ISSUES IN DATAFLOW COMPUTING

Ben Lee* and A. R. Hurson**

*Oregon State University
Department of Electrical and Computer Engineering
Corvallis, Oregon 97331-3211

**The Pennsylvania State University
Department of Electrical and Computer Engineering
University Park, PA 16802
E-mail: A2H@ecl.psu.edu
Phone: (814) 863-1167

Abstract

Early advocates of dataflow computers predicted that the characteristics of the dataflow model of execution would provide plenty of computing power. However, a direct implementation of computers based on the dataflow model has been found to be a monumental challenge. This article presents a survey of issues and evolutionary developments in dataflow computing. The discussion includes the major problems encountered in past dataflow projects and the emergence novel ideas which have shed a new light in dataflow computing.

ISSUES IN DATAFLOW COMPUTING

A. R. Hurson* and Ben Lee**

*The Pennsylvania State University
Department of Electrical and Computer Engineering
University Park, PA 16802
E-mail: A2H@ecl.psu.edu
Phone: (814) 863-1167

**Oregon State University
Department of Electrical and Computer Engineering
Corvallis, Oregon 97331-3211

Abstract

Early advocates of dataflow computers predicted that the characteristics of the dataflow model of execution would provide plenty of computing power. However, a direct implementation of computers based on the dataflow model has been found to be a monumental challenge. This article presents a survey of issues and evolutionary developments in dataflow computing. The discussion includes the major problems encountered in past dataflow projects and the emergence novel ideas which have shed a new light in dataflow computing.

I. INTRODUCTION

The dataflow model of computation offers many attractive properties for parallel processing. First, the dataflow model of execution is asynchronous, i.e., the execution of an instruction is based on the availability of its operands. Therefore, the synchronization of parallel activities is implicit in the dataflow model. Second, instructions in the dataflow model do not impose any constraints on sequencing except the data dependencies in the program. Hence, the dataflow graph representation of a program exposes all forms of parallelism eliminating the need to explicitly manage parallel execution of a program. For high speed computations, the advantage of the dataflow approach over the control-flow method stems from the inherent parallelism embedded at the instruction level. This allows efficient exploitation of fine-grain parallelism in application programs.

Due to its simplicity and elegance in describing parallelism and data dependencies, the dataflow execution model has been the subject of many research efforts. Since the early 1970s, a number of hardware prototypes have been built and evaluated [24, 44] and simulation studies of different architectural designs and compiling technologies have been performed [5, 46]. The experience gained from these efforts has led to progressive development in dataflow computing. However, the question still remains as to whether the dataflow approach is a viable means for developing powerful computers to meet today's and future computing demands.

Studies from past dataflow projects have revealed a number of inefficiencies in dataflow computing [17, 49]. For example, the dataflow model incurs more overhead in the execution of an instruction cycle compared to its control-flow counterpart due to its fine-grained approach to parallelism. The overhead involved in the detection of enabled instructions and the construction of result tokens will generally result in poor performance in applications with low

degree of parallelism. Another problem with the dataflow model is its inefficiency in handling data structures (e.g., arrays of data). The execution of an instruction involves *consuming* tokens at the input arcs and *generating* result token(s) at the output arc(s). Since tokens represent scalar values, representing data structures, which are collections of many tokens, poses serious problems.

In light of these shortcomings, a renewed interest has emerged in the area of dataflow computing. This revival is facilitated by a lack of developments in the conventional parallel processing arena, as well as a change in viewpoint on the actual concept of dataflow and its implementation. The foremost change is a shift from the exploitation of fine- to medium- and large-grain parallelism. This allows conventional control-flow sequencing to be incorporated into the dataflow approach alleviating the inefficiencies associated with the pure dataflow method. In addition, the recent proposals to develop efficient mechanisms to detect enabled nodes, utilizes advantages offered by both dataflow and control-flow methods. Finally, there is also a viewpoint that the dataflow concept should be supported by an appropriate compiling technology and program representation scheme, rather than with specific hardware support. This allows existing control-flow processors to implement the dataflow model of execution. The aforementioned developments are accompanied by experimental evidence that the dataflow approach is very successful in exposing substantial parallelism in application programs [14, 24]. Therefore, issues such as allocation of dataflow programs onto processors and resource requirements are important problems to resolve before the dataflow model can be considered a viable alternative to the conventional control-flow approach for high-speed computations. The objective then, of this article is to survey the various issues and developments in dataflow computing.

The organization of this article is as follows: Section II reviews the basic principles of the dataflow model. The discussion includes the language support for dataflow and its major attributes. Section III provides a general description of the dataflow model of execution based on the pioneering works of research groups at MIT and the University of Manchester—namely, the Static Dataflow Machine, Tagged-Token Dataflow Architecture, and Manchester Machine. Major problems encountered in the design of these machines will be outlined. Section IV surveys current dataflow projects. The discussion includes a comparison of the architectural characteristics and evolutionary improvements in dataflow computing. Section V outlines the difficulties in handling data structures in a dataflow environment. A brief overview of the methods proposed in literature for representing data structures will be provided. This will be supplemented by various issues required to handle data structures in a dataflow environment. Section VI addresses the issue of program allocation. Several proposed methodologies will be presented and the various issues involved in program allocation will be discussed. Finally, Section VII discusses the resource requirements for dataflow computations.

II. DATAFLOW PRINCIPLES

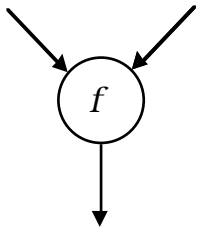
The dataflow model of computation deviates from the conventional control-flow method in two basic principles; *asynchrony* and *functionality*. First, dataflow operations are asynchronous in that an instruction is *fired* (executed) only when all the required operands are available. This is radically different from the control-flow model of computation in the sense that the program counter, which is used to sequentially order the instruction execution, is not used. A dataflow program is represented as a directed graph, $G \equiv G(N, A)$, where *nodes* (or actors) in N represent instructions and *arcs* in A represent data dependencies

between the nodes. The operands are conveyed from one node to another in data packets called *tokens*.

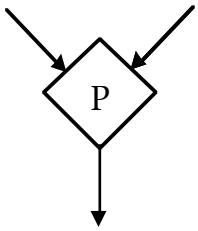
Second, the functionality rule of dataflow implies that there are no side-effects. Any two enabled instructions can be executed in either order or concurrently. This *implicit* parallelism is achieved by allowing side-effect free *expressions* and *functions* to be evaluated in parallel. In a dataflow environment, conventional language concepts such as "variables" and "memory updating" are non-existent. Instead, objects (data structures or scalar values) are consumed by an actor (instruction) yielding a result object which is passed to the next actor(s). The same object can be supplied to different functions at the same time without any possibility of side-effects.

Dataflow Graphs

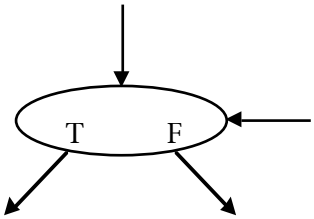
The basis of dataflow computation is the dataflow graph. In dataflow computers, the machine level language is represented by dataflow graphs. As mentioned previously, dataflow graphs consist of nodes and arcs. The basic primitives of the dataflow graph are shown in Figure 1. A data value is produced by an Operator as a result of some operation, *f*. A True or False control value is generated by a Decider (a predicate) depending on its input tokens. Data values are directed by means of either a Switch or a Merge actor. For example, a Switch actor directs an input data token to one of its outputs depending on the control input. Similarly, a Merge actor passes one of its input tokens to the output depending on the input control token. Finally, a Copy is an identity operator which duplicates input tokens.



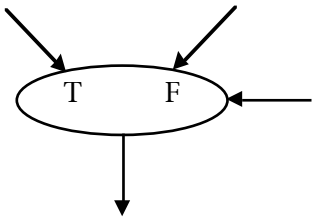
Operator



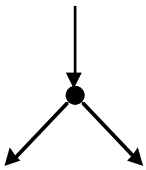
Predicate



Switch



Merge



Copy

Figure 1: Basic primitives of the dataflow graph.

In order to illustrate the capability of the dataflow primitives mentioned previously, consider the dataflow graph shown in Figure 2 which computes the following expression:

$$sum = \sum_{i=1}^{N} f(i)$$

The control inputs to the Merge actors are set to False for initialization. The input values *i* and *sum* are admitted as the initial values of the iteration. The predicate $i \leq N$ is then tested. If it is true, the values of *i* and *sum* are routed to the TRUE sides of the Switch actors. This initiates the firing of the function *f* as well as incrementation of *i*. Once the execution of the body of the loop (i.e., the evaluation of *f*, summation, and $i=i+1$) completes, the next iteration is initiated at the Merge actors. Assuming the body of the loop is a well-behaved graph, the iteration continues until the condition $i \leq N$ is no longer true. The final sum is then routed out of the loop and the initial Boolean values at Merge actors are restore to False.

Note the elegance and flexibility of the dataflow graph for describing parallel computation. In this example, all the implicit parallelism within an iteration is exposed. Furthermore, suppose that the function *f* requires a long time to execute (compare to the other actors in the graph). The index variable *i*, which is independent of the function *f*, will continue to circulate in the loop causing many computations of *f* to be initiated. Thus, given sufficient amount of resources, N iterations of function *f* can be executed concurrently.

Dataflow Languages

There is a special need to provide a high-level language for dataflow computers since their machine language, the dataflow graph, is not an

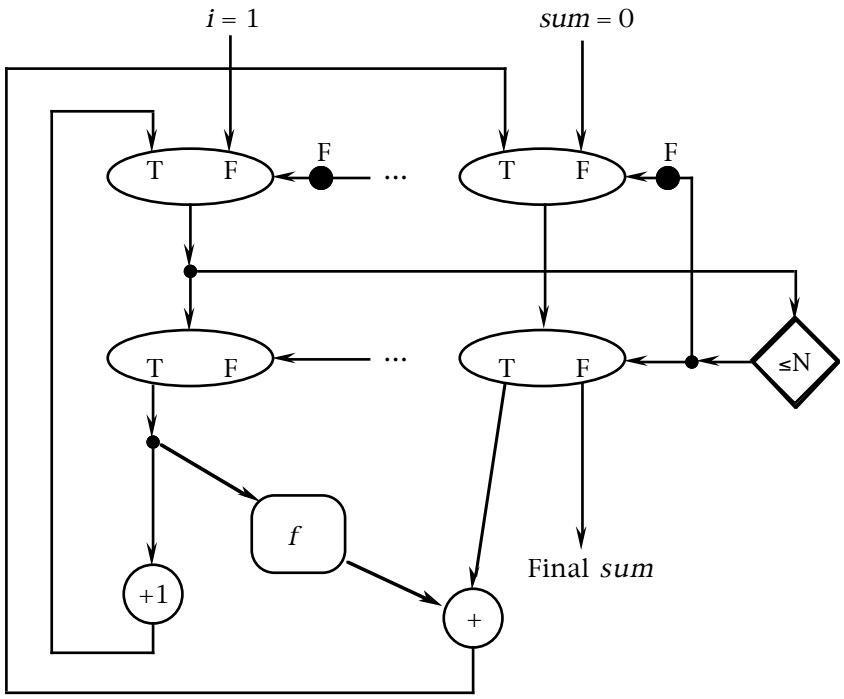


Figure 2: A dataflow graph representation of $sum = \sum_{i=1}^{N} f(i)$.

appropriate programming medium. They are error-prone and difficult to manipulate.

There are basically two high-level language classes which have been considered by dataflow researchers. The first is the *imperative* class. The aim is to map existing conventional languages to directed graphs using dataflow analysis often used in optimizing compilers. For example, the Texas Instruments group used a modified Advanced Scientific Computer (ASC) FORTRAN compiler for their dataflow machine [31]. Compiler techniques for translating imperative languages have also been studied at Iowa State University [6].

Programming dataflow computers in conventional imperative languages would tend to be inefficient since they are by their nature sequential. Therefore, a number of dataflow or *applicative* languages has been developed to provide a more efficient way of expressing parallelism in a program [47]. The goal is to develop high-level dataflow languages which can express parallelism in a program more naturally and to facilitate close interaction between algorithm constructs and hardware structures. Examples of dataflow languages include the Value Algorithmic Language (VAL), Irvine Dataflow language (Id), and Stream and Iteration in a Single-Assignment Language (SISAL) which have been proposed by dataflow projects at MIT [1], the University of California at Irvine [47], and Lawrence Livermore National Laboratories [32], respectively.

The goal of the dataflow concept is to exploit maximum parallelism inherent in a program. There are several useful properties in dataflow languages which allow sequencing of instructions constrained only by the data dependencies and nothing else. These properties are:

- *Freedom from side-effects* - This property is necessary to ensure that the data dependencies are consistent with the sequencing constraints. Dataflow model of execution imposes a strict restriction by prohibiting

any variables from being modified. This is done by using "call-by-value" rather than "call-by-reference" scheme. A call-by-value procedure copies rather than modifies its argument which avoids side-effects.

- *Single assignment rule* - This offers a method to promote parallelism in a program. The rule prohibits the use of the same variable name more than once on the left-hand side of any statement. The single assignment rule offers both clarity and ease of verification, which generally outweigh the convenience of reusing the same name.
- *Locality of effect* - This means that instructions do not have unnecessary far-reaching data dependencies. This is achieved by assigning every variable a definite scope or region of the program in which it is active and carefully restricting the entry to and exit from the blocks that constitute scopes.

The aforementioned properties of dataflow language allow high concurrency in a dataflow computer by easily exposing parallelism in a program. Moreover, it supports easier program verification, enhancing the clarity of programs, and increasing programmer productivity [4].

III. DATAFLOW ARCHITECTURES

In this section, a general description of dataflow machines is provided. Although there have been numerous dataflow proposals[1], the discussion is based on three classical dataflow machines: Static Dataflow Machine [16], Tagged-Token Dataflow Architecture (TTDA) [10], and Manchester Machine [24]. These projects represent the pioneering work in the area dataflow and the foundation they

[1] A good survey of these machines can be found in [44, 46].

provided has inspired many of the current dataflow projects. The major shortcomings of these machines is also discussed.

In the abstract dataflow model, data values are carried by tokens. These tokens travel along the arcs connecting various instructions in the program graph where the arcs are assumed to be FIFO queues of unbound capacity. A direct implementation of this model however is an impossible task. Instead, the dataflow execution model has been traditionally classified as either *static* or *dynamic*. The static approach allows at most one instance of a node to be enabled for firing. A dataflow actor can be executed only when all of the tokens are available on its input arcs and no tokens exist on any of its output arcs. On the other hand, the characteristics of the dynamic approach permit activation of several instances of a node at the same time during run-time. To distinguish between different instances of a node, a *tag* is associated with each token that identifies the context in which a particular token was generated. An actor is considered executable when its input arcs contain a set of tokens with identical tags.

The *static dataflow model* was proposed by Dennis and his research group at MIT [16]. The general organization of the Static Dataflow Machine is depicted in Figure 3. The Program Memory contains instruction templates which represent the nodes in a dataflow graph. Each instruction template contains an operation code, slots for the operands, and destination addresses (Figure 4). To determine the availability of the operands, slots contain presence bits (PBs). The Update Unit is responsible for detecting the executability of instructions. When this condition is verified, the Update Unit

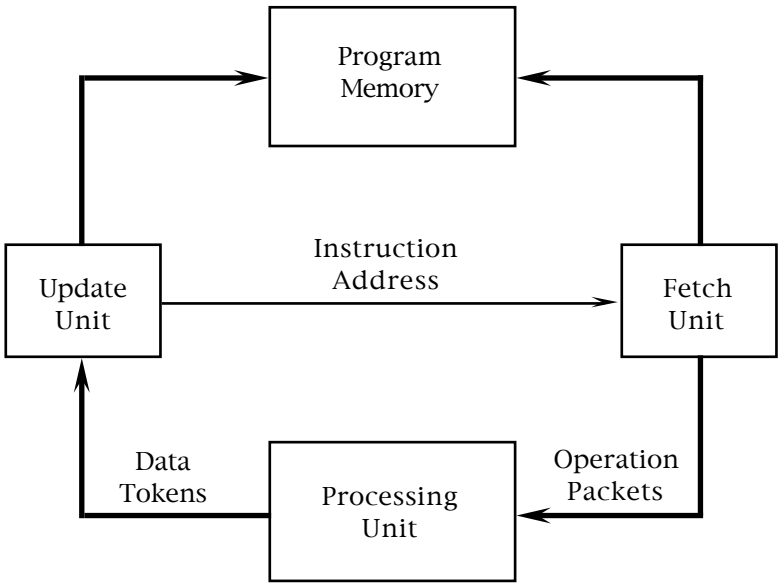


Figure 3: The basic organization of the static dataflow model.

Opcode	
PB	Operand 1
PB	Operand 2
Destination 1	
Destination 2	

Figure 4: An instruction template for the static dataflow model.

sends the address of the enabled instruction to the Fetch Unit. The Fetch Unit fetches and sends a complete operation packet containing the corresponding opcode, data, and destination list to the Processing Unit and also clears the presence bits. The Processing Unit performs the operation, forms a result packets, and sends them to the Update Unit. The Update Unit stores each result in the appropriate operand slot and checks the presence bits to determine whether the activity is enabled.

The *dynamic dataflow model* was proposed by Arvind at MIT [5] and by Gurd and Watson at the University of Manchester [24]. The basic organization of the dynamic dataflow model is shown in Figure 5. Tokens are received by the Matching Unit, which is a memory containing a pool of waiting tokens. The basic operation of the Matching Unit is to bring together tokens with identical tags. If a match exists, the corresponding token is extracted from the Matching Unit and the matched token set is passed on to the Fetch Unit. If no match is found, the token is stored in the Matching Unit to await a partner. In the Fetch Unit, the tags of the token pair uniquely identify an instruction to be fetched from the Program Memory. A typical instruction format for the dynamic dataflow model is shown in Figure 6. It consists of an operational code, a literal/constant field, and destination fields. The instruction together with the token pair forms the enabled instruction and is sent to the Processing Unit. The Processing Unit executes the enabled instructions and produces result tokens to be sent to the Matching Unit.

Note that dataflow architectures can also be classified as *centralized* or *distributed* systems based on the organization of their instruction memories. In a centralized organization, the communication cost of conveying one token from one actor to another is independent of the actual allocation instruction in the program memory. In a distributed organization, the instructions are

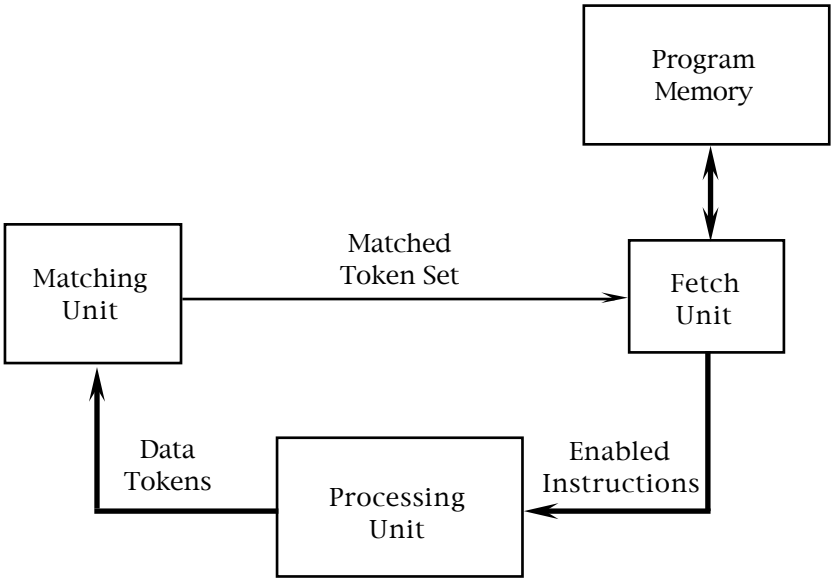


Figure 5: The basic organization of the dynamic dataflow model.

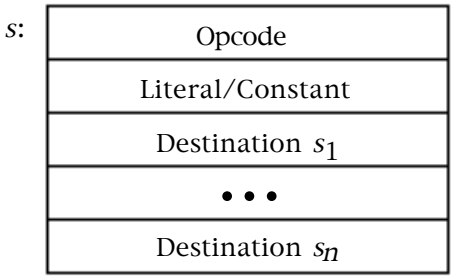


Figure 6: An instruction format for the dataflow model.

distributed among the processing elements (PEs). Therefore, inter-PE communication costs are higher than intra-PE communication costs. Static dataflow machine and the Manchester machine both have centralized memory organization. MIT's dynamic dataflow organization is a multiprocessor system where the instruction memory is distributed among the PEs. The choice between centralized and distributed memory organization has direct affect on the issue of program allocation (see Section VI).

The static dataflow model has inherently simplified mechanism for detecting enabled nodes. Presence bits (or a counter) are utilized to determine when all the required operands are available. However, the static dataflow model has a drawback concerning performance when dealing with iterative constructs and reentrancy. The attractiveness of the dataflow concept is due to the fact that all independent nodes can execute concurrently. It should be noted that in the case of reentrant graphs, strict enforcement of the static firing rule is required; otherwise, it can lead to nondeterminate behavior [46].

To guard against nondeterminacy, Dennis proposed the use of acknowledge signals [16]. This can be implemented by the addition of extra arcs (e.g., acknowledge arcs) from a consuming node to a producing node. These acknowledge signals ensure that no arc will contain more than one token. The acknowledge method can transform a reentrant code into an equivalent graph that allows the execution of consecutive iterations in a pipeline fashion. However, this transformation comes at the expense of increasing the number of arcs and tokens.

The major advantage of the dynamic over the static dataflow model of computation is the higher performance that can be obtained by allowing multiple tokens to exist on an arc. For example, a loop is dynamically unfolded at run-time by creating multiple instances of the loop body and allowing the execution of the

instances concurrently (see Section II). Although the dynamic dataflow model can provide greater exploitation of parallelism, the overhead involved in matching tokens has been a major problem. To reduce the execution time overhead of matching tokens, dynamic dataflow machines require associative memory implementation. In practice, because of the high cost of associative memory in the past, pseudo-associative matching mechanism was used that typically requires several memory accesses [24]. This increase in the number of memory accesses will reduce the performance and the efficiency of dataflow machines.

A more subtle problem with the token matching is the complexity involved in allocation of resources (i.e., memory cells). A failure to find a match implicitly allocates memory within the matching hardware. In other words, when a code-block is mapped to a processor, an unspecified commitment is placed on the matching unit of that processor. If this resource becomes overcommitted, the program may deadlock [35]. In addition, due to the hardware complexity and cost, one cannot assume this resource is so plentiful it can be wasted. The issue of resource requirements in dataflow is discussed more in detail in Section VII.

IV. A SURVEY OF CURRENT DATAFLOW PROPOSALS

In this section, a number of current dataflow projects will be overviewed. The dataflow architectures surveyed here consist of Monsoon, Epsilon-2, EM-4, P-RISC and TAM. These proposals represent the current trend in dataflow computing.

IV.1 Current Dataflow Proposals

Monsoon

Monsoon is a dataflow multiprocessor under development at MIT in conjunction with Motorola [35]. The formulation of the Monsoon began as an outgrowth of the MIT Tagged-Token Dataflow Architecture (TTDA). It is a dynamic dataflow architecture, however, unlike the TTDA, the scheme for matching tokens is implemented more efficiently.

The architecture of the PEs in the Monsoon multiprocessor is based on the Explicit Token Store (ETS) model which is a greatly simplified approach to dataflow execution. The basic idea of the ETS model is to eliminate the expensive and complex process of associative search used in previous dynamic dataflow architectures to match pairs of tokens. Instead, presence bits are employed to specify the disposition of slots within an activation frame which hold data values. The dynamic dataflow firing rule is realized by a simple state transition on these presence bits. This is implemented by dynamically allocating storage for tokens in a code-block. The actual usage of locations within a block is determined at compile-time, however, the actual allocation of activation frames is determined during run-time (this is similar to the conventional control-flow model). For example, when a function is invoked, an activation frame is allocated to provide storage for all the tokens generated by the invocation.

A token in the ETS model consists of a tag and a value. A tag consists of a pointer to an instruction (IP) and a pointer to an activation frame (FP). An instruction pointed to by the IP specifies an opcode, an offset (*r*) in the activation frame where the match will take place, and one or more displacements (*dests*) which define the destination instructions that will receive the results. Each destination is also accompanied by an input port (left/right) indicator which specifies the appropriate input arc for a destination actor.

To illustrate the operation of the ETS model, consider an example shown in Figure 7 of a code-block invocation and its corresponding instruction and frame memory. As a token arrives at an actor (e.g., ADD), the IP part of the tag points to the instruction which contains an offset *r* as well as displacement(s) for the destination instruction(s). The actual matching process is achieved by checking the disposition of the slot in the frame memory pointed to by FP+*r*. If the slot is empty, the value of the token is written in the slot and its presence bit is set to indicate that the slot is full. If the slot is full, the value is extracted, leaving the slot empty, and the corresponding instruction is executed. The result(s) generated from the operation is communicated to the destination instruction(s) by updating the IP according to the displacement(s) encoded in the instruction (e.g., execution of the ADD operation produces two result tokens <FP.IP+1, 3.55> and $<FP.IP+2, 3.55>_L$).

Monsoon multiprocessor consists of a collection of PEs, which are based on the ETS model, connected to each other by a multistage packet switching network and to a set of interleaved I-structure memory modules. Each PE consists of eight pipeline stages which operate as follows (Figure 8): The Instruction Fetch stage fetches an instruction—in the form of <opcode, *r*, *dests*>—from the local Instruction Memory according to the IP part of the incoming token. The calculation of the effective address (FP+*r*) of a slot in the Frame Store where the match will take place occurs in the Effective Address stage. The presence bits associated with the Frame Store location are then read, modified, and written back to the same location. In the Frame Store Operation stage, depending on the status of the presence bit, the value part of

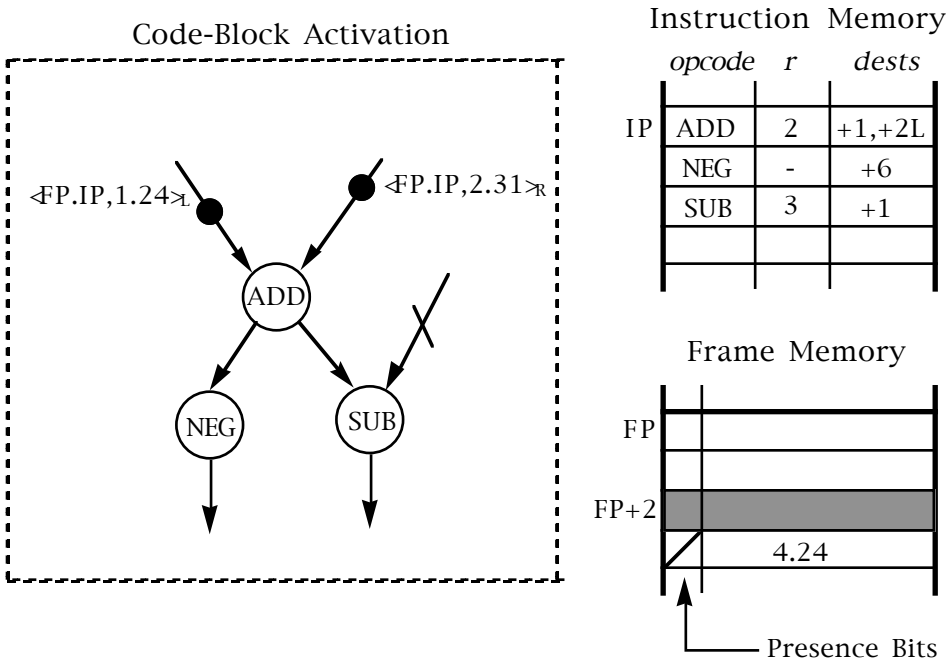


Figure 7: ETS representation of a dataflow program execution.

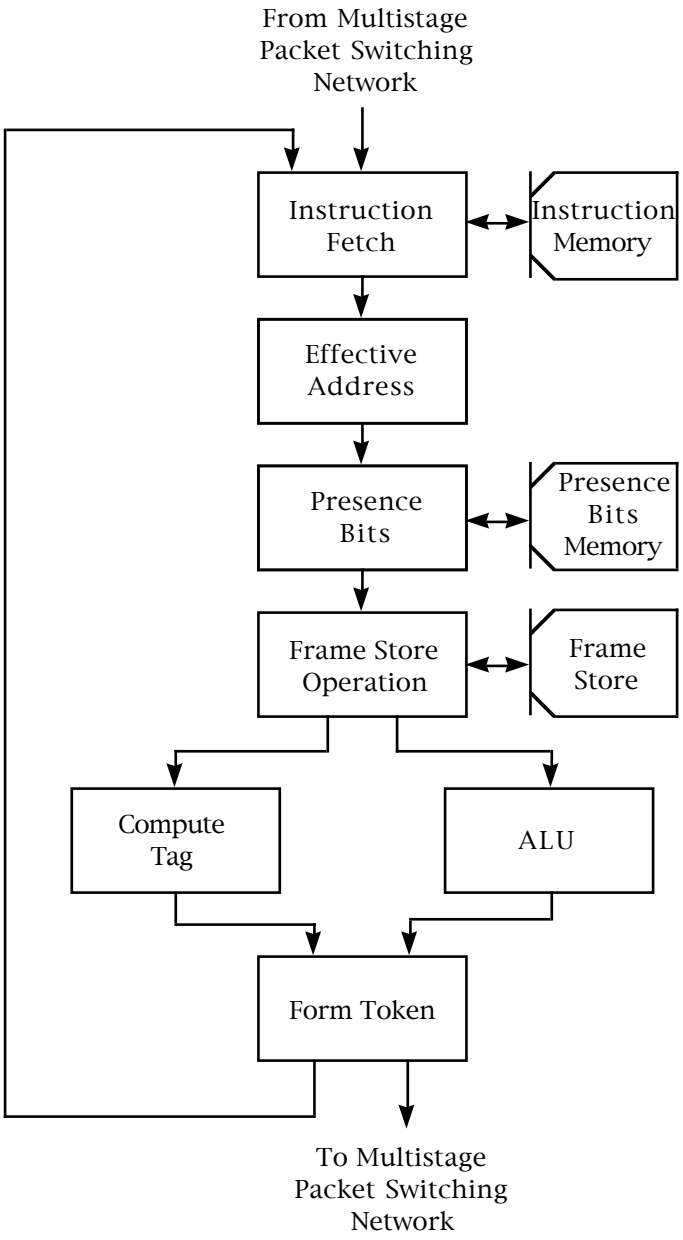


Figure 8: Organization of a PE for the Monsoon.

the appropriate Frame Store location is ignored, read, written, or exchanged with the value on the token.[2]

The ALU consists of three stages and operates concurrently with the Compute Tag stage. In the first stage of the ALU, the value from the token and the value extracted from the Frame Store are sorted into left and right values according to the input port indicator of the incoming token. In the last two stages, the operands are processed by one of the functional units. The tag generation occurs in conjunction with the ALU processing. Finally, the Form Token stage creates result tokens by concatenating the computed tag with the result from the ALU.

A single PE Monsoon prototype has been operational since 1988 and a second prototype is currently under evaluation. The prototype PE was designed to process six million tokens per second. In spite of serious memory limitation, some large codes have been executed on Monsoon and the preliminary results are very encouraging [35]. Currently, the research group at MIT is working with Motorola to develop Monsoon multiprocessor prototypes. The new PEs are expected to be faster (10 million tokens per second) with larger frame storage.

EM-4 Dataflow Multiprocessor

The EM-4 dataflow multiprocessor is under development by the research group at the Electrotechnical Laboratories in Japan [39, 49]. The EM-4 is a highly parallel dataflow multiprocessor with a target structure of more than 1000 PEs based on the SIGMA-1 project [45]. Its design goals are to (1) simplify the architecture by a RISC-based single-chip design and a direct matching scheme,

---

[2] These operations facilitate the handling of loop constants, I-structures, and accumulators [35].

and (2) introduce and utilize strongly connected arc model to enhance the dataflow model.

The EM-4 dataflow multiprocessor utilizes a *strongly connected arc model* to optimized dataflow execution. In this model, arcs in the dataflow graph are divided into normal arcs and strongly connected arcs. A dataflow subgraph whose nodes are connected by strongly connected arcs is called the strongly connected block. Figure 9 shows two examples of a strongly connected block (A and B). The execution strategy for strongly connected blocks is that once the execution of a block begins, the PE would exclusively carry out the execution of the nodes of the block until its completion. Therefore, a strongly connected block acts as a macro node which includes several instructions and is executed as if it was a single basic node, i.e., a thread. Simple nodes falling on critical paths of a dataflow graph are often used to form strongly connected blocks [49].

There are several advantages to the strongly connected arc model. First, a simple control-flow pipeline can be utilized. Second, the instruction cycle can be reduced by utilizing a register file to store tokens in a strongly connected block. This is possible since data in the register file are not violated by any other data. Third, the communication overhead is reduced because there are no token transfers in a strongly connected block. Fourth, the matching can be realized much more easily and efficiently in an intra-block execution because at each moment only one activation of a strongly connected block is executed (i.e., it is not necessary to match function identifiers).

Note that token matching for normal nodes of a dataflow graph is still needed. The EM-4 employs a direct matching scheme which is similar to the frame organization of the Monsoon. Whenever a function is invoked, an

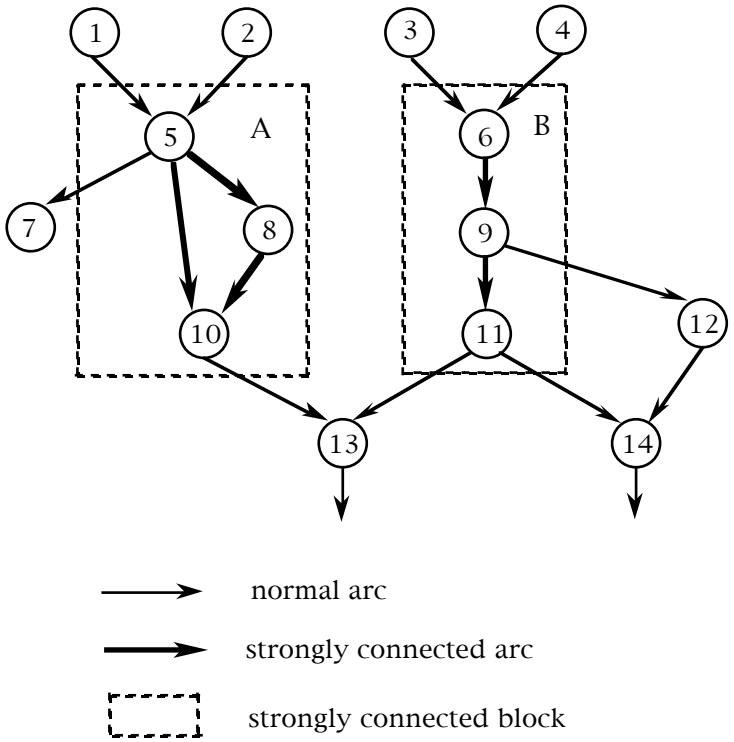


Figure 9: An example of a strongly connected block.

instance of storage, called an *operand segment*, is allocated to this function. The matching is implemented by checking the stored data in the operand segment and storing the new data if its partner is absent. For each instruction within a code-block, there is a corresponding entry in the operand segment which is used to store its operand. Therefore, the address of the instruction can be used to access its operand in the operand segment. The compiled codes for the function are stored in another area which is called a *template segment*. The binding of an operand segment and a template segment is also performed at the time function is invoked.

The PE of the EM-4 prototype consists of a memory module and a single chip processor called EMC-R. Figure 10 shows the block diagram of a PE. The Switching Unit controls the flow of tokens within the PE, to and from the communication network as well as to neighboring PEs. The Input Buffer Unit is a buffer for tokens waiting for execution. The Fetch and Matching Unit performs matching operations for tokens, fetches enabled instructions, and sequences operations in strongly connected blocks. Moreover, it controls the pipelines of a PE by integrating two types of pipelines (register-based advanced control pipeline and token-based circular pipeline). The Execution Unit employs a RISC-based ALU for high-speed processing. The Memory Control Unit arbitrates memory access requests from the Input Buffer Unit, the Fetch and Matching Unit, and the Execution Unit and communicates data between the Off-Chip Memory and the EMC-R.

The pipeline organization of the EMC-R is presented in Figure 11. It basically consists of a four-stage pipeline. When a token arrives, the corresponding template segment number is fetched from the Off-Chip Memory in the first stage (TNF). The number is stored at the beginning of the operand segment when the function is invoked. If an incoming token does not require

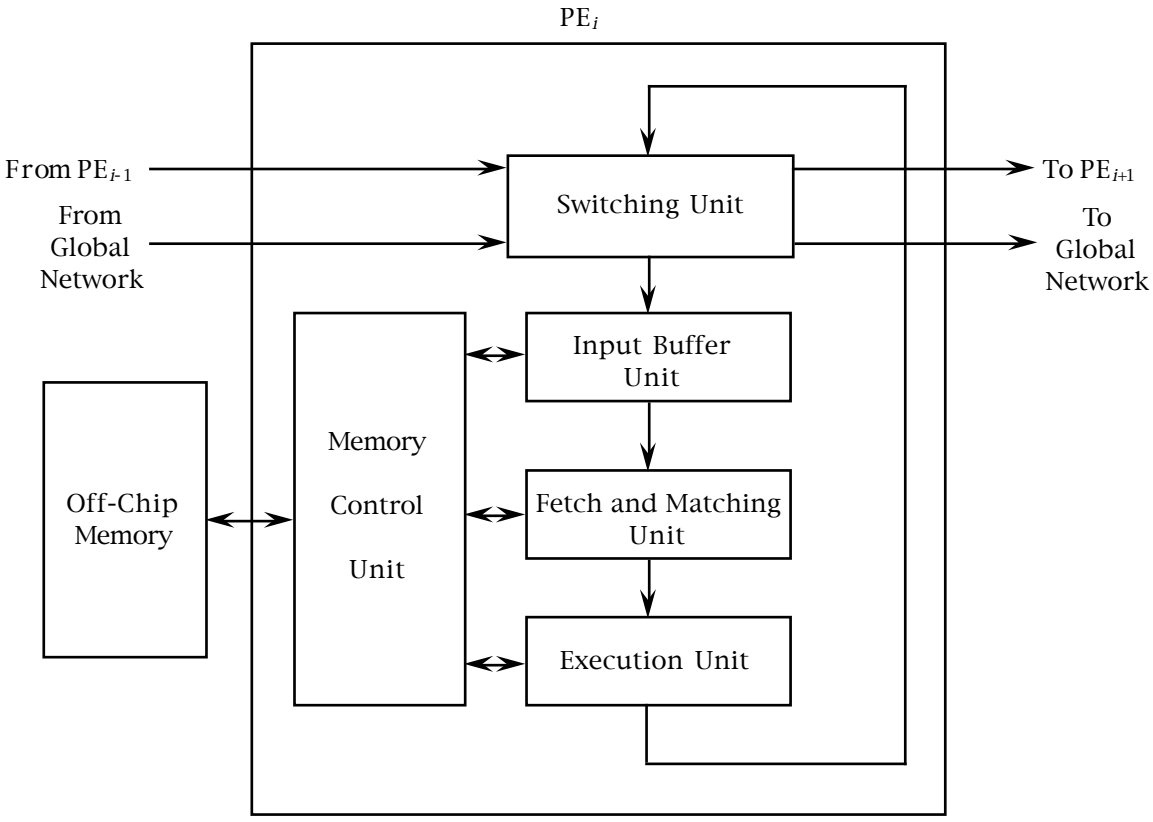


Figure 10: Organization of the EMC-R.

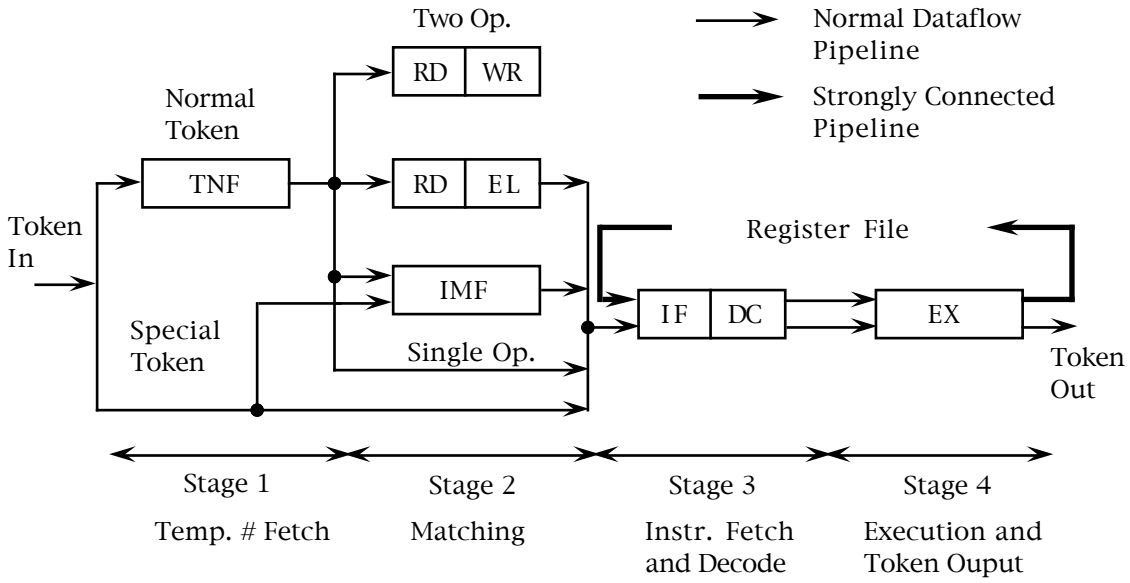


Figure 11: Pipeline organization of the EMC-R.

matching, the first stage is bypassed. The second stage is the matching stage. If the token is for a single-operand instruction, it is sent to the next stage. If the matching is with an immediate value, an immediate fetch (IMF) occurs. If the token is for a two-operand instruction, the corresponding data in the operand segment is read (RD). If a match exists, the flag for the data is cleared (EL) to signal the firing of the instruction; otherwise, the token is written (WR) into the operand segment to await its partner.

The third stage fetches and decodes the instruction. The fourth stage is the execution stage. If the next instruction is strongly connected with the current instruction, the result is written in the register file. The execution of the third and fourth stage is overlapped until there are no executable instructions in the concerned strongly connected block. Therefore, the organization of the EMC-R allows the integration of a token-based circular pipeline and register-based advanced control pipeline.

The EMC-R has been fabricated on a chip which contain 50,000 CMOS gates and 256 signal lines. Each PE has a peak processing performance of more than 12 MIPS. Currently, EM-4 multiprocessor which contains 80 PEs is being constructed. It will consist of 16 processor boards, each containing 5 PEs and a mother board which will contain the communication network. The total peak performance of the prototype is expected to be more than 1 GIPS.

Epsilon-2 Multiprocessor

The Epsilon-2 dataflow multiprocessor was proposed by Grafe and Hoch at Sandia National Laboratories [22]. The design of the Epsilon-2 is based on the dynamic dataflow model and evolved from the Epsilon-1 project [21, 23]. Two prototypes of the Epsilon-1 have already been built and demonstrated sustained

uniprocessor performance comparable to that of commercial mini-supercomputers [21].

The basic organization of the Epsilon multiprocessor is shown in Figure 12. The system is a collection of Processing Modules communicating with each other via a Global Interconnect. Each module consists of a PE, a Structure Memory, and an I/O port connected to each other by a 4×4 Crossbar Switch. Tokens in the system are of fixed length containing a target and a data portion. The target portion of the token represents the tag and consists of an instruction and a frame pointer pair <IP.FP>. Both target and data portions are augmented with type fields. The type field of the target portion points to a particular resource in the system (e.g., a PE, a Structure Memory, an I/O, etc.). On the other hand, the type field of the data portion defines the type of information which the token carries (e.g., a floating point, an integer, a pointer, etc.).

The basic organization of an Epsilon-2 PE is depicted in Figure 13. Tokens arriving at the PE are buffered in the Token Queue. Tokens are read from the Token Queue and the instruction pointer IP of the tag field is used to access the Instruction Memory. An Epsilon-2 instruction word is shown in Figure 14. It contains two offsets for accessing operands—the operand mode determines the item of selection, e.g., current activation frame, a constant, or a register. The match offset field is used to select a rendezvous point in the Match Memory for pair of tokens. The actual synchronization is performed by reading the value in the selected location in the Match Memory. If the value equals the match count encoded in the opcode, the instruction is enabled for execution and the match count is reinitialized to zero. If the value does not equal the match count, the value is incremented and written back to the match location.

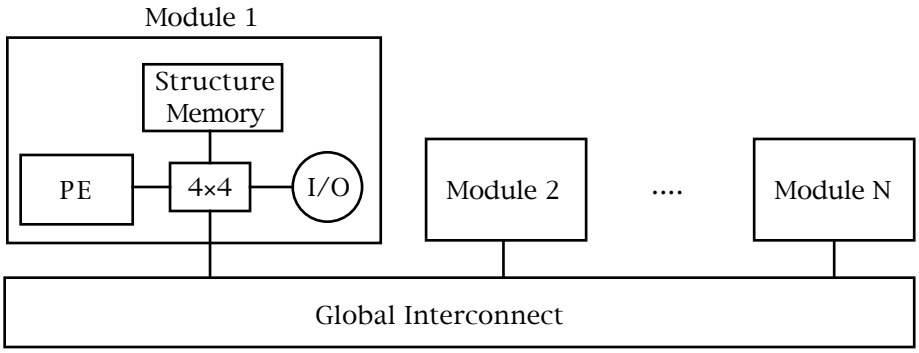


Figure 12: The organization of the Epsilon-2 multiprocessor.

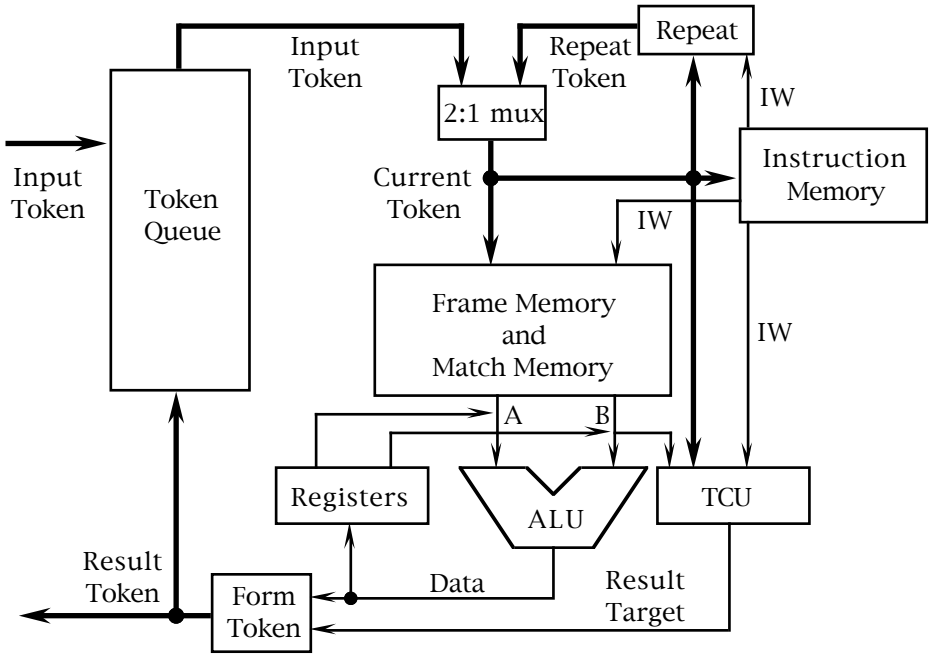


Figure 13: An Epsilon-2 PE.

Repeat Offset	Left Operand Offset	Left Operand Mode	Right Operand Offset	Right Operand Mode	Match Offset	Result Register	Target Offset	Opcode

Figure 14: Epsilon-2 instruction word.

The ALU combines two operands specified by the opcode. Results generated from operations are written into a register directed by the result register field of an instruction word. These registers can be reference by any succeeding instructions. Note that register contents are not necessarily preserved across grain boundaries; therefore, registers serve as buffers for intermediate values within a grain. Finally, the Target Calculation Unit (TCU) is responsible for generating the target portion of output tokens. Targets local to the current frame retain the current frame pointer and a new instruction pointer is generated by adding the target offset from the instruction word to the current instruction pointer.

The Epsilon-2 processor also contains a Repeat unit which is used to generate repeat tokens. Repeat tokens efficiently implements data fanout in dataflow graphs and significantly reduces the overhead required to copy tokens. The repeat offset in the instruction word is used to generate a repeat token. This is done by adding the repeat offset to the current token's instruction pointer to generate a new token. Thus, the Repeat unit basically converts a tree of Copy operations (where the leaves represent the instructions) to a linked list of instructions. The Repeat unit is also used to schedule a grain of computation. This is achieved by representing a thread of computation as a linked list and utilizing registers to buffer the results between instructions.

As mentioned previously, the design of the Epsilon-2 Multiprocessor is based on the Epsilon-1 project. Despite the fact that the Epsilon-1 processor is a single board, wire-wrap, 10 MHz CMOS prototype based on the static dataflow model, it showed experimental evidence that a dataflow computer can provide better performance than comparable control-flow processors. Based on this experience, many refinements have been made in the development of the Epsilon-2.

P-RISC

P-RISC (or Parallel-Reduced Instruction Set Computer) was proposed by Nikhil and Arvind at MIT [33]. The basic idea behind P-RISC is strongly influenced by Iannucci's dataflow/von Neumann hybrid architecture [28]. P-RISC takes existing RISC-like instruction set, generalize it to be multithreaded to achieve a fine-grained dataflow capability. Since it is an extension of the von Neumann architecture, it can exploit conventional as well as dataflow compiling technology. More important, it can be considered as a dataflow machine that provides complete software capability with conventional control-flow machines.

P-RISC multiprocessor consists of PEs and Structure Memory Elements connected by an interconnection network. The organization of a P-RISC PE is shown in Figure 15. The Local Memory contains instructions and frames. In order to support fine-grained interleaving of multiple threads, a thread of computation is completely described by an instruction pointer (IP) and a frame pointer (FP). The pair of pointers, <FP.IP>, is regarded as a *continuation* and it corresponds to the tag part of a token (terminology used in TTDA). The Instruction Fetch and Operand Fetch units fetch appropriate instructions and operands pointed to by the <FP.IP> part of the incoming tokens. The Function Units perform the general RISC operations and the Operand Store is responsible for storing results in the appropriate slots of the frame. Notice that for most part, the P-RISC executes instructions in the same manner as conventional RISCs. For example, an arithmetic/logic instruction generates a

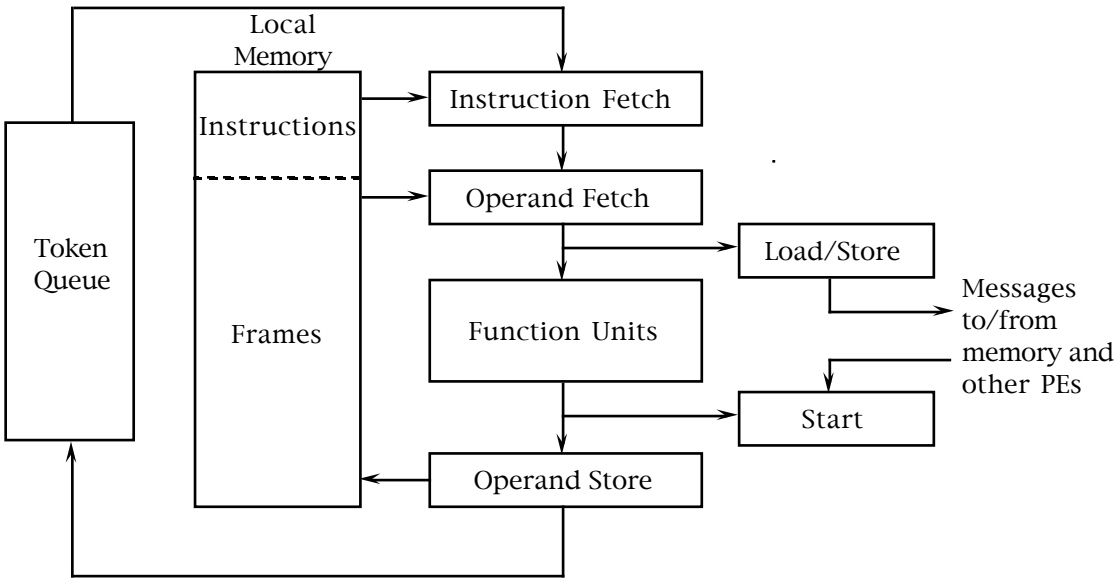


Figure 15: Organization of a P-RISC PE.

continuation which is simply <FP.IP+1>. For a Jump *x* instruction, the continuation is <FP.*x*>.

To initiate new threads and to synchronize two threads, P-RISC contains two special instructions—Fork and Join. It is important to note that these are simple instructions which are executed within the normal processor pipeline and are not operating system calls. Fork *x* is similar to a Jump in the conventional RISC semantic except that two tokens, <FP.*x*> and <FP.IP+1>, are generated as continuations. Join *x*, on the other hand, toggles the contents of the frame location FP+*x*. If the frame location FP+*x* is empty, it produces no continuation. If it is full, it produces <FP.IP+1>. Note that the Join operation can be generalized to support *n*-way synchronization, i.e., the frame location *x* is initialized to *n*-1 and different join operations decrement it until it reaches zero.

Currently, efforts are under way to take existing RISC implementation and extended it to realize P-RISC while maintaining software compatibility.

Threaded Abstract Machine

Threaded Abstract Machine (TAM) was proposed by Culler *et al.* at University of California, Berkeley [15]. The basic idea behind TAM is to place all synchronization, scheduling, and storage management requirements for execution of fine-grain parallelism explicit and under compiler control. This relieves the overwhelming demands placed on the hardware, gives the compiler the responsibility for scheduling low-level program entities, and optimizes the use of critical processor resources, e.g., such as high-speed register storage.

In TAM, the basic storage resources are code-blocks and frames. A code-block consists of a collection of threads where each thread represents a sequence of instructions. Whenever a code-block is invoked, a frame is allocated. Each frame contains slots for depositing argument values associated with the invoked

code block. Instructions may reference slots within the current frame or registers. It is the responsibility of the compiler to statically determine the frame size for each code-block as well as correctly using the slots and registers. The compiler also allocates a portion of the frame as a *continuation vector* which is used to hold the pointers to enabled threads.

To provide synchronization among the threads, a frame slot contains the entry count for the thread. Each fork to such a thread causes the entry count to be decremented. When the entry count reaches zero, the thread is initiated and executes to completion. A Fork operation causes additional threads to be scheduled for execution. An explicit Stop instruction signals the end of a thread execution and initiates another enabled thread. Conditional flow of execution is supported by a Switch, which forks one of two threads depending on a Boolean value.

Since all the activations of a program cannot be maintained in high speed processor storage, TAM provides a storage hierarchy. In order to execute a thread from an activation, it must be made resident on a processor. Only those threads which are resident have access to processor registers. Once an activation is made resident, all enabled threads within the activation execute to completion. The set of threads executed during a residency is called a *quantum.* The processor registers are considered as an extension to the frame with a lifetime of a single quantum. They carry operands between instructions within a thread or between threads within a quantum. Inter-frame interactions are performed by having a set of inlets with each code-block that defines its external interface. An inlet is a simple sequence of instructions which receives the corresponding message, stores values into the specified frame slots, and inserts a thread in the corresponding continuation vector.

A prototype TAM instruction set, TL0 (Threaded Language Version 0), has been developed at the University of California at Berkeley. Id programs are first compiled to generate TL0, then to the native machine code, using C. Preliminary studies have shown that a dataflow language can be compiled to execute with performance which is comparable to conventional sequential language on stock hardware [15].

IV.2 Architectural Trend of the Current Dataflow Machines

Table 1 outlines the main architectural features of current dataflow proposals discussed in the previous subsection. Observations of current dataflow projects show that there is a trend towards adopting the dynamic dataflow model. This is due to the emergence of a novel and simplified process of matching tags—direct matching. In previous dynamic dataflow machines, tags contained all the information associated with activity names. An activity name uniquely identifies the particular instance of computation in which the token was generated. In a direct matching scheme, storage (e.g., activation frame) is dynamically allocated for all the token pairs generated by an activation. For example, "unfolding" a loop body is achieved by allocating an activation frame for each loop iteration. Thus, matching pair of tokens generated within an iteration has a unique slot in the activation frame in which to converge. The actual matching process involves checking the disposition of the slot in the frame memory. Another major architectural change observed is the integration of the control-flow sequencing with the dataflow model. Dataflow architectures which are based on the pure dataflow model, such as the TTDA and the Manchester dataflow machine, provide well-integrated synchronization at a very basic level, i.e., at the instruction level. A typical dataflow instruction cycle involves detection of enabled nodes by matching tokens, computing the results, and

generating and communicating the result tokens to appropriate target nodes. However, it is clear that characteristics of the pure dataflow model are not efficient. For example, consider the synchronization requirements of the instructions with in a procedure. It should not be necessary to generate and match tokens for scheduling every instruction within the body of a procedure. It is intuitively obvious some of this responsibility can be assigned to the compiler and instead a simpler control-flow sequencing can be employed. Moreover, the overhead of construction and communication of result tokens can be reduced by introducing a fast processor register which is used to temporary hold data. The incorporation of control-flow sequencing and registers appear in almost all of the current dataflow projects; e.g., the EM-4, the Epsilon-2, and TAM.

In contrast to dataflow proposals discussed thus far, TAM provides a conceptually different perspective on the implementation of the dataflow model of computation. In TAM, the execution model for fine-grain parallelism is supported by an appropriate compilation strategy and program representation rather than through elaborate hardware. By assigning the synchronization, scheduling, and storage management tasks to the compiler, the use of processor resources can be optimized for the expected case, rather than the worst-case. In addition, since the scheduling of threads is visible to the compiler, TAM allows a more flexible use of registers across thread boundaries. Since the basic structure of TAM is similar to a number of dynamic architectures discussed in this article, it could be directly realized in hardware. However, the concept of TAM is currently used to study the type of

TABLE 1: A comparison of the architectural features.

Architecture	Key features
Monsoon	• Direct matching of tokens based on the Explicit Token Store concept.
EM-4	• The use of strongly connected arc model to enhance the dataflow model. • Use of registers to reduce the instruction cycle and the communication overhead of transferring tokens. • Integration of a token-based circular pipeline and a register-based advanced control pipeline. • Direct matching of tokens. • RISC-based processing element.
Epsilon-2	• Direct matching of tokens. • Repeat fan-out mechanism to reduce the overhead in copying tokens. • Control-flow type of sequencing and use of registers. • Load balancing (adaptive routing).
P-RISC	• Utilization of existing RISC-like Instruction set and its generalization for parallel-RISC. Can use both conventional and dataflow compiling technologies. • Application of multithread using a token queue and circulating thread descriptors. • Introduction of Fork and Join instructions spawn and synchronize multiple threads. • Synchronization of memory accesses through the use of I-structure semantics.
TAM	• Placing all synchronization, scheduling, and storage management responsibility under compiler control that allows execution of dataflow languages on conventional control-flow processors. • Providing a basis for scheduling a number of threads within an activation as a quantum while carrying values in registers across threads. • Having the compiler produce specialized message handlers as inlets to each code-block.

architectural support needed for full-scale parallel programs on large parallel machines.

V. DATA STRUCTURES

As discussed in Section II, the main implication of the functionality principle of dataflow is that all operations are side-effect free, i.e., when a scalar operation is performed, new tokens are generated after the input tokens have been consumed. However, if tokens are allowed to carry vectors, arrays, or other complex structures in general, the absence of side-effects implies that an operation on a structure element must result in an entirely new structure. Although this solution is acceptable from a theoretical point-of-view, it creates excessive overhead at the level of system performance.

In order to provide a facility to handle data structures while preserving the functionality of dataflow operations, a number of schemes have been proposed. There are two basic approaches to representing data structures, such as arrays: *direct* and *indirect* access [19]. A direct access scheme treats each array element as individual data tokens—the notion of array is completely removed at the lowest level of computation. The tokens are identified by their tags which carry information about the relative location of the element within an array. In an indirect access scheme, arrays are stored in special memory units and their elements are accessed through explicit "read" and "write" operations.

V.1 A Survey of Data Structure Handling Schemes

The various proposed solutions to the problem of handling data structures is now presented. The characteristics of each scheme will be outlined and the major shortcomings of each method will be identified and analyzed.

V.1.1 Direct Access Method

Token Relabeling Scheme

The token relabeling scheme was proposed by Gaudiot [18, 19]. Its basic idea is to provide direct data forwarding between a producer and a consumer of an array by relabeling the tag associated with each token. This is possible since the information in the tag indicates the token's origin in the program and in particular the iteration in which the element was created; therefore, the iteration number can usually be directly interpreted as the index of the array element. This eliminates the need to maintain the notion of an array at the lowest level of execution and hence an array is represented by a set of individual tokens.

The basis of the token relabeling scheme can be illustrated by the scatter and gather operation corresponding to the following Fortran code [19]:

```
      DO 1 i = 1, n
1     C(i) = B(i) + A(F(i))
```

The corresponding graph implementation is shown in Figure 16. In each iteration, array elements $B(i)$ and $A(F(i))$ are tagged with an index number i and $F(i)$ (denoted as $B(i)_{[i]}$ and $A(F(i))_{[F(i)]}$), respectively. In order to match the pair of tokens, the tags of elements in array A must be mapped from $F(i)$ to i. This requires a relabeling function F^{-1} which is usually unknown and difficult to obtain at compile-time. In the token relabeling scheme, the effect of F^{-1} is simulated by the following technique. A relabeling actor δ_r is used to extract the tag value in $B(i)_{[i]}$ and produces $i_{[i]}$. This is mapped to $F(i)_{[i]}$ through the given access function F. Another relabeling function χ swaps the

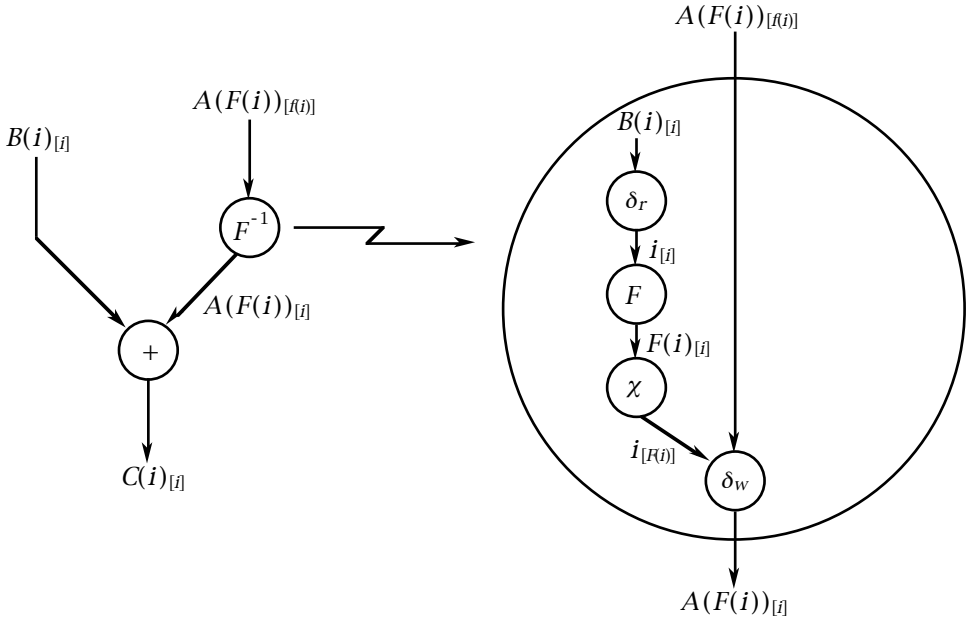


Figure 16: Token relabeling of the Gather operation.

data value and the tag value of $F(i)_{[i]}$ to obtain $i_{F(i)}$. This matches with $A(F(i))_{[F(i)]}$ at the third relabeling function δ_W to generate the $A(F(i))_{[i]}$.

Since the direct access method is compatible with the dataflow principles of execution, it leads to the use of a more homogeneous architecture (i.e., no complex structure controller is required) and a better speedup factor since data tokens need not transit through a separate structure controller. However, the direct access method has the following shortcomings: First, preserving the dataflow semantics of side-effect free operation implies that entire data structures must be passed from one node to the next or duplicated among different nodes. Since tokens are treated as scalar elements, floating in the system and stored in the token matching unit, a large storage space for these array copies will be required in the matching storage. Second, in many applications, the notion of array as a single entity cannot be completely done away with—e.g., as in table look-up applications.

V.1.2 Indirect Access Approach

Heaps

In MIT Static Dataflow Machine, arrays are represented as directed acyclic graphs stored in a separate auxiliary memory [2]. Directed acyclic graphs consist of a set of <selector:value> pairs where the "selector" is an integer or a string and the "value" is any data value including another substructure. Directed acyclic graphs (or heaps) always form a tree having one root node with the property that each leaf of the graph can be reached by a directed path from the root node.

The actual representation of arrays is accomplished by tokens carrying pointers only to the root nodes of the structures. This avoids the communication overhead involved in sending a large token containing the entire array to a requesting actor. An example of the tree representation of an array is shown in

Figure 17. The major motivation behind the use of heaps is to alleviate excessive copying by allowing the common substructures to be shared among several structures. In order to keep track of the number of pointers created and destroyed during the course of a program execution, a *reference count* is associated with each node. When the reference count of a node becomes zero, the node is inaccessible and is placed on a *free storage list* for future usage.

SELECT and APPEND are exclusive operations devoted to manipulating the trees. A value can be retrieved from a structure by a SELECT operation. Depending on the selector argument, the corresponding value from the input structure is placed on the output arc. An APPEND operation creates a new structure which is a version of the input structure containing the modified component. This is illustrated in Figure 18. As can be seen, the APPEND creates a structure (A') which is a version of the input structure (A) containing the new or modified component. Note that elements are shared between the two structures A and A'. Thus, no existing structure is ever modified, rather a new and different structure is constructed.

Although tree structures are capable of representing complex data structures, several disadvantages have been identified when arrays are represented as trees. First, unlike sequentially stored arrays, accessing an element in a tree requires $O(\log n)$ time. Therefore, depending on the depth of the tree, representing arrays as trees may degrade the performance. Second, operations can only be performed on individual array elements due to the low level at which the dataflow model of sequencing is applied. Therefore, performing two independent append operations on an array still requires sequential execution (e.g., *restrained* or *strict* structure). As a result, even

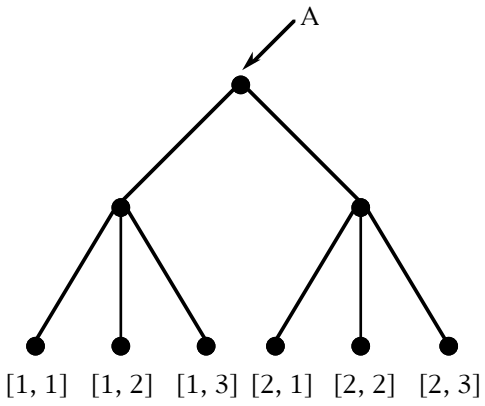


Figure 17: A heap representation of an array.

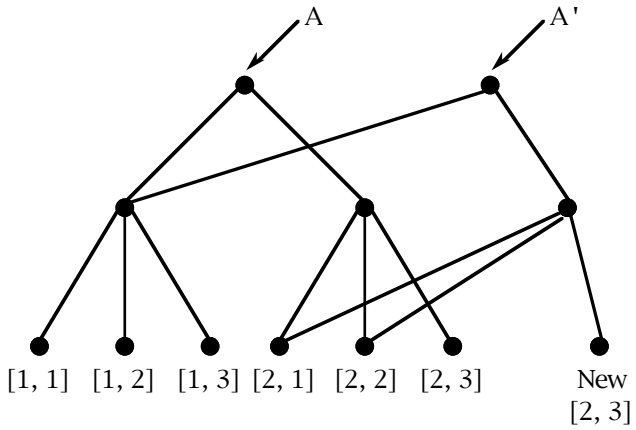


Figure 18: An append operation.

simple operations, such as setting a column or a row of a matrix to zero, require the creation of intermediary structures that significantly degrades the performance [17]. Finally, the garbage collection process of unused nodes is a difficult and an expensive procedure in the tree representation [2]. This is due to the fact that when the reference count of a node is reduced to zero, the reference counts of the nodes indicated by the pointers in the reclaimed node must also be decreased (i.e., recursion).

I-structures

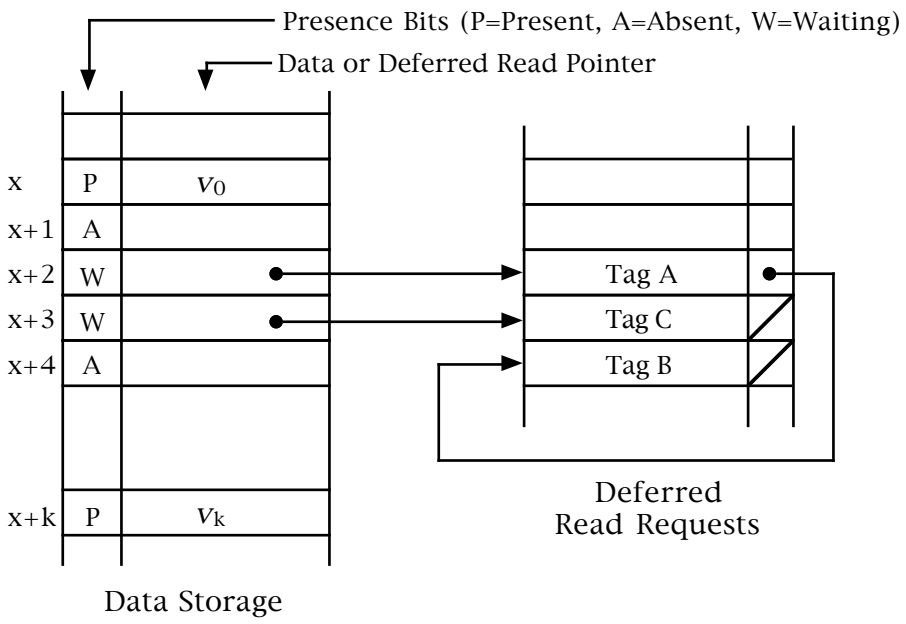
I-structures were proposed by Arvind and Thomas to provide a data structuring mechanism for the TTDA [9, 12]. I-structures are asynchronous array-like structures with constraints on their creation and access, i.e., an I-structure is defined once and each of its elements can be written at most once. An I-structure is asynchronous in the sense that the construction of arrays is not strictly ordered; therefore, it is possible for a part of a program to attempt to select an element before that element is appended (e.g., *unrestrained* or *non-strict* structure).

In order to implement the concept of "consumption before complete production," each storage cell contains *presence bits* to indicate that the cell is in one of three possible states (Figure 19):

(a) PRESENT - The cell contains a valid data that can be read as in conventional memory.

(b) ABSENT - No data has been written into the cell since it was last allocated.

(c) WAITING - No data has been written into to the cell; however, at least one attempt has been made to read it. Such a request is deferred in a



A sequence of operations producing this structure:

- Attempt to READ(x+2) for instruction A
- WRITE(x+k)
- Attempt to READ(x+3) for instruction C
- WRITE(x)
- Attempt to READ(x+2) for instruction B
- READ(x)

Figure 19: An example of I-structure storage.

linked list called the *deferred read request list*. When the data is eventually written into the cell, all deferred reads are serviced.

Despite improvements over the tree structure method, such as reduced access time and storage requirement, some problems are associated with the I-structures. Although I-structures are useful in large class of program constructs where array operations are performed in parallel (e.g., loops), they are not suitable for every programming environment. First, due to the non-strictness of I-structures, problems arise when modeling certain classes of problems which require the mode of production to be strict—e.g., termination of accumulation [41]. In addition, since I-structure elements are appended (written) at most once, there is a potential for race conditions for APPENDs [18].

University of Manchester Approach

The data structure representation in the prototype Manchester Dataflow Machine was proposed by the research group at the University of Manchester [41]. The implementation combines the concept of streams (i.e., a sequence of values communicated between two portions of a code [41]) with conventional arrays. However, in contrast to streams, the size of the structure must be known at the time it is created. Thus, in this approach a *finite component,* defined as a collection of a finite sequence of elements, is regarded as the "unit" on which the basic storage operations are performed.

There are special operations exclusively used to handle finite component structures. A STORE operation converts a token stream into a stored component and returns a pointer to the component. Note that the STORE operation, which releases the pointer immediately, is *unrestrained*. A FETCH is the inverse of STORE; it converts a component back into a token stream. The combination of STORE and FETCH operations is used to communicate elements of a data structure between

processes. A SELECT operation can be used to randomly access an element of a component. A COLLECT operation is used to determine when all the elements of a component have arrived. This restrained operation ensures that the pointer token is not released until all the elements of a component have been created. An INCREMENT/DECREMENT operation, associated with the reference count scheme, is concerned with garbage collection.

The University of Manchester design offers advantages brought forth by combining the concept of streams with conventional arrays. However, modification of any element(s) in an array still requires copying the entire array [40, 41].

EXtended MANchester Approach

In this approach, an enhanced array structure is used to extend the basic Manchester machine [36]. The machine, referred to as the EXtended MANchester (EXMAN) computer, alleviates the problem of memory overhead by having token pointers pointing to the starting location of the array and increases the access speed by utilizing conventional random access structures.

Initially an array is sequentially stored as a random access structure. To perform an APPEND operation on an array, a new node is created with its value and index field appropriately filled. The node is then linked to the array and a pointer referencing the new node is returned. This is performed by an APPEND operator with three inputs, namely, the array pointer, the index, and the modified value. A SELECT operation is performed by taking the pointer reference given at the input and traversing the linked list until the input index matches the index field of a node. When a match is found, the value field of this node is returned at the output. Otherwise, the required element is obtained from the original array

with one more access. A reference count is associated with each node to perform a run-time garbage collection on the unused appended nodes.

Figure 20 illustrates a series of APPEND operations on array A. The first APPEND operation results in a modified array with the first element changed. Similarly, repeated APPEND operations result in a linked list shown in Figure 21. A SELECT operation is performed by taking the pointer reference given at the input and traversing the linked list until the input index matches the index field of a record. When a match is found, the value field of this node is returned at the output. For example, the select operator in Figure 20 requires the traversal of the nodes 3 and 2 in Figure 21. If a match is not found in the linked list, the required element is obtained from the original random access array.

To perform a run-time garbage collection on the unused appended nodes, each node contains four fields—index, value, reference count and array number. A reference count indicates the number of references pointing to the node. With each select operation, the array reference pointer is absorbed
and the reference count of the node is decreased by one. When the reference count reaches zero, the node is garbage collected and is returned to the free storage list. The select operation in Figure 23 will therefore absorb the array reference pointers S and R and the reference count of node 1 will be decreased to one.

The EXMAN approach avoids excessive copying by utilizing dynamic pointers with conventional random access structures. This in turn leads to a gain in memory space; however, disadvantages still exist. The most obvious problem is the number of append operations to an array which consequently affects the search time for a select operation.

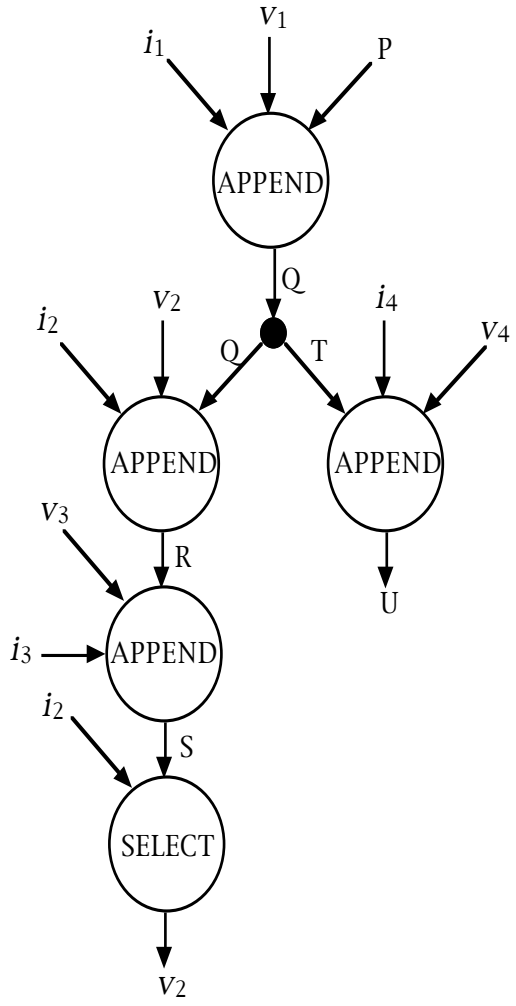


Figure 20: A dataflow graph

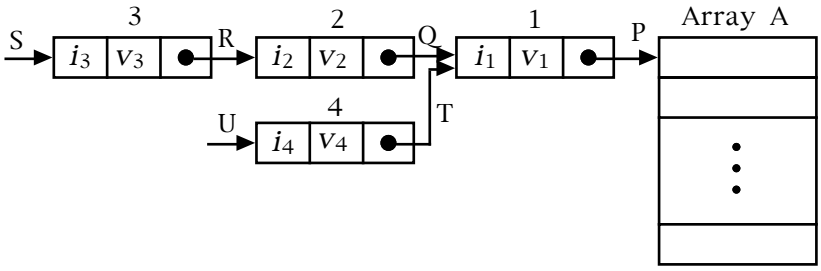


Figure 21: Linked lists to represent the modification on array A after the execution of the dataflow graph in Figure 20.

Hybrid Scheme

The hybrid scheme was proposed by Lee and Hurson [30, 25]. The basic idea behind the hybrid scheme is to associate a template, called the *structure template*, with each conceptual array. For selective updates, this minimizes copying by allowing only the modified elements to be appended to the new array.

The basic representation of an array in the hybrid scheme is shown in Figure 22. Each array is represented by a *hybrid structure* which consists of a structure template and a vector of array elements. A structure template is subdivided into three fields. The reference count field (RC) is an integer indicating the number of references to the array. The location field (LOC) contains a string of 1's and 0's where the length of the string equals the total number of elements in the array. Each location bit determines whether the desired array element resides in the vector indicated by either the left ("0") or the right ("1") pointer.

When an array is initially created, the status bit (S) is initialized to “0” indicating that the vector contains the *original* array. Whenever a modification is made to an array with more than one reference, a new hybrid structure is created (the status bit set to “1”) where all the modified elements can be accessed from the vector pointed by the right pointer. The sharing of array elements between the original and the modified array is achieved by linking the left pointer of the modified hybrid structure back to the original hybrid structure. The location bits in the structure template can then be used to access the desired element, i.e., the i^{th} location bit having a value of "0" or "1" indicates that the corresponding array element can be found in the vector linked to the left or right pointer, respectively.

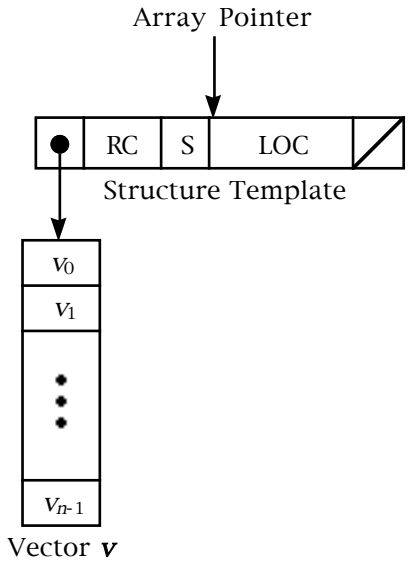


Figure 22: The proposed array representation.

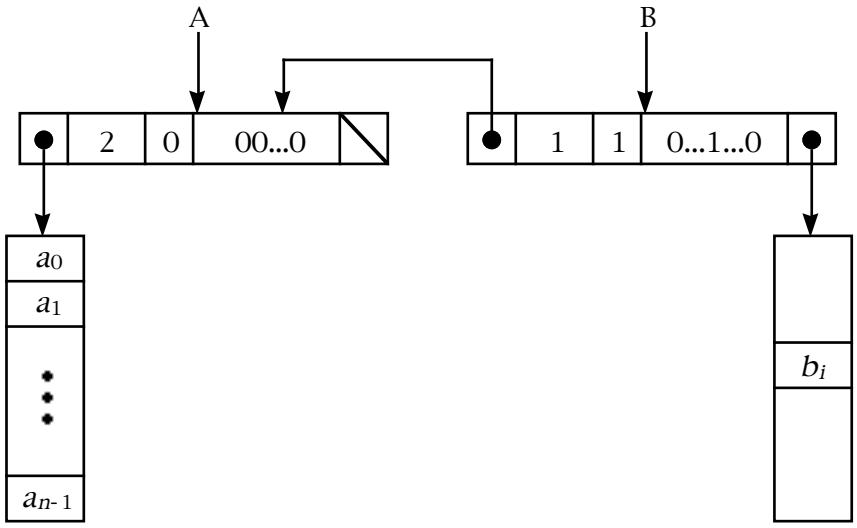


Figure 23: An example of an APPEND operation.

An example of a simple operation, B[i] = *f*(A[i]), is illustrated in Figure 23. Initiation of this operation first requires the selection of the element a_i from array A—i.e., SELECT. After performing the required function *f*, an APPEND operation then generates a new hybrid structure with its right pointer pointing to a vector containing only the modified element b_i and its left pointer pointing back to the original hybrid structure. If an append operation is performed on a modified array (status bit equal to "1"), all the existing elements in the modified array are copied to the new hybrid structure before the element is appended. In the cases where the reference counts of arrays to be modified are ones, append operations are simplified since the elements cane be "modified-in-place" without introducing any side-effects.

The hybrid scheme also handles non-strict operations—such as in I-structures. As discussed before, I-structures exploit producer-consumer parallelism by allowing the selection of elements before the entire structure is constructed. In order to preserve the determinacy property of the dataflow model, the synchronization of SELECTs and APPENDs is provided by the use of presence bits. In the hybrid scheme, the same kind of information is embedded in the structure template when an array is defined. Therefore, whenever the compiler detects a construct which is suitable as an I-structure producer or explicitly declared by the programmer as non-strict, a structure template is defined with the status bit set to zero, location bits all set to 1's, and an empty vector linked to the left pointer. Once an I-structure is allocated, the pointer token is immediately released. As the elements are appended to the I-structure, the corresponding location bit is reset to zero indicating the availability of datum. The synchronization between a producer and a consumer is provided by examining the status bit *S* and the location bit *i* to determine how the element *i* is accessed and whether it is ready for consumption (see Table 2). If there is a

request to select an element which has not yet been appended, the request is deferred to a linked-list.

TABLE 2: The meaning of the *S* and *i* bits.

Bits	Meaning
00	The element is in the vector pointed to by the left pointer
01	The element has not yet been appended (the request is deferred).
10	The element is in the hybrid structure pointed to by the left pointer
11	The element is in the vector pointed to by the right pointer

Note that every APPEND operation to an array with a reference count greater than one requires the allocation of an entirely new array. This is appropriate if a) the arrays are small, or b) all or most of the allocated vector elements are eventually used to hold the new values—e.g., updating an entire row or column of an array. Unfortunately, this is not always the case; it is obviously impractical in terms of memory requirements to allocate the entire array for a single random append. Therefore, the hybrid scheme allows array elements to be partitioned into blocks and then an access table is used to keep a list of pointers to all the blocks associated with a modified array. By utilizing an access table, a search operation for a block which contains the desired element takes a constant time. In addition, the concept of "allocation as required" is employed to reduce the memory overhead.

V.2 Issues in Handling Data Structures

In the previous subsection, various data structure representations were surveyed. Despite the advantages brought forth by each method, it is apparent that no single proposed method succeeds in handling data structures for all instances of program constructs. All the proposals have succeeded in preserving

functionality, which is crucial to the dataflow model of computation. However, in pursuing the optimum data structure representation, several implementation choices can be characterized.

A comparison between direct and indirect access schemes can be made based on several implementation issues. Most important, in many program constructs, a better speedup can be obtained due to the simplification of access pattern since data tokens need not transit through a separate structure controller. In addition, the elimination of a structure storage reduces the overall hardware complexity as well as the communication network load. Only a careful design in the compiler is needed to generate the relabeling functions.

However, the aforementioned advantages do not totally overshadow the importance of indirect access schemes. There are desirable properties for having data structures temporarily stored in a separate storage. First of all, retaining the concept of arrays as a single entity (at least conceptually) is important in order to maintain programmability. Second, in many applications, the replication of arrays is often needed and the structure storage provides the opportunity for performing multiple reads. Third, storing large arrays in a separate storage reduces the resource requirement for a frame store (see Section VII). Finally, one of the attractiveness of the dataflow model of execution is the ability to tolerate long latency incurred during a structure operation. Therefore, given sufficient amount of parallelism to keep the processor busy, these latencies can be masked by other independent computations (i.e., context switching).

The indirect access approach, however, is not free of obstacles. When array elements are stored in a separate storage, in order to preserve the functionality principle of dataflow, even a simple operation which modifies an element of an array requires, at least conceptually, copying the entire array except for the element specified by the index. This results in excessive processing

and storage overhead. As a result, one major implementation issue is how can copying be reduced without violating the functionality principle of dataflow?

The indirect access schemes surveyed here attempt to alleviate complete copying while preserving the semantics of the program during array update operations. Rather than copying elements which have not been updated, they are shared between the original and the updated structure. Heaps attempt this by having common substructures to be shared. The EXMAN approach and the Hybrid Scheme provide an overlapping effect of array elements which are not modified. Only the updated array elements are made distinct; the EXMAN approach uses linked-lists while the Hybrid Scheme utilizes arrays. In general, these approaches reduce the amount of copying and storage requirements.

Another important issue, which directly affects the performance of data structure operations, is the comparison between *strict* and *non-strict* structures. According to the dataflow principles, no element of a structure can be accessed before its creation. This means that the pointer token is not released until all the elements of a structure are constructed. Although the strictness property conforms well to dataflow semantics, in many cases it reduces the potential parallelism. Non-strict operations, on the other hand, improve performance by allowing an element of a structure to be consumed before its complete production. This type of structure is especially useful in exploiting parallelism between the producer and the consumer of an array. One inherent problem with non-strict structures is that in many situations an element is accessed before it is actually stored. Handling the deferred accesses is expensive in terms of the hardware overhead as well as performance.

The choice between strict and non-strict structure still remains a difficult one. Although strict operations decrease parallelism, they make collection of termination signals easier [40]. From a performance point of view, unnecessary

delay should be avoided whenever possible. Thus, a data structure handling scheme should be flexible so that both strict and non-strict structures can be implemented.

Finally, one problem which is common to all the methods based on the indirect access scheme is the issue of memory management. Due to its fully distributed nature, implementation of data structure in a dataflow environment requires a memory system capable of handling concurrent accesses. This requires careful considerations of issues such as memory organization, dynamic memory allocation, and garbage collection. Another issue which greatly influences the overall performance is the partitioning and the distribution of data structure elements among the structure memory. The distribution of data structures is strongly influenced by the organization of the target machine (e.g., the type of interconnection network), the program characteristics, and the program allocation scheme used to partition and assign subgraphs to processors. It is clear that the solution to the problem of distributing data structures remains a difficult one and therefore requires better knowledge of concrete program characteristics—the amount and kind of parallelism in a program and the type of structure accesses performed during program execution.

VI. PROGRAM ALLOCATION

Similar to the conventional multiprocessors, the issue of program allocation is also of major interest to dataflow multiprocessors. The proper allocation of tasks to processing elements (PEs) has direct effect on the overall performance of dataflow computers. Therefore, in this section the issues involved in developing an allocation scheme that maximizes the utilization of parallelism embedded in the dataflow model of computation are discussed.

Two major issues in allocation of programs can be identified as *partitioning* and *assignment* [27]. Partitioning is the division of an algorithm into procedures, modules and processes. Assignment, on the other hand, refers to the mapping of these units to processors. The ultimate goal is to develop an efficient allocation scheme which avoids processor contention and at the same time reduces the overall network communication. However, since these two goals are in direct conflict with each other, the problem is not a trivial one. The allocation problem is further complicated due to the existence of variety of architectural differences as well as interconnection topologies. Therefore, a single allocation scheme which achieves high performance for all classes of dataflow computer is a difficult if not an impossible task.

There are two main approaches to allocating subtasks of a dataflow graph: static and dynamic [27]. In *static* allocation, the tasks are allocated at compile-time to processors using global information about the program and system organization. The allocation of a given program is done only once even though it may be executed repeatedly. However, one major drawback of the static allocation policies is that they are inefficient when estimates of run-time dependent characteristics are inaccurate. A *dynamic* allocation policy, on the other hand, is based on measuring loads at run-time depending on the program behavior and assigning activated tasks to the least loaded processor. The

disadvantage of a dynamic scheme is the overhead involved in determining the processor loads and allocation of tasks at run-time.

VI.1 Proposed Allocation Schemes

In light of the discussion presented previously, it is apparent that there are two common goals in the allocation of programs: maximizing the inherent concurrency in a program graph by minimizing contention for processing resources. It has been shown that obtaining an optimal allocation of a graph with precedences is NP-complete problem [37]. Therefore, heuristic solutions are the only possible approach to solving the allocation problem.

A number of heuristic algorithms have been developed for the allocation problem based on *critical path list schedules* [1, 30]. The basic idea behind the critical path list scheduling is to assign each node of a directed graph a weight that equals the maximum execution time from that node to an exit node (i.e., critical path). An ordered list of nodes is constructed according to their weights, which is then used to dynamically assign nodes with highest weights to processors as they become idle.

One major problem with critical path list schedules is a result of communication delays among the nodes. Enforcing only critical path scheduling, without considering the communication overhead associated with the predecessor-successor nodes assigned to different processors, will not necessarily minimize the overall execution time.

In response to this shortcoming, a number of allocation schemes have been proposed which considers the effects of communication costs. One such a scheme, which is a variation of the critical path list scheduling, was proposed by Ravi *et al.* [38]. In this method, rather than simply choosing the topmost node on the list, several top *candidates* whose critical paths fall within a certain range are

considered for allocation. From this set of candidates, a node is selected which maximizes savings in communication time. To determine the actual execution time (i.e., computation and communication time) of a node, the program graph is reversed. This procedure allows the correct communication costs to be associated with particular nodes, since a node is allocated after its actual successors.

It is important to note that the effectiveness of the scheme depends primarily on the range chosen for selection of a node. It has been shown that, when the deviation is chosen to be very large, the scheduling of nodes no longer conforms to the critical path and the nodes are allocated based only on the minimization of communication delays. On the other hand, when the set of candidates is chosen to be very small, the effectiveness of minimizing the communication costs diminishes [38]. Therefore, the success of this algorithm relies on a parameter which is difficult to generalize for arbitrary program graphs and their implementations on different underlying architectures.

To determine the proper compromise between computation and communication costs, Lee *et al.* proposed the Vertically Layered (VL) allocation scheme [31]. The VL allocation scheme consists of two separate phases: separation and optimization phase. The basic idea behind the separation phase is to arrange nodes of a dataflow graph into vertical layers such that each vertical layer can be allocated to a processor. This is achieved by utilizing Critical Path (CP) and Longest Directed Paths (LDP) heuristics. These heuristics give the highest priority to the critical path for allocation and then recursively determine the vertical layers by finding longest directed paths emanating from the nodes which have already been assigned to vertical layers. A *density factor* is then used to determine how the longest directed paths are assigned to vertical layers. Therefore, the CP and LDP heuristics minimize contention and inter-processor

communication time by assigning each set of serially connected nodes to a single PE.

The separation phase initially attempts to minimize the inter-PE communication costs by identifying as many longest directed paths as possible for allocation to PEs. Unfortunately, this is done with very little information about the communication overhead. Once the initial allocation is completed, more information regarding the relative assignment of predecessor-successor nodes and their inter-PE communication costs is known. Therefore, the Communication to execution Time Ratio (CTR) heuristic algorithm utilizes this information to optimize the final allocation. This is done by considering whether the inter-PE communication overhead offsets the advantage gained by overlapping the execution of two subsets of nodes in separate processing elements.

To apply the CTR heuristic algorithm, the execution time of a new critical path which includes the effects of inter-PE communication costs is determined. The type of communication behavior which resulted in the new critical path is then considered. If an improvement results after applying the CTR heuristic, the nodes are combined into a single PE. Since combining two parallel subsets of nodes into a single processing element forces them to be executed sequentially, a new critical path may emerge from the optimization process. Therefore, this process is repeated in an iterative manner until no improvement in performance can be obtained by combining two subsets of nodes associated with the critical path.

To illustrate the VL allocation scheme, consider a dataflow graph shown in Figure 24 which is to be allocated to two PEs with inter-PE communication cost of 5 time units. The bold letters indicate the execution times of the nodes After performing the partitioning phase using CP and LDP heuristics, the

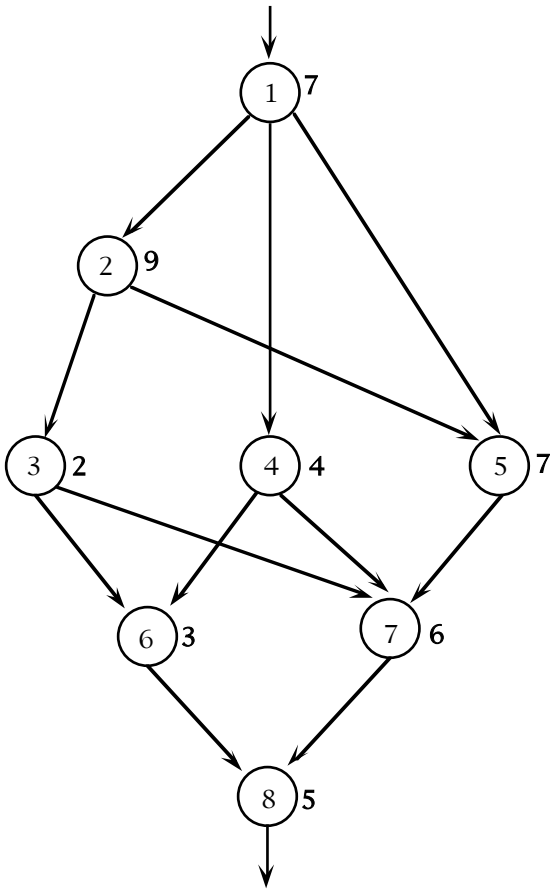


Figure 24: An example of a dataflow graph. Bold letters indicate the execution times of the nodes.

following vertically layered graph is obtained (see Figure 25.a). Note that the critical path length has changed due to the inter-PE communication costs from nodes 2 to 3 and nodes 3 to 7. If the node 3 is merged together with the node 5 using CTR heuristic, the overall execution time is improved (Figure 25.b). Note that no further improvements can be made and therefore the algorithm terminates

Performance analysis indicates that the proposed scheme is very effective in reducing the communications overhead. For comparison, the VL allocation scheme showed considerable improvement over the critical path list schedule in the presence of inter-PE communication delays [31]. However, one potential drawback exists in the VL allocation scheme. When the number of available PEs is small, the VL allocation scheme assigns a number of LDPs to the same PE without considering the interaction between the LDPs. Although, the VL allocation scheme succeeds in balancing the load among the PEs, it may not always reduce the total execution time.

A more general approach to program allocation was proposed by Sarkar and Hennessy [42]. This method also involves determining a partition that minimizes the critical path length of a program graph. However, in contrast to the VL allocation scheme which utilizes the CTR heuristic algorithm to reduce communication costs, the scheme uses a greedy approximation algorithm. The algorithm begins with the trivial partition that places each node in a separate *block*. A table which represents the decrease in the critical path length obtained from merging a pair of blocks is maintained. It then iteratively merges the pair of blocks that results in the largest decrease in the critical path length. The algorithm is terminated when no remaining merger could possibly reduce the critical path length.

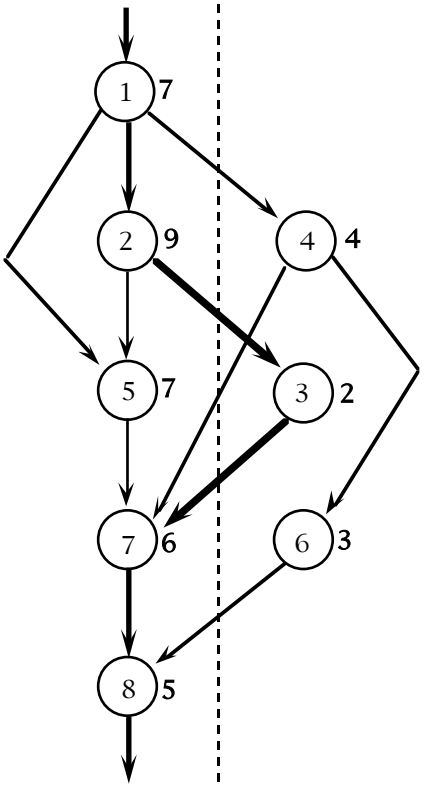


Figure 25.a: Vertically Layered graph of Figure 24 on two PEs with inter-PE communication delay of 5 time units. The new critical path length is 39 time units.

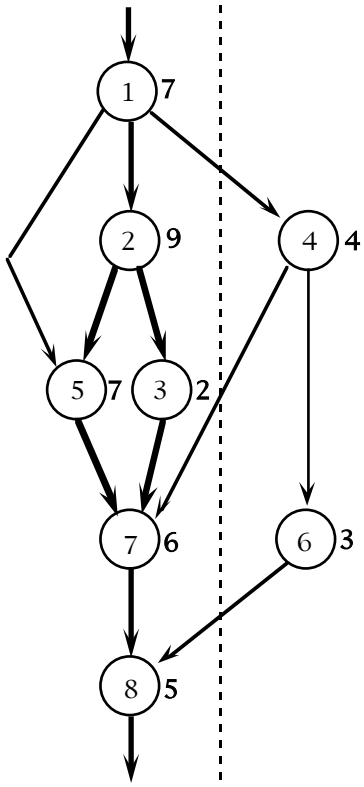


Figure 25.b: After merging nodes 3 and 5 on the same PE using CTR heuristics. The new critical path length is 36 time units (worst case).

VI.2 Issues in Program Allocation

Although a number of proposed allocation schemes has been reviewed, one major problem still remains unresolved—the issue of handling dynamic parallelism. As discussed in Section II, the dynamic dataflow model of execution permits simultaneous firing of multiple instances of a node. In theory, this increases both the asynchrony and parallelism of dataflow graphs. For example, consider the implementation of a loop schema. A dynamic architecture unfolds the loop at run-time by generating multiple instances of the loop body and attempts to execute the instances concurrently. However, since a single PE does not allow two simultaneous executions of a node, mapping the source dataflow graphs to processors without special provisions results in the inability to fully exploit the maximum parallelism.

One solution is to provide a code-copying facility, where an instruction within a code block is duplicated among the available resources. Arvind has proposed a mapping scheme in which the instructions within a code block (called the logical domain) are mapped onto available PEs (called the physical domain) based on a hashing scheme [11]. For example, if a physical domain consists of n PEs, then the destination PE number can be $PE_{base} + i \bmod n$, where i is the iteration number. This will distribute the code uniformly over the physical domain. Since each of the n PEs has a copy of the code, n iterations may be executed simultaneously. However, since not all program constructs can be unfolded in this manner, the question still remains as to how effectively dynamic parallelism can be detected at compile-time.

Another important issue to consider in program allocation is the granularity of partitions. Although the dataflow model of computation exposes fine-grain parallelism embedded in a program, in practice there are advantages to adopting a more coarse-grain approach. Most important, to reduces

communication costs between predecessor-succesor nodes by having a group of instructions execute on the same processor. The question then becomes what is an appropriate granularity? Due to the syntax structure of programs, code blocks (i.e., instances of a loop or user defined functions) adheres naturally to decomposition. Such partitions can also simplify resource management involved in allocation of frames associated with code blocks [14]. However, since a single code block may contain considerable amount of parallelism, such a coarse-grain partition may severely degrade the performance.

Sarkar and Hennessy suggest that the optimal granularity should be dictated the trade-off between parallelism and the overhead of exploiting parallelism [43]. As mentioned before, fine granularity execution is inefficient on dataflow multiprocessors due to enormous scheduling and communication overhead. On the other hand, determining the level of granularity by language constructs causes the programming style to drastically affect the system performance. In their approach, the optimal granularity is dictated by performance characteristics. The proposed compile-time partitioning algorithm determines each partition based on the *cost function* that considers execution time, communication overhead, and scheduling overhead. The algorithm begins with a trivial partition that places each node in a separate subgraph. For each iteration, a subgraph with the largest cost function is merged with another subgraph that yield the smallest cost function. At the same time, a cost history is maintained. This is repeated until all the nodes have been merged into a single subgraph. The cost history is then used to identify and reconstruct the iteration with the best partition.

VII. RESOURCE REQUIREMENTS IN DATAFLOW

The major motivation behind dataflow research is to expose as much parallelism as possible in a given program. However, studies have shown that the dataflow execution model may be too successful in this regard [14]. In general, parallel execution requires more complex resource management than sequential execution. As parallelism increases, more resources are required to fully exploit such concurrency. If a program contains excess amount of parallelism, precautionary measures must be taken to prevent the machine from saturation and even deadlock. Ideally, sufficient parallelism should be exposed to fully utilize the machine on which the program is running, while minimizing the resource requirement of the program. In this section, the issue of resource management is discused.

In the dataflow model of execution, the storage model is represented as a tree of activation frames. A code block, such as loops, may be unfolded allowing many iterations to be active at the same time; therefore, the invocation tree can branch with arbitrary large degree. Variables in a dataflow program denote arcs in the graph. Although a token carries a data on an arc, a certain amount of storage is implicitly required to represent the data value. For example, consider TTDA which employs Waiting-Matching units. Tokens which are waiting for partners occupy storage in the Waiting-Matching unit. Even architectures which utilize direct matching schemes must allocate storage in the frames. Thus, one measure of the storage requirement of a dataflow program is the number of tokens in existence.

A study performed by Arvind and Culler shows that under the ideal case (i.e., unlimited resources and zero communication latencies) considerable parallelism exists in many programs [14]. Of course, such excessive parallelism cannot be fully exploited in practice. Even when the number of processors is

reduced, the storage requirement for waiting tokens will not necessarily decrease. This is due to the fact that under the dynamic dataflow model of execution there exists a significant amount of exposed yet unexploited parallelism. Limiting the number of processors constrains the actual number of executable nodes; however, the remaining nodes are scheduled with resource requirements close to that of the ideal case.

In order to reduce the resource requirements of a program, constrain must be placed on its execution so that less parallelism is exposed. Culler and Arvind suggested placing a limit on the unfolding of loops which is the source of parallelism in many programs [14]. This is achieved by introducing an artificial data dependence between iterations. An addition gate is place on the output of the predicate operator which inhibits new iterations from starting unless a *trigger* token is present at the control input to the gate. The bounds on unfolding can be determined at run-time according to the problem size and number of processors.

The loop bounding approach to reduce the resource requirements of dataflow program still require extensive research. Thus, one of the main challenge that remains is the appropriate bounding of loops to reduce useless parallelism. In other words, when a program unfolds, a major portion of it allocates resources and then wait until data becomes available. The proper detection of this useless parallelism for a broad class of programs will be the key to allowing large scientific programs to be executed effectively on dataflow machines.

VIII. CONCLUSION

There is no question that dataflow computing has matured considerably in the past decade. With the knowledge gained from the pioneering works such as Static Dataflow Machine and TTDA, dataflow computing has experienced

tremendous growth. In addition, the ground breaking advances from the development of current dataflow projects indicate that high performance dataflow machines may soon become a reality. However, before a successful implementation of dataflow machines is possible, the various issues discussed in this article must be resolved.

The handling of data structures still remains a formidable problem to resolve. It is clear that the solutions presented in this article do not fully satisfy all the requirements of the dataflow model of execution. For example, the concept of the direct access scheme (i.e., token relabeling scheme) is more consistent with the semantics of the dataflow model resulting in improved performance due to simplification of access patterns. On the other hand, indirect access schemes provide better resource requirements by allowing multiple reads from a separate structure storage.

In the past, a major emphasis has been place on data structure representations which are based on indirect access schemes [18, 30]. However, with the development of the token relabeling scheme, direct access approach is viewed as a practical alternative to the traditional indirect access methods. There is obviously no question that a more extensive research is required to determine the appropriate strategy for representing data structures—direct or indirect access. Many advocates of indirect access schemes suggest that the ability of dataflow computers to tolerate long latencies will mask the overhead involved during data structure operations. Therefore, a major study is needed to evaluate the tradeoff between direct and indirect schemes in order to utilize advantages offered by both methods. For example, the direct scheme is appropriate for most array operations while indirect schemes are suitable for pointer accessing or table lookup applications [19]. This will require a careful analysis of concrete program characteristics—the amount and kind of parallelism in a program, the size of data

structures, and the kinds of structure accesses performed during program execution.

The issue of program allocation is as old as parallel processing. With the current dataflow proposals adapting a multiprocessor organization, the proper partitioning of programs into modules or processes and mapping of these units to processors will have a direct implication on the success of dataflow computers. The main objective in program allocation is to maximize the parallelism in a program while minimizing the communication costs between predecessor and successor nodes. In order to achieve this compromise effectively, an appropriate granularity must be chosen for the partitions which can maximally utilize the architectural features of the target machine—e.g., the type of interconnection network and the organization of the processing element.

Handling dynamic parallelism is another area in program allocation which requires additional research. An analysis of static dataflow graph representation of a program reveals only a portion of the potential parallelism. A simple solution of providing a code-copying facility may cause resources to be overcommitted. Therefore, a careful coordination of theses two interrelated issues (program allocation and resource requirements) is required to successfully manage and exploit the ample parallelism that exists in many programs.

It is clear that the issues discussed in this article are all important problem to resolve before dataflow computers can fulfill the promise of delivering substantial computing power. The topics presented in this article are by no mean exhaustive. However, it is also clear that the problem of handling data structures, program allocation, and resource management pose an arduous challenge to dataflow researchers. These crucial issues will require an extensive study of program behaviors to provide a better assessment of the problems.

IX. REFERENCES

[1] Ackerman, W. B. and Dennis, J. B., "VAL—A Value-Oriented Algorithmic Language: Preliminary Reference Manual," *MIT Laboratory for Computer Science Technical Report TR-218*, MIT, Cambridge, Mass., June 1979.

[2] Ackerman, W. B., "A Structure Processing Facility for Dataflow Computers," *Proc. of the International Conference on Parallel Processing*, Aug. 1978, pp. 166-172.

[3] Ackerman, W. B., "Dataflow languages," *Computer*, Vol. 15, No. 2, Feb. 1982, pp. 15-23.

[4] Allan, S. J. and Oldehoeft, "A Flow Analysis Procedure for the Translation of High Level Language to a Dataflow Language," *Proc. of the International Conference on Parallel Processing*, August 1979, pp. 26-34.

[5] Arvind and Culler, D. E, "Dataflow Architectures," *Annual Review in Computer Science*, 1986, Vol. 1, pp. 225-253.

[6] Arvind, Culler, D. E., and Ekanadham, K., "The Price of Fine-Grain Asynchronous Parallelism: An Analysis of Dataflow Methods," *Proc. CONPAR 88*, Sept. 1988, pp. 541-555.

[7] Arvind and Goestelow, K. P., "The U-Interpreter," *Computer*, Vol. 15, No. 2, Feb. 1982, pp. 42-50.

[8] Arvind and Iannucci, R. A., "A Critique of Multiprocessing von Neumann Style," *Proc. 10th Annual Symposium Computer Architecture*, June 1983, pp. 426-436.

[9] Arvind and Thomas, "I-Structures: An Efficient Datatype for Functional Languages," *MIT Laboratory for Computer Science Technical Report TM-178*, Cambridge, Mass., Sept. 1980.

[10] Arvind and Nikhil, R. S., "Executing a Program on the MIT Tagged-Token Dataflow Architecture," *Proc. Parallel Architectures and Languages, Europe*, June 1987, pp. 1-29.

[11] Arvind, "Decomposing a Program for Multiprocessor System," *Proc. of the International Conference on Parallel Processing*, 1980, pp. 7-14.

[12] Arvind, Nikhil, R. S., and Pingali, K. K., "I-structures: Data Structures for Parallel Computing," *Proceedings of the Workshop on Graph Reduction*, Los Alamos, NM, 1986.

[13] Cornish, M., "The TI Dataflow Architecture—The Power of Concurrency for Avionics," *Proceedings of Third Conference on Digital Avionics Systems*, 1979, pp. 9-26.

[14] Culler, D. E. and Arvind, “Resource Requirements of Dataflow Programs,” *Proc. 15th Annual Int'l. Symposium on Computer Architecture*, 1988, pp. 141-150.

[15] Culler, D. E. *et al.*, “Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstracted Machine,” *4th Int’l. Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991.

[16] Dennis, J. B., "Data-Flow Supercomputers," *Computer*, Nov. 1980, pp. 48-56.

[17] Gajski, D. D., Padua, D. A., Kuck, D. J., and Kuhn, R. H., "A Second Opinion on Dataflow Machines and Languages," *Computer*, Feb 1982, pp. 58-69.

[18] Gaudiot, J. -L., "Structure Handling in Data-Flow Systems," *IEEE Transactions on Computers*, Vol. C-35, No. 6, June 1986, pp. 489-502.

[19] Gaudiot, J. -L. and Wei, Y. H., "Token Relabeling in a Tagged Token Data-Flow Architecture," *IEEE Transactions on Computers*, Vol. 38, No. 9, Sept. 1989, pp. 1225-1239.

[20] Ghosal, D. and Bhuyan, L. N., "Analytical Modeling and Architectural Modifications of a Dataflow Computer," *Proc. 14th Annual Int'l. Symposium on Computer Architecture*, 1987, pp. 81-89.

[21] Grafe, V. G. and Hoch, J. E., "Implementation of the εpsilon Dataflow Processor," *Proc. 17th Annual Int'l. Symposium on Computer Architecture*, 1990, pp. 19-29.

[22] Grafe, V. G. and Hoch, J. E., "The Epsilon-2 Multiprocessor System," *Journal of Parallel and Distributed Computing*, 10, 1990, pp. 309-18.

[23] Grafe, V. G. *et al.*, "The εpsilon Dataflow Processor," *Proc. 16th Annual Int'l. Symposium on Computer Architecture*, 1989, pp. 36-45.

[24] Gurd, J. R., Kirkham C. C. and Watson, I., "The Manchester Prototype Data-Flow Computer," *Commun. ACM*, Vol. 28, pp. 34-52, Jan. 1985.

[25] Hurson, A. R., Lee, B., and Shirazi, B., "Hybrid Structure: A Scheme for handling Data Structures in a Dataflow Environment," *Proc. Parallel Architectures and Languages, Lecture Notes in Computer Science*, Vol. 365, 1989, pp. 323-340.

[26] Hurson, A. R., Lee, B., Shirazi, B., and Wang, M. "A Program Allocation Scheme for Dataflow Computers," *Proc. 1990 Int'l. Conf. on Parallel Processing*, pp. 415-423.

[27] Hwang, K. and Briggs, F. A., Computer Architecture and Parallel Processing. New York: McGraw-Hill, 1981.

[28] Iannucci, “Towards a Dataflow/von Neumann Hybrid Architecture,” *Proc. 15th Annual Int'l. Symposium on Computer Architecture*, 1988, pp. 131-140.

[29] Kohler, W. H., "A Preliminary Evaluation of Critical Path Method for Scheduling Tasks on Multiprocessor Systems," *IEEE Transactions on Computers*, Dec. 1975, pp. 1235-1238.

[30] Lee, B. and Hurson, A. R., "A Hybrid Scheme for Processing data Structures in a Dataflow Environment," *IEEE Transactions on Parallel and Distributed Systems*, accepted, 1990.

[31] Lee, B., Hurson, A. R., and Feng, T. Y. "A Vertically Layered Allocation Scheme for Dataflow Computers," *Journal of Parallel and Distributed Computing*, Vol. 11, 1991. pp. 175-187.

[32] McGraw, J. R. *et al.*, "SISAL: Streams and Iteration in a Single-Assignment Language," *Language Reference Manual, Version 1.2 M-146.*, Lawrence Livermore National Lab., March, 1985.

[33] Nikhil, R. S. and Arvind, "Can Dataflow Subsume von Neumann Computing?," *Proc. 16th Annual Int'l. Symposium on Computer Architecture*, 1989, pp. 262-272.

[34] Nikil, R. S., "Id World Reference Manual" MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA.

[35] Papadopoulos, G. M. and Culler, D. E., "Monsoon: An Explicit Token Store Architecture," *Proc. 17th Annual Int'l. Symposium on Computer Architecture*, May 1990.

[36] Patnaik, L. M., Govindarajan, R., and Ramadoss, N. S., "Design and Performance Evaluation of EXMAN: An EXtended MANchester Dataflow Computer," *IEEE Transactions on Computers*, Vol. C-35, No. 3, March 1986, pp. 229-243.

[37] Polychronopoulos, C. D. and Banerjee, U., "Processor Allocation for Horizontal and Vertical Parallelism and Related Speedup Bounds," *IEEE Transactions on Computers*, Vol. C-36, No. 4, April 1987, pp. 410-420.

[38] Ravi, T. M., Ercegovac, M. D., Lang, T., and Muntz, R. R., "Static Allocation For a Dataflow Multiprocessor System," *2nd Int'l. Conference on Supercomputing*, May, 1987.

[39] Sakai, S. *et al.*, "An Architecture of a Dataflow Single Chip Processor," *Proc. 16th Annual Int'l. Symposium on Computer Architecture*, 1989, pp. 46-53.

[40] Samet, S. and Gokhale, M., "Data Structures On Dataflow Computers: Implementations and Problems," *IEEE 1984 Micro-Delcon*, pp. 84-96.

[41] Sargeant, J. and Kirkham, C. C., "Stored Data Structures on the Manchester Dataflow Machine," *Proc. 1986 Symposium on Computer Architecture*, pp. 235-242.

[42] Sarkar, V. and Hennessy, J., "Compile-Time Partitioning and Scheduling of Parallel Programs," *Proc. SIGPLAN '86 Symposium on Compiler Construction*, 1986, pp. 17-26.

[43] Sarkar, V. and Hennessy, J., "Partitioning Parallel Programs for Macro-Dataflow," *ACM Conf. on Lisp and Functional Programming*, 1986, pp. 202-211.

[44] Sirini, V. P., "An Architectural Comparison of Dataflow Systems," *Computer*, Vol. 19, No. 3, March 1986, pp. 68-86.

[45] Shimada, T. et al., "Evaluation of a Prototype Data Flow Processor of the SIGMA-1 for Scientific Computations," *Proc. 13th Annual Int'l. Symposium on Computer Architecture*, 1986, pp. 226-234.

[46] Veen, A. H., "Dataflow Machine Architecture," *Computing Surveys*, Vol. 18, No. 4, December 1986.

[47] Vegdahl, S. R., "A Survey of Proposed Architectures for the Execution of Functional Languages," *IEEE Transaction on Computers*, Vol. V-33, No. 12, Dec. 1984, pp. 1050-1071.

[48] Wang, M., Lee, B., Shirazi, B., and Hurson, A. R., "Accurate Communication Cost Estimate in Static Task Scheduling," *24th Hawaii International Conference on System Science*, 1991, pp. 10-16.

[49] Yamaguchi, Y. *et al.*, "An Architectural Design of a Highly Parallel Dataflow Machine," *Proc. IFIP Congress 1989*, pp. 1155-1160.