Dataflow Machine Architecture

ARTHUR H. VEEN

Center for Mathematics and Computer Science, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Dataflow machines are programmable computers of which the hardware is optimized for fine-grain data-driven parallel computation. The principles and complications of data-driven execution are explained, as well as the advantages and costs of fine-grain parallelism. A general model for a dataflow machine is presented and the major design options are discussed.

Most dataflow machines described in the literature are surveyed on the basis of this model and its associated technology. For general-purpose computing the most promising dataflow machines are those that employ packet-switching communication and support general recursion. Such a recursion mechanism requires an extremely fast mechanism to map a sparsely occupied virtual space to a physical space of realistic size. No solution has yet proved fully satisfactory.

A working prototype of one processing element is described in detail. On the basis of experience with this prototype, some of the objections raised against the dataflow approach are discussed. It appears that the overhead due to fine-grain parallelism can be made acceptable by sophisticated compiling and employing special hardware for the storage of data structures. Many computing-intensive programs show sufficient parallelism. In fact, a major problem is to restrain parallelism when machine resources tend to get overloaded. Another issue that requires further investigation is the distribution of computation and data structures over the processing elements.

Categories and Subject Descriptors: A.1 [**General Literature**]: Introductory and Survey; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures—*multiple-instruction-stream, multiple-data-stream processors (MIMD)*; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*data-flow architectures*; C.4 [**Computer Systems Organization**]: Performance of Systems—*design studies*

General Terms: Design, Performance

Additional Key Words and Phrases: Data-driven architectures, dataflow machines, data structure storage

INTRODUCTION

Early advocates of data-driven parallel computers had grand visions of plentiful computing power provided by machines that were based on simple architectural principles and that were easy to program, maintain, and extend. Experimental dataflow machines have now been around for more than a decade, but still there is no consensus as to whether data-driven execution, besides being intuitively appealing, is also a viable means to make these visions become reality.

Author's current address: Computing Science Department, University of Amsterdam, P.O. Box 41882, 1009 DB Amsterdam, The Netherlands; or Arthur e MCVAX.cwl.NL.

CONTENTS

The concept of data-driven computation is as old as electronic computing. It is ironic that the same von Neumann, who is sometimes blamed for having created a bottleneck that dataflow architecture tries to remove, made an extensive study of neural nets, which have a data-driven nature. Asynchronously operating in/out channels, introduced in the 1950s, which communicate according to a ready/acknowledge protocol, are among the first implementations of data-driven execution. The development, in the 1960s, of multiprogrammed operating systems provided the first experience with the complexities of large-scale asynchronous parallelism. After exposure to these problems in the MULTICS project, Dennis [1969] developed the model of dataflow schemas, building on work by Karp and Miller [1966] and Rodriquez [1969]. These dataflow graphs, as they were later called, evolved rapidly from a method for designing and verifying operating systems to a base language for a new architecture. The first designs for such machines [Dennis and Misunas 1974; Rumbaugh 1975] were made at Massachusetts Institute of Technology. The first dataflow machine became operational in July 1976 [Davis 1979].

The dataflow field has matured considerably in the past decade. Realistic hardware prototypes have become operational, experience with compiling and large-scale simulation has been gained, and the execution of large programs has been studied. Early optimism has often been replaced by an appreciation of the problems involved. A keen understanding of these problems is, however, still lacking. In the years ahead the emphasis may shift from exploratory research to evaluation and to a thorough analysis of the problems deemed most crucial. Many of these problems have counterparts in other parallel computers, certainly in those based on fine-grain parallelism.

To facilitate such an analysis, we shall attempt to summarize the work done so far. A clear view of the common properties of different dataflow machines is sometimes obscured by trivial matters such as differences in terminology, choice of illustrations, or emphasis. In order to reduce such confusion, all designs are described as instances of a general dataflow machine. All necessary terminology is introduced when this general model is presented; the reader does not have to be familiar with dataflow or general graph terminology. Some understanding of the problems encountered in parallel architecture is, however, helpful.

There is no sharp definition of dataflow machines in the sense of a widely accepted set of criteria to distinguish dataflow machines from all other computers. For the sake of this survey we consider dataflow machines to be all *programmable* computers of which the hardware is optimized for *fine-grain data-driven* parallel computation. *Fine grain* means that the processes that

run in parallel are approximately of the size of a conventional machine code instruction. *Data driven* means that the activation of a process is solely determined by the availability of its input data. This definition excludes simulators as well as nonprogrammable machines, for instance, those that implement the dataflow graph directly in hardware, an approach that is popular for the construction of dedicated asynchronous signal processors. We also exclude data-driven computers that use coarse-grain parallelism such as the MAUD system [Lecouffe 1979], or medium-grain parallelism [Hartimo et al. 1986; Preiss and Hamacher 1985] and computers that are not purely data driven [Treleaven et al. 1982a].

Comparisons of dataflow machines have appeared elsewhere, but they were mostly limited to a few machines [Dennis 1980; Hazra 1982]. Recently Srini [1986] presented a comparison of eight machines. Excellent surveys with a wider scope, including sequential and demand-driven computers, can be found in Treleaven et al. [1982b] and Vegdahl [1984].

This paper is only concerned with the *architecture* of dataflow machines, that is, the machine language and its implementation. To limit the size of the paper all issues related to the equally important areas of compiling and programming have been omitted. Introductory material on dataflow languages, which are commonly used to program dataflow machines, can be found in McGraw [1982] and Glauert [1984]. There is extensive literature on the closely related group of functional languages [Darlington et al. 1982]; a recommended introduction is Peyton-Jones [1984]. Positive experiences with programming in imperative languages have also been reported [Allan and Oldehoeft 1980; Veen 1985a]. Discussions on programming techniques suitable for dataflow machines are just beginning to appear [Dennis et al. 1984; Gurd and Böhm 1987].

The first section of this paper places dataflow machines in the context of other parallel computers. In the next section we introduce dataflow graphs, the machine language of most dataflow machines. Readers familiar with dataflow concepts can skip these first two sections. In Section 3 we describe the execution of a program on a dataflow machine, and discuss different types of machine organizations. Section 4 is a presentation of a comparative survey of a wide variety of machine proposals and is suitable as a starting point for a literature study. Section 5 contains a detailed study of one operational prototype, and in Section 6 the feasibility of the dataflow concept is discussed on the basis of this prototype.

1. DATAFLOW MACHINES VERSUS OTHER PARALLEL COMPUTERS

The efficiency of a parallel computer is influenced by several conflicting factors. A major problem is *contention* for a shared resource, usually shared memory or some other communication channel. Contention can often be reduced by careful coordination, allocation, and scheduling, but if this is done at run time, it increases the *overhead* due to parallelism, that is, processing that would be unnecessary without parallelism. If, during a significant part of a computation, a major part of the processing power is not engaged in useful computation, we speak of *underutilization*. A useful measure of the quality of a parallel computer is its *effective utilization*, that is, utilization corrected for overhead. The best one can hope for is that the effective utilization of a parallel computer approaches that of a well-designed sequential computer. Another desirable quality is *scalability*, that is, the property that the performance of the machine can always be improved by adding more processing elements. We speak of *linear speed-up* if the effective utilization does not drop when the machine is extended.

Flynn [1972] introduced the distinction between parallel computers with a single instruction stream (SIMD) and those with multiple instruction streams (MIMD). The characteristic feature of SIMD computers is that they are synchronous at the machine language level. In the programming of such computers, the timing of concurrent computations plays a prominent role. They require skillful programming to bring utilization to an acceptable level since

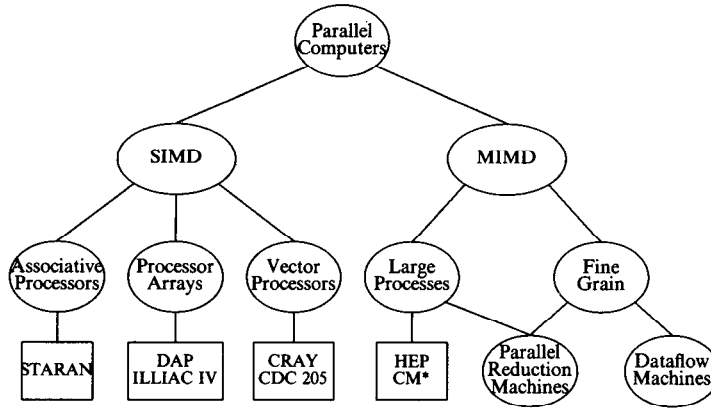

Figure 1. Some of the design options for parallel computers. In SIMD machines the parallel operations are synchronized at the machine language level, and scheduling and allocation needs to be done by the programmer. In MIMD machines the processes that run in parallel need to be synchronized whenever they communicate with each other.

scheduling and allocation, that is, deciding when and where a computation will be executed, has to be done statically, either by the programmer or by a sophisticated compiler. For certain kinds of applications this is quite feasible. For instance, in low-level signal processing massive numbers of data have to be processed in exactly the same way: The algorithms exhibit a high degree of regular parallelism. Various parallel computers have been successfully employed for these kind of applications.

SIMD computers show a great variety in both the power of individual processors and the access paths between processors and memory (see Figure 1). In associative processors (e.g., STARAN) many primitive processing elements are directly connected to their own data; those processing elements that are active in a given cycle all execute the same instruction. Contention is thus minimized at the cost of low utilization. Achieving a reasonable utilization is also problematic for processor arrays such as ILLIAC IV, DAP, PEPE, and the Connection Machine. The most popular of today's supercomputers are pipelined vector processors, such as the CRAY-1S and the CDC 205. These machines attain their speed through a combination of fast technology and strong reliance on pipelining geared toward floating-point vector arithmetic. The performance of vector processors is highly dependent on the algorithms used and especially on the access patterns to data structures. The reason for this is the large discrepancy between the performance of the machine when it is doing what it is designed to do, that is, processing vectors of the right size, and when it is doing something else.

In many areas that have great needs for processing power, the behavior of algorithms is irregular and highly dependent on the input data, making it necessary to perform scheduling at run time. This calls for asynchronous machines in which computations are free to follow their own instruction stream with little interference from other computations. MIMD computers are asynchronous at the level of the machine language:[1] As long as two concurrent computations are independent, no assumptions can be made about their relative timing. Computations are seldom completely independent, however, and at the points where interaction occurs they need to be synchronized by some special mechanism. This synchronization overhead is the price to be paid for the higher utilization allowed by asynchronous operation.

There are different strategies to keep this price to an acceptable level. One is to keep

[1] This does not imply that the organization of the machine is also asynchronous.

the communication between computations to a minimum by dividing the task into large processes that operate mainly on their own private data. Although in such machines scheduling is done at run time, the programmer has to be aware of segmentation, that is, the partitioning of program and data into separate processes. Again the difficulty of this task is highly dependent on the regularity of the algorithm. Assistance from the compiler is feasible, but hardly any work in this area has been reported [Hudak and Goldberg 1985]. Another problem is that processes may have to be suspended, leading to complications such as process swapping and the possibility of deadlock. Examples of such *coarse-grain* parallel computers are the HEP [Smith 1978] and the CM* [Swan et al. 1977].

A different strategy to minimize synchronization overhead is to make communication quick and simple by providing special hardware and coding the program in a special format. Dataflow machines are examples of such *fine-grain* parallel computers. Because communication is quick, the processes can be made very small, about the size of a single instruction in a conventional computer. This makes segmentation trivial and improves scalability since the program is effectively divided into many processes and special hardware determines which of them can execute concurrently. The applications for which fine-grain parallel computers can be expected to be competitive are those with great computing demands that can be formulated with a high average but quite irregular degree of parallelism. However, Dennis et al. [1984] have achieved high utilization (greater than 90 percent) for a regular problem that ran at 20 percent utilization on vector processors.

In dataflow machines scheduling is based on availability of data; this is called *data-driven* execution. In reduction machines scheduling is based on the need for data; this is known as *demand-driven* execution. Demand-driven machines (also known as reduction machines) are currently under extensive study. Various parallel reduction machines have been proposed ranging from coarse-grain to very-fine-grain machines. There are close parallels between dataflow machines and fine-grain reduction machines, but the relative merits of each type remain unclear. Most of the crucial implementation problems are probably shared by both types of machines, but in this paper we do not investigate these parallels. See Treleaven et al. [1982b] and Vegdahl [1984] for a comparative survey.

2. DATAFLOW MACHINE LANGUAGE

Although each dataflow machine has a different machine language, they are all based on the same principles. These shared principles are treated in this section. Because we are concerned with a wide variety of machines, we often have to be somewhat imprecise. More specific information is provided in Section 5, which deals with one particular machine. We start with a description of dataflow programs and the ways in which they differ from conventional programs. Dataflow programs are usually presented in the form of a graph; a short summary of the terminology of dataflow graphs is given. In the rest of this section we show how these graphs can be used to specify a computation.

2.1 Dataflow Programs

In most dataflow machines the programs are stored in an unconventional form called a *dataflow program*. Although a dataflow program does not differ much from a control flow program, it nevertheless calls for a completely different machine organization. Figure 2 serves to illustrate the difference. A control flow program contains two kinds of references: those pointing to instructions and those pointing to data. The first kind indicates control flow, and the second kind organizes data flow. The coordination of data and control flow creates only minor problems in sequential processing (e.g., reference to an uninitialized variable), but becomes a major issue in parallel processing. In particular, when the processors work asynchronously, references to shared memory must be carefully coordinated. Dataflow machines use a different coordination scheme called *data-driven*

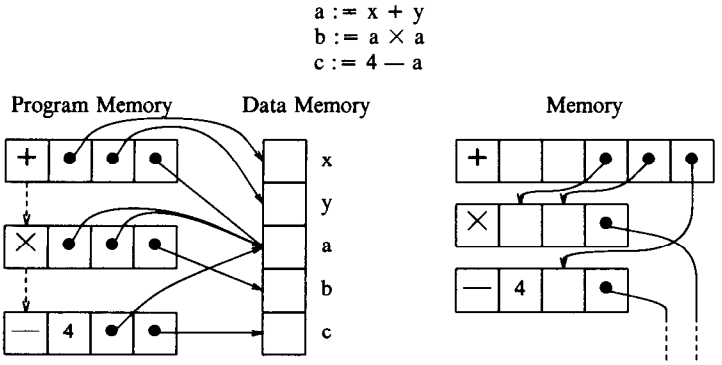

Figure 2. A comparison of control flow and dataflow programs. On the left a control flow program for a computer with memory-to-memory instructions. The arcs point to the locations of data that are to be used or created. Control flow arcs are indicated with dashed arrows; usually most of them are implicit. In the equivalent dataflow program on the right only one memory is involved. Each instruction contains pointers to all instructions that consume its results.

execution: The arrival of a data item serves as the signal that may enable the execution of an instruction, obviating the need for separate control flow arcs.

In dataflow machines each instruction is considered to be a separate process. To facilitate data-driven execution each instruction that produces a value contains pointers to all its consumers. Since an instruction in such a *dataflow program* contains only references to other instructions, it can be viewed as a node in a graph; the dataflow program in Figure 2 is therefore often represented as in Figure 3 (see Section 2.2). In this notation, referred to as a *dataflow graph*, each node with its associated constants and its outgoing arcs corresponds to one instruction.

Because the control flow arcs have been eliminated, the problem of synchronizing data and control flow has disappeared. This is the main reason why dataflow programs are well suited for parallel processing. In a dataflow graph, the arcs between the instructions directly reflect the partial ordering imposed by their data dependencies; instructions between which there is no path can safely be executed concurrently.

2.2 Dataflow Graphs

The prevalent description of dataflow programs as graphs has led to a characteristic and sometimes confusing terminology stemming from Petri net and graph theory. Instructions are known as *nodes*, and instead of data items one talks of *tokens*. A producing node is connected to a consuming node by an *arc*, and the "point" where an arc enters a node is called an *input port*. The execution of an instruction is called the *firing* of a node. This can only occur if the node is *enabled*, which is determined by the *enabling rule*. Usually a *strict* enabling rule is specified, which states that a node is enabled only when each input port contains a token. In the examples in this section all nodes are strict unless noted otherwise. When a node fires, it removes one token from each input port and places at most one token on each of its output arcs. In so-called queued architectures, arcs behave like first-in-first-out (FIFO) queues. In other machines each port acts as a bag: The tokens present at a port can be absorbed in any order.

Figure 3 serves to illustrate these notions. It shows an acyclic graph comprising three nodes, with a token present in each of the two input ports of the PLUS node (marked with the operator "+"). This node is therefore enabled, and it will fire at some unspecified time. Firing involves the removal of the two input tokens, the computation of the result, and the production of three identical tokens on the input ports of the other two nodes. Both of these nodes are then enabled, and they may fire in any

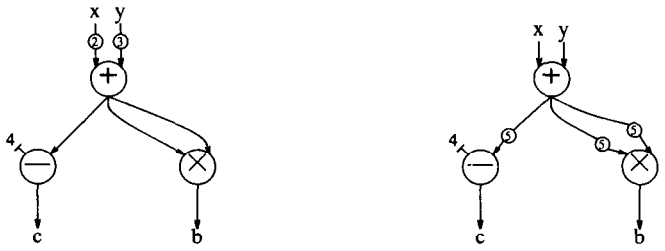

Figure 3. The dataflow program of Figure 2 depicted as a graph. The small circles indicate tokens. The symbol at the left input of the subtraction node indicates a constant input. In the situation depicted on the left the first node is enabled since a token is present on each of its input ports. The graph on the right depicts the situation after the firing of that node.

order or concurrently. Note that, on the average, a node that produces more tokens than it absorbs increases the level of concurrency. All three nodes in this example are *functional*; that is, the value of their output tokens is fully determined by the node descriptions and the values of their input tokens.

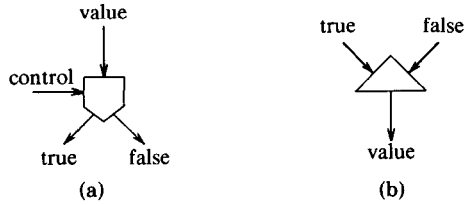

Figure 4. BRANCH and MERGE nodes. (a) A BRANCH node. (b) A nondeterministic MERGE node.

2.3 Conditional Constructs

Conditional execution and repetition require nodes that implement controlled branching. The conditional jump of a control flow program is represented in a dataflow graph by BRANCH nodes. The most common form is the one depicted in Figure 4.

A copy of the token absorbed from the *value* port is placed on the *true* or on the *false* output arc, depending on the value of the control token. Variations of this node with more than two alternative output arcs or with more than one *value* port (compound BRANCH) have also been proposed. As we shall see shortly, the complement of the BRANCH node is also needed. Such a MERGE node does not have a strict enabling rule; that is, not all input ports have to contain a token before the node can fire. In the deterministic variety the value of a control token determines from which of the two input ports a token is absorbed. A copy of the absorbed token is sent to the output arc. The nondeterministic MERGE node (i.e., a MERGE node without control input) is enabled as soon as one of its input ports contains a token; when it fires, it simply copies the token that it receives to its successors. Nonqueued architectures usually do not have MERGE nodes, but allow two arcs to end at the same port. The advantage is that only a strict enabling rule has to be supported.

Figure 5 shows an implementation of a conditional construct. If one token enters at each of the three arcs at the top of the graph, the two BRANCH nodes will each send a token to subgraph *f* or to subgraph *g*. Only the activated subgraph will eventually send a token to the MERGE node. It can easily be shown [Veen 1981] that this graph preserves *safety*; that is, it is safe provided that subgraphs *f* and *g* are safe. A graph is safe if it can be shown that, when presented with at most one token on each input arc, no port will ever contain more than one token. Safety ensures determinate behavior even in the presence of nondeterministic MERGE nodes.

If BRANCH and (nondeterministic) MERGE nodes are used in an improper manner, unsafe graphs can be constructed, in which two tokens may end up at the same port (see Figure 6).

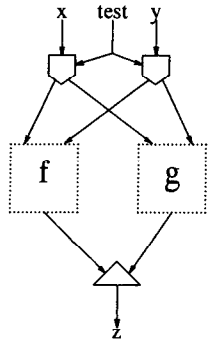

Figure 5. Conditional expression. The graph corresponding to the expression $z :=$ **if** test **then** $f(x, y)$ **else** $g(x, y)$ **fi**. If *test* succeeds, both BRANCH nodes send a token to the left; otherwise, the tokens go to the right. Note the use of nondeterministic MERGE node.

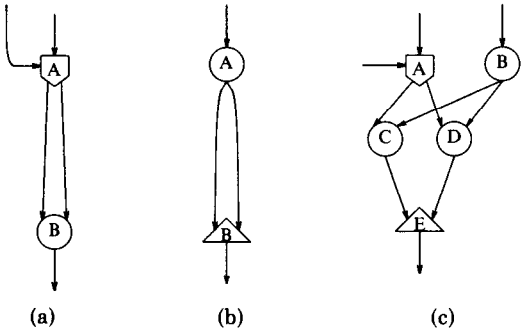

Figure 6. Problems resulting from the improper use of BRANCH and MERGE nodes. All nodes are strict, except the MERGE nodes, and produce tokens on all output arcs when they fire, except the BRANCH nodes. (a) When a pair of tokens arrives at the input ports of node A, the node is enabled, but its firing will not enable node B, since the latter receives only one token on one of its input ports. (b) When a token enters the graph, node A fires and places a token on each of the input ports of the MERGE node. This node then sends two tokens to its output arc. (c) A token will be left behind at an input port of either node C or node D, depending on the value of the control token of the BRANCH node.

2.4 Iterative Constructs and Reentrancy

Figure 7 illustrates problems that may arise when the graph contains a cycle. The simple graph on the left will *deadlock* unless it is possible to initialize the graph with a token on the feedback arc. The node in the graph on the right will never stop firing once started. Although these are not realistic graphs, similar problems may arise in any cyclic graph unless special precautions are taken.

Figure 7. Problems with cyclic graphs. The graph on the left will deadlock; the one on the right will never finish.

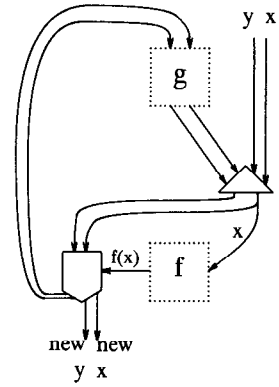

Figure 8. A loop construct according to the lock method. An implementation of the expression **while** $f(x)$ **do** $(x, y) := g(x, y)$ **od**, using the lock method to protect the reentrant subgraphs f and g. The triangular shaped node indicates a compound MERGE node, which functions just like a pair of nondeterministic MERGE nodes. On the left is a compound BRANCH node, which copies its two value inputs either to its left or to its right output arcs, depending on the value of its third input token.

A correct way to implement a loop construct is shown in Figure 8. Note the use of a compound BRANCH node rather than a series of simple BRANCH nodes as in Figure 5. The strict enabling rule of this node ensures that it does not fire before subgraph g has released both its output tokens. If we assume that g is such that no tokens stay behind when all its output tokens are produced, then tokens for the next iteration can be safely sent into the same subgraph. Subgraph g is an example of a *reentrant* graph; its nodes can fire repeatedly. The way reentrancy is handled is a key issue in dataflow architecture. A dataflow graph is

attractive as a machine language for a parallel machine since all nodes that are not data dependent can fire concurrently. In case of reentrancy, however, this maximum concurrency can lead to nondeterminate behavior unless special measures are taken, as we show in the remainder of this section.

A graph in which reentrancy can lead to nondeterminacy is illustrated in Figure 9, where the cycles fox x and y lead through separate MERGE and BRANCH nodes. In the first iteration the first PLUS node calculates the value for x and sends copies to subgraph h and to one of the MERGE nodes. Subgraph h may postpone the absorption of its input token. Meanwhile, the nodes on the cycle for x may fire again, and the PLUS node may send a second token to subgraph h.

The use of the compound BRANCH node in Figure 8 is therefore essential for its safety. We call this method the *lock* method. It is safe and simple, but not very attractive for parallel machines: The level of concurrency is low since the BRANCH node acts as a lock that prevents the initiation of a new iteration before the previous one has been concluded.

An alternative approach is the *acknowledge* method. This can be implemented by adding extra acknowledge arcs from consuming to producing node. These acknowledge arcs ensure that no arc will ever contain more than one token and the graph is therefore safe. One arc provides space for one token. In a manner too complicated to show here, the proper addition of dummy nodes and arcs can transform a reentrant graph into an equivalent one, allowing overlap of consecutive iterations in a pipelined fashion. The acknowledge method therefore allows more concurrency than the lock method, but at the cost of at least doubling the number of arcs and tokens. Through proper analysis, however, a substantial part of these arcs can be eliminated without impairing the safety of the graph [Brock and Montz 1979; Montz 1980].

Both of these methods can also be implemented at the architecture level by modifying the enabling rule. In some machines locking is implemented by specifying that nodes in a reentrant subgraph can only be

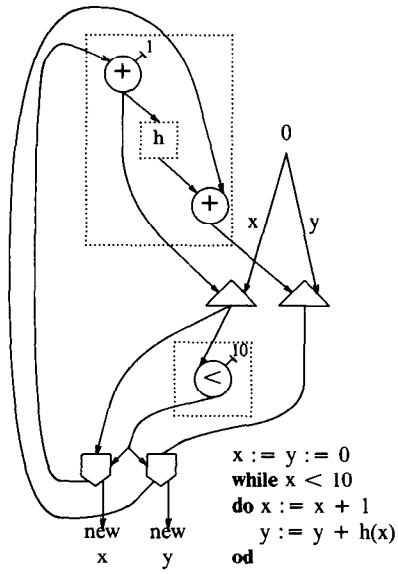

Figure 9. An unsafe way to implement a loop. A new token may arrive at the input of subgraph h before the previous one is absorbed.

enabled a second time after all tokens of a previous activation have left the subgraph [Syre et al. 1977]. The architectures of other machines implement acknowledgment by enabling a node only after all its output arcs are empty [Dennis et al. 1983].

A higher level of concurrency is obtained when each iteration is executed in a separate instance (or copy) of the reentrant subgraph. This *code-copying* method requires a machine with facilities to create a new instance of a subgraph and to direct tokens to the appropriate instance. A potentially more efficient way to implement code copying is to share the node descriptions between the different instances of a graph without confusing tokens that belong to separate instances. This is accomplished by attaching a *tag* to each token that identifies the instance of the node that it is directed to. These so-called *tagged-token* architectures have an enabling rule that states that a node is enabled if each input arc contains a token *with identical tags.* Safety in these machines means that no port ever contains more than one token with the same tag. A tag is sometimes referred to as a *color* or a *label.*

The tagged nature of the architecture

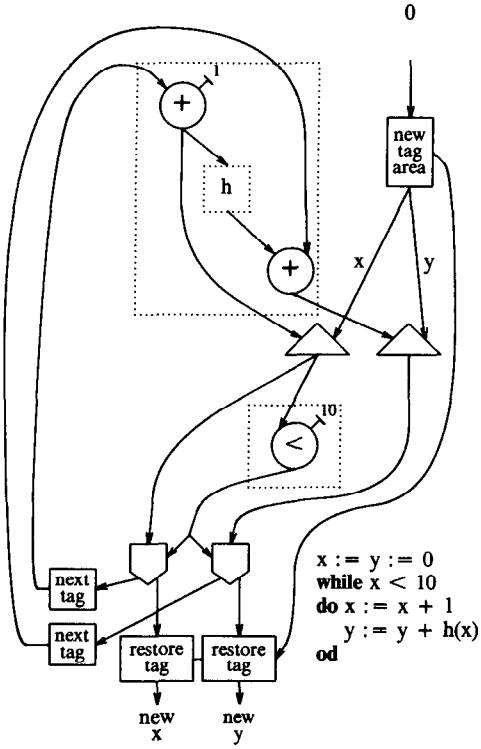

Figure 10. An implementation of a loop using tagged tokens. At the start of the loop a new tag area is allocated. Tokens belonging to consecutive iterations receive consecutive tags within this area. The tag from before the loop is restored on tokens that exit from the loop.

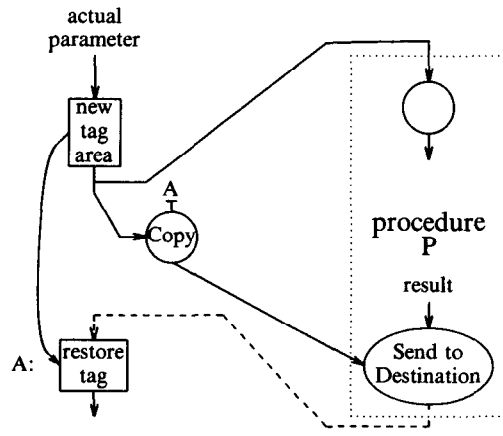

Figure 11. Interface for a procedure call. On the left a call of procedure P whose graph is on the right. P has one parameter and one return value. The actual parameter receives a new tag and is sent to the input node of P and concurrently a token containing address A is sent to the output node. This SEND-TO-DESTINATION node transmits the other input token to a node of which the address is contained in the first token. The effect is that, when the return value of the procedure becomes available, the output node sends the result to node A, which restores the tag belonging to the calling expression.

shows up in the program in the form of nodes that modify tags. Figure 10 shows the implementation of the example in Figure 9 on a tagged-token architecture. The proper execution of nested loops requires that the tags used within a loop be distinct from those in the surrounding expression. A new area in the tag space is therefore allocated at the start of the loop. An efficient implementation of this allocation is not easy, as we shall see in the next section. Within the area tags are ordered; tokens entering the loop receive the first tag, and tokens for consecutive iterations receive consecutively ordered tags within the allocated area. On tokens that exit the loop, the tag corresponding to the surrounding expression is restored. This method can lead to a high level of concurrency because the cycle for x can safely send a whole series of tokens with different tags into subgraph h, with each token initiating a separate and possibly concurrent execution of h.

Machines that handle reentrancy by the lock or acknowledge method are called *static*; those employing code copying or tagged tokens are called *dynamic*. Static machines are much simpler than dynamic machines, but for most algorithms their effective concurrency is lower. Algorithms with a predominantly pipelining type of parallelism, however, execute efficiently on static machines with acknowledging.

2.5 Procedure Invocation

The invocation of a procedure introduces similar problems with reentrancy, to which the methods described above can also be applied. In code-copying architectures a copy of the called procedure is made. In tagged-token architectures a new tag area is allocated for each procedure call so that each invocation executes in its own context. Nested procedure calls, recursion, and coroutines can therefore be implemented without problems. The method is, how-

ever, wasteful of tag space, an important resource, as we shall see below. A good compiler may recognize tail recursion and generate code as efficient as for loops.

An extra facility is required to direct the output tokens of the procedure activation back to the proper calling site. This is usually implemented as shown in Figure 11. The procedure body receives a token that contains a reference to a node at the calling site. This token is then used by the output nodes of the procedure body to direct the return values to the proper places. These output nodes are special nodes capable of sending tokens to nodes to which they have no static arc.

3. THE ARCHITECTURE OF DATAFLOW MACHINES

In this section dataflow machines at the level that directly supports the machine language are described. First, the basic execution mechanism of a processing element and then the overall structure of a dataflow multiprocessor is described.

3.1 A Processing Element

A typical dataflow machine consists of a number of processing elements, which can communicate with each other. Figure 12 shows a functional diagram of one processing element.

The nodes of the dataflow program are often stored in the form of a *template* containing a description of the node and space for input tokens. The node description consists of the operand code (a shorthand for the mapping from input values to output values) and a list of destination addresses (the outgoing arcs). We can think of the movement of a token between two nodes as the progress of a locus of activity. A node that produces more tokens than it consumes increases the number of concurrent activities. Concurrent activities interact at nodes that consume more than one token. Coordination has to take place at these nodes. In dataflow machines coordination therefore amounts to the administration of the enabling rule for those nodes that require more than one input. We call the unit that manages the storage of the tokens the *enabling unit*. It sequentially accepts a token and stores it in memory. If this causes the node to which the token is addressed to become enabled (i.e., each input port contains a token), its input tokens are extracted from memory and, together with a copy of the node, formed into a packet and sent to the functional unit. Such an *executable packet* consists of the values of the input tokens, the operand code, and a list of destinations. The *functional unit* computes the output values and combines them with the destination addresses into tokens. Tokens are sent back to the enabling unit, where they may enable other nodes. Since the enabling and the functional stage work concurrently, this is often referred to as the circular pipeline.

Dividing a processing element into two stages is just one of the possibilities. In some machines the processing elements do not have to be so powerful and they just consist of a memory connected to a unit that handles both token storage and the execution of nodes. In other machines the circular pipeline consists of more concurrent stages, as, for instance, in most machines that use tagged tokens to protect reentrant code. Since, in such a machine, nodes are shared between different instances of a graph, the space in a template to be reserved for storage of input tokens may become arbitrarily large. This makes it impractical to store tokens in the nodes themselves. Token storage is therefore separated from node storage, and the enabling unit is split into two stages: the *matching unit* and the *fetching unit*, usually arranged as shown in Figure 13.

For each token that the matching unit accepts, it has to check whether the addressed node is enabled. In most tagged-token machines this is facilitated by limiting the number of input arcs to two and providing each token with an extra bit that indicates whether the addressed node is monadic or dyadic. Only for dyadic nodes the matching unit has to check whether its memory already contains a matching token, that is, a token with the same destination and tag. Conceptually, the matching unit simply combines destination and tag

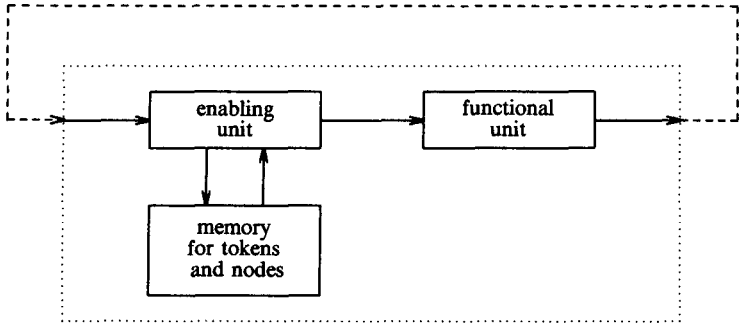

Figure 12. Functional diagram of a processing element. The enabling unit accepts tokens from the left and stores them at the addressed node. If this node is enabled, an executable packet is sent to the functional unit where it is processed. The output tokens, with the destination addresses, are sent back to the enabling unit. Modules dedicated to buffering or communication have been left out of this diagram.

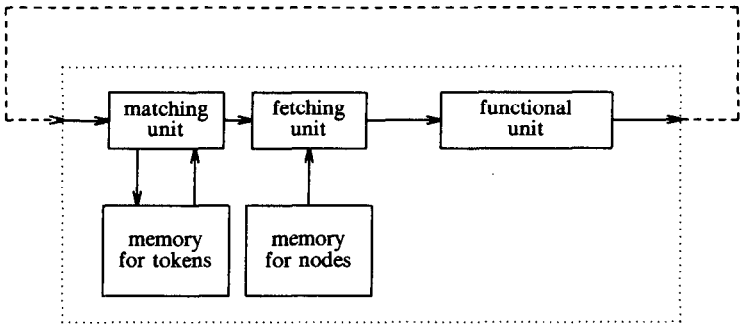

Figure 13. Functional diagram of a processing element of a tagged-token machine. The matching unit stores tokens in its memory and checks whether an instance of the destination node is enabled. This requires a match of both destination address and tag. Tokens are stored in the memory connected to the matching unit. When all tokens for a particular instance of a node have arrived, they are sent to the fetching unit, which combines them with a copy of the node description into an executable packet to be passed on to the functional unit.

into an address and checks whether the location denoted by the address contains a token. The set of locations addressed by tag and destination forms a space that we call the *matching space.* Managing this space and representing it in a physical memory is one of the key problems in tagged-token dataflow architectures.

Although not apparent at first, the problem of matching space management is quite similar to the problems encountered in code-copying machines and in fact involves problems that have plagued parallel architectures from the beginning. At the entrance to a loop, and during procedure invocation, a unique tag area has to be allocated. Guaranteeing uniqueness in a parallel computer is problematic. The fundamental trade-off is between the bottleneck created by a centralized approach and the communication overhead or inefficient use of space offered by a distributed approach. Arvind and Gostelow [1977] proposed an extremely distributed approach in which the uniqueness of a new tag area can be deduced from the existing tag. Since a tag in this scheme effectively encodes the calling stack of a procedure invocation, its

size grows linearly with calling depth. Usually a partly distributed solution is used, amounting to statically distributing the matching space over a set of managers, each of which manages the allocated area locally. An example is a centralized counter per processing element, which, together with a unique identification of the processing element, provides a unique tag [Gurd et al. 1985]. To prevent the local areas from becoming exhausted the matching space must be large and, consequently, at any given time sparsely occupied. Large, sparsely occupied spaces cause several problems. First, addressing an item requires many bits. Second, implementing the space involves a difficult trade-off between storage waste (e.g., a sparsely occupied array) and access time overhead (e.g., a linked list). Hashing techniques offer a compromise. Actual implementations of the approaches just described are few, but it appears that this space or time overhead is a fundamental problem of the fine-grain approach and that a purely fine-grain machine may not be implemented efficiently. In Section 6 we see that the introduction of a manager based on coarse-grain principles may alleviate this problem.

It is interesting to note that the tradeoffs for code-copying machines are virtually identical. When a copy of a subgraph needs to be created, a storage area has to be allocated. A centralized allocator may be space efficient, but may also create a bottleneck. A virtual memory scheme with space allocation can be used, but addresses become large and an efficient mapping to physical memory is needed. Paging techniques that exploit locality in instruction execution may be useful. A good memory manager would avoid these problems, but has the same drawbacks as described above.

3.2 Dataflow Multiprocessors

Figure 14 is a schematic view of the structure of a complete dataflow machine. Although each description of a dataflow machine in the literature seemingly presents a different picture, most designs conform to one of the three structures illustrated.

In a *one-level* dataflow machine (e.g., Arvind and Kathail [1981]), there is only pipeline concurrency within a processing element. Instructions are executed in the processing elements, and the resulting tokens are used in the same processing element or communicated to other processing elements. In some machines one or several processing elements are replaced by *structure units* for the storage and low-level manipulation of data structures.

The two structures illustrated in Figure 14b and c exploit the fact that the processing of executable packets is independent and can be done in any order or concurrently since they contain all the information that the functional unit needs to fire the node and to construct the output tokens. In a *two-level* machine (e.g., Gurd et al. [1985]), each functional unit consists of many *functional elements*, which process executable packets concurrently. Scheduling is trivial: An executable packet is allocated to any idle functional element. By adjusting the number of functional elements, the power of the functional unit can be tuned to that of the rest of the processing element. In a *two-stage* machine (e.g., Dennis and Misunas [1974]), the processing elements are split into two stages, and between the two stages there is an extra communication medium that sends executable packets to functional elements. This two-stage structure is advantageous if the functional stage is heterogeneous, for instance, when some functional elements have specialized capabilities.

3.3 Communication

Figure 14 is merely intended to indicate that there is a way to communicate between different processing elements without suggesting any particular topology. In an actual machine the communication medium can have the structure of a tree, a ring, a binary n-cube, or an equidistant $n \times n$ switch. An even more important difference lies in the nature of the connections that the communication medium provides. Just as there are circuit switching and packet-switching networks, a dataflow machine

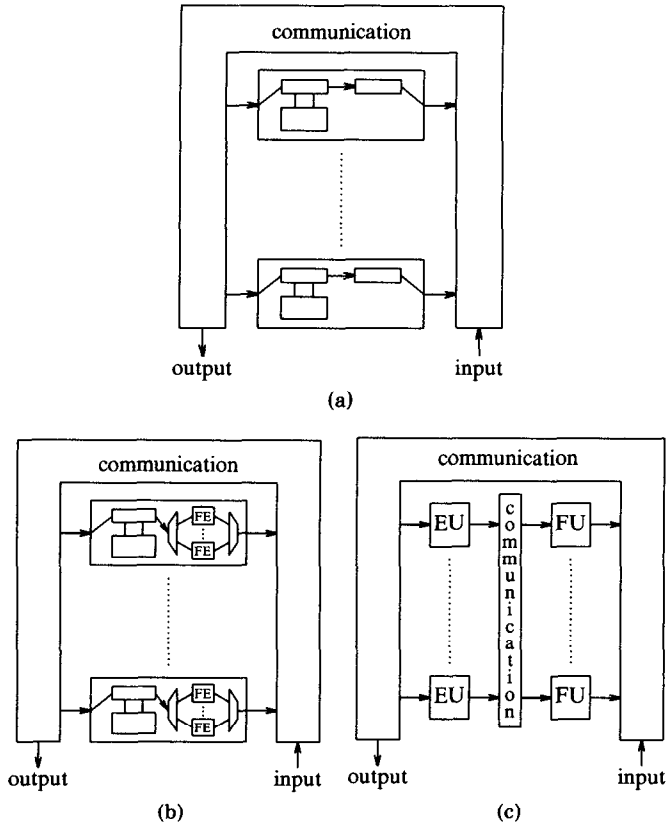

Figure 14. Overall structure of various dataflow multiprocessors. (a) One-level dataflow machine. The structure of each processing element is as shown in Figure 12. Communication facilities deliver tokens that are produced by a functional unit to the enabling unit of the correct processing element, as determined by the destination address and the allocation policy. (b) Two-level dataflow machine. Each functional unit consists of several functional elements (FE), which concurrently process executable packets. (c) Two-stage dataflow machine. Each enabling unit (EU) can send executable packets to each functional unit (FU).

can have a direct communication or a packet communication architecture.

In *direct communication* machines adjacent nodes in the graph are allocated to the same processing element or to processing elements that have a direct connection with each other. An important property of a direct communication architecture is that the communication medium delivers tokens in the same order as they were received. If the communication medium is equipped with queues, unsafe graphs (dataflow graphs in which arcs can contain more than one token) can be executed without impairing determinacy.

Packet communication offers the greatest opportunity for load distribution and parallelism in the communication unit since it can be constructed from asynchronously operating packet-switching modules, with parallelism and redundancy in this critical resource. Such a module can accept a token and forward it to another module, depending on its destination address. In general, such store-and-forward communication units need safeguards to avoid deadlock: Contention may block an essential path. Some machines have redundant communication paths, and consequently the order of packets is not necessarily maintained. On these machines, the arcs of the graph do not necessarily behave as FIFO queues, and determinate execution can only be guaranteed for safe graphs. The best structure for the communication unit and its limitations in size and performance are a matter of debate among dataflow architects. One approach is to have a large number of slow and simple processing elements connected to a high-bandwidth communication unit. A one-level machine structure is usually appropriate for this approach. Other architects claim that as soon as the machine contains more than a few dozen processing elements, insurmountable bottlenecks in the communication unit are created. They

therefore concentrate on the construction of powerful processing elements, which usually involves a two-level design. These architects tend to postpone the design of the higher level until later, and sometimes one processing element is presented as a complete machine [Gurd and Watson 1980]. The performance of one processing element, however, is limited by the inherent bottlenecks in the enabling section.

3.4 Data Structures

In a dataflow graph values flow from one node to another and are, at least at that level of abstraction, not stored in memory. If a value is input to more than one node, a copy is sent to each node. Conceptually, data structures are treated in the same way as other values. In a tagged-token machine with limited token size a complete structure can be sent to a node by packaging each element as a separate token distinguished by subsequent tags. A retrieve operation, for instance, consumes a complete structure and an index and produces a copy of the retrieved element. Directly implementing this concept is known as *structure copying*. Copying is appropriate for small structures. Unfortunately, data structures tend to be large, and implementing these by the conceptually simple structure copying method would place an unacceptable burden on the machine. Many machines therefore have a facility to *store* structures. In such machines an element can be retrieved by sending a request to the unit where the structure has been stored.

The dataflow equivalent of a selective update operation (changing one element of a structure) is an operation that consumes the old structure, the index, and the new value and produces a completely new structure. This involves copying of structures even when they are stored. There are several ways to reduce excessive structure copying. Structures that are not shared do not have to be copied before an update. A reference count mechanism can be used to detect this, and is helpful for garbage collection as well. For shared structures, copying can be further reduced by storing the structure in the form of a tree and copying only the updated node and its ancestors.

Another approach is to provide restrictive access primitives in the programming language. This leads to the concept of *streams*, which are structures that can only be produced and consumed sequentially. These may be processed more efficiently in some machines and increase the effective parallelism because elements of a stream can be consumed before the stream is completed. This increase in parallelism can also be achieved by treating the structures *nonstrictly*, that is, allowing access to elements before the structure has been completely created. Arvind and Thomas [1980] invented this concept and coined the term *I-structures* (for incomplete structures). It requires special hardware to defer fetches of elements that are not yet available.

Gaudiot [1986] gives an excellent comparison of some of the proposed solutions to the structure-handling problem and concludes that its complexity precludes a universal solution. He suggests two additional approaches to avoid storing: defining medium-grain structure operations or defining tag manipulation instructions that exploit the frequent interaction between array indexing on the one hand and either iteration or recursion on the other hand. Tag manipulation has first been proposed by Bowen [1981] and has since been used extensively in Manchester (see, e.g., Böhm and Sargeant [1985] or Veen [1985a]). Its use is restricted to those algorithms in which data structures are consumed completely. With simulation studies on two of those algorithms, Gaudiot and Wei [1986] showed that tag manipulation gave a much better performance than I-structures.

4. A SURVEY OF DATAFLOW MACHINES

Figure 15 and Table 1 illustrate our classification of dataflow machines. The choice of properties used for the classification is limited by the fact that many descriptions (and some designs) are vague and incomplete. In Figure 15 dataflow machines are categorized according to the nature of the communication unit and the architecture of the processing elements. The topology of the communication unit is not used as a criterion, since it does not really help to characterize a dataflow machine and is

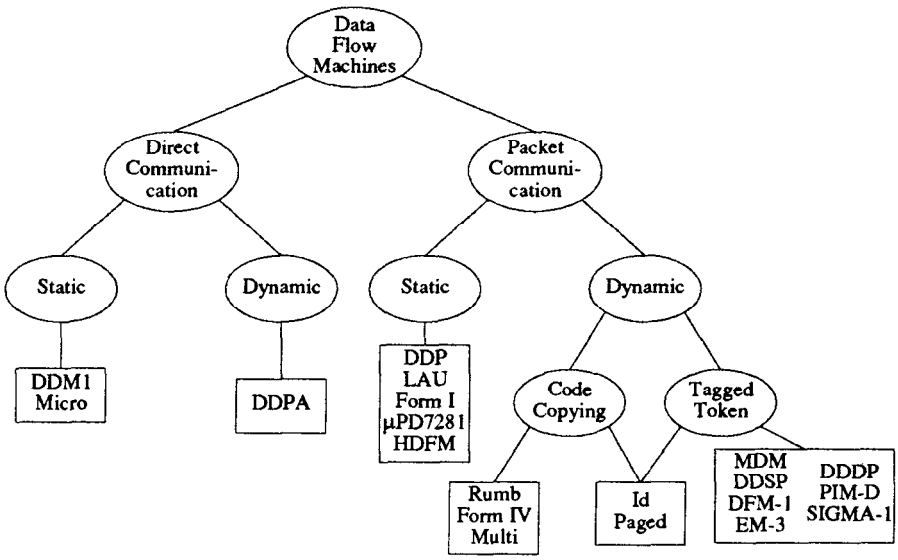

Figure 15. A survey of dataflow machines, categorized according to their architecture and implementation. The keys in the boxes refer to the machines that are summarized in Table 1.

often left unspecified. In the rest of this section all machines appearing in Figure 15 are described separately, using the common terminology established in the previous two sections. A few features of some designs are summarized in Table 2 at the end of this section.

4.1 Direct Communication Machines

The main drawback of direct communication machines is that for many graphs it is difficult to find a good mapping onto the network (*allocation*). It may be a fruitful approach, however, for applications that have predictable and regular communication patterns matching the machine's topology. The most important member of this class is the oldest working dataflow machine, the DDM1 [Davis 1977, 1979]. The processing elements of this machine are arranged as a tree. Allocation is simplified by preserving the hierarchical tree structure of the program. Any internal node of the processing tree can allocate a part of its program (a subtree) to any of its descendants. Allocation is simple and distributed, but far from optimal with respect to even load distribution over the processing elements. The root of the tree forms a bottleneck in the communication between processing elements.

Another less elaborate example is provided by a machine developed in Warsaw, in which the processing elements receive the node descriptions in the form of microprograms [Marczyński and Milewski 1983].

In Japan an interesting dynamic direct communication machine has been developed for large-scale scientific calculations, such as solving partial differential equations [Takahashi and Amamiya 1983]. The processing elements are arranged on a two-dimensional grid and use tags to distinguish tokens belonging to different activations. To avoid the necessity to allocate unique tag areas dynamically, the input language is somewhat restricted (no general recursion) so that static allocation is possible. A hardware simulator, consisting of 4 × 4 processing elements, each connected to eight neighbors, has been used to study small applications. It confirmed analytical predictions that communication delay does not seriously degrade performance, provided that programs have enough parallelism.

Table 1. A Summary of the Dataflow Machines That Are Described in the Text[a]

Key	Machine	Group	Start project	Operational
		Direct Communication Machines		
DDM1	Data-Driven Machine #1	Davis, Burroughs	1972	1976
Micro	Microprogrammed	Marczyński, Warsaw		—
DDPA	Data-Driven Processor Array	Takahashi, Tokyo		1983
		Static Packet Communication Machines		
DDP	Distributed Data Processor	Cornish, Texas Instruments	1976	1978
LAU	LAU System Prototype #0	Syre, Toulouse	1975	1980
Form I	Prototype Basic Dataflow Processor	Dennis, M.I.T.	1971	1982
μPD7281	Dataflow Image Processor	Iwashita, NEC		1984
HDFM	Hughes Data Flow Multiprocessor	Vedder, Hughes	1982	
		Dynamic Packet Communication Machines		
Rumb	Dataflow multiprocessor	Rumbaugh, M.I.T.	1974	—
Form IV	Dynamic dataflow processor	Misunas, M.I.T.	1976	—
Multi	Multiuser dataflow machine	Burkowski, Winnipeg		
Id	Id Machine	Arvind, M.I.T.	1974	1985
Paged	Paged memory dataflow machine	Caluwaerts, Leuven	1979	—
MDM	Manchester Dataflow Machine	Gurd and Watson, Manchester	1976	1981
DDSP	Data-driven signal processor	Hogenauer, ESL	1980	—
DFM-1	List-processing oriented dataflow machine	Amamiya, Tokyo	1980	1983
EM-3	ETL Data-Driven Machine-3	Yuba, ETL		1984
DDDP	Distributed data-driven processor	Kishi, Tokyo		1982
PIM-D	Parallel inference machine based on dataflow	Ito, ICOT		1986
SIGMA-1	Dataflow computer for scientific computations	Hiraki, Ibaraki	1985	

[a] The dates are in most cases estimates and are merely meant as an indication of the relative chronology. Machines without an operational date are paper designs only.

4.2 Static Packet Communication Machines

The first packet communication dataflow machine that became operational is the Distributed Data Processor [Cornish et al. 1979; Johnson et al. 1980], built at Texas Instruments. The references suggest that the DDP uses a locking method to protect reentrant graphs. Although the compiler may create additional copies of a procedure to increase parallelism, this copying occurs statically. It is a one-level machine with a ring-structured communication unit, augmented with a direct feedback link for tokens that stay within the same processing element. A prototype comprising four processing elements has been built.

Around the same time the LAU project in Toulouse, France, designed another static dataflow machine [Comte et al. 1980; Syre 1980; Syre et al. 1977]. LAU stands for *langage à assignation unique* (single assignment language). The group concentrated on the construction of a powerful processing element and left the higher level structure more or less unspecified. In 1980 the LAU system prototype #0, a processing element with 32 functional elements, was completed. Most functional elements are built around a conventional microprocessor. The machine is not programmed by pure dataflow programs as described in Section 2. The program and data memory are separate, and programs are represented as conventional control flow programs, in which control flow arcs have been replaced by additional pointers in data memory to all consuming instructions. This requires a

multiphase communication between functional unit and token memory, and it also complicates the communication with other processing elements. Safety is guaranteed by a hardware-supported locking mechanism. As in the DDP, the programmer can instruct the compiler to create copies of reentrant subgraphs to increase parallelism. The instruction set includes nodes that manage all copies of a subgraph and choose the copy to be used dynamically.

Dennis and his colleagues at the Massachusetts Institute of Technology have been in the vanguard of the dataflow field [Dennis 1974] and produced the first designs for dataflow machines. The earliest design [Dennis and Misunas 1974] had a two-stage structure, with each enabling unit (called an instruction cell) dedicated to one node and with heterogeneous functional units. This design was later extended into a series of machines differing in the way they handled reentrancy and data structures. They ranged from the elementary Form I processor, which was static and could only handle elementary data, to the full-fledged Form IV processor, which had extensive structure facilities and could copy subgraphs on demand (Forms II and III have never been elaborated; Form IV is described below). When it was discovered that an unsafe graph might deadlock the machine and acknowledge arcs had to be introduced, it became clear that it was wasteful to dedicate the processing power needed in one instruction cell just to one instruction. This hardware was therefore shared between a group of nodes and called a cell block. A prototype has been built in which the different parts are emulated by microprogrammable microprocessors [Dennis et al. 1980, 1983]. Since this single unit can emulate both a cell block and a functional unit, the prototype has the single-stage structure of Figure 14a. The prototype that is now operational consists of eight processing elements and an equidistant packet routing network built from 2×2 routing elements.

NEC Electronics has developed the μPD7281 Dataflow Image Processor that may be used as a small processing element in a dataflow machine [Iwashita et al. 1983]. The chip contains memory for 64 instructions and 560 tokens. It has a seven-stage circular pipeline; tokens communicating between instructions that are allocated to the same processing element do not leave the chip. The pipeline contains a mechanism to dynamically regulate the level of parallelism: When the chip is underutilized, preference is given to tokens addressed to instructions that increase parallelism (we come back to this issue of throttling in Section 6). A maximum of 14 processing elements can be connected into an asynchronous packet-switching ring. The ring topology as well as the instruction set are suitable for image processing. The peak performance is reported to be 5 million tokens per second [Jeffery 1985].

At Hughes Aircraft Company another static dataflow architecture for signal processing has been developed [Vedder and Finn 1985]. Its processing elements are arranged on a three-dimensional bussed cube network; the distance between processing elements is at most three. Much attention has been given to static fault tolerance; a faulty processing element can be isolated rapidly. The program graph is statically distributed over the processing elements. Simulation studies showed that a good allocation algorithm could give 30–80 percent better performance than random allocation. Two VLSI chips have been designed that, together with memory chips, constitute a complete processing element. The peak performance is expected to be 2–5 million instructions per second.

4.3 Machines with Code-Copying Facilities

The dataflow machines with potentially the highest level of parallelism are the dynamic dataflow machines; they employ either code copying or tags to protect reentrant graphs. It is characteristic for a code-copying machine that the physical address of a node cannot always be determined statically. The first detailed design of a dataflow machine was of this type [Rumbaugh 1975]. Allocation in this machine is per procedure: All the nodes and intermediate results of one procedure are stored in the memory of one processing element. There is a fast

connection from the output to the input port of a processing element such that a circular pipeline is created. Tokens stay within this pipeline unless they are directed to another procedure, in which case they are routed to a special processing element called the scheduler. This scheduler sends a copy of the called procedure and its input values to an idle processing element. If there is no idle processing element, it waits until a processing element becomes dormant and then saves its state (i.e., all the unprocessed tokens) and declares it idle.

The Massachusetts Institute of Technology *Form IV dataflow processor* is not one machine, but refers to a whole family of designs: There have been a number of articles from the dataflow group at the Massachusetts Institute of Technology, each specifying part of a full-fledged dataflow machine. They are all based on an extension of the basic architecture originally described by Dennis and Misunas [1974], but include special units to store data structures in the form of a tree using hardware-supported reference counts. There have been different proposals for the handling of reentrancy. Misunas [1978] rejected locking and acknowledgment because it limits parallelism and proposed to program the machine without iteration. Procedure bodies would be stored just like data structures, and presumably the invocation of a procedure would result in the storing of a copy of the procedure in the cell blocks. Weng [1979] is more specific about this mechanism. Miranker [1977] suggests a sort of virtual memory for nodes. Translation from virtual to physical address is handled by a relocation box, which manages both the physical and the virtual space. A node is copied into physical memory when it receives its first token. A procedure call generates a unique suffix, which identifies a particular activation. The relocation mechanism ensures that all tokens in that invocation receive the same suffix. This is similar to the tagged-token method. All nodes in a procedure are relocated, not only those that get executed. Code copying is needed because in all machines of this family tokens and nodes are stored together as templates.

A proposal that is surprisingly similar to this is presented by Burkowski [1981]. He produced a detailed hardware design for the static Form I processor, including the acknowledge scheme to protect reentrant graphs, but added memory management facilities, so that the machine can safely be shared between independent tasks. This feature makes it into a dynamic machine, since nodes can be allocated and removed under program control. Although this makes code copying at procedure invocation feasible, no reference to this can be found in the description.

4.4 Machines with Both Tagged-Token and Code-Copying Facilities

Arvind and Gostelow began their study of dataflow languages and architectures at the University of California, Irvine, a decade ago [Arvind and Gostelow 1977]. They designed the language Id (Irvine Dataflow), which introduced many interesting concepts. Independently from similar work in Manchester, they developed the concept of tags (originally known as colors) and showed that it helped to extract more of the parallelism available in a dataflow graph [Arvind and Gostelow 1977]. Simulation studies were also carried out [Gostelow and Thomas .1980]. All data structures are implemented as I-structures. This increases the effective parallelism of a program and facilitates the asynchronous activation of parts of a procedure (i.e., nonstrict procedure call). Arvind and his group, now at the Massachusetts Institute of Technology, constructed a large-scale emulator comprising 32 LISP machines [Arvind and Kathail 1981; Arvind et al. 1980]. Each machine emulates one processing element, and can communicate through a packet-routing network consisting of specially designed switching elements. The physical connections between these switching elements favor a binary n-cube topology, but the network can be programmed to emulate other topologies. Since the paths between processing elements are unequal in length, with the path from a processing element back to itself the shortest, the allocation of nodes and structures can have

a great influence on the performance of the machine. Since elaborate facilities are needed to make this allocation as flexible as possible, allocation of memory and tags is under control of a software manager. An advantage of the combined managing of these two resources is that dynamic tradeoff is possible. The tag space (limited by the maximum size of a tag) is kept small and is used rather densely. When the tag supply is exhausted, new copies of a subgraph are allocated (code copying).

In Leuven, Belgium a machine has been designed with an elaborate memory management scheme [Caluwaerts et al. 1982]. Each processing element has its own memory manager, but these managers can also communicate with each other, so that the total memory space is shared. A procedure call results in the allocation of a fresh memory area for the tokens belonging to the new invocation. A pointer to this area serves as the tag. To facilitate an even load distribution, the area is allocated in a neighboring processing element. Therefore, when a node is enabled, its description must be fetched from another processing element. Caches are used to create local copies. In fact, memory is paged and complete pages are copied. An interesting feature of the memory system is that it treats data structures in the same way as programs, just as in the Form IV processor, and they can be converted into each other. This facilitates the implementation of higher order functions.

4.5 Tagged-Token Machines

The first tagged-token dataflow machine built was the Manchester Dataflow Machine [Gurd and Watson 1980; Watson and Gurd 1982]. This machine is treated in detail in the next section. All machines described in this section are derived from this machine or from Arvind's design.

The data-driven signal processor (DDSP), designed at ESL Inc. [Hogenauer et al. 1982], can accommodate a maximum of 32 processing elements. It is optimized for signal processing using a special allocation algorithm combined with an unorthodox communication topology, which appears to be a combination of a ring and a tree. No hardware was ever built.

In Japan several tagged-token dataflow machines are in various stages of construction. The machine constructed at the Electrical Communication Laboratory of NTT is optimized for list processing [Amamiya et al. 1982, 1986]. It contains separate processing and structure elements. Functional units are integrated with the structure elements as well, since many nodes are expected to operate on structures. The matching unit contains one content addressable memory for each function activation. The references do not indicate how these are allocated. Load balancing is realized by a centralized unit that allocates a function invocation to the processing element with the lowest load level. The design is guided by the primitive operations available in pure LISP; all structures are lists. The central structure operation *cons* is implemented lenient:[2] A pointer token is generated before its arguments are available. This provides the same advantages as other nonstrict structures such as I-structures. A prototype is in operation consisting of two processing and two structure elements.

Nonstrict data structures are also supported by the Electro Technical Data-Driven Machine-3 (EM-3), another LISP-based machine [Yamaguchi et al. 1983]. This nonstrict mechanism is extended to increase the concurrency of a procedure call. At the start of a procedure invocation *pseudoresults* are sent to the consumers of the results of the procedure call. Concurrent with the execution of the procedure body, most nodes will process these pseudoresults just as if they were normal tokens. When a node requires the actual value, its execution is delayed until it becomes available. This mechanism seems to provide the same computational capability as lazy evaluation. A hardware emulator comprising eight processing elements has been constructed, reaching a speed of a few thousand instructions per second. Bottlenecks are now being analyzed [Toda et al. 1985].

[2] See Keller et al. [1979] for the origin of this term.

Table 2. A Comparison of Some Interesting Features of the Most Important Dynamic Machines[a]

Feature	Form IV	Id	Paged	MDM	DFM-1	EM-3	DDDP	PIM-D	SIGMA-1
MS	2S	1L	O	2L	O	1L	1L	2L	1L
Top	E	C	?	E	B	E	B	B	B
Power	L	M	H	H	H	M	M	H	H
Data	St	NS	St	NS	NS	NS	NS	NS	NS
Dyna	C	CT	CT	T	T	T	T	T	T
Alloc	H	M	H	S	H	S	H	S	H

[a] The features are as follows:

MS Machine structure is one level (1L), two level (2L), two stage (2S), or other (O).
Top Topology of communication unit is equidistant (E), bus (B), or cube (C).
Power Computational power per processing element is high (H), medium (M), or low (L).
Data Hardware data structure support for streams (St) or general nonstrict data structures (NS).
Dyna Dynamic mechanism uses code copying (C) and/or tags (T).
Alloc Allocation of data or activity is static (S), hardware supported (H), or by means of a software manager (M).

The distributed data-driven processor built at Systems Laboratory [Kishi et al. 1983] is distinguished by a centralized tag manager. Although this manager may introduce a bottleneck, it uses the tag space rather densely and simplifies the restoration of tags after a procedure invocation. Token matching is by means of a hardware hashing mechanism similar to the one described in the next section. The machine has a dedicated unit for nonstrict structures. A prototype comprising four processing elements communicating through a two-way ring has been constructed. The study of simple hand-coded bench marks revealed that simple allocation results in a reasonable utilization, which can be markedly improved by more sophisticated allocation schemes.

The Institute for New Generation Computer Architecture (ICOT) has stimulated research on the parallel execution of logic programs. One result is the design for a parallel inference machine based on dataflow, with primitives to support nondeterminate merging of streams. Such a feature, or something equivalent, is required for the efficient implementation of unification. Streams are manipulated by separate structure memories implementing I-structures. A distributed mechanism allocates a function invocation on the same processing element, on a neighboring element, or on a distant element, depending on the value of a load factor, which is maintained by periodic exchange of information between processing elements. A prototype has been constructed consisting of four processing elements and three structure memories connected by a two-level bus [Ito et al. 1985, 1986].

A comparison of some features of the most important machines is given in Table 2.

5. THE MANCHESTER DATAFLOW MACHINE

Around 1976 John Gurd and Ian Watson started a research project on dataflow computing at the University of Manchester. They conceived a two-level machine such as that shown in Figure 14b. Since they believe that the construction of an asynchronously operating packet communication network serving more than a few dozen processing elements is not realistic at present, the emphasis of their work has been on constructing a powerful processing element.

This section is a description of this computer in terms of the model presented in Section 3, based on Gurd and Watson [1980], Kirkham [1981], Watson and Gurd [1982], da Silva and Watson [1983], and personal communication. Since Gurd et al. [1985, 1987] contain excellent overviews of the machine, we concentrate on details not

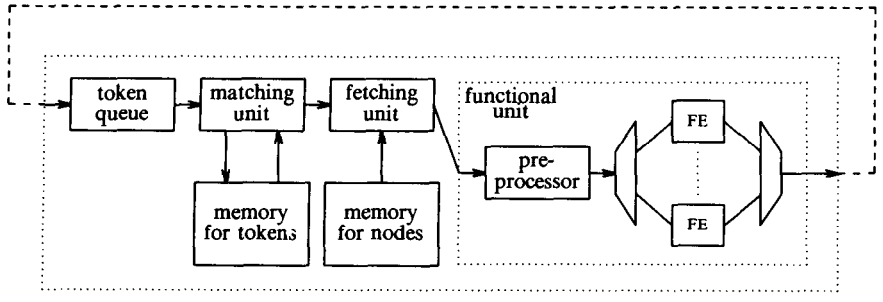

Figure 16. Functional diagram of a processing element in the Manchester Dataflow Machine.

covered there that will be needed for the evaluation in the next section.

5.1 Overview

The group developed the tagged-token concept to increase parallelism for reentrant graphs independently from similar work elsewhere [Arvind and Gostelow 1977]. The structure of their processing element (Figure 16) is similar to that shown in Figure 13. It is a pipeline of four units: token queue, matching unit, fetching unit, and functional unit. Each unit works internally synchronous, but they communicate via asynchronous protocols. More than 30 packets can be processed simultaneously in the various stages of the pipeline. To maximize communication speed the data paths are all parallel (up to 166 bits wide) transmitting a complete packet at a time. Consequently the sizes of packets, and thus of tokens, are fixed.

The *token queue* is a simple FIFO buffer currently accommodating 32K tokens. It serves to smooth the irregular output rates of two other units in the pipeline: the matching unit and the functional unit.

The *matching unit* accepts tokens from the token queue and sends complete sets of input tokens to the fetching unit. Currently it can store 1M tokens. Since in this machine the number of input arcs of a node is limited to two, the destination node is either monadic or dyadic. Each token carries information to distinguish the two cases. In the former case the token is simply passed on to the fetching unit. For dyadic nodes a *match operation* is performed, as described below. A match operation may or may not result in the production of an output packet and this accounts for the variable rate of this unit.

The *fetching unit* combines the set of input tokens with the description of the destination node into an executable packet. The prototype currently accommodates 64K nodes. Each node may contain up to two destination descriptions, each consisting of an address and an indication whether the destination node is monadic or dyadic. A dyadic instruction may be made monadic by replacing one of the destination descriptions by a constant input token for one of the two input ports.

The *functional unit* consists of a preprocessor and a set of functional elements connected via a distributor and an arbiter. The preprocessor executes instructions that require access to a counter memory. Most counters are used to monitor performance. One counter, called the *activation name* counter, is used for the generation of unique tag areas and can be manipulated by the program proper. Although this is not a functional operation, the instruction set is such that this in itself cannot lead to nonfunctional programs. The functional elements are microprogrammed bit-slice processors. The processing time per instruction varies from 3 to 30 microseconds, with an average of 6 microseconds. This variation, combined with the fact that an instruction may produce zero, one, or two tokens, accounts for the irregular rate of the functional unit.

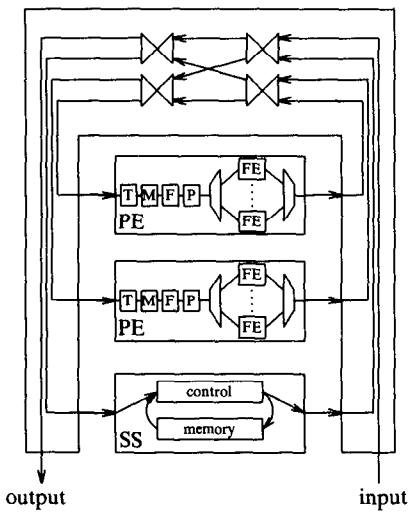

Figure 17. The Manchester Dataflow Machine with two processing elements and one structure store.

Figure 17 illustrates the structure of a multiprocessor with two processing elements and one structure store. In principle, the machine can be expanded to an arbitrary size, but the communication structure becomes impractical when much more than 100 processing elements are employed. The communication unit consists of 2×2 routing elements, each of which may accept a token from one of its input lines and send it to one of its output lines. For $n - 1$ processing elements or structure stores $\log_2 n$ layers of $n/2$ routing elements each are needed. The communication unit is equidistant: The distance between any pair of processing elements is the same as that between the output and input of one processing element. The routing of tokens is determined by the destination address and/or the tag, depending on which allocation strategy is chosen. Since the communication unit has no locality properties that the allocation policy could take advantage of, only an even load distribution over the processing elements has to be ensured. At present a pseudorandom distribution is envisioned, implemented by hashing on both address and tag. Distributing structures over several structure stores is another allocation problem that still needs to be investigated.

5.2 The Match Operation

When a token destined for a dyadic node arrives at the matching unit, it performs a match operation; that is, it searches its memory for a token with the same destination and tag. In a dataflow machine matching implements the synchronization of and the communication between concurrent threads of execution. An efficient implementation is crucial for the performance of the whole machine. The unit can be considered to implement a sparsely occupied virtual memory with the pair ⟨destination, tag⟩ as memory address. The search consists of retrieving the addressed memory cell. If it is empty, the match *fails*. If the cell contains a token destined for the other port, the two tokens are partners and the match *succeeds*. They are combined into a packet and sent to the fetching unit.

The virtual matching memory is occupied so sparsely that it cannot be implemented directly, but has to be mapped onto a physical memory of realistic size. An associative memory could be used, but it was determined that simulating this by means of a hardware-hashing mechanism is more cost effective. The 54-bit matching key (18 bits for the destination and 36 bits for the tag) is hashed to a 16-bit address to access a memory of 64K cells. Each cell has room for one token including destination address, tag, and an extra bit to indicate an empty cell. If the accessed cell is empty, the match fails. If it contains a token, its address, tag, and port are compared with those of the incoming token leading to either success or failure. If the match fails, the incoming token has to be stored at the same address. At present, 16 of these memory banks work in parallel, and so 16 tokens that hash to the same address can be accommodated simultaneously. When a token needs to be stored for which all 16 slots are occupied, it is diverted to the *overflow unit*. The matching unit uses an extra 64K bit memory to indicate which hash keys have overflowed and routes each failing token hashed to an overflowed address to the overflow unit to continue its search for a partner. Other tokens can be processed concurrently since the order in which

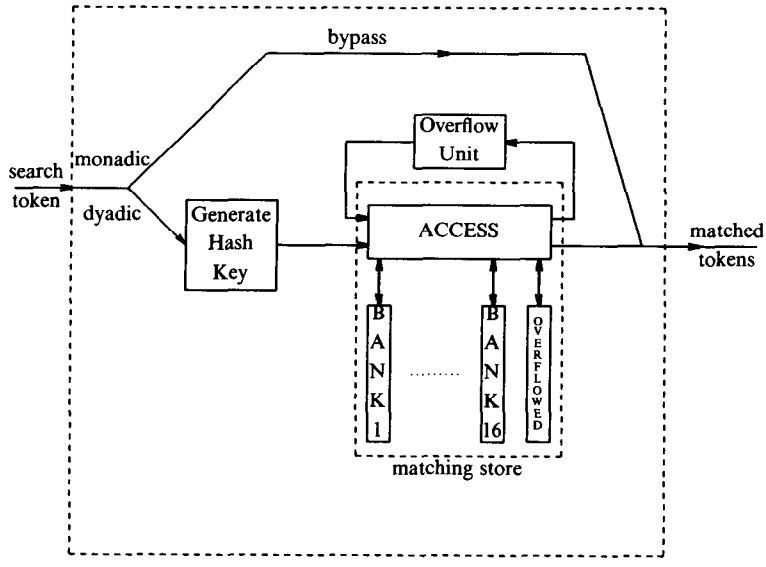

Figure 18. Matching unit. For each token destined for a dyadic node a hash key is generated on the basis of destination address and tag. In the second stage of the pipeline the 16 slots of the memory banks are accessed in parallel. If the partner is present, the match succeeds. If the match fails, the token is stored unless all slots are already occupied. In that case the token is diverted to the overflow unit and an overflow bit is set. A token for which the match fails and the corresponding overflow bit is on is always sent to the overflow unit.

matching occurs does not affect the computation. The matching unit is shown in Figure 18.

A hardware overflow unit is being constructed that maintains a linked list for each overflowed address. For each token that enters the overflow unit the appropriate list is searched sequentially. At present, this mechanism is simulated by the host computer, which makes the processing of overflowed tokens much slower than normal matching. A small fraction of overflowed addresses (less than 1 percent) may therefore have a considerable effect on the overall performance. When such level of overflow is reached, 50 percent of the memory is occupied on the average [Veen 1985b]. It is expected that the real overflow unit will support 10 percent overflow without seriously degrading the average matching speed.

5.3 Tag Space

The tag is divided into the *activation name*, used to separate tag areas, and the *index*, used to distinguish elements of a data structure. Through clever encoding the sizes of these fields are determined at run time although the total tag size is fixed. The fields are not distinguished outside the functional unit, but there are separate instructions to manipulate the different fields.

The activation name space is considered to be an unordered set of unique names. The GENERATE-ACTIVATION-NAME instruction generates a new activation name by causing the preprocessor to increment its activation name counter and prefixing its value by the processing element identifier. Consequently the activation names are unique, but their supply is rather limited. Since tokens of this type may not be converted to any other type, the nonfunctionality of this instruction is harmless. In contrast, the index may be set to an integer and may be subject to arithmetic. Such operations make an efficient implementation of loops possible.

5.4 Data Structures

A data structure can be sent over an arc with each element as a separate token

distinguished by the index field of the tag. The elements can be produced and accessed in any order and concurrently. Retrieving a single element in structure-copying mode (acceptable for small structures) is accomplished by sending all tokens of a structure to a node that transmits the token with the proper index field and discards all other tokens. Many algorithms exhibit a pipeline type of parallelism, calling for implementation with streams, which are produced and consumed in order. Streams are nonstrict; that is, elements can be read before the complete stream is produced. With the aid of special tag manipulation instructions (see, e.g., Bowen [1981]) the interaction between subsequent iterations and subsequent data structure elements can be made quite efficient. Great efficiency improvements have been made using *scatter* instructions, which produce a whole series of tokens with consecutive values or tags within a specified range [Böhm and Sargeant 1985].

Copying large data structures is in general unacceptable. Fixed-size structures can be stored in the *structure store* [Sargeant and Kirkham 1986]. A CREATE-STRUCTURE instruction reserves a memory area large enough to store the complete structure and returns a pointer. The WRITE-ELEMENT instruction stores the value part of a token without tag or destination. The READ-ELEMENT instruction returns the value of an element if it is available; otherwise, the read request is appended to a list of pending requests, which are fulfilled when the element is written. This makes the storage of nonstrict structures possible. Garbage collection is by means of reference counts that are maintained by explicit INCREMENT and DECREMENT instructions. When there are several structure stores, allocation of a data structure is not easy. Small structures (e.g., less than 32 elements) can best be allocated in one structure store and large structures interleaved over all structure stores. This requires a global structure manager.

Before the structure store was installed, the matching unit had special facilities to store data structures. Compared with the old scheme the structure store provides three main advantages. Storing elements without tags or destination is three times more space efficient. It also saves instructions because no tags need to be manipulated when a structure is accessed. The list of pending requests makes efficient support of nonstrict data structures possible.

5.5 State of the Project

The first prototype processing element, which became operational in the fall of 1981, has been subjected to numerous performance studies, and unsatisfactory parts have been improved. For a set of benchmark programs a performance of 1–2 million instructions per second has been reached [Gurd et al. 1985]. Since the prototype is implemented in medium-performance technology, an upgrading to around 10 million instructions per second for one processing element seems feasible. An emulator has been constructed to study the behavior of the communication unit. The emulator consists of 16 pairs of microprocessors, each pair emulating one processing element, connected via a synchronously operating packet switching network [Foley 1985]. In the summer of 1985, a structure store with space for 500K elements has been installed, and later expanded to 1M elements [Kawakami and Gurd 1986]. A fast overflow unit is under construction, which will have a basic speed comparable to that of the matching unit.

6. FEASIBILITY OF DATAFLOW MACHINES

We saw in the previous section that a processing element for a dataflow machine can be constructed with a speed of close to 10 million instructions per second. Since dataflow machines are in principle extensible, a machine consisting of more than 100 processing elements could conceivably reach a speed in the range of 1 billion instructions per second. It is too early to tell whether this potential can indeed be realized; much work needs to be done on allocation schemes, and experience needs be gained with data structure support and networks that connect many processing elements. But even if a machine with such a performance could be constructed, .the

question remains as to whether the amount of hardware needed for such a machine would not be better used by an alternative architecture. In fact, most of the objections raised against the dataflow approach are concerned with factors that are believed to reduce the effective utilization of a dataflow machine to an unacceptably low level. A well-argued case is made by Gajski et al. [1982]. They claim that most programs do not contain enough parallelism to utilize a realistic dataflow machine except when large arrays are processed in parallel. They also claim that the handling of large data structures involves considerable overhead in the form of either excessive storage or excessive processing requirements. With several prototypes operational the validity of such objections can now be judged on the basis of actual experience. In this section this question is addressed with respect to the Manchester Dataflow Machine. Related discussions can be found in Shimada et al. [1986] and Hiraki et al. [1986]. Here we concentrate on underutilization and overhead, treated together as *resource waste*. Roughly speaking, wasted resources are considered to be those that are needed beyond those in a reasonably high-performance sequential computer.

Most of the hardware of the Manchester Dataflow Machine can be classified as being used either for *processing* or for *storage*. We shall only consider the functional elements as processing hardware. Storage consists of data and instruction memories in the token queue, the matching unit, the fetching unit, and the structure store. The rest of the hardware we classify as being used for *communication*. The total resource waste in this machine can be estimated if we know the relative sizes of the three categories and the level of waste within each category. As a rough measure of the amount of hardware we use the number of printed circuit boards, ignoring differences in board and chip density.

A multiprocessor consists of a number of processing elements connected with a communication switch. The amount of hardware in the switch per processing element grows logarithmically with the size of the machine. A machine containing a few dozen processing elements would require per processing element or structure store about two printed circuit boards for the switch alone. One processing element is currently implemented with about 15 printed circuit boards for processing, 22 for storage, and 9 for internal communication. The structure store requires four printed circuit boards for communication and two for storage.

If there are an equal number of processing elements as there are structure stores, about 45 percent of the hardware will be devoted to storage, with the rest equally divided between processing and communication. Half of the communication hardware is needed for the asynchronous communication between units within one processing element. The same architecture can easily be implemented synchronously. Since in that case the communication hardware is relatively small (15 percent), we concentrate on the other two categories.

6.1 Waste of Processing Power

Processing power is wasted either because a functional element is idle or because it is performing overhead computation, that is, computation that would not be needed in a sequential implementation. We treat these two factors in order.

6.1.1 Underutilization of Functional Elements

A functional element is idle because of a poor hardware balance, lack of parallelism in the program, or poor distribution over the processing elements. Balancing the hardware amounts to adjusting the number of functional elements to the speed of the matching unit and providing enough buffering to smooth irregularities. This has been done by analysis and by experiment [Gurd and Watson 1983; Gurd et al. 1985], and it has been concluded that the functional unit should contain between 12 and 20 elements.

In such a configuration there are 30–40 stages in the pipeline that can concurrently be active. The parallelism in a program should thus be at least 30 per processing element to avoid starvation of functional elements and preferably more to accom-

modate the smoothing buffers. Experiments with simple programs run on one processing element indicate that an average parallelism of 50 is sufficient to keep above this minimum. A reasonably sized multiprocessor would therefore need programs with an average rate of parallelism close to 1000. Experience so far suggests that realistic programs can indeed achieve such rates of parallelism, if the programmer carefully avoids sequential constructs. Programs with a regular type of parallelism, for which the average rate of parallelism is close to the maximum rate, do not create problems. Such programs, however, run well or even better on static dataflow machines or on more conventional parallel computers. Programs with irregular parallelism often create excessive storage demands. We come back to this below.

Distributing the work load over the processing elements is in general a complicated allocation problem that needs to take the locality of instruction and data access into account. In the Manchester machine the problem is simplified since all communication paths are of equal length, so that there is no physical locality that the allocator needs to exploit. Simple experiments with both the simulator and the micro-based emulator suggest that a pseudorandom distribution based on similar hashing techniques as those used in the matching store will provide an even distribution of the processing load [Barahona and Gurd 1985; Foley 1985]. Only simple programs have been simulated, however, and structure allocation has been ignored. Experiments with more realistic programs are necessary to substantiate these claims.

6.1.2 Overhead Computation

Even if the functional elements are sufficiently utilized, processing power can still be wasted if many instructions are in fact overhead. One source of this type of overhead mentioned by Gajski et al. [1982] is the distributed nature of flow control. A manifestation of this problem is the separate BRANCH instructions that need to be executed for each data item that enters a conditional expression compared to the single-jump instruction in a control flow computer. Another manifestation is the tag manipulation instruction that is needed for each data item entering a reentrant subgraph. Possibly the largest source of overhead computation is in the handling of large data structures.

For certain numerical programs an indication of the amount of overhead computation is provided by the *floating-point fraction*, that is, the fraction of executed instructions that perform floating point operations.[3] Studies of bench-mark programs run on conventional supercomputers at Lawrence Livermore National Laboratory showed that assembly language programmers achieve a floating-point fraction of 30 percent, whereas FORTRAN compilers reach 15–20 percent [Gurd et al. 1985]. Straightforward compilers for the Manchester Dataflow Machine achieve a floating-point fraction of 3 percent for large programs. There is, however, much room for optimization, and a good compiler can reduce this overhead considerably. Recent work on optimization in Manchester has achieved floating-point fractions of 15 percent for realistic programs [Böhm and Sargeant 1985]. Shimada et al. [1986] are working in the same direction.

6.2 Waste of Storage Space

An even distribution of the work over a multiprocessor is greatly simplified if each instruction is available on each processing element. Because of all the copies of the program, most instruction storage would be wasted. This waste is, however, insignificant compared with the waste in data storage.

The processing element that is currently operating contains an enormous amount of memory, practically all of it situated in the matching unit and the structure store. The total hardware cost of the machine is dominated by the cost of this 20-Mbyte high-speed data memory.

The structure store needs 5 Mbyte of memory because most programs with

[3] The Manchester Dataflow group calls the inverse of this figure the MIPS/MFLOPS ratio.

sufficient parallelism operate on large data structures, and sometimes several copies of these need to be maintained concurrently. It seems possible that with compiling techniques that analyze access patterns to structures this duplication could be reduced substantially without impairing parallelism. Experience with compiling for vector processors may be helpful in this.

The 15-Mbyte of memory in the matching unit is barely sufficient to run realistic programs. One reason for this is that its effective utilization is less than 20 percent [Veen 1985b]. This results from a combination of two factors:

- Each token carries a destination and a tag in addition to its data. Two-thirds of each cell is thus dedicated to overhead.
- The occupancy needs to be limited to less than 50 percent to avoid serious performance degradation due to overflow.

When the hardware overflow unit is installed, the effective utilization may rise to 25 percent.

The second reason for the size of the matching unit is that programs with sufficient irregular parallelism occasionally flood the matching unit with intermediate results. It has become clear that a mechanism is needed to limit parallelism if resources tend to get overloaded. Such a dynamic mechanism has been called a *throttle*. In the NEC dataflow chip (see Section 4.2) we saw a fine-grain throttle: Tokens are classified in different categories, depending on the effect they are expected to have on the level of parallelism, and the token queue favors a particular token category, depending on the machine load. A suggestion for such a mechanism also appears in Veen [1980]. An effective classification would need assistance from the compiler.

Currently a coarse-grain throttle is under investigation in Manchester that manages procedure and loop bodies by controlling the generation of activation names. The execution of a GENERATE-ACTIVATION-NAME instruction can be seen as the initiation of a new process. The throttle would allow only a limited number of processes to be active concurrently and may suspend a process until a previous one has terminated. In order to avoid deadlock the throttle has to maintain an explicit representation of the dependencies between processes. The mechanism seems complex but feasible, and simulation experiments have been very promising.

A consequence of this type of throttling is that the matching space can be much smaller because activation names can be reused. The matching space is treated as a critical resource managed by the throttle, and its consumption is reduced by enforcing locality. With assistance from the compiler, the index field of the tag can also be reduced. A tag of half its current size appears sufficient. There is also some redundancy in the matching space addressing that can be removed. A combination of these improvements affects the utilization of the matching unit in two ways: The overhead per cell may be reduced to 30 percent and, because of greater locality, a more efficient hashing function can be used. In this way a utilization of 40 percent can be achieved [Veen 1985b].

7. SUMMARY

The concept of dataflow architecture is some 15 years old, and the first dataflow machine became operational more than a decade ago. In the last five years considerable progress has been made. Several designs have been emulated in hardware or have reached the prototype stage. They all have a communication mechanism based on packet switching, and most of them support general recursion. Recursion mechanisms are based on tagged tokens, sometimes combined with code copying. Whether such a combination is advantageous is not yet clear. The handling of data structures is an area of active investigation. The dataflow model calls for separate copies of each value to be sent from the producer to each consumer. It is clear that such copying is not feasible when large data structures are involved and that data structures have to be stored. Since this storage is visible in the dataflow graph, it constitutes a deviation from the pure functional

approach. There are signs that a deviation is also necessary from the fine-grain approach. Mechanisms have been proposed and implemented that schedule on coarse- or medium-grain levels such as procedure bodies. Scheduling occurs in a load balancer and in a throttle, that is, a mechanism to limit parallelism when resources tend to get overloaded.

An objection frequently raised against the dataflow approach is that its fine granularity leads to an excessive consumption of resources. Experience with the Manchester Dataflow Machine, the most advanced prototype to date, shows that the major waste of resources occurs in data storage. The memory needed to store structures can probably be reduced by compiling techniques. More serious is the consumption of matching unit memory. An effective throttle could alleviate this problem. Besides its primary goal, reducing the number of tokens that have to be stored concurrently, it has two additional effects: The size of the tag can be reduced and the matching space is used more densely, which makes a more efficient representation possible.

The second area of concern is the waste of processing power due to underutilization of functional elements. One source of this may be an insufficient level of parallelism in the program. It has been shown that with careful programming and sophisticated compilers a high level of parallelism can be sustained even for realistic programs. It remains to be seen whether the required reprogramming effort stays within reasonable bounds for a wide range of large programs. Underutilization may also occur if processing or communication load is not evenly balanced. Simple experiments with load balancing have shown promising results, but conclusive experiments await the construction of multiprocessors of sufficient power.

It has been assumed in the past that fine granularity would require a great proportion of overhead instructions, which would also waste processing power. It has been shown recently that sophisticated compilers can reduce this overhead to an acceptable level.

The crucial questions are concerned with handling of data structures, load balancing, and control of parallelism. They all require study of the execution of large programs for which simulation or analysis is difficult. Prototypes of sufficient power that are, or soon will be, available together with their supporting software provide excellent test beds for further research in these areas.

ACKNOWLEDGMENTS

Wim Böhm, John Gurd, Jan Heering, Paul Klint, Martin Rem, and Marleen Sint read earlier versions of this paper. I have benefited greatly from their comments. The work was supported by the Centre for Mathematics and Computer Science and the Dutch Parallel Reduction Machine project.

REFERENCES

ALLAN, S. J., AND OLDEHOEFT, A. E. 1980. A flow analysis procedure for the translation of high-level languages to a data flow language. *IEEE Trans. Comput. C-29*, 9 (Sept.) 826–831.

AMAMIYA, M., HASEGAWA, R., NAKAMURA, O., AND MIKAMI, H. 1982. A list-processing-oriented data flow machine architecture. In *Proceedings of the AFIPS National Computer Conference 82*. AFIPS Press, Reston, Va., pp. 143–151.

AMAMIYA, M., TAKESUE, M., HASEGAWA, R., AND MIKAMI, H. 1986. Implementation and evaluation of a list-processing-oriented data flow machine. In *Proceedings of the 13th Annual Symposium on Computer Architecture* (Tokyo, June 2–5). ACM, New York, pp. 10–19.

ARVIND AND GOSTELOW, K. P. 1977. A computer capable of exchanging processors for time. In *Information Processing 77*, B. Gilchrist, Ed. North-Holland, New York, pp. 849–853.

ARVIND AND KATHAIL, V. 1981. A multiple processor dataflow machine that supports generalized procedures. In *Proceedings of the 8th Annual Symposium on Computer Architecture* (Minneapolis, Minn., May 12–14). ACM, New York, pp. 291–302.

ARVIND AND THOMAS, R. E. 1980. I-Structures: An efficient data type for functional languages. Tech. Memo. 210, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass.

ARVIND, KATHAIL, V., AND PINGALI, K. 1980. A dataflow architecture with tagged tokens. Tech. Memo. 174, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass.

BARAHONA, P., AND GURD, J. R. 1985. Simulated performance of the Manchester multi-ring dataflow machine. In *Parallel Computing 85*. North-Holland, Amsterdam, pp. 419–424.

BÖHM, A. P. W., AND SARGEANT, J. 1985. Efficient dataflow code generation. In *Parallel Computing 85*. North-Holland, Amsterdam, pp. 339–344.

BOWEN, D. L. 1981. Implementation of data structures on a data flow computer. Ph.D. dissertation, Dept. of Computer Science, Victoria Univ. of Manchester, Manchester, England.

BROCK, J. D., AND MONTZ, L. B. 1979. Translation and optimization of data flow programs. CSG Memo. 181, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass.

BURKOWSKI, F. J. 1981. A multi-user data flow architecture. In *Proceedings of the 8th Annual Symposium on Computer Architecture* (Minneapolis, Minn., May 12–14). ACM, New York, pp. 327–340.

CALUWAERTS, L. J., DEBACKER, J., AND PEPERSTRAETE, J.A. 1982. Implementing code reentrancy in functional programs on a dataflow computer system with a paged memory. In *The International Workshop on High-Level Language Computer Architecture* (Fort Lauderdale, Fla., Dec. 1–3).

COMTE, D., HIFDI, N., AND SYRE, J. C. 1980. The data driven LAU multiprocessor system: Results and perspectives. In *Proceedings of the IFIP Congress 80* (Tokyo and Melbourne, Australia, Oct. 6–17), S. Lavington, Ed. North-Holland, Amsterdam, pp. 175–180.

CORNISH, M., ET AL. 1979. The TI data flow architectures: The power of concurrency for avionics. In *Proceedings of the 3rd Conference on Digital Avionics Systems* (Fort Worth, Tex., Nov.). IEEE, New York, pp. 19–25.

DARLINGTON, J., HENDERSON, P., AND TURNER, D. A. 1982. *Functional Programming and its Applications*. Cambridge University Press, Cambridge, England.

DA SILVA, J. G. D., AND WATSON, I. 1983. Pseudo-associative store with hardware hashing. *IEEE Proceedings, Pt. E 130*, 1, 19–24.

DAVIS, A. L. 1977. Architecture of DDM1: A recursively structured data driven machine. Tech. Rep., Dept. of Computer Science, Univ. of Utah, Salt Lake City, Utah.

DAVIS, A. L. 1979. A data flow evaluation system based on the concept of recursive locality. In *Proceedings of the National Computing Conference*. AFIPS Press, Reston, Va., pp. 1079–1086.

DENNIS, J. B. 1969. Programming generality, parallelism and computer architecture. *Information Processing 68* (Amsterdam). North-Holland, Amsterdam, pp. 484–492.

DENNIS, J. B. 1974. First version of a data flow procedure language. In *Lecture Notes in Computer Science*, vol. 19, G. Goos and J. Hartmanis, Eds. Springer, New York, pp. 362–376.

DENNIS, J. B. 1980. Data flow supercomputers. *Computer 13*, 4 (Nov.), 48–56.

DENNIS, J. B., AND MISUNAS, R. P. 1974. A preliminary architecture for a basic data flow processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture* (Houston, Tex., Jan. 20–22). *Comput. Archit. News 3*, 4, 126–132.

DENNIS, J. B., BOUGHTON, G. A., AND LEUNG, C. K. C. 1980. Building blocks for data flow prototypes. In *Proceedings of the 7th Annual Symposium on Computer Architecture* (La Baule, France, May 6–8). ACM, New York, pp. 1–8.

DENNIS, J. B., LIM, W. Y. P., AND ACKERMAN, W. B. 1983. The MIT data flow engineering model. In *Proceedings of the IFIP 9th World Computer Congress* (Paris, Sept. 19–23), R. E. A. Mason, Ed. North-Holland, New York, pp. 553–560.

DENNIS, J. B., GAO, G. R., AND TODD, K. W. 1984. Modeling the weather with a data flow supercomputer. *IEEE Trans. Comput. C-33*, 7 (July), 592–603.

FLYNN, M. J., 1972. Some computer organizations and their effectiveness. *IEEE Trans. Comput. C-21*, 9 (Sept.), 948–960.

FOLEY, J. 1985. A hardware simulator for a multiring dataflow machine. Ph.D. dissertation, Dept. of Computer Science, Victoria Univ. of Manchester, Manchester, England.

GAJSKI, D. D., PADUA, D. A., KUCK, D. J., AND KUHN, R. H. 1982. A second opinion on data flow machines and languages. *Computer 15*, 2 (Feb.), 58–69.

GAUDIOT, J. L. 1986. Structure handling in dataflow systems. *IEEE Trans. Comput. C-35*, 6 (June), 489–502.

GAUDIOT, J. L., AND WEI, Y. H. 1986. Token relabeling in a tagged data-flow architecture. In *Proceedings of the 1986 International Conference on Parallel Processing* (St. Charles, Aug. 19–22). IEEE, New York, pp. 592–599.

GLAUERT, J. R. W. 1984. High level dataflow programming. In *Distributed Computing*, F. B. Chambers, D. A. Duce and G. P. Jones, Eds. Academic Press, Orlando, Fla., pp. 43–53.

GOSTELOW, K. P., AND THOMAS, R. E. 1980. Performance of a simulated dataflow computer. *IEEE Trans. Comput. C-29*, 10 (Oct.), 905–919.

GURD, J., AND BÖHM, A. P.W. 1987. Implicit parallel processing: SISAL on the manchester dataflow computer. In *Proceedings of the IBM-Europe Institute on Parallel Processing* (Oberlech, Aug.), G. Amalsi, R. W. Hockney, and G. Paul, Eds. North-Holland, Amsterdam.

GURD, J., AND WATSON, I. 1980. A data driven system for high speed parallel computing. *Comput. Design 9*, 6 and 7 (June), 91–100 and 97–106.

GURD, J., AND WATSON, I. 1983. Preliminary evaluation of a prototype dataflow computer. In *Proceedings of the 9th IFIP World Computer Congress* (Paris, Sept. 19–23), R. Mason, Ed. North-Holland, Amsterdam, pp. 545–551.

GURD, J. R., KIRKHAM, C. C., AND WATSON, I. 1985. The Manchester prototype dataflow computer. *Commun. ACM 28*, 1 (Jan.), 34–52.

GURD, J., KIRKHAM, C. C., AND BÖHM, A. P. W. 1987. The Manchester dataflow computing sys-

tem. In *Experimental Parallel Computing Architecture*, J. Dongarra, Ed. North-Holland, Amsterdam, pp. 177–219.

HARTIMO, I., KVONLÖF, K., SIMULA, O., AND SKYTTA, J. 1986. DFSP: A data flow signal processor. *IEEE Trans. Comput. C-35*, 1 (June), 23–33.

HAZRA, A. 1982. A description method and a classification scheme for data flow architectures. In *Proceedings of the 3rd International Conference on Distributed Computing Systems* (Oct.). IEEE, New York, pp. 645–651.

HIRAKI, K., NISHIDA, K., SEKIGUCHI, S., AND SHIMADA, T. 1986. Maintenance architecture and its LSI implementation of a dataflow computer with a large number of processors. In *Proceedings of the International Conference on Parallel Processing*. IEEE, New York, pp. 584–591.

HOGENAUER, E. B., NEWBOLD, R. F., AND INN, Y. J. 1982. DDSP—A data flow computer for signal processing. In *Proceedings of the International Conference on Parallel Processing*. IEEE, New York, pp. 126–133.

HUDAK, P., AND GOLDBERG, B. 1985. Distributed execution of functional programs using serial combinators. *IEEE Trans. Comput. C-34*, 10 (Oct.), 881–891.

ITO, N., KISHI, M., KUNO, E., AND ROKUSAWA, K. 1985. The dataflow-based parallel inference machine to support two basic languages in KL-1. In *IFIP TC-10 Working Conference on Fifth Generation Computer Architecture* (Manchester, England, July 15–18). J. V. Woods, Ed. Elsevier, New York, pp. 123–145.

ITO, N., SATO, M., KUNO, E., AND ROKUSAWA, K. 1986. The architecture and preliminary evaluation results of the experimental parallel inference machine PIM-D. In *Proceedings of the 13th Annual Symposium on Computer Architecture* (Tokyo, June 2–5). ACM, New York, pp. 149–156.

IWASHITA, M., TEMMA, T., MATSUMOTO, K., AND KUROKAWA, H. 1983. Modular dataflow image processor. In *Spring 83 COMPCON*. IEEE, New York, pp. 464–467.

JEFFERY, T. 1985. The μPD7281 Processor. *Byte* (Nov.) 237–246.

JOHNSON, D., ET AL. 1980. Automatic partitioning of programs in multiprocessor systems. In *Proceedings of the Spring 80 COMPCON*. IEEE, New York.

KARP, R. M., AND MILLER, R. E. 1966. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM J. Appl. Math.* 14, 6 (Nov.), 1390–1411.

KAWAKAMI, K., AND GURD, J. R. 1986. A scalable dataflow structure store. In *Proceedings of the 13th Annual Symposium on Computer Architecture* (Tokyo, June 2–5). ACM, New York, pp. 243–250.

KELLER, R. M., LINDSTROM, G., AND PATIL, S. 1979. A loosely-coupled applicative multiprocessing system. In *Proceedings of the National Computer Conference* (New York, June 4–7). AFIPS Press, Reston, Va., pp. 613–622.

KIRKHAM, C. C. 1981. Basic programming manual of the Manchester prototype dataflow system, 2nd ed. Dataflow Research Group, Manchester Univ., Manchester, England.

KISHI, M., YASUHARA, H., AND KAWAMURA, Y. 1983. DDDP: A distributed data driven processor. In *Proceedings of the 10th Annual Symposium on Computer Architecture* (Stockholm, June 13–17). IEEE, New York, pp. 236–242.

LECOUFFE, M. P. 1979. MAUD: A dynamic single-assignment system. *Comput. Digital Tech. 2*, 2 (Apr.), 75–79.

MARCZYŃSKI, R. W., AND MILEWSKI, J. 1983. A data driven system based on a microprogrammed processor module. In *Proceedings of the 10th Annual Symposium on Computer Architecture* (Stockholm, June 13–17). IEEE, New York, pp. 98–106.

MCGRAW, J. R. 1982. The VAL language: Description and analysis. *Trans. Program. Lang. Syst. 4*, 1 (Jan.), 44–82.

MIRANKER, G. S. 1977. Implementation of procedures on a class of data flow processors. In *Proceedings of the 1977 International Conference on Parallel Processing* (Detroit, Mich., Aug. 23–26), J. L. Baer, Ed. IEEE, New York, pp. 77–86.

MISUNAS, D. P. 1978. A computer architecture for data flow computation. Tech. Memo. 100, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass.

MONTZ, L. B. 1980. Safety and optimization transformations for data flow programs. Tech. Rep. 240, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass.

PEYTON-JONES, S. L. 1984. Directions in functional programming research. In *Distributed Computing Systems Programme*, D. A. Duce, Ed. Peter Peregrinus, London, pp. 220–249.

PREISS, B. R., AND HAMACHER, V. C. 1985. Dataflow on a queue machine. In *Proceedings of the 12th Annual Symposium on Computer Architecture* (Boston, Mass., June 17–19). IEEE, New York, pp. 342–351.

RODRIGUEZ, J. E. 1969. A graph model for parallel computation. Tech. Rep. 64, Project MAC, Massachusetts Institute of Technology, Cambridge, Mass.

RUMBAUGH, J. 1975. A data flow multiprocessor. In *Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing* (Sagamore, N.Y.), pp. 220–223.

SARGEANT, J., AND KIRKHAM, C. C. 1986. Stored data structures on the Manchester dataflow machine. In *Proceedings of the 13th Annual Symposium on Computer Architecture* (Tokyo, June 2–5). ACM, New York, pp. 235–242.

SHIMADA, T., HIRAKI, K., NISHIDA, K., AND SEKIGUCHI, S. 1986. Evaluation of a prototype data flow processor of the SIGMA-1 for scientific com-

putations. In *Proceedings of the 13th Annual Symposium on Computer Architecture* (Tokyo, June 2–5). ACM, New York, pp. 226–234.

SMITH, B. J. 1978. A pipelined shared resource MIMD computer. In *Proceedings of the 1978 International Conference on Parallel Processing.* IEEE, New York.

SRINI, V. P. 1986. An architectural comparison of dataflow systems. *Computer 19*, 9 (Mar.), 68–88.

SWAN, R. J., FULLER, S. H., AND SIEWIOREK, D. P. 1977. Cm*—A modular, multi-microprocessor. In *Proceedings of the National Computer Conference* (Dallas, Tex., June 13–16). AFIPS Press, Arlington, Va., pp. 637–644.

SYRE, J. C. 1980. Êtude et realisation d'un systeme multiprocesseur MIMD en assignation unique. Thèse, Univ. Paul Sabartier de Toulouse, Toulouse, France.

SYRE, J. C., COMTE, D., AND NIFDI, N. 1977. Pipelining, parallelism and asynchronism in the LAU system. In *Proceedings of the 1977 International Conference on Parallel Processing* (Detroit, Mich., Aug. 23–26), J. L. Baer, Ed. IEEE, New York, pp. 87–92.

TAKAHASHI, N., AND AMAMIYA, M. 1983. A data flow processor array system: Design and analysis. In *Proceedings of the 10th Annual Symposium on Computer Architecture* (Stockholm, June 13–17). IEEE, New York, pp. 243–250.

TODA, K., YAMAGUCHI, Y., UCHIBORI, Y., AND YUBA, T. 1985. Preliminary measurements of the ETL Lisp-Based data-driven machine. In *IFIP TC-10 Working Conference on Fifth Generation Computer Architecture* (Manchester, England, July 15–18), J. V. Woods, Ed. Elsevier, New York, pp. 235–253.

TRELEAVEN, P. C., HOPKINS, R. P., AND RAUTENBACH, P. W. 1982a. Combining data flow and control flow computing. *Comput. J. 25*, 2 (Feb.), 207–217.

TRELEAVEN, P. C., BROWNBRIDGE, D. R., AND HOPKINS, R. P. 1982b. Data-driven and demand-driven computer architecture. *Comput. Surv. 14*, 1 (Mar.), 93–143.

VEDDER, R., AND FINN, D. 1985. The Hughes data flow multiprocessor: Architecture for efficient signal and data processing. In *Proceedings of the 12th Annual Symposium on Computer Architecture* (Boston, Mass., June 17–19). IEEE, New York, pp. 324–332.

VEEN, A. H. 1980. Data flow computers. In *Colloquium Hogere Programmeertalen en Computerarchitectuur–Syllabus 45*, P. Klint, Ed. Centre for Mathematics and Computer Science, Amsterdam, pp. 99–132 (in Dutch).

VEEN, A. H. 1981. A formal model for data flow programs with token coloring. Tech. Rep. IW 179, Centre for Mathematics and Computer Science, Amsterdam.

VEEN, A. H. 1985a. The misconstrued semicolon—Reconciling imperative languages and dataflow machines. Tract 26, Centre for Mathematics and Computer Science, Amsterdam.

VEEN, A. H. 1985b. Locality in the matching store. Internal report. Dept. of Computer Science, Victoria Univ. of Manchester, Manchester, England.

VEGDAHL, S. R. 1984. A survey of proposed architectures for the execution of functional languages. *IEEE Trans. Comput. C-33*, 12 (Dec.), 1050–1071.

WATSON, I., AND GURD, J. 1982. A practical data flow computer. *Computer 15*, 2 (Feb.) 51–57.

WENG, K. S. 1979. An abstract implementation for a generalized data flow language. Tech. Rep. 228, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass.

YAMAGUCHI, Y., TODA, K., AND YUBA, T. 1983. A performance evaluation of a LISP-based data-driven machine (EM-3). In *Proceedings of the 10th Annual Symposium on Computer Architecture* (Stockholm, June 13–17). IEEE, New York, pp. 363–369.

Received November 1985; final revision accepted March 1987.