in this field. The only general statement we can make is that giving the programmer or the compiler a greater control over the management of resources increases his responsibility and burden but may provide significant performance improvements and may simplify the design of the machine.

3. DATA STRUCTURES

Section 1 described how data structures can be incorporated in the dataflow model without sacrificing its elegance or utility for parallel computation. We now illustrate the difficulties in implementing "functional" data structures efficiently and describe an alternative view known as I-structures. This latter approach offers an efficient implementation without sacrificing determinacy and allows more parallelism to be exploited in programs than the "functional" approach.

3.1 *Functional Operations On Data Structures*

The simplest form of "functional" data structures is reflected in the operations *cons, first,* and *rest. Cons* glues two values together to form a pair; *first* and *rest* select values from such pairs. Clearly, we cannot allow arbitrarily large values to be carried on a token, so pairs must be maintained in storage with tokens carrying the addresses of these pairs. To this end, dataflow machines provide *structure storage,* which should be considered as a special operation unit with internal storage. The unit is shared by all PEs and is capable of performing many concurrent structure operations.

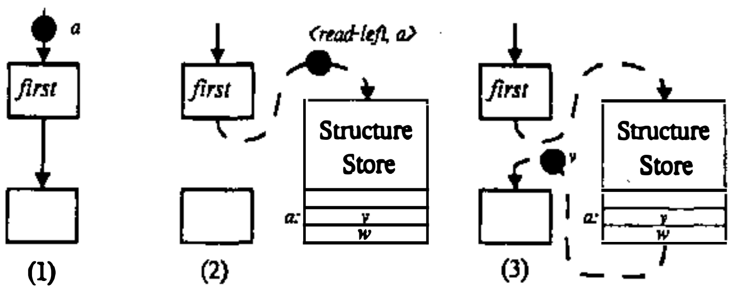


Figure 6 Action of a *first* operation.

To see how the structure store and its associated operations behave, we can step through the execution of a *first* operation. A *first* operation is enabled by the arrival of a token carrying a pointer. Neither the *fetch* unit in the static machine nor the ALU in the tagged-token machine can access the structure storage directly.[3] Thus, a new packet containing the *read* request and the address or tag of the destination node of the *first* operation is sent to the structure storage. Upon receipt of such a request, the structure storage controller produces a token containing the left value of the pair and sends it to the appropriate destination instruction; this is depicted in Figure 6.

Similarly, for the *cons* operator, two input data values together with the destination node address (or tag) are sent to a structure storage unit. The structure controller allocates storage for the pair, writes the elements, and sends a pointer for the newly allocated storage to the destination instruction.

The implementation of large, flat data structures, such as arrays, presents difficult design trade-offs. If arrays are implemented as linked lists using *cons,* selection operations are inefficient. If, instead, array elements are stored contiguously, as a generalization of the pairing operation, the *append* operation becomes costly. This is because *append* involves creating a new array and copying all except one element from the old array. Efficient implementations of arrays have been researched extensively (Ackerman 1978; Guharoy 1985), and two key ideas have emerged to reduce copying. First, if the array descriptor (or pointer) fed to the *append* operator is the only descriptor in existence for the corresponding array, the array can be updated in place without risk of causing a read-write race. Second, if the array is

[3]Not providing direct access to a large storage shared by many PEs is certainly a design choice, but a fundamental one. In a machine with many processors and many structure controllers, the time to access a particular memory controller may be very large. If the instruction processing pipeline blocks for structure operations, the performance of the machine will be greatly affected by the latency of the communication system. One advantage of dataflow machines is that they can be made extremely tolerant of latency and thus can sustain high performance with many processors working on a single problem. Detailed arguments concerning these points can be found in Arvind & Iannucci (1983a).

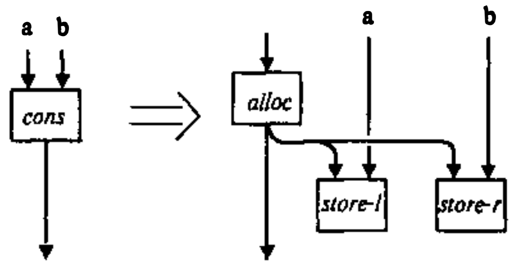


Figure 7 Implementation of nonstrict cons.

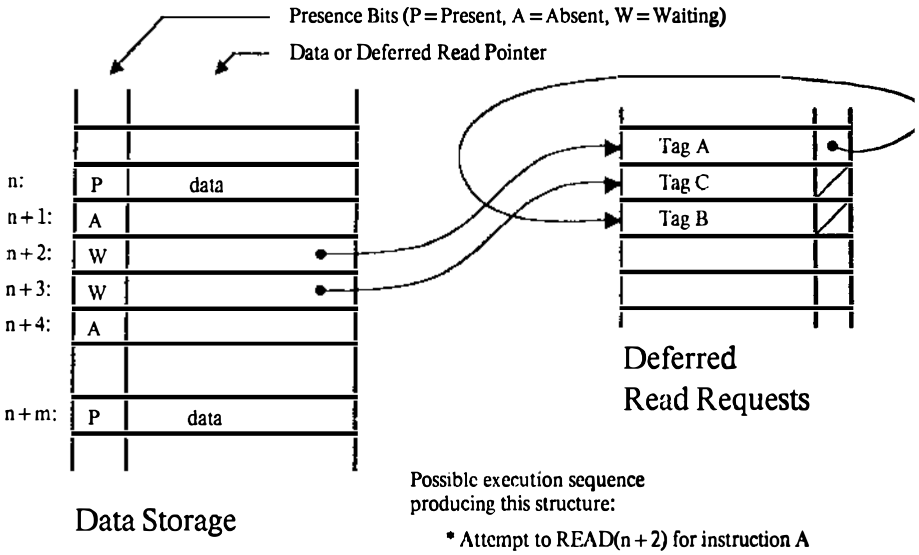
represented as a tree, then only the nodes along the path to the appended element need be regenerated; the rest of the tree can be shared. This reduces the amount of allocation and copying but increases the time for selection.

3.2 *I-structures*

The "functional" view of structures imposes unnecessary restrictions on program execution, regardless of how efficiently it is implemented. Consider the simple example *cons*(f(a),g(a)); the *cons* will not be enabled until both f(a) and g(a) have completed. Thus, another part of the program that uses the first element of the pair, but not the second, must wait until both elements have been computed. In programming language jargon, such data structures are called *strict*. In contrast, *cons* can be treated as a *nonstrict* operator (Friedman & Wise 1976), which allows an element of a pair to be used regardless of whether the other element has been produced. The resultant increase in parallelism is far greater than one might naively imagine.

The firing rule for nonstrict *cons* is difficult to implement. One way to circumvent this difficulty is to treat *cons* as a triplet of operations, as shown in Figure 7. The implicit storage allocation of strict *cons* becomes visible as a new type of node in the dataflow graph. The descriptor produced by the *allocate* operator is passed to the two store operations, in addition to the subsequent select operations. This allows consumption of a structure to proceed in parallel with production but also raises an awkward problem: A *first* or *rest* operation may be executed before the corresponding *store*. This seemingly catastrophic situation can be resolved with the help of a smart structure-storage controller. If a *read* request arrives for a storage cell that has not been written, the controller defers the *read* until a write arrives. This is the basic idea behind I-structure storage.

Referring to Figure 8, each storage cell contains status bits to indicate that the cell is in one of three possible states. (*a*) PRESENT: The word contains valid data that can be freely read as in a conventional memory. Any attempt to write it will be signalled as an error. (*b*) ABSENT: Nothing has been written into the cell since it was last allocated. No attempt has been made to read the



Possible execution sequence producing this structure:

* Attempt to READ(n + 2) for instruction A
* WRITE(n + m)
* Attempt to READ(n + 3) for instruction C
* WRITE(n)
* Attempt to READ(n + 2) for instruction B
* READ(n)

Figure 8 I-structure storage.

cell; it may be written as for conventional memory. (*c*) WAITING: Nothing has been written into the cell, but at least one attempt has been made to read it. When it is written, all deferred *reads* must be satisfied. Cells change state in the obvious ways when presented with requests. Destination tags of deferred *read* requests are stored in a part of the I-structure storage specially reserved for that purpose.

While I-structure storage can be used to implement nonstrict *cons,* to exploit the full potential of this form of storage, functional languages can be augmented with explicit allocate and store operations. From a programmer's perspective, an I-structure is an array of slots (Nikhil & Arvind 1985) that are initially empty and that can be written at most once. Regardless of when or how many times a *select* instruction for a particular slot is executed, the value returned is always the same. This preserves the determinacy property of the model. I-structures are not "functional" data structures; they are "monotonic objects" that are constructed incrementally, hence their name.

I-structures provide the kind of synchronization needed for exploiting producer-consumer parallelism without risk of read-write races. I-structure *read* requests for which the data is present require about the same time as conventional *reads,* and with special hardware (Heller 1983) deferred *reads* can be processed quickly. Thus, as long as most *read* requests follow the corresponding *write,* the overhead of I-structure memory is small, and the utility is enormous.

The benefit of nonstrict structures in terms of the amount of parallelism exhibited by programs is surprisingly large. For example, methods in which a large mesh is repeatedly transformed into a new version by performing some calculation for each point are common in numerical computing. Some such methods show tremendous parallelism because all mesh points can be computed simultaneously. However, even when this is not possible because of data dependencies, it is usually possible to overlap the computation of several versions of the mesh. This latter form of parallelism can be exploited only if the mesh is represented as a nonstrict structure.

4. CURRENT DATAFLOW PROJECTS

We now present an overview of some of the more important dataflow projects; we restrict our attention to those that have built or are currently building a dataflow machine. In particular, we do not address how dataflow concepts have influenced high-performance von Neumann computers being designed today.

4.1 *Static Machine Projects*

It is no exaggeration to say that *all* dataflow projects started in the seventies were directly based on seminal work by Dennis (1974). Such projects, besides Dennis's own project, include the LAU project in Toulouse, France (Comte et al 1980), the Texas Instruments dataflow project (Johnson et al 1980), the Hughes dataflow machine (Gaudiot et al 1985), and several projects in Japan (Temma et al 1980; NEC 1985). Even the work on tagged-token machines at the University of Manchester in England and the University of California at Irvine was inspired by Dennis's work.

4.1.1 THE MIT STATIC MACHINE DATAFLOW PROJECT Dennis's group at MIT has proposed and refined several static dataflow architectures over the years (Dennis & Misunas 1974; Rumbaugh 1977; Dennis et al 1980, 1984a), and has implemented an eight-processor engineering model of the static machine shown in Figure 3 (Dennis et al 1980). The processing elements (PE) were built out of AMD bit-slice microprocessors and were connected by a packet-switched butterfly network composed of 2 × 2, byte-serial routers with send-acknowledge protocol. The structure controller was not implemented. Dataflow graphs for the machine were compiled from the language VAL (Ackerman & Dennis 1978). A PDP-11 served as a front end. While the machine operated successfully, it was only large enough to run toy programs. Also, because of microcoding, the PEs were far slower than the routers. The Texas Instruments machine (Johnson et al 1980), which was architecturally similar to Dennis's machine, was built by modifying four conventional processors. Even though these machines proved to be too slow to generate commercial interest in dataflow machines, they have had a marked influence on instruction scheduling in high-performance machines intended for scientific computing.

4.1.2 THE NEC DATAFLOW MACHINES The latest machines that may be classified as static machines are NEC's NEDIPS (Temma et al 1980) and image pipelined processor (IPP) μPD7281 (NEC 1985). NEDIPS is a 32-bit machine that is intended for scientific computation and uses high-speed logic, while the IPP is a single chip processor of similar architecture that is intended as a building block for highly parallel image processing systems. We focus on the latter machine. Generally, image processing involves applying a succession of filters to a stream of image data. Thus, each IPP chip may be loaded with a dataflow program for a specific filter or several filters.

The NEC designers have generalized the machine described in Section 2.1 by allowing multiple tokens per arc. To see how this is done, consider once again the static machine in Figure 3. Instruction templates must be enlarged to include a collection of operand slots. If we assume that the operands of an enabled instruction are immediately removed from the activity store and forwarded to the operation units, then tokens cannot accrue in the slots for both the left and right arcs simultaneously. Thus, both arcs can share the same slots as long as a flag is provided in the instruction template to indicate on which arc (left or right) the current tokens reside. Further, the collection of slots in an instruction are managed as a cyclic buffer, with two pointers marking the head and tail of the queue. When an incoming token is for the same arc as the arc to which the previously arrived tokens in the instruction belong, the update unit adds the data value of the incoming token to the tail of the queue. Otherwise, the data value at the head is removed and placed in the instruction queue, along with incoming token. Notice that it is not necessary for all instruction templates to contain the same number of operand slots.

In the IPP implementation, the three components of the instruction template—op-code, operand slots, and destination list—are placed in three separate memories so that they can be accessed at consecutive stages of the instruction pipeline. Each IPP provides storage for 64 instructions, 128 arcs, and 512 16-bit data elements, which can be partitioned into queues of up to 16 slots per instruction. The IPP also allows regions of the data memory to be used for constants and tables. In addition, special hardware operations are provided for generating, coalescing, splitting, and merging *streams* of tokens. A novel technique is employed to govern the level of activity in the instruction pipeline: Instructions with multiple destinations are queued separately from those with single destinations, so when the pipeline is starved the multiple-destination instruction queue is given priority, and when the instruction pipeline is full the other queue is favored. Buffered input/output ports, which support a full send-acknowledge protocol, are provided and allow up to 14 IPPs to be connected in a ring. The system relies on a host processor to provide input/output, bookkeeping, and operating system support.

IPP does not handle acknowledgments specially and requires that operand storage be allocated statically, i.e. by the programmer or compiler. The programmer must tune the program graph to avoid buffer overflows and to ensure that tokens do not get out of order. As a result, program development for this machine is a tedious task. The buffer overflow problem is much less severe in NEDIPS because it provides much more data memory (64K words) than IPP. Nevertheless, the problem is serious enough to cause the designers to modify NEDIPS so that operand buffers can be extended or shrunk dynamically in 128-word increments. As discussed in Section 2.3, this extension also makes it difficult to classify NEDIPS as a static machine.

NEDIPS and IPP are the first commercially available dataflow processors, and regardless of their commercial success, which only time will tell, they are major milestones in non-von Neumann architectures.

4.2 *Tagged-Token Machine Projects*

The tagged-token dataflow approach was conceived independently by two research groups, one at Manchester University in Manchester, England, and one at the University of California at Irvine. The tagged-token architecture presented in Section 2.2 is based on work by the latter group, which has since moved to the Massachusetts Institute of Technology. The prototype tagged-token machine completed at the University of Manchester in 1981 (Gurd et al 1985) presents some interesting variations on the machine described above. A number of other prototype efforts are in progress in Japan, most notably in Amamiya's group at Nippon Telephone and Telegraph Company (Amamiya et al 1982; Takahashi & Amamiya 1983), and Sigma-1 at the Electrotechnical Laboratory, which is discussed later in this section.

4.2.1 THE MANCHESTER DATAFLOW PROJECT The Manchester machine is essentially like the instruction processing section shown in Figure 5. It is a single ring consisting of a token queue, a matching unit, an instruction store, and a bank of ALUs. The ALUs are microcoded and fairly slow. It has demonstrated reasonable performance (1.2 MIPS) with this arrangement, although the choice of many slow ALUs has received some criticism because all the ALUs can be easily replaced by a single fast ALU. Tokens are 96 bits wide, including 37 bits for data, 36 for tag, and 22 for destination address.

The matching unit is a two-level store. The first level has a capacity of 1M tokens and uses a parallel hashing scheme to map an incoming tag into a set of eight slots. The contents of the selected slots are associatively matched against the incoming tag. The second-level overflow store uses hashing with linked lists.

The Manchester machine has no structure store per se. Instead, a host of exotic matching operations are provided so that the matching store can function as a structure store as well (Watson & Gurd 1982). The analog of an invocation ID can be treated as an array descriptor, and the iteration ID can function as the index, so a tag can represent an array element. A store operation generates a token that goes to the matching unit and *sticks* there. A read operation generates a token that matches with an element *stuck* in the store, extracts a copy of it, and forwards the copy to the destination of the read operation, but leaves the sticky element in the store. If the read token fails to find a partner in the store, it cycles through the ring, busy-waiting. When the structure is deallocated, its elements must be purged from the store. This approach has not proved very successful. It increases the already large load on the matching unit and communication network, degrades the performance of the matching unit on standard operations, and makes its design much more complex. To resolve these problems, the Manchester group is developing a structure store similar to the I-structure store. Sticky tokens are also used for loop constants (discussed in Section 2.2). The iteration part of the tag is ignored in performing the match, and the sticky token remains in the store even when a match is performed. Cleaning up the matching store when a loop terminates presents difficulties.

The Manchester machine has provided a target for a number of dataflow languages and has run a number of sizable applications. Extensions to multi-ring machines are being studied through simulation. Work continues in areas related to controlling parallelism and instruction set design.

4.2.2 SIGMA-1 AT ELECTROTECHNICAL LABORATORY, JAPAN Under the auspices of the Japanese National Supercomputer Project, the Electrotechnical Laboratory is developing a machine (Yuba et al 1984) based on the MIT tagged-token architecture. The current proposal is to produce a prototype 32-bit machine capable of 100 Mflops, by the end of 1986. The individual processors are pipelined and operate on a 110-ns clock. The pipeline beat varies from 2 to 16 clock periods and is typically 3 to 4 periods. The network is packet-switched and composed of 10×10 routers. The engineering effort involved in this project is substantial, including the development of a 1-board PE and a 1-board structure memory. Together, these will require eight to ten custom cMOS gate-array chips and a custom VLSI chip. The PE will contain 64K words of program memory, 8K words of token buffering, and 64K words of waiting-matching store; the structure memory will contain 256K words.

The machine will have up to 304 boards, divided roughly half and half between the structure memory and ALU boards. A 6-board version of the PE has been operational since November 1984.

A number of interesting design choices have been made in Sigma-1. A short latency two-stage processor pipeline is employed to efficiently execute code with low parallelism. In the first stage, instruction fetch and matching are performed simultaneously. If the match fails, the fetched instruction is discarded. In the second stage, destination tags are generated in parallel with the ALU operation. Tokens are transferred through the network as 80-bit packets. Two cycles are required to receive a packet, but the first stage of the processor pipeline operates on the first 40 bits of the packet (the tag) while the second 40 bits are received. The waiting-matching store is implemented as a chained hash table. The first operand of a pair is inserted in the matching store in 4 cycles; matching the second token of a pair has an expected time of 2.6 cycles. Sticky tokens are employed for loop constants; however, the designers of the ETL machine have intimated that the utility of this approach may not warrant the added complexity in the matching unit. The structure controllers support deferred *reads*. Rather than support a general heap storage model, in which data objects may have arbitrary lifetimes, structures are deleted when the procedure that created the structure terminates. This simplifies storage management and is probably acceptable for writing numerical applications, the intended application area for the machine.

4.2.3 THE MIT TAGGED-TOKEN PROJECT Not surprisingly, the tagged-token machine presented in Section 2.2 reflects the approach of the authors' group at MIT. This machine developed through a sequence of stages (Arvind & Gostelow 1977a; Gostelow & Thomas 1980; Arvind et al 1980, 1983a; Arvind & Kathail 1981; Arvind & Iannucci 1983b) from theoretical work on the U-interpreter model (Arvind & Gostelow 1977b, 1982). The MIT group has focused on developing an entire dataflow system, rather than on hardware development per se. Two soft prototypes have been implemented to serve as vehicles for studying architectures, program development, and resource management. A simulator provides a detailed model of the machine, including internal timings, while a dataflow emulator, which runs on the multiprocessor emulation facility (Arvind et al 1983b) (MEF), supports studying the dynamic behavior of larger applications. The MEF is a collection of 32 Texas Instruments Explorer Lisp machines connected by a high bandwidth packet-switched network. Each Lisp machine emulates a dataflow PE. Both the simulator and emulator execute graphs produced by our compiler from the high-level dataflow language Id (Arvind et al 1978; Nikhil & Arvind 1985). A number of reasonably large benchmarks are being studied on the soft-prototypes of the MIT tagged-token machine, including a complex hydrodynamics and heat conduction code.

5. PROGNOSIS

In this review we have outlined two salient issues in dataflow architectures—token storage mechanisms and data structures. We have also surveyed several dataflow machines. We have not attempted to cover all the current research topics; for the interested reader, these include demand-driven evaluation (Pingali & Arvind 1985), controlled program unfolding and deadlock avoidance (Culler 1985; Ruggiero & Sargeant 1985; Arvind & Culler 1985), efficient procedure invocation, storage reclamation, relationships with parallel reduction architectures (Keller et al 1979; Darlington & Reeve 1981; Keller 1984), network design and topology, and semantics of programming languages with I-structures. However, dataflow architectures are of more than academic interest, so in conclusion we consider their potential in the real world.

Today a vast collection of single-board computers are available and offer roughly 1 MIPS at low cost; these are touted as building blocks for multiprocessors. Can dataflow machines compete? It is not clear if a single dataflow processor can achieve the performance of a von Neumann processor at the same hardware cost. The dataflow instruction-scheduling mechanism is clearly more complex than incrementing a program counter. An engineering effort substantially beyond any of the current dataflow projects is required to make a fair comparison. The Sigma-1 project is an important step in this direction. The question becomes more interesting when we consider machines with multiple processors, where the dataflow scheduling mechanism yields significant benefits. In the basic von Neumann machine the processor issues a memory request and waits for the result to be produced. The memory cycle time is invariably greater than the processor cycle time, so computer architects devote tremendous effort to reduce the amount of waiting. This problem is much more severe in a multiprocessor context because the time to process a memory request is generally much greater than in a single processor and is unpredictable. Furthermore, most traditional techniques for reducing the effects of memory latency do not work well in a multiprocessor setting. The dataflow approach can be viewed as an extreme solution to the memory latency problem: the processor never waits for responses from memory; it continues processing other instructions. Instructions are scheduled based on the availability of data, so memory responses are simply routed along with the tokens produced by processors. Thus, even if individual dataflow processors do not yield the performance per dollar of a conventional processor, we can expect them to be better utilized than a conventional processor in a multiprocessor setting. For large enough collections of processors they should be cost effective and should show absolute performance not achievable by conventional processors. But it is not yet clear where this threshold lies.

The preceding discussion suggests that dataflow machines are likely to be competitive in high-performance range; however, we do not make such a claim lightly. It is unlikely that a large collection of 1 MIPS machines of any ilk will compete with a few very high performance processors, i.e. processors that can perform 10 to 100 MFLOPs each. To compete among supercomputers, it may be necessary to engineer a dataflow machine with the technology and finesse employed in conventional supercomputers. This is a major undertaking, far beyond any of the dataflow projects currently proposed. Most supercomputers include vector accelerators to improve performance on a restricted class of programs. It remains to be seen how effective these will be in a multiprocessor context and the extent to which analogous accelerators will be needed for dataflow machines.

This paper has focused on architectural issues, and accordingly has scarcely touched on the high-level programming model that accompanies dataflow machines. Nonetheless, programmability of parallel machines is critical. Conventional programming languages are imperative and sequential in nature: do this, then do that, etc. Efforts to use these languages for describing parallel computation have been ad hoc and unwieldy, greatly increasing the difficulty of the already onerous programming task. The programmer must determine what synchronization is required to avoid read-write races. Even so, subtle timing bugs are common. A class of languages, called *functional* languages, completely avoid these synchronization problems by disallowing "updatable" variables. Functional languages employ function composition, rather than command sequencing, as the basic concept and can be translated into dataflow graphs easily, thereby exposing parallelism. These languages can be augmented with I-structures to make data structures more efficient, without sacrificing determinacy or parallelism. It is our belief that dataflow architectures together with these new languages will show the programming generality, performance, and cost effectiveness needed to make parallel machines widely applicable.

Acknowledgments

We gratefully acknowledge Robert Iannucci's drawings of various dataflow architectures, from which we have "borrowed" liberally. Many ideas in this paper derive from the common heritage of the Computation Structures Group at M.I.T. Laboratory for Computer Science, and we are indebted to its members for providing a stimulating research environment. We are grateful to Steven Brobst, Jack Dennis, K. Ekanadham, Bhaskar Guharoy, Gino Maa, Hitoshi Nohmi, Greg Papadopoulos, Natalie Tarbet, and Ken Traub for their valuable comments on drafts of this paper. Of course, we take responsibility for the opinions presented and any remaining errors.

This work was performed at the M.I.T. Laboratory for Computer Science under the Tagged-Token Dataflow project. Funding is provided in part by the Advanced Research Projects Agency of the U.S. Department of Defense, contract N00014-75-C-0661.

Literature Cited

Ackerman, W. B. 1978. A structure processing facility for dataflow computers. *Proc. Int. Conf. Parallel Process.*, pp. 166–72

Ackerman, W. B., Dennis, J. B. 1978. *VAL—A Value-Oriented Algorithmic Language: Preliminary Reference Manual.* Tech. Rep. TR-218, Lab. Comput. Sci., MIT, Cambridge, Mass.

Amamiya, M., Hasegawa, R., Nakamura, O., Mikami, H. 1982. A list-oriented data flow machine architecture. *Proc. Natl. Comput. Conf.*, pp. 143–51

Arvind, Culler, D. E. 1985. Managing resources in a parallel machine. *Proc. IFIP TC-10 Conf. Fifth-Generation Comput. Archit., Manchester, UK*

Arvind, Culler, D. E., Iannucci, R. A., Kathail, V., Pingali, K., Thomas, R. E. 1983a. *The Tagged Token Dataflow Architecture.* Tech. Rep., Lab. Comput. Sci., MIT, Cambridge, Mass. (Prepared for MIT Subject 6.83s)

Arvind, Dertouzos, M. L., Iannucci, R. A. 1983b. *A Multiprocessor Emulation Facility.* Tech. Rep. TR-302, Lab. Comput. Sci., MIT, Cambridge, Mass.

Arvind, Gostelow, K. P. 1977a. A computer capable of exchanging processors for time. *Proc. IFIP Congr. 77, Toronto, Canada,* pp. 849–53

Arvind, Gostelow, K. P. 1977b. Some relationships between asynchronous interpreters of a dataflow language. *Proc. IFIP WG2.2 Conf. Formal Description of Program. Lang., St. Andrews, Canada*

Arvind, Gostelow, K. P. 1982. The U-interpreter. *Computer* 15(2):42–49

Arvind, Gostelow, K. P., Plouffe, W. 1978. *An Asynchronous Programming Language and Computing Machine.* Tech. Rep. 114a, Dep. Inf. Comput. Sci., Univ. Calif., Irvine

Arvind, Iannucci, R. A. 1983a. A critique of multiprocessing von Neumann style. *Proc. 10th Int. Symp. Comput. Archit., Stockholm, Sweden,* pp. 426–36

Arvind, Iannucci, R. A. 1983b. *Instruction Set Definition for a Tagged-token Dataflow Machine.* Tech. Rep. CSG 212-3, Lab. Comput. Sci., MIT, Cambridge, Mass.

Arvind, Kathail, V. 1981. A multiple processor dataflow machine that supports generalized procedures. *Proc. 8th Ann. Symp. Comput. Archit., Minneapolis, Minn.,* pp. 291–302

Arvind, Kathail, V., Pingali, K. 1980. *A Dataflow Architecture with Tagged Tokens.* Tech. Rep. TM-174, Lab. Comput. Sci., MIT, Cambridge, Mass.

Brobst, S. A. 1986. *Token Storage Requirements in a Dataflow Supercomputer.* Tech. Rep., Lab. Comput. Sci., MIT, Cambridge, Mass. Submitted for publication

Comte, D., Hifdi, N., Syre, J. 1980. The data driven LAU multiprocessor system: results and perspectives. *Proc. IFIP Congr. 80, Tokyo, Japan,* pp. 175–80

Culler, D. E. 1985. *Resource Management for the Tagged-Token Dataflow Architecture.* Tech. Rep. TR-332, Lab. Comput. Sci., MIT, Cambridge, Mass.

Darlington, J., Reeve, M. 1981. ALICE: a multi-processor reduction machine for the parallel evaluation of applicative languages. *Proc. Conf. Functional Program. Lang. Comput. Archit., Portsmouth, NH,* pp. 65–76

Dennis, J. B. 1974. First version of a data flow procedure language. *Proceedings of the Colloque sur la Programmation, Vol. 19: Lecture Notes in Computer Science,* pp. 362–76. New York: Springer-Verlag

Dennis, J. B. 1980. Data flow supercomputers. *Computer* 13(11):48–56

Dennis, J. B., Boughton, G. A., Leung, C. K-C. 1980. Building blocks for data flow prototypes. *Proc. 7th Ann. Symp. Comput. Archit., La Boule, France,* pp. 1–8

Dennis, J. B., Fosseen, J., Linderman, J. 1972. Data flow schemas. *Proc. Symp. Theor. Program., Novosibirsk, USSR,* pp. 187–216

Dennis, J. B., Gao, G. R. 1983. Maximum pipelining of array operations on a static dataflow machine. *Proc. Int. Conf. Parallel Process.*

Dennis, J. B., Gao, G. R., Todd, K. 1984a. Modeling the weather with a data flow supercomputer. *IEEE Trans. Comput.* C33(7):592–603

Dennis, J. B., Misunas, D. 1974. *A Preliminary Architecture for a Basic Data Flow Processor.* Tech. Rep. CSG Memo 102, Lab. Comput. Sci., MIT, Cambridge, Mass.

Dennis, J. B., Stoy, J. E., Guharoy, B. 1984b. VIM: an experimental multi-user system supporting functional programming. *Proc. Int. Workshop High-Level Comput. Archit., Los Angeles, Calif.*, pp. 1.1–1.9

Friedman, D. P., Wise, D. S. 1976. CONS should not evaluate its arguments. In *Automata, Languages, and Programming,* ed. Michaelson, Milner. Edinburgh: Univ. Press

Gaudiot, J., Vedder, R., Tucker, G., Finn, D., Campbell, M. 1985. A distributed VLSI architecture for efficient signal and data processing. *IEEE Trans. Comput.* C34(12):1072–87

Gostelow, K. P., Thomas, R. E. 1980. Performance of a simulated dataflow computer. *IEEE Trans. Comput.* C29(10):905–19

Guharoy, B. 1985. *Structure management in a dataflow computer. Master's thesis. Dep. Elect. Eng. Comput. Sci., MIT, Cambridge, Mass.*

Gurd, J. R., Kirkham, C. C., Watson, I. 1985. The Manchester dataflow prototype computer. *Commun. Assoc. Comput. Mach.* 28(1):34–52

Heller, S. K. 1983. *An I-structure memory controller. Master's thesis. Dep. Elect. Eng. Comput. Sci., MIT, Cambridge, Mass.*

Hiraki, K., Nishida, K., Shimada, T. 1984. Evaluation of associative memory using parallel chained hashing. *IEEE Trans. Comput.* C33(9):851–55

Jaffe, J. M. 1979. *The Equivalence of R. E. Programs and Data Flow Schemes.* Tech. Rep. TM-121, Lab. Comput. Sci., MIT, Cambridge, Mass.

Johnson, D., et al. 1980. Automatic partitioning of programs in multiprocessor systems. *Proc. Compcon 80,* pp. 175–78

Kahn, G. 1974. The semantics of a simple language for parallel programming. *Proc. IFIP Cong. 74,* pp. 471–75

Keller, R. 1984. Rediflow multiprocessing. *Proc. Compcon 84*

Keller, R. M., Lindstrom, G., Patil, S. 1979. A loosely-coupled applicative multiprocessing system. *Proc. Natl. Comput. Conf., New York,* pp. 613–22

Misunas, D. 1975. Deadlock avoidance in a data-flow architecture. *Proc. Milwaukee Symp. Autom. Comput. Control*

Montz, L. B. 1980. Safety and Optimization Transformations for Data Flow Programs. *Tech. Rep. TR-240, Lab. Comput. Sci., MIT, Cambridge, Mass.*

NEC. 1985. *Advanced Product Information User's Manual: μPD7281 Image Pipelined Processor*

Nikhil, R., Arvind. 1985. *Id/83s. Tech. Rep., Lab. Comput. Sci., MIT, Cambridge, Mass.* (Prepared for MIT Subject 6.83s)

Pingali, K., Arvind. 1985. Efficient demand-driven evaluation. Pt. I. *ACM TOPLAS* 7(2):311–33

Preiss, B. R., Hamacher, V. C. 1985. Data flow on a queue machine. *Proc. 12th Ann. Int. Symp. Comput. Archit., Boston, Mass.,* pp. 342–51

Ruggiero, J., Sargeant, J. 1985. *Hardware and Software Mechanisms for Control of Parallelism.* Tech. Rep., Comput. Sci. Dep., Univ. Manchester, UK

Rumbaugh, J. A. 1977. Data flow multiprocessor. *IEEE Trans. Comput.* C26(2):138–46

Takahashi, N., Amamiya, M. 1983. A dataflow processor array system: design and analysis. *Proc. 10th Int. Symp. Comput. Archit., Stockholm, Sweden,* pp. 243–50

Temma, T., Hasegawa, S., Hanaki, S. 1980. Dataflow processor for image processing. *Proc. 11th Int. Symp. Mini and Microcomput., Monterey, Calif.,* pp. 52–56

Watson, I., Gurd, J. R. 1982. A practical dataflow computer. *Computer* 15(2):51–57

Yuba, T., Shimada, T., Hiraki, K., Kashiwagi, H. 1984. *Sigma-1: A Dataflow Computer For Scientific Computation.* Tech. Rep., Electrotech. Lab.

Annual Review of Computer Science
Volume 1, 1986

CONTENTS