LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

The Evolution of Dataflow Architectures from Static Dataflow to P-RISC

Computation Structures Group Memo 316
August 2, 1990

Arvind

Stephen Brobst

To appear in *Proceedings of Workshop on Massive Parallelism: Hardware, Programming and Applications, Amalfi, Italy, October 1989.* Academic Press, 1990.

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory for Computer Science is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-89-J-1988.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

The Evolution of Dataflow Architectures from Static Dataflow to P-RISC

Arvind and Stephen Brobst
Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square
Cambridge, Massachusetts 02139

1 Introduction

Dataflow architectures have undergone substantial evolutionary changes since they were first proposed fifteen years ago. There has been a consistent convergence toward a "practical" architectural framework for implementing dataflow machines. In this paper we trace the evolution of dataflow architectures since their origin. We identify what we believe have been the most significant milestones in this evolution, and give examples of the underlying theme to these important advances.

Section 2 of the paper discusses potential pitfalls in designing a fundamentally new architecture. In Section 3, we examine the origins of dataflow computing with static dataflow computers. Section 4 traces the evolution of data structures in the context of dataflow architectures. The evolutionary steps in the development of dynamic dataflow architectures are presented in Section 5. The dataflow/von Neumann hybrid architecture, P-RISC (for parallel RISC), is introduced in Section 6 as a new phase of evolution in dataflow architectures. Section 7 concludes with our prognosis of dataflow architectures.

2 How to Go Wrong in Designing a New Architecture

Designing a fundamentally new computer architecture is an inherently difficult task. The field of computer science is littered with computer architectures which have failed for various reasons. Many of these architectures were extremely well-implemented, but never-the-less fell short of the expectations of their architects in terms of overall success. In general, our

observations indicate that fundamentally new computer architectures go wrong[1] because of one or more of the following factors:

- An architecture is likely to fail if it leaves the programmability issue as an afterthought.
- An architecture is likely to fail if it institutes overly complex hardware mechanisms to support its intended programming model.
- An architecture is likely to fail if it ignores compatibility with existing computing environments.

In this section we will discuss each of these factors in detail. In the following sections we will look at the evolution of dataflow architectures and how various improvements address shortcomings related to these factors.

2.1 Leave the Programmability Issue as an Afterthought

Many computer architectures are driven via a bottom-up approach. Perhaps the best example of this phenomena is given by the proliferation of microprocessor-based parallel computer architectures. The economic allure of VLSI microprocessor technology, along with the high availability of such parts, gives very strong motivation to architects for interconnecting n 68000s or Transputers on a bus or network. This bottom-up approach is an important force in computer architecture because it often uncovers new and innovative uses of emerging (and existing) technologies.

The main problem with this approach, however, is that the programmability of such architectures is usually left as an afterthought. A technology-based architecture, by its very nature, will generally result in a very powerful computing engine in terms of raw processing power. For example, it is clear that there is plenty of processing power available by interconnecting any of the high-end microprocessors currently on the market [41, 10, 42]. However, automatic partitioning of a large application and hand-mapping parallel tasks to multiple processors has turned out to be a very difficult undertaking. Even when successful, the peculiarities of an architecture are often reflected in such a way that code maintenance becomes inherently difficult.

That is not to say that all technology driven architectures end up without a usable programming paradigm. Take the Connection Machine [27], for example. This SIMD machine, consisting of 64K bit-serial processors and a novel network architecture, was made possible specifically due to the wide availability of tools to design custom VLSI chips. In the Connection Machine the programming model was also somewhat of an afterthought, but has

[1] For the purposes of this paper we are considering only those computer systems which are well-implemented, not those which fail due to poor engineering, technology choices, etc.

been extremely successful never-the-less. The programming model and application domain originally proposed for the machine [26] had very little impact on its development. Moreover, this original programming model has very little resemblance to the *data parallel* [28] model currently in use. The data parallel model, through use of virtual processors and algorithms based on parallel prefix computations, has been extremely successful in harnessing the underlying hardware of the Connection Machine. It is unlikely that this approach would have been developed without the motivation provided by a "real" machine. These developments are also a credit to the ingenuity of the persons who have programmed and contributed to the development of the Connection Machine over the years.

We believe that the fundamental reason behind the success of the data parallel model is that it simultaneously abstracts a lot of detail regarding the underlying hardware without sacrificing the programmer's intuition about the efficiency of an algorithm's implementation on the machine. Moreover, this model has applied well across a relatively wide range of applications. Variations on this model have even begun to be applied in large MIMD machines such as the Hypercube [20].

However, it is not always the case that such an elegant programming paradigm emerges. The Illiac IV [11], for example, is a relatively unsuccessful architecture which is very similar to the Connection Machine in both its intent and motivations. Unfortunately, a suitable programming paradigm was never quite developed for the Illiac IV. There was no means of harnessing the processing power of the machine without building programs that explicitly mapped to the network topology and number of processors. Although Illiac IV had a number of hardware implementation problems, there is no doubt that lack of a high-level programming model was instrumental in the demise of this architecture.

The bottom-up approach to development of new computer architectures is an important force for leveraging new technologies. However, the tendency to leave the programmability issue as an afterthought in such architectures is extremely risky. A programming paradigm appropriate to a new architecture, if left to ad-hoc development, may or may not emerge.

This phenomena is the least of our worries in the development of dataflow architectures; from the very beginning our approach has been language-directed, so the programming paradigm is at the forefront of the architectural developments. Of course, many technology-based architectures emerge with elegant programming paradigms as well. The Connection Machine and Hypercube are just two of many examples in which programming methodologies have been developed to take advantage of new architectures. Yet there is always the danger that such clever solutions will not emerge, once the technology is fully in place. Herein lies the warning behind our first observation of how to go wrong in developing a new computer architecture.

2.2 Build Complex Hardware to Support the Programming Model

The opposite extreme to the dilemma raised in the previous section is to be influenced too far by the programming model. The problem is that if the architect goes too far in providing support for a particular programming paradigm, the complexity and efficiency of the architecture will suffer. The Intel iAPX432 [31] is a prime example of this phenomena. The Intel machine implemented what was close to a silicon operating system for managing capabilities in an object-oriented programming model. Although the objective was noble, the magnitude of hardware support overwhelmed the architecture and resulted in an unbearably inefficient machine implementation. This is an easy trap to fall into whenever the development of a language-directed architecture is undertaken.

The challenge to the architect is to provide the "right" set of hardware primitives to support the programming paradigm, but not to end up with a hardware implementation of the model. This is a very tricky issue because it is often difficult to determine the "right" set of hardware primitives. The architect must bare in mind that the concrete implementation of an abstract programming model does not necessarily require that the model be fully implemented in hardware. Tools such as compilers, when coupled with simple hardware primitives, can go a long way toward realization of a powerful programming model without overly complex hardware support.

Support of Algol's high-level programming model presented exactly these challenges to architects in the early 1960's. The high-level programming paradigm with lexical scoping and other innovations in language design explicitly addressed the programmability issue, but how was the model supposed be supported in hardware? The Burroughs B6700 [23] went to great lengths to build hardware support for efficient compilation and execution of Algol programs. A hardware supported stack mechanism was implemented to allow for efficient evaluation of expressions which could be easily translated to a postfix structure by a high-level language compiler. In addition, hardware *display registers* were provided to explicitly support lexical scoping by keeping track of the location of each set of variables within the scope of of the currently active procedure block. Microprogrammed support allowed for automatic management of the display registers.

Although the stack mechanism and display registers of the B6700 are conceptually elegant, they are overkill as hardware primitives in a language-directed architecture. Rather than providing built-in hardware support for evaluating expressions on a stack to make things "easier" for the compiler, it is much more beneficial to provide a simple, symmetric GPR instruction set [3] to the compiler writer. Providing non-primitive instructions with hardware support often makes the compiler writer's job more difficult in generating efficient code [44]. This happens because the compiler has much less flexibility in optimizing code when explicit hardware support for "higher-level" semantics is built into the instruction set. The difficulty in optimizing the utilization of fast register storage in a stack machine is one such example. Maintaining the display registers in the Burroughs B6700 results in a similar effect:

the execution of microcode and hardware management of the registers is no more efficient than compiler management of the lexical scoping via tracing of static pointers embedded in the activation frames of each procedure along the current calling path. The key point to recognize is that although stacks are necessary as a conceptual framework for implementing a lexically scoped high-level language, they are *not* necessary as a hardware implementation. Instead, the provision of general purpose registers that can be used to store top-of-stack and frame-base pointers, along with addressing primitives to index into the stack relative to a frame-base pointer register, is much more appropriate for efficiently executing Algol-like languages in the von Neumann framework. Providing a simple set of well-thought primitives goes a long way toward the design of an architecture well-suited to its purpose.

Symbolics 3600 LISP machines [35] are another case in point. Significant hardware is dedicated to the support of the programming model. Run-time type checking hardware, built-in instructions for CAR, CDR, etc. with hardware and microcode support, hardware support for chasing forwarding pointers, and so on all contribute to the complexity of the machine. The complexity of the specialized instruction set and the language-directed hardware support resulted in a highly microcoded execution engine. As a consequence, the Symbolics LISP machine completely missed the opportunities associated with RISC hardware. We believe that a well-designed RISC instruction set (extended with hardware support for garbage collection [34]) would have led to the most efficient implementation for this architecture. Run-time type checking could easily be implemented using compiler generated type checking code rather than hardware support[2]. Similarly, the hardware and microcode support for CDR-coding and other specialized mechanisms to support the LISP language directly appear to be overkill when a RISC instruction set and a good compiler could achieve the same results more cost effectively. The theme, again, is to find the right primitives to support a language-directed architecture rather than implementing a programming paradigm directly in hardware.

Our thesis is that although throwing hardware at the programmability issue may appear appealing in isolated cases, it is not a panacea. In fact, excessive use of hardware support will often be detrimental to the overall cost and performance of the machine. We are proponents of the RISC philosophy: provide the right primitives and an efficient hardware implementation and let the compiler do the rest. Of course, identification of the "right" primitives is no easy task. As language-directed architectures, dataflow machines have fallen victim to the hardware complexity issue on more than one occasion. Throughout this paper, we will show how these instances of hardware complexity have been unearthed and rectified during the evolution of dataflow architectures.

[2]Note that support of weak typing in LISP is a bit suspect in itself, but we will put that aside as a language design issue and assume that it is the architects intent to support these semantics.

2.3 Ignore the Compatibility Issue

Compatibility is often thought of as the worst enemy of the computer architect. The architects of the IBM 360/91 [4] went to great lengths to obtain high-performance while maintaining strict compatibility with the 360 instruction set definition. The result was incredible complexity in the hardware design of the pipeline and multiple functional units for this computer. Clearly, compatibility can act as an overly restrictive ball and chain when attempting to pursue breakthrough innovation in designing a new computer architecture. Yet, looking at the brighter side of the issue, compatibility can also be a saving grace in that it means that the architect can avoid re-inventing the wheel when adherence to an existing standard is enforced.

There are many different levels within a computer architecture where compatibility can be defined. The goal of the architect in considering compatibility as a design tradeoff should be to get the most leverage out of the existing environment without sacrificing significant possibilities for innovation within the new architecture. With the proliferation of high-level language compilers, instruction set compatibility has perhaps taken a back seat to language compatibility. For many architectures the existence of standardized language definitions such as ANSI C facilitate a leveraging of large code foundations simply by re-compiling for a new instruction set. Yet, is it acceptable to sacrifice the ability to innovate in language design in order to maintain source code compatibility? To what extent does compatibility with existing operating systems, I/O devices, and the plethora of other standards help or hinder an architecture? These are not questions that can be answered with a blanket statement. Rather, the architect must make an informed choice as to what level of compatibility will be adopted in a new computer architecture.

From one perspective, compatibility is regressive because it prevents an architecture from moving forward. Yet, we believe it is extremely valuable because it allows the architect to leverage the work of others and focus on areas of highest relevance to the success of the particular computer system being designed. The hard part is to determine which areas of a new computer architecture should be conquered via compatibility and which areas should be focused on for innovation. Up until now, the evolution of dataflow architectures has fallen into a trap similar to that of many fundamentally new computer architectures: to almost completely ignore the compatibility issue. However, with the advent of P-RISC [38] and von Neumann hybrids [30], as well as the interfaces we have begun to build to the UNIX operating system for many of our services, dataflow architectures are beginning to address the compatibility issue. We will discuss these developments in further detail later in the paper.

3 Static Dataflow Machines

The origin of dataflow architectures lies in the work of Dennis *et. al.* In this section we describe Dennis' dataflow graphs and the architecture that was developed to facilitate their execution.

3.1 Dataflow Graphs

A dataflow graph is a directed graph in which nodes represent machine instructions and arcs correspond to the data dependencies between instructions [16]. Nodes in the dataflow graph implement functions of one or two arguments and arcs convey data tokens between nodes. A given node (instruction) may execute when and only when all of its incoming arcs possess a token. That is, an instruction is allowed to fire only when all of its operands become available. Note the time independence of the execution model; the firing of graph nodes can occur asynchronously with no violation of the execution model, as long as data dependencies are obeyed. This strategy implicitly introduces sequencing between instructions which depend on each other, but allows instructions to execute in parallel if there exists no arc (dependency) between them. Upon execution, an instruction consumes its input tokens and produces one or more output tokens for subsequent instructions in the dataflow graph. A program terminates when there are no enabled instructions left in its flow graph.

The Figure shown below illustrates the *acyclic* dataflow graph that represents computation of the following expression:

```
let x = a - b;
    y = c - d;
in  (x^2 + y^2)
```

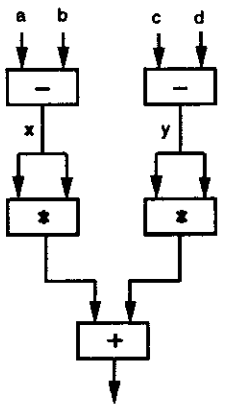

Figure 1: Acyclic Dataflow Graph

```
Def trapezoidal f a b h =
   { result = (f(a) + f(b)) / 2;
     {For i From 1 To n Do
       result = result + f(a + i*h)}
   In
     result};
```

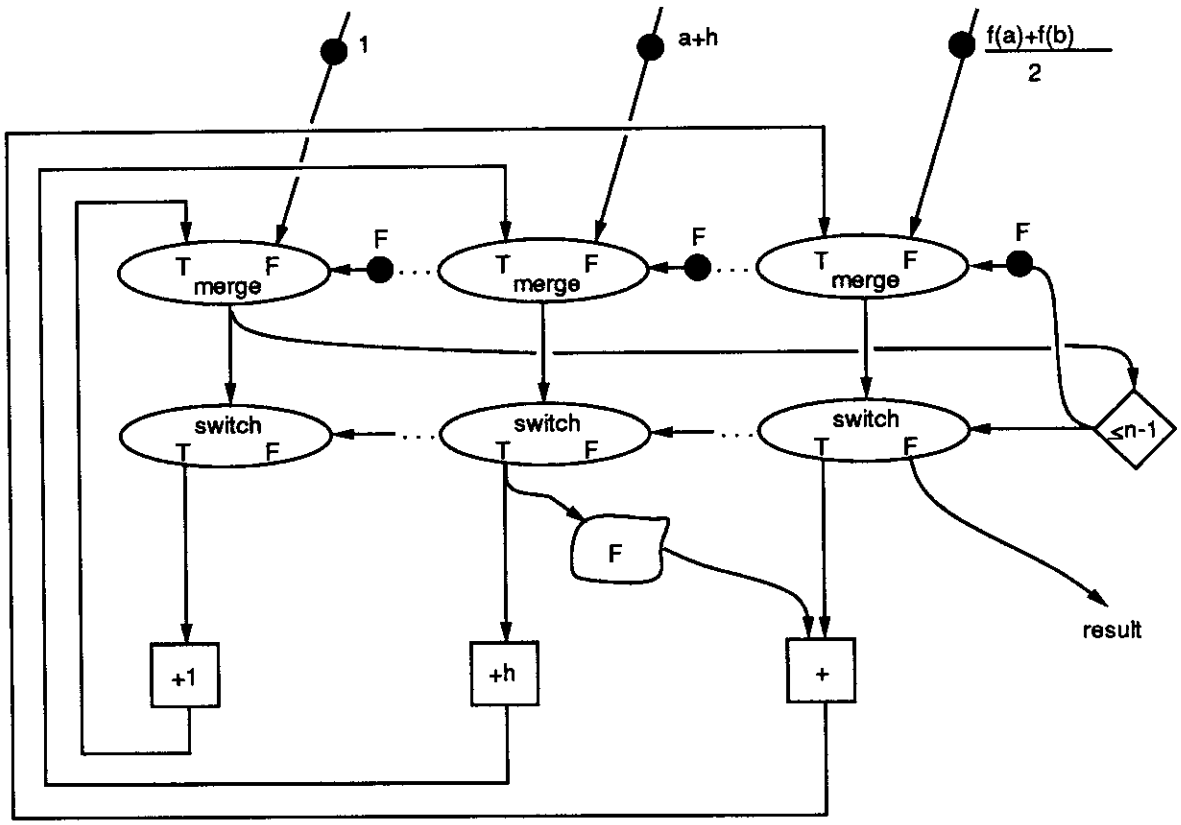

Figure 2: Cyclic Dataflow Graph for the Trapezoidal Rule

3.2 Static Dataflow Architectures

Dennis, in his landmark paper [16], suggested that dataflow graphs be treated as a parallel machine language. Dennis subsequently went on to propose the architecture shown in Figure 4 to directly execute dataflow graphs [19].

To support loops and conditional expressions, it is necessary to introduce two additional primitives to the repertoire of the machine language. The *switch* and *merge* operators allow for navigation of tokens through the dataflow graph in the presence of conditional constructs. Consider the dataflow graph for implementing the trapezoidal rule shown in Figure 2. The switch operators, shown at the top of the graph, allow selection between input arcs for this graph node. The boolean input to the switch nodes, shown coming in from the left, provides the selection criteria for the input arc. A merge operator, examples of which are shown just below each of the switch operators, plays a converse role. Its purpose is to navigate an input token to one of two output arcs based on a boolean input. Together, the switch and merge operators are able to orchestrate the execution of conditional constructs.

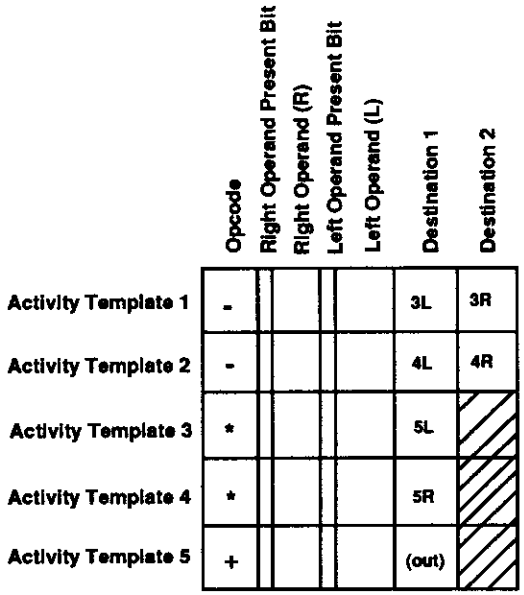

Figure 3: Static Dataflow Representation of a Program Graph

The method for storing operands and scheduling instruction execution in a static dataflow machine is through use of *activity templates*. Each activity template corresponds to a node in the dataflow graph. An activity template consists of (1) an operation code to specify the ALU operation to be performed, (2) slots in which the incoming operands should be stored, (3) status flags to indicate which operands are present in the activity template, and (4) destination specifications to indicate to which node(s) the result of the operation should be forwarded. See Figure 3. An activity (instruction) is ready to fire once each of the operand slots in its corresponding activity template have been filled. The result of the instruction execution is then forwarded to successor nodes in the dataflow graph to be used as operands in subsequent computations.

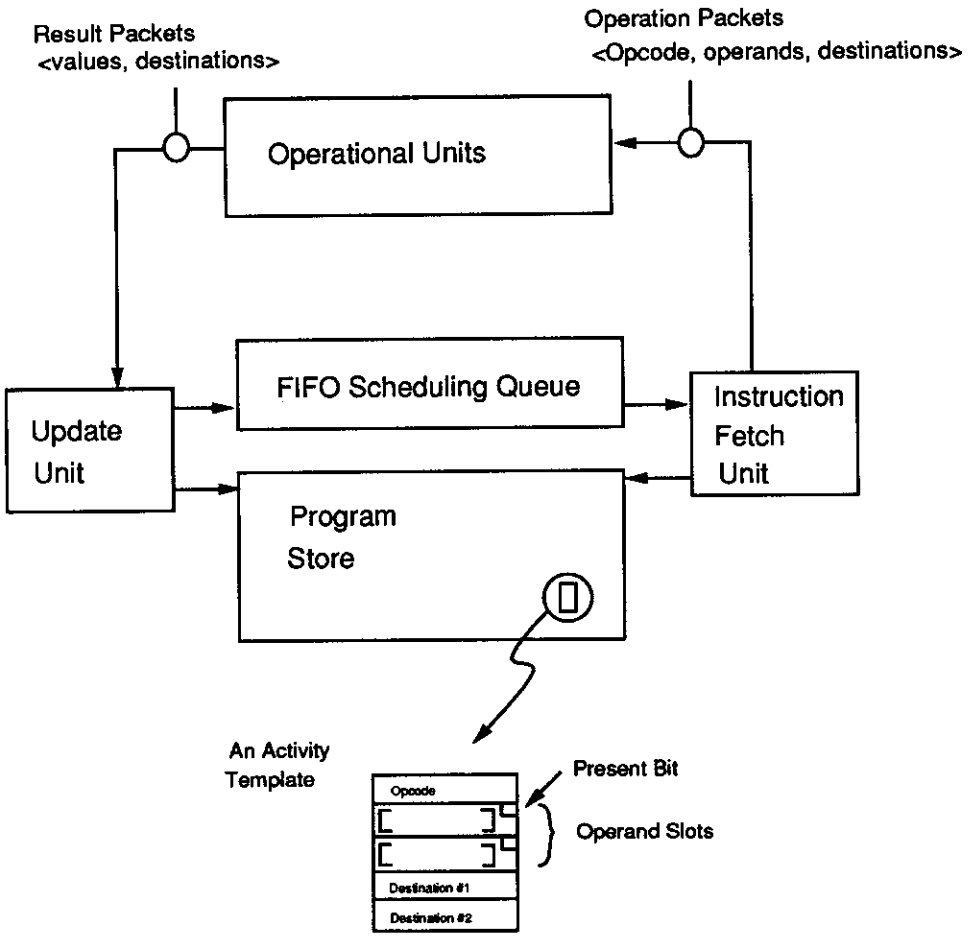

Figure 4: Architecture of a Static Dataflow Processing Element

The activity templates corresponding to the portion of a dataflow program graph assigned to a particular processing element are stored in the *program memory* of the element. Result packets produced by the execution of dataflow operators are received by the *update unit* of the processing element to which they are delivered. These result packets are packaged as a triplet of information containing (1) the activity template address in program memory, (2) a port specification to indicate which operand it is delivering, and (3) the data value of the operand. The data value carried in the result packet will be extracted and placed in the specified operand field of the activity template for which it is destined. In addition, the update unit will then check to see if all the input operands for the instruction have arrived. If so, then the firing rule for a dataflow computation has been satisfied. Consequently, the instruction becomes enabled and its address in program memory is placed on a FIFO *scheduling queue.*

The *instruction fetch* unit repeatedly takes the address of the next enabled instruction from the scheduling queue and proceeds to fetch the instruction at that address along with its operands and destination list. This information is bundled into an operation packet and

forwarded to the appropriate operational unit (functional unit or array memory) for processing. Since the execution of dataflow instructions are independent of one another (except as indicated by explicit data dependencies), each of the many operational units in a dataflow processor can operate without external synchronization. There is one result packet formed for each destination contained in the activity template corresponding to the instruction executed by the operational unit. These result packets are sent to the appropriate destinations, as specified in the destination list of the activity template.

In some respects, dataflow computation is very similar to conventional von Neumann processing [18]. Both models are realized using stored program computers in which a machine-level program is comprised of individual instructions which dictate the activity of the machine. Primitive instructions in these machines include operations for implementing both floating point and integer arithmetic as well as standard logical functions. A fundamental difference, however, is the way in which instructions are scheduled for execution in these two models of computation. In a dataflow computer, an instruction is scheduled for execution when and only when all of its input operands are available, whereas in a von Neumann computer instruction execution is determined by the program counter.

3.3 Multiprocessor Operations

Multiprocessing fits naturally into the distributed execution model provided by dataflow graphs. To facilitate multiprocessor operations in a static dataflow computer, we simply interconnect individual dataflow processing elements using a communication network. Each processing element stores a portion of the dataflow graph to be executed. Each of the processing elements in the static dataflow computer maintains its own queue of enabled instructions to be scheduled for execution. The result produced by an operational unit may be routed locally within the same processing element (as before) or it can be routed to another processing element via a *distribution routing network*. See Figure 5 below.

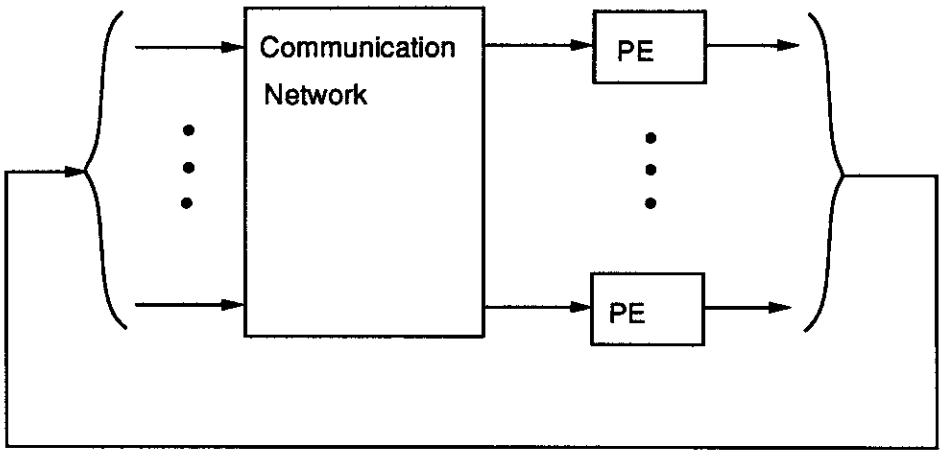

Figure 5: Multiprocessor Architecture of a Static Dataflow Computer

The activity templates that make up an executable program are statically mapped to the processing elements in the machine. The compiler or programmer must pre-assign partitions of a program to individual processing elements. To take advantage of local communication, activity templates which are connected in the graph must be assigned to the same physical processing element. Of course, local communication will often be sacrificed in order to get appropriate work distribution in the machine.

3.4 Some Observations on Static Dataflow Computing

The parallel machine language developed by Dennis has served as the foundation for the evolution of architectures designed to support the dataflow model of execution. The parallel machine language described provides exposure of inherent parallelism in a compiled program. Moreover, this program parallelism can be used to hide memory and communication latencies in the machine. The architecture allows a fine-grained interleaving of instructions from different computational threads within the pipeline of a processing element. This allows long latencies to be hidden as the processor is able to turn its attention to other threads of computation while awaiting a remote request to be satisfied.

There are, however, some significant shortcomings in the static dataflow implementation of the model. Most notably, these include:

- The *merge* operator requires special hardware support to accommodate its own variant on the normal dataflow firing rule.
- The operand slots of an activity template provide only one token per arc in the dataflow graph, whereas the pure model is based upon unlimited token storage per arc.

The remainder of this section will address these points and their influence on the architectural development.

The *merge* operator selects an input token from either the TRUE or FALSE input arc, as directed by the boolean control input (see Figure 2), and propagates its value along its output arc. The firing rule for *merge* requires different treatment because it is not enough to merely sense arrival of both operands for the operator; it is critical that the *correct* set of operands are paired together for execution of this instruction. Consider, for example, if the TRUE input arc token has arrived, as well as a control input with a value of FALSE. In this case, the operator should not fire because the FALSE value requires its other input to come from the FALSE input arc. Unlike other dataflow operators, a firing of merge does not remove all tokens from the input arcs.

The variance of the *merge* firing rule from that of other dataflow operators imposes additional complexity in the hardware. The update unit must read the opcode of an instruction and the value of the incoming token to decide how the firing rule should be handled. In an

effort to remove this burden from the hardware design, Dennis [17] attempted to omit *merge* as operator in the static dataflow graphs through compilation of highly-structured graphs. However, without the *merge* operator, the execution of loop and conditional constructs in the static model can lead to erroneous (*i.e.*, non-deterministic) results.

The underlying problem with the static approach is that the architecture does not adequately support the dataflow model. The pure dataflow model allows unbounded unfolding of parallelism as long as data dependencies are obeyed. Pure dataflow requires *unbounded* FIFO token queues to be provided for each of the arcs in the dataflow graph. The operand slots of a static dataflow architecture allow only one token to be stored along each arc in the dataflow graph. These statically allocated operand slots prevent multiple, parallel instantiations of an instruction within a loop or from multiple function invocations. This is because the operand values in an activity template could become overwritten by operands corresponding to subsequent instantiations of the instruction.

One approach to supporting the pure dataflow model in the static architecture is to provide buffering of input operands to eliminate the possibility of overwriting operand slots. In fact, the network and functional units already provide an implicit form of additional token storage in the machine. However, the problem is not so easily solved. The FIFO ordering of tokens along the arcs in the dataflow graph must also be maintained. This imposes unreasonable demands upon the hardware: the communication network needs to maintain FIFO ordering, and compensation must be made for variations in the execution time of different functional units. Such constraints are beyond the realm of reason for the architecture of a large-scale multiprocessor.

The approach taken by Dennis *et. al.* [18] was to transform dataflow programs in such a way as to guarantee no more than one token per arc in the dataflow graph. The firing rule for each operator was modified to prevent the arrival of data before the firing of its previous instantiation. This was done by augmenting graphs in the static dataflow architecture with *signal arcs*, as shown in Figure 6. Thus, in addition to *result arcs* which are used to forward data values produced by the firing of an instruction, it is necessary to provide *signal arcs* to indicate when an successor instruction has consumed its input operands. There is a one-to-one correspondence between result arcs leading out of an activity template and signal arcs leading into an activity template for each node in a static dataflow graph.

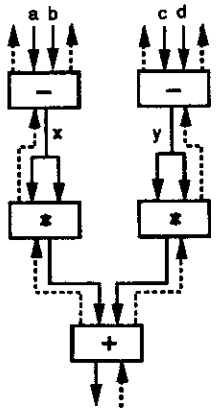

Figure 6: Acyclic Dataflow Graph with Signal Arcs

When an instruction fires, it will send the resultant data value to the target instructions on its destination list as well as providing signals to each of the instructions which supply its input operands. The signal indicates that the instruction is ready to receive its next set of input operands. The firing rule for an instruction needs to be extended to include the transmission of signals from each of the target instructions on its destination list (in addition to the presence of all input operands). This added provision to the firing rule ensures that each target instruction has a chance to use its current input value before it is overwritten by the value produced by the next firing of the instruction which produces its input operands. The additional clause to the firing rule can be implemented through use of two integer fields: *signals-needed* and *signal reset*. The signals-needed field is decremented by one for each signal received by the activity template and the signal clause of the firing rule is satisfied when this field reaches a value of zero. After each firing of an instruction, its signals-needed field is re-initialized to be equal to the number of target instructions on the destination list of the instruction. This constant is stored in the signal reset field. Thus, if an instruction produces input data for two target instructions (i.e., it possesses two result arcs) in the dataflow program graph, then its signal reset field will be equal to two and each of the two target instructions should send a signal back to the instruction when they fire. Although some of these signal arcs can be eliminated [33], this protocol will generally increase token traffic by a factor of 1.5 to 2, and severely impacts the rate at which successive firings of a given node can take place.

The problem with the solution described in the previous two paragraphs is that it is too microscopic in its approach. Transformation of the firing rule for operators led to complicated hardware and significant losses in available program parallelism. Advances in compilation technology have opened up a new approach. The idea is to think of the program transformations at a higher level: to use a graph schema that *enforces the single token per arc rule without additional hardware support.* Culler observed in [14] that many loops were naturally bounded. That is, data dependencies often prevented a loop from unfolding more than some bounded number of iterations. This observation gave impetus for the development of

a bounded loop schema [5] whereby the compiler introduces artificial dependencies into the program graph to ensure that no loop unfolds more than a bounded number of concurrent iterations. We will discuss the bounded loop schema in more detail in Section 5.2. Note that a bounded loop schema actually changes the model of computation: no longer is there an *unbounded* number of tokens per arc in the dataflow graph, but rather a compiler or run-time determined upper bound on the number of tokens per arc.

4 The Evolution of Data Structures in Dataflow Architectures

Support for data structures is a critical component in dataflow systems. This aspect of the dataflow architecture has demonstrated significant evolution, resulting in a fundamentally different programming model and set of hardware primitives in the current realizations of dataflow machines.

4.1 Functional Data Structures and the Copying Problem

In the dataflow model, structures have the property that a functional semantics is enforced. In particular, this means that multiple writes to a single memory location in a data structure is explicitly disallowed in the programming model. The functional semantics ensure that determinacy is preserved in the face of parallel execution of a dataflow program. Functional semantics present a new dimension to data structure handling.

Let us consider the functional array operations described by Dennis [16] for his first version of a dataflow procedure language. Figure 6 illustrates these operations. Conceptually, we can assume that a token carries the whole array. However, at the implementation level, it is clear that carrying a large, variable size value on a token is patently absurd. Thus, we envision that arrays are stored in a data structure storage and tokens carry only pointers to an array, not the array itself. The *select* and *append* operations must communicate with the data structure storage.

Given a pointer to the array (A) along with an integer index (i), the *select* operator "selects" the desired value (v) from the array and passes it on to a subsequent node in the dataflow graph. The real issue arises when updating an existing array. The *append* operator, given a pointer to an array along with an integer index and a new value, produces a new version of the array which is identical to the old one except that the value at the specified index.

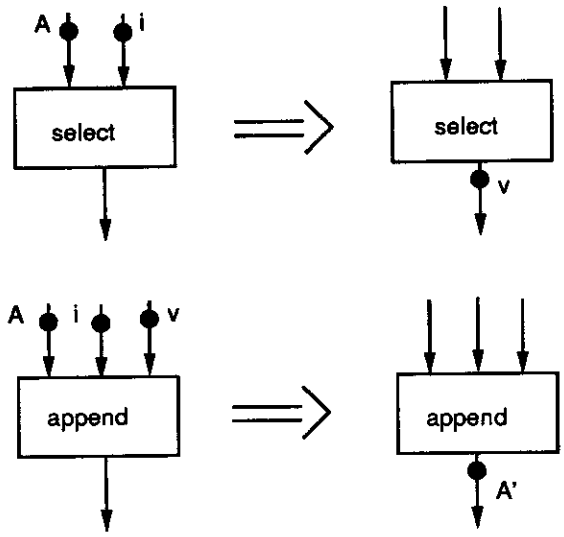

Figure 7: Data Structure Operators for Functional Arrays

Of course, the functional semantics of append require that the original array remain intact, so the new array must be constructed via a copying operation on the original. This is quite troublesome, since such an operation is very expensive in the machine. The evolution of data structure implementations in dataflow architectures has been primarily aimed at attacking this problem.

4.2 Data Structure Storage as a Functional Unit

Ackerman's proposed implementation [1] of data structure storage made it look just like another functional unit in the machine, although the internal operations of this unit were extremely complex. This approach is illustrated in Figure 8. The implication of this approach is that the select operation can issue its request to the structure handling unit and then step out of the pipeline to let other instructions proceed. This is a big win in contrast to the traditional approach whereby the pipeline would stall while a (potentially lengthy) memory access was in progress.

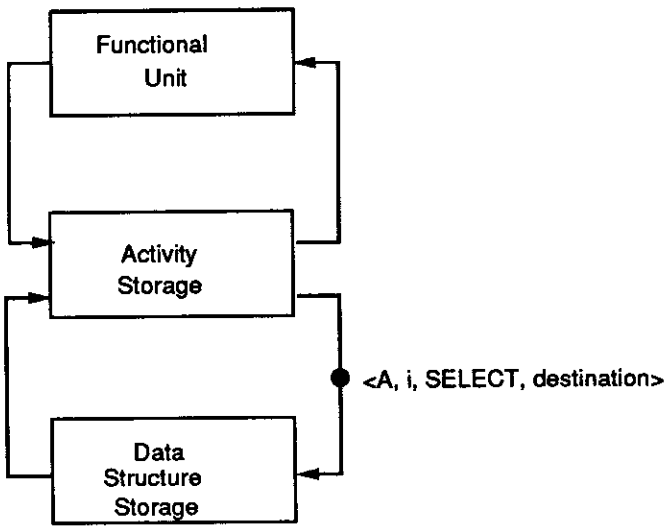

Figure 8: Data Structure Storage as a Functional Unit

The data structure storage unit had clever mechanisms for tree-structured organization of the data to minimize copying as well as automatic allocation and deallocation of structures. However, the implementation complexity associated with these mechanisms was enormous. *The flaw is not so much in the architectural implementation, but rather in the approach that involved solving a problem in hardware which really should have been tackled in software.* The append operator is far too complex in its semantics to be considered a primitive in the hardware implementation of the machine. Management of copying overhead for a potentially large data structure clearly goes beyond the scope of a primitive hardware-implemented operator.

4.3 I-Structures

The movement away from complex hardware mechanisms for implementing the semantics of functional data structures has led us expose the data structure handling within the programmer's model. Our current approach is to shift much of the burden for managing the structures away from the hardware and back up to the compiler and programmer. Our implementation of data structures, I-Structures [8], forces the compiler/programmer to perform explicit management of the structure resources. There is no implicit allocation of storage. Whereas an update to a data structure in Ackerman's model would perform an implicit copy of the structure to maintain functional semantics by constructing a new "version" of the data structure, our model simply does not allow multiple writes to a single location in a structure. The programmer, via the compiler, must *explicitly* create a new version of the structure with the desired update directed to the appropriate location. The compiler will generate the proper code to allocate the structure and fill in its contents, as well as to reclaim the old structure when appropriate. In this simplicity we believe there is much to

be gained in terms of a predictable performance model for the programmer and efficiency in our implementation.

I-structure operations retain the view presented in [1] of data structure storage as a separate functional unit. This view is made possible through use of *split-phase* operations, as shown below, in which the I-structure request and reply are *not* synchronous. A request token is sent to an I-structure unit (potentially across the communication network) and the processor is then free to continue executing other instructions while the request is being delivered and handled. For example, in the select shown below, an address (addr) is constructed from the base (A) of an array and an index (i), and the "read" request is shipped off to the I-structure storage unit along with a tag specifying the context (c) and target instruction (t) for the result value (v). The eventual arrival of the answer to the request will enable the target instruction to fire. However, since all operations are split-phase, an I-structure request will never cause the processor pipeline to stall.

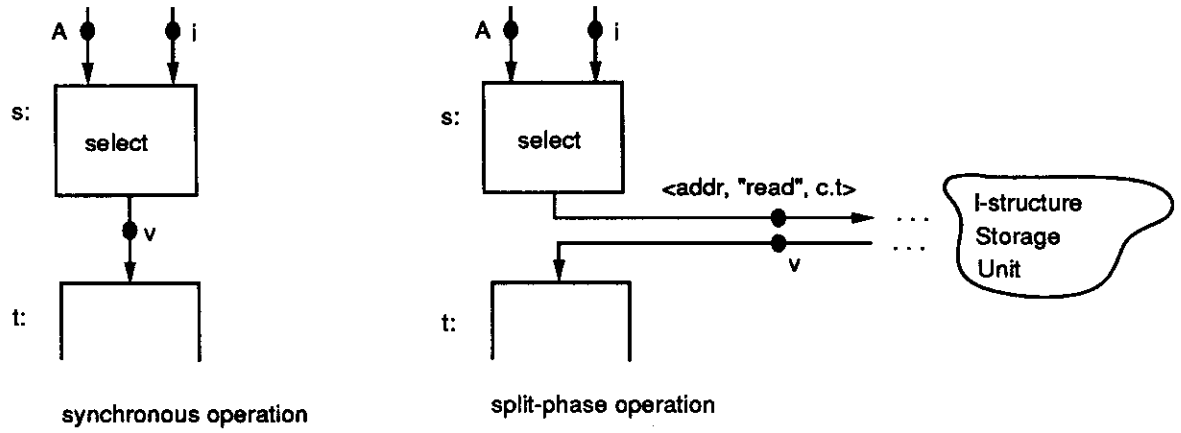

Figure 9: Split-phase Select Operation to I-Structure Store

Our compiler-oriented approach also allows us to leverage some very simple hardware primitives to implement variations on the functional storage model. By adding presence bits to associate a small amount of state with structure cells, non-strict data structure operations [8] can be supported. Non-strict data structures allow a program to access parts of a data structure before all of its cells are filled in. Multiple reads and writes are allowed to proceed in parallel as long as all write operations are directed to different elements in the structure. In fact, reads and writes may even get out of order in the communication network. Synchronization of reads and writes is performed for each cell of the structure, so that there is no problem when a read precedes the write to a cell. In such a case, a deferred read is created whereby the read is put aside on a list with a promise to fulfill the read when the cell is written. This producer-consumer synchronization is implemented using just two bits to indicate whether a cell is in EMPTY, FULL or DEFERRED READ state. A pointer to the deferred read list, if one exists, is kept in the empty cell of the structure

until the point when the expected write takes place. As long as most reads take place after their corresponding write, the overhead of maintaining the deferred read list is small [24]. In accordance with our philosophy of minimalist hardware implementation of primitives to support the programming model, we rely primarily upon the compiler for the realization of non-strict structures. The hardware support necessary is simply the presence bits to help manage the out-of-order arrival of reads and writes. The compiler takes an active role in the deferred read management, and has full control over all copying and garbage collection activities.

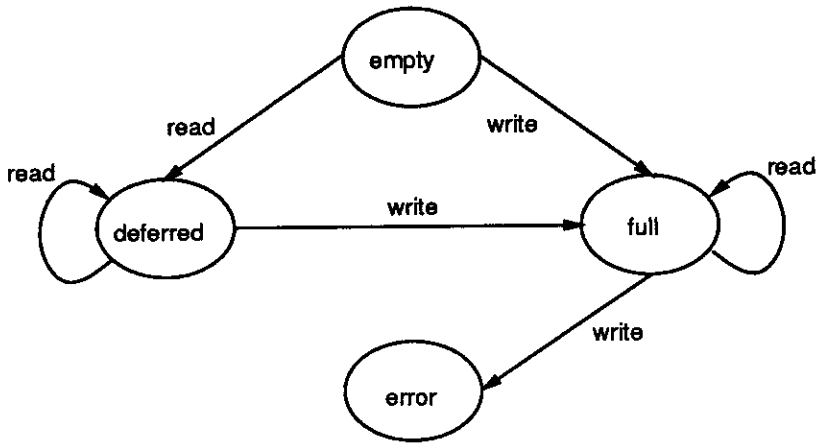

Figure 10: State-Transition Diagram for an I-Structure Cell

The fact that the storage model is managed by the compiler allows a variety of other storage models to be implemented with no hardware modifications to the machine. Heller [25] has shown how lazy structures can be implemented in this framework. In addition, Barth [9] has developed non-functional TAKE and PUT operations that provide semaphore semantics for systems programming tasks.

5 Dynamic Dataflow Architectures

Dynamic dataflow architectures evolved as a means to address the mismatch between static dataflow architectures and the pure dataflow model. The intent of dynamic dataflow architectures is to provide a framework with *unbounded* FIFO token queues for each of the arcs in a dataflow graph. Within this framework, the unfolding of parallelism is not constrained by the architecture (as is the case with static dataflow architectures), but rather only by the availability of hardware resources (such as token store) which are presumably managed at the (software) run-time system level. As a result, there is substantially more parallelism that can be exploited to speed up program execution. There are three components of the evolution from static to dynamic dataflow architectures:

- Separation of the code and operand store.
- Provide for re-entrant (sharable) code.
- Dynamic allocation and deallocation of operand store.

This approach allows the architecture to provide data-driven computation without the single instantiation limitation on instructions inherent in the static dataflow approach. There are two methods for supporting dynamic dataflow computation: run-time code copying and dynamic tagging [6, 22]. In either implementation, the goal is to provide a means for distinguishing between different instantiations of an instruction within a dataflow program. We will focus on the dynamic tagging approach. One result of the dynamic approach is that recursive functions can be supported, in contrast to the static model in which they are strictly prohibited. Another result is that with dynamic tagging the merge operator can be eliminated from the repertoire of the instruction set without introducing non-determinacy into the program execution.

The dynamic tagging approach to dataflow computation associates a unique tag with each instantiation of an instruction within a program. Thus, each data operand for an instruction carries a tag that specifies the particular instantiation for which the data is intended. Using this mechanism resolves any ambiguities arising from the existence of multiple data operands for different instantiations of an instruction; the unique tag values allow the instruction scheduling unit to determine which operands correspond to the same instantiation of an instruction. The architectural challenge in designing a tagged token dataflow machine hinges upon efficient storage and matching of large numbers of data operands that may arrive in any order during program execution.

5.1 Automatic Management of Token Storage

Our first design for the Tagged Token Dataflow Architecture called for automatic management of token storage [6]. This function is performed in the waiting-matching unit, as shown in Figure 11. The figure shown illustrates the pipeline corresponding to the execution of basic dataflow instructions in a processing element. The *waiting-matching* unit is designed to pair tokens necessary for the execution of dual-operand instructions. Some form of associative match is performed on the tag field of tokens in the store to determine if two operands are partners for execution. All tagged token dataflow machines require this capability to facilitate detection of instructions which are ready to fire according to the data-driven execution rules. Tokens corresponding to single-operand instructions may bypass the waiting-matching station without delay. The next stage in the pipeline is the *instruction fetch* unit. The purpose of this station is to retrieve the instruction corresponding to the operand(s) which it has received from the waiting-matching station. The instruction fetch station subsequently passes the instruction, along with its operands, to the ALU which computes the results of

the instruction. The *compute tag* unit executes in parallel with the ALU. It performs address translation to construct new tags for the data result packets from the ALU station. These tags specify the set of destinations to which the result packets should be sent. The *form token* unit then compresses the contents of the resultant packet from the ALU with the tags generated in the compute tag unit to yield a collection of "finished" tokens. These tokens are then dispatched to the network, or along an internal path, as appropriate.

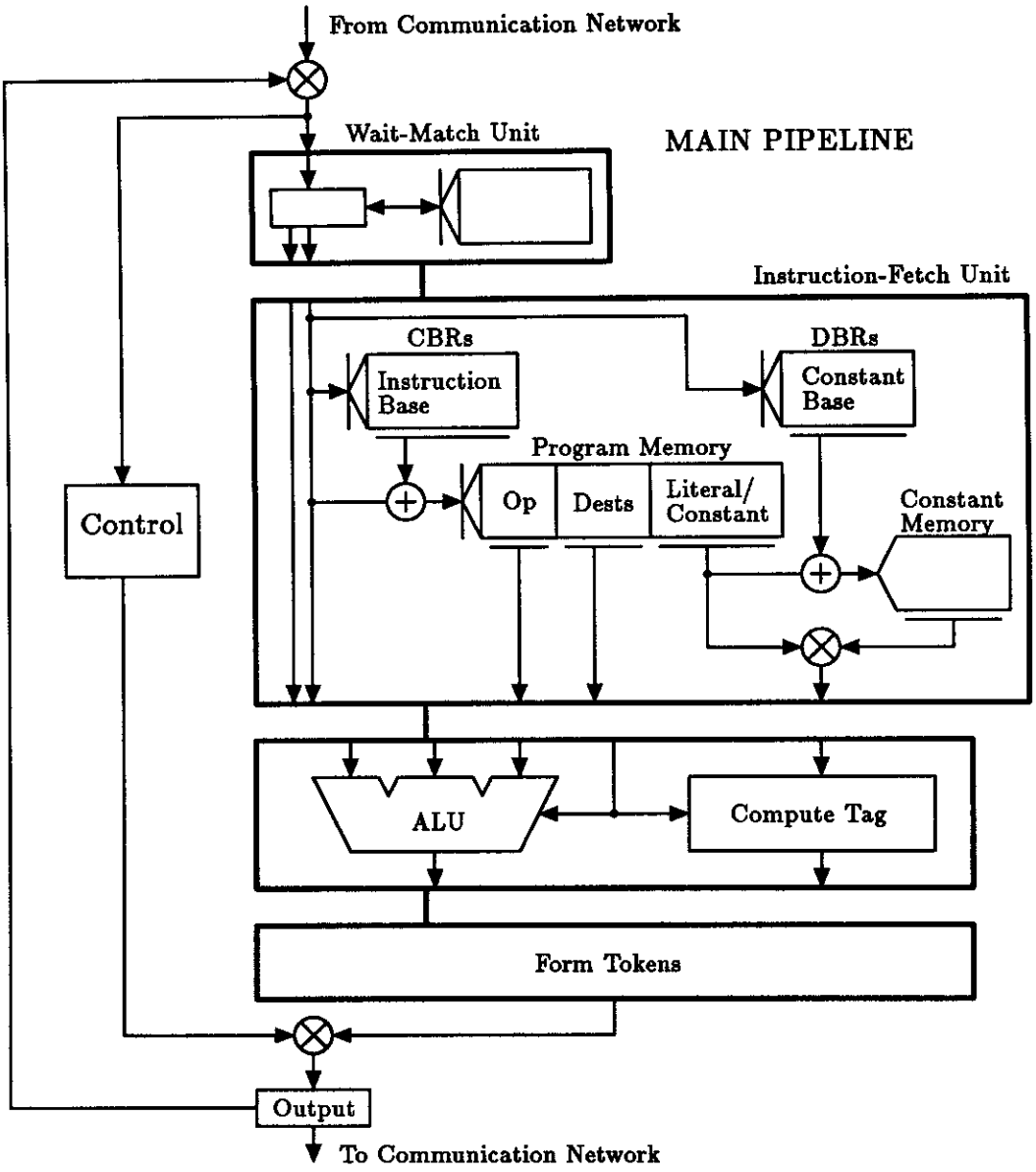

Figure 11: The Tagged-Token Dataflow Architecture (originally appeared in [7])

Unmatched data operands in the Tagged Token Dataflow Architecture are stored in a common pool of token storage; this token storage is managed at the hardware level with automatic allocation and deallocation of token store as operands come and go. The token store

was designed with fully associative matching capabilities to allow tokens to be matched with their partners based on tag values which specify the code block instantiation[3], iteration identifier (for loops), and instruction address for the data operand. This token store was originally envisioned as a fully associative content addressible memory with parallel matching capabilities on the tag of each waiting data operand. The reasoning behind this design was that only a small number of waiting tokens needed to be stored in order to support enough parallel threads of computation to keep the pipeline of each processing element fully utilized. Unfortunately, however, this was not the case.

As it turns out, there are a substantial number of tokens that need to be stored to maintain the context for suspended functions [14, 13]. These tokens represent data local to inactive functions which are awaiting the return of values undergoing computation in other functions invoked from within their bodies. These tokens are analogous to entries in old stack frames on a von Neumann machine designed to support block-structured programming languages. Once these data operands for old stack frames are considered, token storage requirements on a per processor basis start to enter the realm of tens of thousands of tokens. When these are included as part of the token pool that needs to be maintained in the waiting-matching unit, a content addressible becomes much less appealing as a practical means for realization of the token store. The Manchester [22] and SIGMA-1 [29] machines turned toward hash tables with hardware support to manage the common pool of token storage. A number of other schemes, including caching [13] and virtual memory organizations [32] were proposed as well. The self-managing waiting-matching unit in the Tagged Token Dataflow Architecture was clearly the most complicated component of the architecture, and was a source of concern both due to its cost and efficiency.

5.2 Explicit Management of Token Storage

It soon became apparent that self-managing token store was paramount to *un*managed token store in our implementation of the Tagged Token Dataflow Architecture. The Tagged Token Dataflow Architecture was too abstract for a concrete hardware implementation; a simpler model of token management in the hardware of the machine was required. In 1988, Papadopoulos proposed the Monsoon Architecture [39] which is based on the explicit *software* management of token storage. Upon the invocation of a function, a *frame of storage* is allocated to provide exactly the number of token slots necessary for supporting the function invocation.

Culler's work [14] allowed even loops to be efficiently implemented in this paradigm. Of course, loops could always be treated as tail recursive procedure calls in a dynamic dataflow architecture. However, Culler's schema for controlled unfolding of loops facilitates a more

[3]Each instantiation of a code block receives a unique context identifier to allow multiple instantiations of a single code block without running into any naming conflicts

efficient handling of this paradigm. Within his framework it became possible to determine the token storage requirements needed to support any code block invocation, given the number of loop iterations that would be allowed to proceed in parallel. For loop code blocks a bounded number of iterations will be allowed to proceed in parallel with enough frame storage allocated to support that bounded number of iterations. The frame storage is recycled transparently as one iteration of the loop terminates and another is allowed to begin. The compiler generates straightforward code to orchestrate efficient recycling of frame storage and controlled loop unfolding [15].

Frame storage also provides a logical place to store loop invariants so that tokens do not circulate unnecessarily in the dataflow graph. Previous to the advent of frame storage, there were two approaches that emerged for handling non-recirculating tokens. The Tagged Token Dataflow Architecture provided a dedicated *constant memory* in the instruction fetch unit where loop invariants and program constants were stored. This is yet another variety of memory in the machine, and further makes it difficult for the architect to balance resources in the machine. The alternative approach, implemented in the Manchester machine [22] and Sigma-1 [29], was to introduce specialized hardware support in the wait-match unit. This support came in the form of "sticky tokens" whereby specially designated tokens were allowed to remain in the store after a successful match. Special hardware is required to facilitate the persistence of these tokens in the store and to ensure proper matching with future partners. Special matching operations are also required to purge the tokens after they are no longer needed (*i.e.*, when the loop terminates). Although the "sticky token" approach allows constant store and wait-match token store to be combined, the additional complexity of the matching unit is certainly less than desirable. Monsoon tackles both problems by storing loop invariants and constants in frame memory along with normal tokens - without any special matching operations. These values are placed in compiler specified slots for as long as needed, and the slots are deallocated along with the rest of the frame when the loop terminates.

There are two fundamental concepts in the handling of token storage that Monsoon implements as distinguished from its predecessor, the TTDA. First, *Monsoon mandates explicit allocation and deallocation of token store.* An activation frame of token slots is allocated upon invocation of a function, and released upon function termination. In the TTDA, token store was allocated (and deallocated) implicitly in hardware associative memories. The explicit management of token store facilitates simplicity in the hardware and an efficient implementation of the matching operations required for data-driven execution in the dataflow model. Culler's bounded loop schema even makes it possible to efficiently manage parallel loops in the context of an explicit token store model, as described in the previous paragraph.

The second fundamental concept in the Monsoon architecture is that *the compiler, rather than the machine hardware, takes charge of token slot assignment.* In Monsoon, compiler analysis is performed to pre-assign data tokens to their slots within an activation frame. This compiler analysis is very similar to that used for register allocation in conventional

compilers. This is in contrast to the TTDA in which token slots were assigned dynamically by the hardware based on availability. The result of Monsoon's improvement upon this model is, again, substantially simplified hardware mechanisms leading to a more efficient and economical machine implementation.

6 Dataflow Solution to the Compatibility Problem: P-RISC

Throughout most of the lifetime of our dataflow project, the focus has been on the parallel execution of programs written in our own functional language, Id [36]. Moreover, we have pursued development of a programming environment [37], run-time system [12] and hardware architecture [39] that are specifically designed to suit our parochial, though quite sophisticated, view of general-purpose parallel processing. Of late, however, our approach has broadened. The P-RISC effort within our project is an attempt provide support for running existing applications, written in languages such as C or FORTRAN, on von-Neumann oriented processors.

The P-RISC architecture came about as a bottom-up effort in our project. The Monsoon architecture had a number of registers designed into its pipeline to facilitate efficient temporary storage of values to be used within a thread of computation. Further innovations in the use of these registers in Monsoon, along with motivation provided by [30], led us toward a synthesis of dataflow and von Neumann architectures. The idea is to allow efficient execution of sequential threads of computation in a dataflow computer through use of the inexpensive control mechanism provided by the von Neumann framework.

Approaching these thoughts from the opposite direction, the P-RISC architecture emerged. The idea is to provide extensions to the von Neumann framework that allow the fine-grained parallelism of dataflow execution, while still retaining the efficient control mechanism of von Neumann computing. We require the following modifications to a conventional RISC processor to make it suitable for execution of P-RISC code:

1. Modify the RISC processor implementation to make it suitable for multithreading [43, 2].
2. Extend the instruction set of an existing RISC processor to include three additional instructions: FORK, JOIN, START.
3. Augment the RISC processor with I-Structure storage.

The result is a multithreaded architecture (see Figure 12) that can exploit both von Neumann and dataflow compiling technology [38].

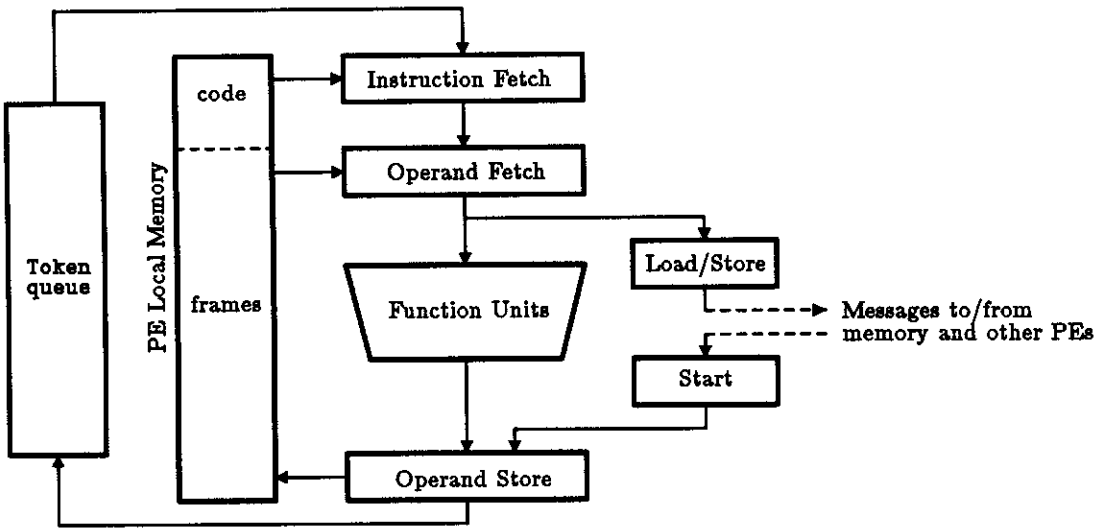

Figure 12: The P-RISC Processor Architecture (originally appeared in [38])

Our framework for a multithreaded architecture is loosely based upon HEP [43]. To transform a RISC processor into a multithreaded RISC, it is necessary to provide a token queue capable of holding data values along multiple threads of computation in the machine. In addition, thread descriptors are required to identify the thread and specific instruction to which a data value corresponds. Upon each pipeline beat in the machine, a token is extracted from the queue and scheduled for execution. In this way, multiple threads can be finely interleaved in the pipeline as tokens from different threads are extracted from the queue. Each thread in the machine has an explicitly allocated frame, managed in the same style as Monsoon frames. Thus, a thread descriptor is really just a (FP.IP) pair to specify pointers to the frame and instruction to be next scheduled for execution.

The instruction set of a P-RISC processor is intended to be a simple extension of the RISC model. The addition of the FORK, JOIN and START instructions will allow us to support fine-grained dataflow execution in the machine. The FORK instruction is a method of spawning a parallel thread of computation from within an existing thread, and the JOIN instruction allows two threads to synchronize execution. With these two instructions, it is possible to simulate the fine-grained, asynchronous parallelism of dataflow execution [38]. The START instruction is a further extension that facilitates communication between frames on different processors in the machine. The START instruction receives a value from another processor and initiates a thread. The START instruction is similarly used to implement the split-phase memory operations as we discussed in Sections 4.2 and 4.3. When a memory operation is issued the thread suspends; upon completion of the operation a value is sent to a START instruction and the thread re-initiates. The full details of these three instructions and their implementation is given in [38].

The last modification we require for ourr P-RISC processor is to extend heap memory with I-structure semantics. By adding presence bits to provide a small amount of state with

each structure location, synchronization between multiple, parallel reads and writes can be implemented. With this addition, the full power of the data structures described in Section 4 can be provided.

The fact that the von Neumann framework is supported by P-RISC allows us to build a dataflow machine that achieves compatibility with a huge base of existing software. Within the context of P-RISC, we hope to run existing systems such as UNIX and FORTRAN on semi-stock hardware. Applications written in FORTRAN or C will be translated into dataflow graphs, and then a back-end will target code to the instruction set of a RISC-oriented, multithreaded von Neumann processor. In addition, this framework will allow us to run parallel languages, such as Id, at any level of granularity in a von Neumann architecture that supports multithreading.

7 Conclusions

An architect's primary task is to simplify complex mechanisms through provision of *appropriate* primitives to support the targeted execution model. Throughout our discussion, we have pointed toward a theme of simplicity: the most significant evolutionary steps in dataflow architecture have involved simple and elegant solutions designed specifically to obviate the need for huge hardware mechanisms. Whenever possible, our approach has been to push the onus of bridging the gap between the programming model and the hardware into the compiler. Effective leverage of compilation technology, when coupled with appropriate hardware primitives, is unbeatable in developing an efficient architecture.

The evolution of dataflow architectures has clearly been in this direction. At the present time, we are only a couple of years away from commercialization of dataflow multiprocessors. Monsoon has emerged with the ideas of explicit frame allocation and effective use of compilation technology that allow an efficient dataflow implementation to take place. This effort is currently underway in a cooperative venture between M.I.T. and Motorola. We have also seen an important trend in the synthesis of ideas between the dataflow and von Neumann frameworks. Numerous projects are pursuing hybrid architectures designed to exploit fine-grain parallelism as well as preserve the efficiency of sequential threads: P-RISC at M.I.T. [38], Empire at IBM [30], Epsilon at Sandia [21], EM-4 at the Electrotechnical Laboratory in Japan [40, 45], and others as well. Architecture work in dataflow has never before been so invigorating.

Acknowledgements

The evolution presented in this paper represents the concerted research efforts of many talented research scientists from numerous different institutions, to whom we are grateful for making the dataflow field what it is today. Jack Dennis, as the founding father of

dataflow research, as well as the Computation Structures Group at the M.I.T. Laboratory for Computer Science, has had a profound influence on the development of the field.

We would like to thank Derek Chiou and Gregory Papadopoulos for their careful reading and comments on an earlier draft of this paper. We are indebted to Rishiyur Nikhil for his artwork in Figures 11 (originally appeared in [7]) and 12 (originally appeared in [38]). The bulk of the work for remaining figures was performed by Agnieszka Adamkiewicz, to whom we are also grateful.

References

[1] W.B. Ackerman. A Structure Processing Facility for Dataflow Computers. In *7th International Conference on Parallel Processing*, August 1978.

[2] Anant Agarwal, Ben-Hong Lim, David Kranz, and John Kubiatowicz. April: A processor architecture for multiprocessing. In *Proc. 17th Annual Intl. Symp. on Computer Architecture, Seattle, Washington, U.S.A.*, pages 104–114, May 28-31 1990.

[3] G.M. Amdahl, G.A. Blaauw, and F.P. Brooks Jr. Architecture of the IBM System/360. *IBM Journal of Research and Development*, pages 87–101, April 1964.

[4] D.W. Anderson, F.J. Sparacio, and R.M. Tomasulo. The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling. *IBM Journal of Research and Development*, pages 8–24, January 1967.

[5] Arvind and D.E. Culler. Managing Resources in a Parallel Machine. In *Proc. of IFIP TC-10 Working Conference on Fifth Generation Computer Architecture*, July 1985.

[6] Arvind, V. Kathail, and K. Pingali. A Dataflow Architecture with Tagged Tokens. Technical Report TM 174, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, 1980.

[7] Arvind and Rishiyur S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[8] Arvind and Robert E. Thomas. I-structures: An Efficient Data Type for Parallel Machines. Technical Report TM 178, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, September 1980.

[9] Paul S. Barth and Rishiyur S. Nikhil. Supporting state-sensitive computation in a dataflow system. Technical Report CSG Memo 294, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, March 1989.

[10] BBN Butterfly Parallel Processor Overview. Technical report, BBN Laboratories Inc., 1986.

[11] W.J. Bouknight, Stewart A. Denenberg, David E. McIntyre, J.M. Randall, Amed H. Sameh, and Daniel L. Slotnick. The Illiac IV System. *Proceedings of the IEEE*, 60(4):369–388, April 1972.

[12] S. Brobst, J. Hicks, G. Papadopoulos, and J. Young. Id Run-time System. Technical report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, 1990.

[13] S.A. Brobst. Instruction Scheduling and Token Storage Requirements in a Tagged Token Dataflow Architecture. In *16th International Conference on Parallel Processing*, August 1987.

[14] David E. Culler. Resource Management for the Tagged Token Dataflow Architecture. Technical Report TR-332, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, 1985.

[15] David Ethan Culler. *Effective Dataflow Execution of Scientific Applications.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, 1989.

[16] Jack B. Dennis. First Version of a Data Flow Procedure Language. In G. Goos and J. Hartmanis, editors, *Proceedings of the Programming Symposium, Paris 1974.* Springer-Verlag, Berlin, 1974. Lecture Notes in Computer Science 19.

[17] Jack B. Dennis. Data Flow Supercomputers. *Computer*, 13(11):48–56, July 1980.

[18] Jack B. Dennis, G.R. Gao, and K.W. Todd. Modeling the Weather with a Data Flow Supercomputer. *IEEE Transactions on Computers*, pages 592–603, July 1984.

[19] Jack B. Dennis and D. Misunas. A Preliminary Architecture for a Basic Dataflow Processor. Technical report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, 1974.

[20] G. Fox, M.A. Johnson, G.A. Lyzenga, S.W. Otto, J.K. Salmon, and D.W. Walker. *Solving Problems on Concurrent Processors.* Prentice Hall, 1988.

[21] V.G. Grafe, J.E. Hoch, and G.S. Davidson. Eps '88: Combining the Best Features of von Neumann and Dataflow Computing. Technical Report SAND88-3128, Sandia National Laboratories, January 1989.

[22] John R. Gurd, C.C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[23] E.A. Hauck and B.A. Dent. Burroughs' B6500/7500 Stack Mechanism. In *Proceedings of the AFIPS Sprint Joint Computer Conference*, pages 245–251, 1968.

[24] Steven K. Heller. An I-structure Memory Controller. Technical report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, 1983.

[25] Steven K. Heller. *Efficient Lazy Data-Structures on a Dataflow Machine.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, 1989.

[26] W. Daniel Hillis. The connection machine. Technical report, Massachusetts Institute of Technology Artificial Intelligence Laboratory Memo 646, 1981.

[27] W. Daniel Hillis. *The Connection Machine.* The MIT Press, 1985.

[28] W. Daniel Hillis and Guy L. Steele, Jr. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[29] Kei Hiraki, Satoshi Sekiguchi, and Toshio Shimada. The SIGMA-1 Dataflow Supercomputer: A Challenge for New Generation Computing Systems. *Journal of Information Processing*, 10(4):219–226, April 1987.

[30] Robert A. Iannucci. *A Dataflow/von Neumann Hybrid Architecture.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, 1988.

[31] Introduction to the iAPX 432 Architecture. Technical report, Intel Corporation, 1981.

[32] G. Maa. Personal Communication. 1985.

[33] L.B. Montz. Safety and Optimization Transformations for Data Flow Programs. Technical report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, 1980.

[34] David A. Moon. Garbage Collection in a Large Lisp System. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–246, 1984.

[35] David A. Moon. Symbolics Architecture. *IEEE Computer*, pages 43–52, 1987.

[36] Rishiyur S. Nikhil. Id World Reference Manual. Technical report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[37] Rishiyur S. Nikhil. Id (Version 88.1) Reference Manual. Technical Report CSG Memo 284, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1988.

[38] Rishiyur S. Nikhil and Arvind. Can Dataflow Subsume von Neumann Computing? In *16th International Symposium on Computer Architecture*, May 1989.

[39] Gregory M. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, 1988.

[40] S. Sakai, Y. Yamaguchi, K. Hiraki, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *16th Annual International Symposium on Computer Architecture*, May 1989.

[41] Charles L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[42] Sequent Computers. Technical report, Sequent Computer Systems, Inc., 67 S. Bedford St., Burlington, MA 01803.

[43] Burton J. Smith. A Pipelined, Shared Resource MIMD Computer. In *Proc. 1978 Int'l Conf. on Parallel Processing*, pages 6–8, 1978.

[44] William A. Wulf. Compilers and Computer Architecture. *IEEE Computer*, pages 41–47, July 1981.

[45] Y. Yamaguch, S. Sakai, K. Hiraki, Y. Kodama, and T. Yuba. An Architectural Design of a Highly Parallel Dataflow Machine. In *Information Processing 89*, August 1989.