Dataflow Architectures

MIT/LCS/TM-294

12 February 1986

Arvind

David E. Culler

To appear in Annual Reviews in Computer Science, 1986

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 N000-1483-K-0125.

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 which 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.

Keywords: Dataflow, Dataflow graphs, Determinacy, Dynamic Dataflow architectures, Functional data structures, I-structures, Multiprocessors, Parallel computation, Parallel computers, Static Dataflow architectures, Structure storage, Tagged-Token Dataflow architectures, Token storage.

Dataflow Architectures

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 [36]. This gives rise to two serious, pragmatic concerns: (1) How should the tokens on arcs be managed? (2) 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 paper, we examine the major variations in dataflow architectures with regard to token storage mechanisms and data structure storage.

The paper is organized as follows. The rest of Section 1 introduces dataflow program graphs and the rules which determine when and how operations are performed. Also, it explains why data structures can not 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 places the restriction that at most one token can 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 which is, conceptually, 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 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 what the future holds for 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 [22]. As an example, Figure 1 shows the acyclic dataflow program graph for the following expression.

let $x = a * b;$
$y = 4 * c$
in $(x + y) * (x - y) / c$

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 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.

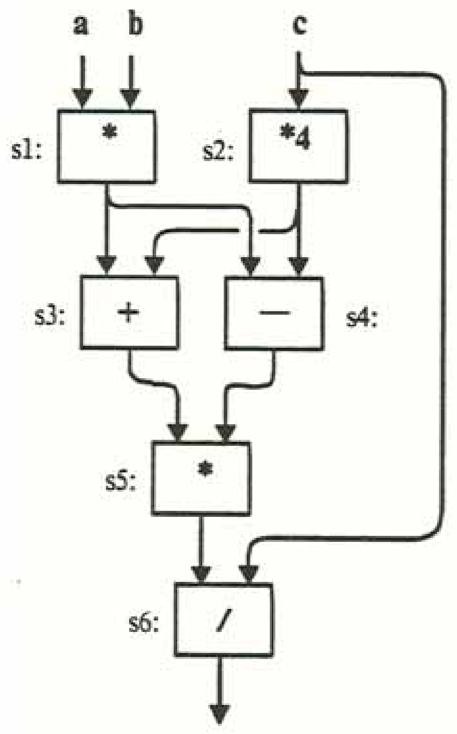

Figure 1: Acyclic Dataflow Graph

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, say through shared memory. This example illustrates two key properties of the dataflow approach: (1) *parallelism*, *i.e.*, nodes may potentially execute in parallel unless there is an explicit data dependence between them, and (2) *determinacy*, *i.e.*, results do not depend on the relative order in which potentially parallel nodes execute[1]. Further, notice that by supplying several sets of input tokens, distinct computations can be

[1]The unbounded FIFO queue model presented in this paper is a generalization of the dataflow model originally formulated by Dennis. His model [22] 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 [36] 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.

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 which have this property are called *well-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 [24]. Consider first the conditional graph in Figure 2.a which 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 which 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, providing 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 is allowed to enter the loop at a time; 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.*

1.2. Data Structures

The dataflow model introduced thus far is fully general in a formal computational sense [34], 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, producing a pair, and selectors *first* and *rest* which access the components of a pair. Since these new operators are functions, they fit easily in the dataflow model, provided we assume tokens can carry composite data values. Note

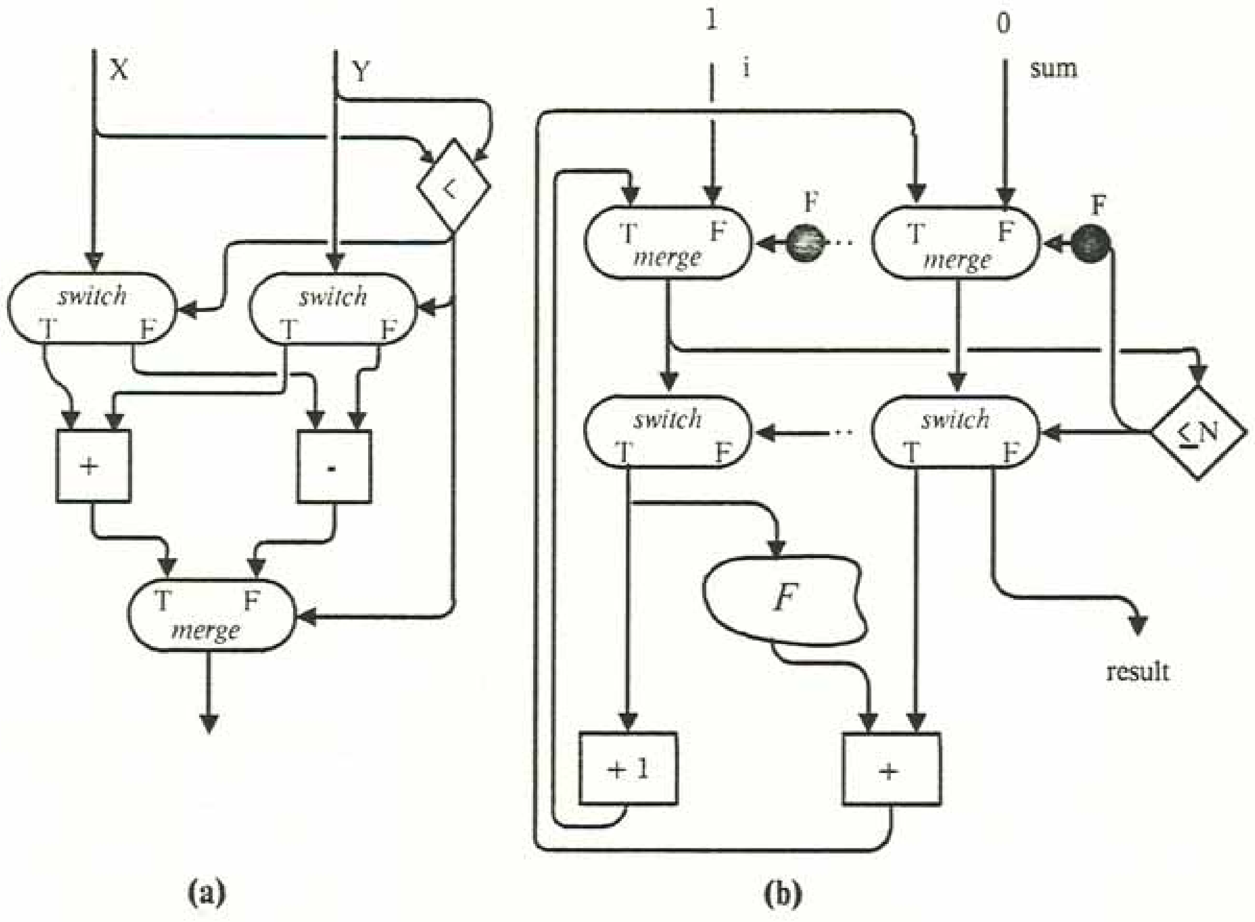

Figure 2: Conditional and Loop Graphs

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 which 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 which 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 determinacy of the model: it is no longer true that instructions can execute in any order consistent with the data dependencies and 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 model of computation is the ability to support user-defined functions. Each of our examples represents a function which, 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 which allows arguments and results to be transmitted properly. Non-recursive functions can be handled by graph expansion at compile time. However, to support user-defined functions more generally, we need an *apply* operator which takes as inputs a function-value, (*i.e.*, description of an encapsulated dataflow graph) and a set of arguments, and 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, an instruction or node may execute whenever a token is available on each of its input arcs, and 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: (1) detect when an operation is enabled (this is tantamount to collecting operand values), (2) determine the operation to be performed, *i.e.*, fetch the instruction, (3) compute results, and (4) 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, the management of token storage and instruction scheduling 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 is straightforward to determine if the other inputs are all present. This idea underlies the static dataflow machines proposed by Dennis and his co-workers [21, 23, 25] (see 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

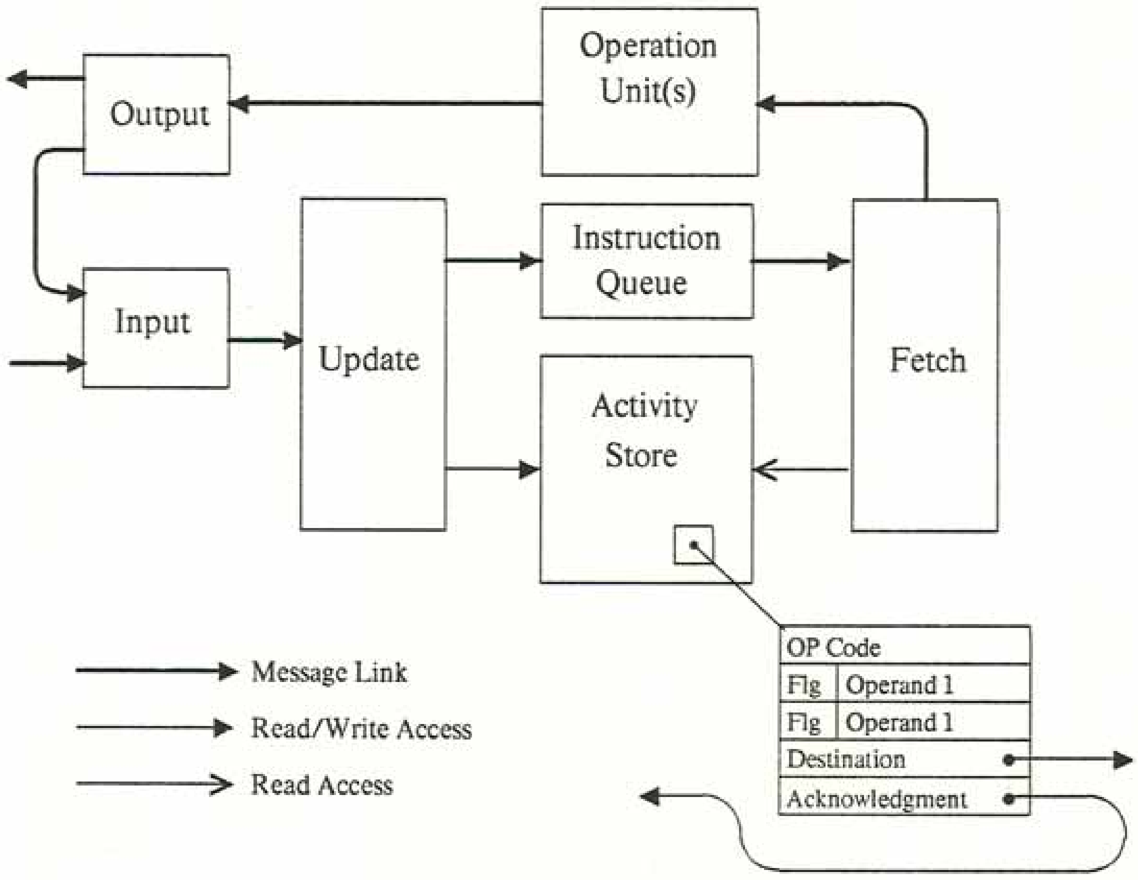

Figure 3: Static Dataflow Architecture

determine if the corresponding activity is enabled. If so, 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' model. Suppose the units 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 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 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 which have the property that 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 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 [40], 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 [20] 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 communication delays between operators are constant. Neither assumption is realistic and balancing becomes computationally intractable without these

[2]Misunas shows [39] that multiple tokens per arc can also cause the machine to deadlock.

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 next.

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 which is supposed to be the i^{th} value to flow along a given arc carries i in its tag. The trick is to give simple tag generation rules for the control operators, *switch* and *merge*. Arvind and Gostelow [7] have given such rules for Dennis' operators [22]. However, if only well-behaved graphs are considered, then it is possible to develop even simpler tag manipulation rules [9]. 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 intend the tagged-token approach to support user-defined functions, so a program is viewed as a collection of graphs, called *code-blocks*, where each graph is either acyclic or a single loop. A node is identified by a pair <code-block, instruction address>. Tags have four parts: <invocation ID, iteration ID, code-block, instruction address>, where 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 which 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 given tag, a data item carrying that tag will arrive on at most 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.

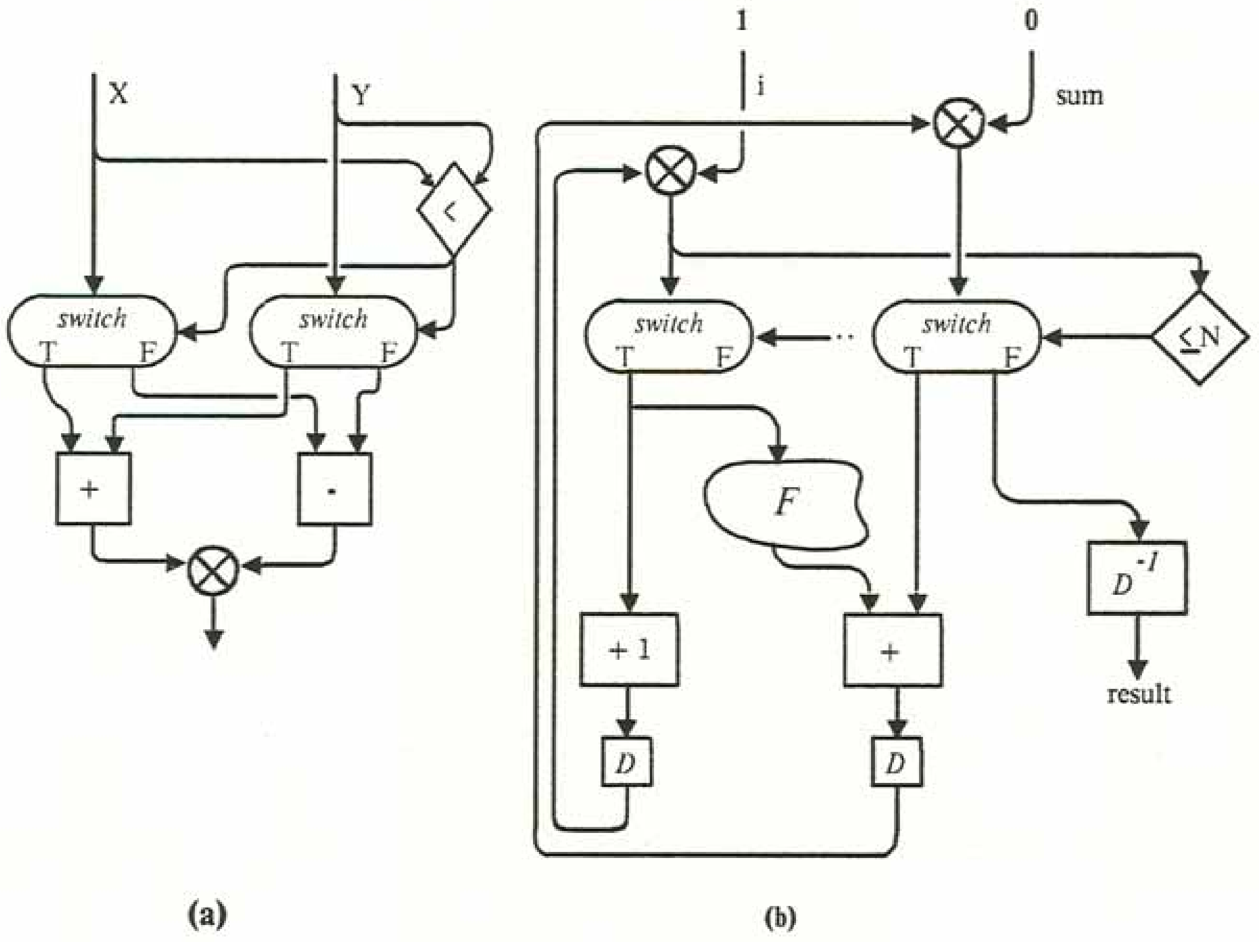

Figure 4: Conditional and Loop Graphs for Tagged Approach

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 ID's. 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 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 [10] 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 [8].

2.2.2. Tagged-Token Dataflow Machine

A machine proposed by Arvind *et al.* [4] is depicted in Figure 5. It comprises a collection of processing elements (PE's) 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 to at most two operands so a single match enables an activity.) Tokens which 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) which 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 which 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 PE's. 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 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 occur [17]. 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 at 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.

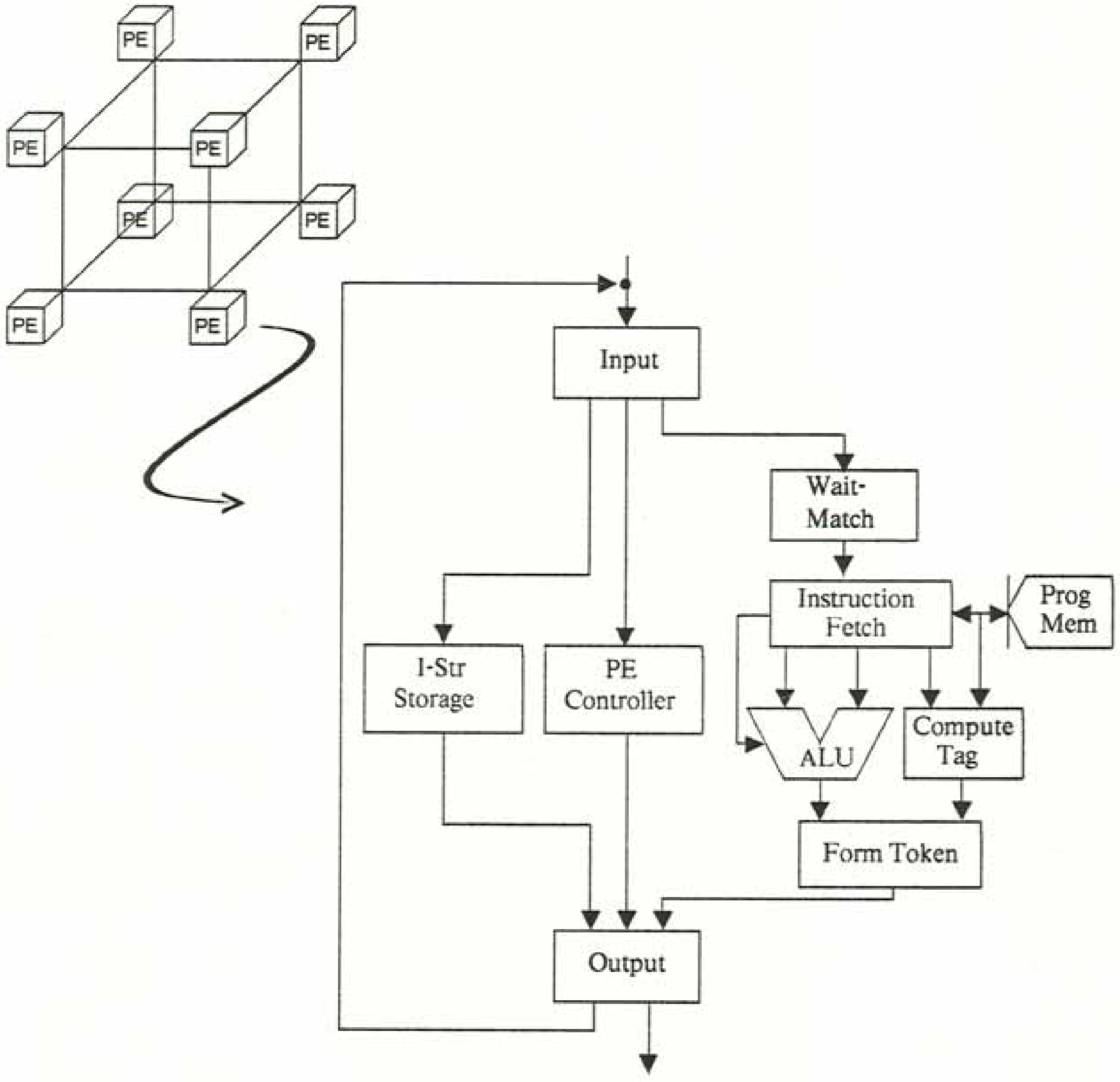

Figure 5: Processing Element of the MIT Tagged Token Dataflow Machine

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 PE's within a collection by "hashing" the tags. A variety of mapping schemes have been developed to distribute the most frequently encountered program structures efficiently. The MAP register assigned to a code-block invocation keeps the hashing function to be used 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 which are invariant over all iterations for a particular invocation of a loop, but vary for different invocations, must be circulated. Variable 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, allowing 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 PE's, 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 [17]. This has turned out to be 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 which 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 (see [33] for examples of the hashing functions used in a tagged-machine).

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 oe larger), where a non-existant 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 as there is evidence to suggest that when an incoming token finds its partner, the partner is usually among the most recently arrived tokens [15]. 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 [26, 44]. 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 due to 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.

Coming from the other direction, a variation of the tagged-token machine that has been proposed by David Culler and Greg 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 the 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 PE's and is capable of performing many concurrent structure operations.

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 [1, 31] 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

[3]Not providing direct access to a large storage shared by many PE's 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 beauty of dataflow machines is they can be made extremely tolerant of latency, and thus can sustain high performance with many processors working on a single problem. Detailed arguments to this account can be found in Arvind and Iannucci [11].

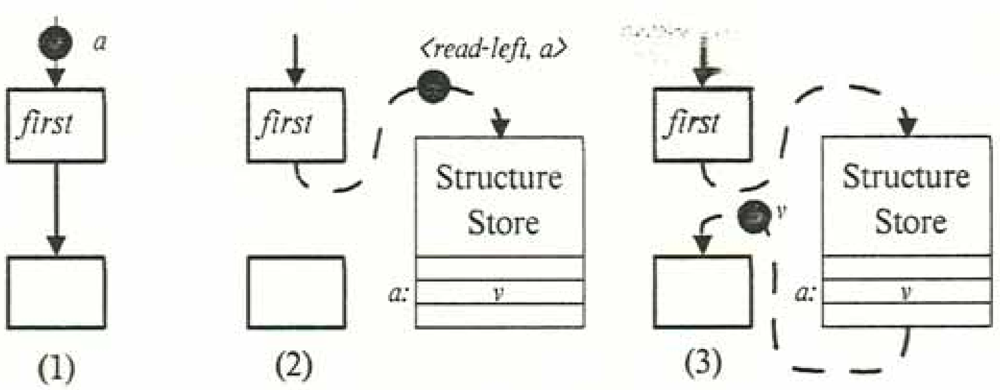

Figure 6: Action of a *first* operation

corresponding array, the array can be updated in place without risk of causing a read-write race. Second, if the array is 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 which uses the first element of the pair, but not the second, must wait until both elements have been computed. Such data structures are called *strict* in programming language jargon. In contrast, *cons* can be treated as a *non-strict* operator [27], allowing 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 non-strict *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 which 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. (1) PRESENT: The word contains valid data which can be freely read as in a conventional memory. Any attempt to write it will be signalled as an error. (2) ABSENT: Nothing has been written into the cell since it was last allocated. No attempt has been made to read the cell;

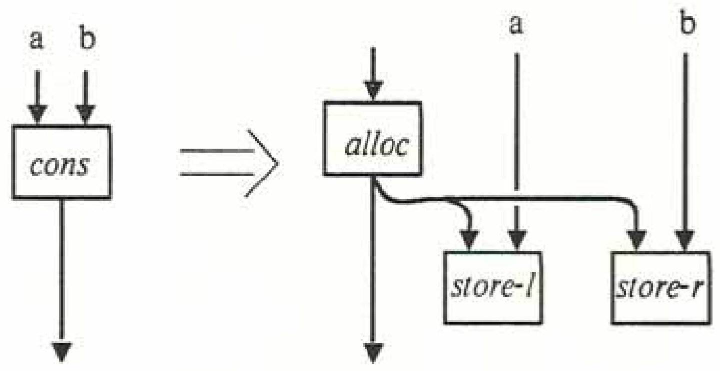

Figure 7: Implementation of Non-strict Cons

it may be written as for conventional memory. (3) 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 non-strict *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 [42] which are initially empty, and which 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" which 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 [32] 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 non-strict 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 non-strict structure.

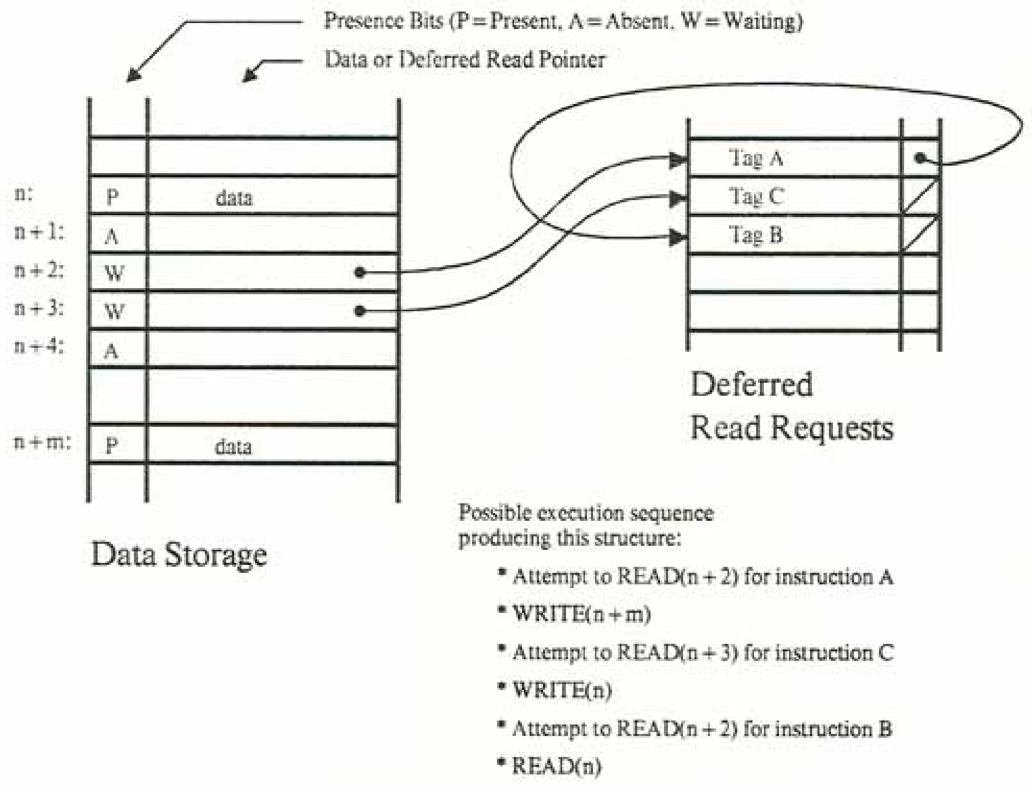
Presence Bits (P=Present, A=Absent, W=Waiting)
Data or Deferred Read Pointer
n:
n+1:
n+2:
n+3:
n+4:
n+m:
P
data
A
W
W
A
P
data
Data Storage
Tag A
Tag C
Tag B
Deferred
Read Requests
Possible execution sequence
producing this structure:
* Attempt to READ(n+2) for instruction A
* WRITE(n+m)
* Attempt to READ(n+3) for instruction C
* WRITE(n)
* Attempt to READ(n+2) for instruction B
* READ(n)

Figure 8: I-Structure Storage

4. Current Dataflow Projects

We now present an overview of some of the more important dataflow projects, restricting 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 Dennis' seminal work [22]. Such projects, besides Dennis' own project, include the LAU project in Toulouse, France [16], the Texas Instruments dataflow project [35], the Hughes dataflow machine [28], and several projects in Japan [48, 41]. 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' work.

4.1.1. The MIT Static Machine Dataflow Project

Dennis' group at MIT has proposed and refined several static dataflow architectures over the years [21, 46, 19, 25], and have implemented an eight-processor engineering model of the static machine shown in Figure 3 [19]. The processing elements (PE) were built out of AMD bit-slice micro-processors and were connected by a packet-switched butterfly network composed of 2x2, byte-serial routers with send-acknowledge protocol. The structure controller was not implemented. Dataflow graphs for the machine were compiled from the language VAL [2]. 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 PE's were far slower than the routers. The Texas Instruments machine [35], which was architecturally similar to Dennis' 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 marked influence on instruction scheduling in high-performance machines intended for scientific computing.

4.1.2. The NEC Dataflow Machines

The latest machines which may be classified as static machines are NEC's NEDIPS [48] and Image Pipelined Processor (IPP) μPD7281 [41]. NEDIPS is a 32-bit machine intended for scientific computation and uses high-speed logic, while the IPP is a single chip processor of similar architecture, 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 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 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, allowing up to 14 IPP's 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 is allocated statically, *i.e.*, by the programmer or compiler. The programmer must tune the program graph to avoid buffer overflows and ensure that tokens do not get out of order. This makes program development for this machine a tedious task. The buffer overflow problem is much less severe in NEDIPS because it provides much more data memory (64K words) than IPP. Still the problem is serious enough to cause the designers to modify NEDIPS so 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 [29] is an important milestone, and 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 NTT [3, 47], and Sigma-1 at ETL 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 ALU's. The ALU's are microcoded and fairly slow. It has demonstrated reasonable performance (1.2 MIPS) with this arrangement, although the choice of many slow ALU's has received some criticism because all the ALU's 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 [49]. 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 which goes to the matching unit and *sticks* there. A read operation generates a token which 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, as well as 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 [50] 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 100ns cycle. The network is packet-switched and composed of 4x4 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 16k words of program memory, 8k words of token buffering, and 64k words of waiting-matching store, and the structure memory 256k words. (The memory sizes may be increased by a factor of four by the time the machine is built.) The machine will have up to 180 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 execute code with low parallelism efficiently. 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 which 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 [7, 30, 14, 13, 12, 4] from theoretical work on the U-interpreter model [8, 9]. 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 is being developed to run on the Multiprocessor Emulation Facility [6] (MEF), to study dynamic behavior of larger applications. The MEF is a collection of Lisp machines (38 Texas Instruments Explorers and 8 Symbolics 3600's) which will be connected by a high bandwidth packet-switched network in the near future. 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 [10, 42]. 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 paper we have outlined two salient issues in dataflow architectures: token storage mechanisms and data structures, and surveyed several dataflow machines. We have not attempted to cover all the current research topics; for the interested reader, these include: demand-driven evaluation [43], controlled program unfolding and deadlock avoidance [17, 45, 5], efficient procedure invocation, storage reclamation, relationships with parallel reduction architectures [38, 18, 37], 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 which 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. Further, 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 as well as 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 would 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 which 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 which 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, 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.

References

1. Ackerman, W. B. A Structure Processing Facility for Dataflow Computers. Proceedings of the 1978 International Conference on Parallel Processing, July, 1978, pp. 166-172.

2. Ackerman, W. B., and J.B. Dennis. VAL -- A Value - Oriented Algorithmic Language: Preliminary Reference Manual. TR-218, Laboratory for Computer Science, MIT, Cambridge, MA, December, 1978.

3. Amamiya, M., R. Hasegawa, O. Nakamura and H. Mikami. A List-oriented Data Flow Machine Architecture. Proceedings of the National Computer Conference, AFIPS, 1982, pp. 143-151.

4. Arvind, D. E. Culler, R. A. Iannucci, V. Kathail, K. Pingali and R. E. Thomas. The Tagged Token Dataflow Architecture. Laboratory for Computer Science, MIT, Cambridge, MA, July, 1983. (Prepared for MIT Subject 6.83s).

5. Arvind, and D. E. Culler. Managing Resources in a Parallel Machine. Proceedings of the IFIP TC-10 Conference on Fifth-Generation Computer Architecture, Manchester, U.K., July, 1985.

6. Arvind, M. L. Dertouzos and R. A. Iannucci. A Multiprocessor Emulation Facility. TR-302, Laboratory for Computer Science, MIT, Cambridge, MA, October, 1983.

7. Arvind, and K. P. Gostelow. A Computer Capable of Exchanging Processors for Time. Proceedings of IFIP Congress 77, Toronto, Canada, August, 1977, pp. 849-853.

8. Arvind, and K.P. Gostelow. Some Relationships Between Asynchronous Interpreters of a Dataflow Language. Proceedings of the IFIP WG2.2 Conference on Formal Description of Programming Languages, St. Andrews, Canada, 1977.

9. Arvind, and K. P. Gostelow. "The U-interpreter". *Computer 15*, 2 (February 1982), 42-49.

10. Arvind, K. P. Gostelow and W. Plouffe. An Asynchronous Programming Language and Computing Machine. 114a, Department of Information and Computer Science, University of California, Irvine, CA, December, 1978.

11. Arvind, and R. A. Iannucci. A Critique of Multiprocessing von Neumann Style. Proceedings of the 10th International Symposium on Computer Architecture, Stockholm, Sweden, June, 1983, pp. 426-436.

12. Arvind, and R. A. Iannucci. Instruction Set Definition for a Tagged-token Dataflow Machine. CSG 212-3, Laboratory for Computer Science, MIT, Cambridge, MA, February, 1983.

13. Arvind, and V. Kathail. A Multiple Processor Dataflow Machine that Supports Generalized Procedures. Proceedings of the 8th Annual Symposium on Computer Architecture, Minneapolis, MN, May, 1981, pp. 291-302.

14. Arvind, V. Kathail and K. Pingali. A Dataflow Architecture with Tagged Tokens. TM-174, Laboratory for Computer Science, MIT, Cambridge, MA, September, 1980.

15. Brobst, S. A. Token Storage Requirements in a Dataflow Supercomputer. Laboratory for Computer Science, MIT, Cambridge, MA, May, 1986. "To be published".

16. Comte, D., N. Hifdi and J. Syre. The Data Driven LAU Multiprocessor System: Results and Perspectives. Proceedings of IFIP Congress 80, Tokyo, Japan, October, 1980, pp. 175-180.

17. Culler, D. E. Resource Management for the Tagged-Token Dataflow Architecture. TR-332, Laboratory for Computer Science, MIT, Cambridge, MA, January, 1985.

18. Darlington, J., and M. Reeve. ALICE: A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages. Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture, Portsmouth, NH, 1981, pp. 65-76.

19. Dennis, J. B., G. A. Boughton and C. K-C. Leung. Building Blocks for Data Flow Prototypes. Proceedings of the 7th Annual Symposium on Computer Architecture, La Boule, France, May, 1980, pp. 1 - 8.

20. Dennis, J. B., and G. R. Gao. Maximum Pipelining of Array Operations on a Static Dataflow Machine. Proceedings of the 1983 International Conference on Parallel Processing, August, 1983.

21. Dennis, J. B., and D. Misunas. A Preliminary Architecture for a Basic Data Flow Processor. CSG Memo 102, Laboratory for Computer Science, MIT, Cambridge, MA, August, 1974.

22. Dennis, J. B. First Version of a Data Flow Procedure Language. Proceedings of the Colloque sur la Programmation, Vol. 19, Lecture Notes in Computer Science, Springer-Verlag, 1974, pp. 362-376.

23. Dennis, J. B. "Data Flow Supercomputers". *Computer 13*, 11 (November 1980), 48-56.

24. Dennis, J., J. Fosseen and J. Linderman. Data Flow Schemas. Proceedings of the Symposium on Theoretical Programming, Novosibirsk, USSR, 1972, pp. 187-216.

25. Dennis, J. B., G. R. Gao and K. Todd. "Modeling the Weather with a Data Flow Supercomputer". *IEEE Transactions on Computers C-33*, 7 (July 1984), 592-603.

26. Dennis, J. B., J. E. Stoy and B. Guharoy. VIM: An Experimental Multi-User System Supporting Functional Programming. Proceedings of the 1984 International Workshop on High-Level Computer Architecture, Los Angeles, CA, May, 1984, pp. 1.1-1.9.

27. Friedman, D. P., and D. S. Wise. CONS Should Not Evaluate its Arguments. In *Automata, Languages, and Programming*, Edinburgh University Press, 1976, pp. 257-284.

28. Gaudiot, J., R. Vedder, G. Tucker, D. Finn and M. Campbell. "A Distributed VLSI Architecture for Efficient Signal and Data Processing". *IEEE Transactions on Computers c-34*, 12 (December 1985), 1072-1087.

29. Gurd, J. R., C. C. Kirkham, and I. Watson. "The Manchester Dataflow Prototype Computer". *Communications of the ACM 28*, 1 (January 1985), 34-52.

30. Gostelow, K. P., and R. E. Thomas. "Performance of a Simulated Dataflow Computer". *IEEE Transactions on Computers C-29*, 10 (October 1980), 905-919.

31. Guharoy, B. Structure Management in a Dataflow Computer. Master Th., Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA,May 1985.

32. Heller, S. K. An I-Structure Memory Controller. Master Th., Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA,June 1983.

33. Hiraki, K., K. Nishida and T. Shimada. "Evaluation of Associative Memory Using Parallel Chained Hashing". *IEEE Transactions on Computers C-33*, 9 (September 1984), 851-855.

34. Jaffe, J. M. The Equivalence of R. E. Programs and Data Flow Schemes. TM-121, Laboratory for Computer Science, MIT, Cambridge, MA, January, 1979.

35. Johnson, D., et. al. Automatic Partitioning of Programs in Multiprocessor Systems. Proceedings of Compcon 80, February, 1980, pp. 175-178.

36. Kahn, G. The Semantics of a Simple Language for Parallel Programming. Proceedings of the IFIP Congress 74, 1974, pp. 471-475.

37. Keller, R. Rediflow Multiprocessing. Proceedings of Compcon 84, IEEE, 1984.

38. Keller, R.M., G. Lindstrom and S. Patil. A Loosely-Coupled Applicative Multi-Processing System. Proceedings of the National Computer Conference, New York, NY, June, 1979, pp. 613-622.

39. Misunas, D. Deadlock Avoidance in a Data-Flow Architecture. Proceedings of the Milwaukee Symposium on Automatic Computation and Control, April, 1975.

40. Montz, L. B. Safety and Optimization Transformations for Data Flow Programs. TR-240, Laboratory for Computer Science, MIT, Cambridge, MA, January, 1980.

41. NEC. *Advanced Product Information User's Manual: μPD7281 Image Pipelined Processor.* NEC Electronics Inc., Mountain View, CA, 1985.

42. Nikhil, R., and Arvind. Id/83s. Laboratory for Computer Science, MIT, Cambridge, MA, July, 1985. (Prepared for MIT Subject 6.83s).

43. Pingali, K., and Arvind. "Efficient Demand-driven Evaluation. Part I". *ACM TOPLAS 7*, 2 (May 1985), 311-333.

44. Preiss, B. R., and V. C. Hamacher. Data Flow on a Queue Machine. Proceedings of the 12th Annual International Symposium on Computer Architecture, Boston, MA, June, 1985, pp. 342-351.

45. Ruggiero, J., and J. Sargeant. Hardware and Software Mechanisms for Control of Parallelism. Computer Science Dept., University of Manchester, Manchester, England, April, 1985.

46. Rumbaugh, J. "A Data Flow Multiprocessor". *IEEE Transactions on Computers C-26*, 2 (February 1977), 138-146.

47. Takahashi, N., and M. Amamiya. A Dataflow Processor Array System: Design and Analysis. Proceedings of the 10th International Symposium on Computer Architecture, Stockholm, Sweden, June, 1983, pp. 243-250.

48. Temma, T., S. Hasegawa and S. Hanaki. Dataflow Processor for Image Processing. Proceedings of the 11th International Symposium on Mini and Microcomputers, Monterey, CA, 1980, pp. 52-56.

49. Watson, I., and J. R. Gurd. "A Practical Dataflow Computer". *Computer 15*, 2 (February 1982), 51-57.

50. Yuba, T., T. Shimada, K. Hiraki, and H. Kashiwagi. Sigma-1: A Dataflow Computer For Scientific Computation. Electrotechnical Laboratory, 1-1-4 Umesono, Sakuramura, Niiharigun, Ibaraki 305, Japan, 1984.

Table of Contents

List of Figures