LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Why Dataflow Architectures

Computation Structures Group Memo 229-1
28 September 1983

Arvind

David E. Culler

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this project is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0661 and in part through various grants from the International Business Machines Corporation.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Abstract

The demand for large scale multiprocessor systems has been substantial for many years. The technology for fabrication of such systems is available, but attempts to extend traditional architectures to this context have met only mild success. We claim that fundamental aspects of the von Neumann architecture prohibit its extension to multiprocessor systems and pose Dataflow architectures as an alternative. These two approaches are contrasted on issues of synchronization, memory latency, and the ability to share data without constraining parallelism.

Key words and phrases: data flow, multiported memories, multiprocessors, packet communication network, von Neumann architecture

Why Dataflow Architectures

1. Introduction

The possibility of many independent computers cooperating on a single computational task has been a prevalent issue since the very beginning of modern computing. The technology of single processors has advanced nearly to its inherent physical limits. The production of complex computers on a single chip has become routine. The demands for performance grow exponentially. And yet, successful large scale multi-processor systems remain an elusive goal.

Computers with a handful of CPUs sharing a common memory have been built (*e.g.*, B6500, PLURIBUS, IBM 360-67, and IBM 370) and serious attempts have been made to extend these architectures to systems with many (*i.e.*, more than fifty) processors[1, 2, 3, 4]. However, even on problems with ample parallelism, these systems have not been able to achieve a sufficient level of cooperation to make use of many processors. We believe this is not simply an engineering problem, but a fundamental shortcoming in the conventional von Neumann computer architecture and the programming languages so closely akin to that method of computation. Dataflow architectures and functional languages offer an alternative model of computation, one which is novel and yet natural for parallel computation.

2. Conventional computation

The conventional method of computation is conceptually very simple; it is much like following a recipe. A specific sequence of actions is encoded and stored in a memory. These directives are sequentially read up and carried out; causing data portions of the memory to be modified. A more sophisticated program is formed by a more elaborate sequence of instructions. Programming languages, such as FORTRAN and PASCAL, allow many of the details and vagaries of such a system to be hidden, but have a similar style. The programmer has a variety of control constructs (*e.g.*, loops and procedures) with which to build sequences of instructions. He is not concerned with memory directly, but with many small symbolic pieces of memory, (*i.e.*, variables and arrays). This imperative style of programming is built entirely upon the concept of a single sequence of operations[5]. As such, it is inherently difficult to express parallelism in a conventional language.

Various methods of exploiting (local) parallelism within this context have been explored. Many processors today attempt to overlap the basic execution cycle. High speed computers provide multiple, pipelined instruction units and vector instructions. Simple vector operations can be gleaned from analysis of high level program sources, but the amount of parallelism exploited is small compared to that in the program potentially.

Let us start at the very basics; parallel computation, by its very nature, requires many simultaneous operations, synchronized appropriately, and distributed over many processors. In any multi-processor system, parallelism and dependency must be expressed in the language and exploited in the hardware. This implies certain software and hardware issues must be addressed. We will deal with these in turn. Much of this discussion is abstracted from the work of Arvind and Iannucci[6].

3. Dataflow: Software Issues

There are two basic software issues: independence and synchronization. In order to exploit parallelism it must be possible to determine when portions of the computation are independent. Conversely, if one portion of a program depends on another, they must be performed in proper sequence. The essential aspect of a program is the set of operations it specifies and the operands to each of those operations. One operation can be dependent on another if they make use of a common resource or if the results of one contribute to the input of the other. If two operations are independent, they may be executed in any order or simultaneously. This latter form of dependence, termed *data dependence*, describes the minimal synchronization required for correct execution of the program. Resource conflicts impose additional constraints, but may be removed by providing additional resources.

In a dataflow system, data dependence is the sole driving force for the scheduling of operations, and also the basic mechanism for synchronization. A program consists of a directed graph with machine operations as nodes and data dependencies as edges. A simple example is given in Figure 3-1.

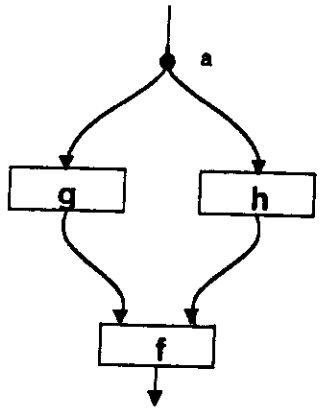

Figure 3-1: Dataflow graph for f(g(a), h(a))

Each operation depends only on the values of its inputs; it has no state (*i.e.*, no memory) and can not effect any other operation except through supplying operands. The processor's task is to propagate data values through a graph, such as this, triggering instructions when the operands are available. Data values are carried on logical entities called *tokens*; a token contains not only a data value, but also the name of the instruction to which it is destined, as part of its *tag*. Conceptually, tokens move about on the edges of the graph. Instructions are enabled for execution when tokens are present on all input edges. Upon execution, the instruction absorbs the input tokens, and produces an output token for the next instruction in the graph. A program is said to *terminate* when no enabled instructions remain[6].

The virtue of this model of computation is perhaps best seen through comparison with conventional methods. To express a computation similar to that in Figure 3-1 in a language such as

ADA, the programmer would have to designate **g** and **h** as tasks and take on the onus of determining that they are in fact independent and have completed before **f** begins. Dataflow priniciples ensure that **g** and **h** are independent, regardless of their internal natures, and that **f** is enabled, by definition, when **g** and **h** complete. This essential form of distribution and synchronization is the basis of dataflow computation.

Data dependence in the conventional method of computation arises from writing and subsequent reading of variables. Thus essential dependencies are nearly indistinguishable from incidental resource conflicts. In fact, the programmer is encouraged to introduce additional constraints in order to reduce memory requirements. This manner of communication thwarts efforts to exploit parallelism in conventional programs. The techniques for extracting parallelism from conventional programs primarily involve building dependence graphs and providing additional storage to remove resource conflicts[7]. However, these techniques do not apply in the presence of aliasing (i.e. two variables denoting the same location), which is impossible to exclude in most conventional languages.

The close relationship between dataflow and functional languages is easily observed in the example presented in Figure 3-1. A program in a functional language is built by composition of functions. Each function behaves as a dataflow operation: receiving operands and producing results. Composition is equivalent to connecting the graphs together. The result is again a function with dataflow properties[8].

4. DataFlow: Hardware Issues

Let us turn now to hardware issues. We have already mentioned mechanisms for determining when operations should be enabled. In dataflow systems this is determined by the availability of operands; in conventional systems it is determined either by the explicit control sequence of the program or by explicit synchronization constructs. The other issues are more subtle: dealing with memory latency and the ability to share data without constraining parallelism.

4.1. Memory Latency

Latency is the time between issuing a memory request and receiving a response. A multiprocessor system can be viewed, abstractly, as a collection of processing elements and memory elements, with connecting elements forming a network between. To a first approximation we can treat the complex of communication and memory elements as a multi-ported memory. Memory latency has a finite lower bound, for a given technology. Moreover, as the number of ports in the network increases, it stands to reason that the average latency of a memory request will increase, if only due to the switching time in the network. Given the rate at which a processing element can issue memory requests and the bandwidth of each memory element (bits per second per port), it is clear that at some point, the memory units, on the average, will not be able to respond to each processor request without causing the processor to idle.

The key observation, then, is that it is absolutely necessary that each processor be able to issue multiple memory requests in succession without intervening memory responses. It is also quite reasonable to assume that, because of contention, the memory responses may arrive out of order unless some constraints are placed on the memories and network interconnections.

Memory latency has always been a crucial architectural issue; the most popular approach is to exploit locality to reduce the number of memory requests. The inherent sequentiality of the von Neumann machine assures substantial spatial locality in instruction references; thus instruction prefetching is effective. However, exploiting spatial locality in data references is far more difficult. General purpose registers serve to reduce the number of references to memory, but their utility is limited in a multiprocessor context where data must be shared. A technique receiving recent popularity is to try to exploit temporal locality through the use of caches. This is particularly difficult to extend to a multiprocessor context because the problem of cache coherency arises. What happens if two processors attempting to communicate through shared memory both write into their caches? There are several solutions to this problem, but they inevitably involve substantial overhead and/or reduced parallelism.

The techniques mentioned above reduce the load on the memory, but fail to address how the processor can do useful work when it must wait for memory requests. One method of dealing with this question is to perform context switching at a very low level (analogous to task switching when an I/O request is made in a multiprogrammed operating system). While one computation waits for the memory to respond, the processor resumes another, parallel computation. Of course, the scheme works only if the context switching itself does not generate any memory references. It is accomplished by duplicating programmer-visible registers (*i.e.*, the processor state). Unfortunately, this method does not allow the system to grow incrementally by addition of processors; as memory elements are added, the depth of the communication network will grow. Hence, the number of low-level contexts to be maintained will also have to increase to match the increase in memory latency[6].

The fundamental problem is that the scheduling of operations is determined by the instruction sequence, rather than the availability of data; the entire sequence must wait because the data for one operation (a memory reference) is slow to arrive. Dataflow systems remove these artificial scheduling constraints. While one operation is delayed due to memory latency, others that do not depend on it may still execute. The critical issue becomes the bandwidth of the memory, not the latency. Memory bandwidth can be increased almost without bound, through interleaving and pipelining. Each piece of data identifies its own destination, thus the order of arrival of data is immaterial.

4.2. Sharing Data

A more troublesome issue is that of sharing data between two or more processes while maintaining proper synchronization; there must be some way to assure data is properly written before it is read. To illustrate this problem, consider two routines running on two different processors, both accessing a two-dimensional array of numbers. One routine is creating the elements, in order, and writing them into the array. The other routine is waiting to read the elements. One way to avoid a read-before-write race is to allow the *entire* array to be written before the consumer routine is allowed to begin processing. By this simple minded transfer of control, there is no synchronization problem, but neither is there any chance for parallelism. This scheme performs well on a single processor, since the computation cannot be expedited by overlapping the production and consumption of a data structure. It defeats the purpose of multiprocessing, however.

A more common scheme is to synchronize on a per-row or per-column basis of the array (as

appropriate); this incurs more overhead, but constrains parallelism less. The extreme approach would be to synchronize the two routines on a per-element basis. It should be obvious that doing so is impractical with current methods and requires fundamental changes at the hardware level.

This example oversimplifies the real situation - consider the case where the elements are not produced in a regular (*i.e.*, row order or column order) way, or the case of a nonuniform data structure. The question remains: is it necessary to sacrifice parallelism for proper synchronization of reads-before-writes?

The Tagged Token Architecture offers one solution to this problem: I-structure storage[6]. If we associate with each memory cell in a machine special flags (called *presence* bits) which indicate the memory cell's status - written or unwritten - we have the ability to solve the read-before-write race problem as follows: assume that a memory module has just received a request to read a particular memory location and to forward the contents to instruction x. The memory module interrogates the presence bits associated with that location. If the bits indicate that the cell has already been written into, the contents are retrieved and forwarded to instruction x. If the bits indicate that the location is empty, the memory module puts the read request aside, and marks the empty location to indicate that a read request is outstanding.

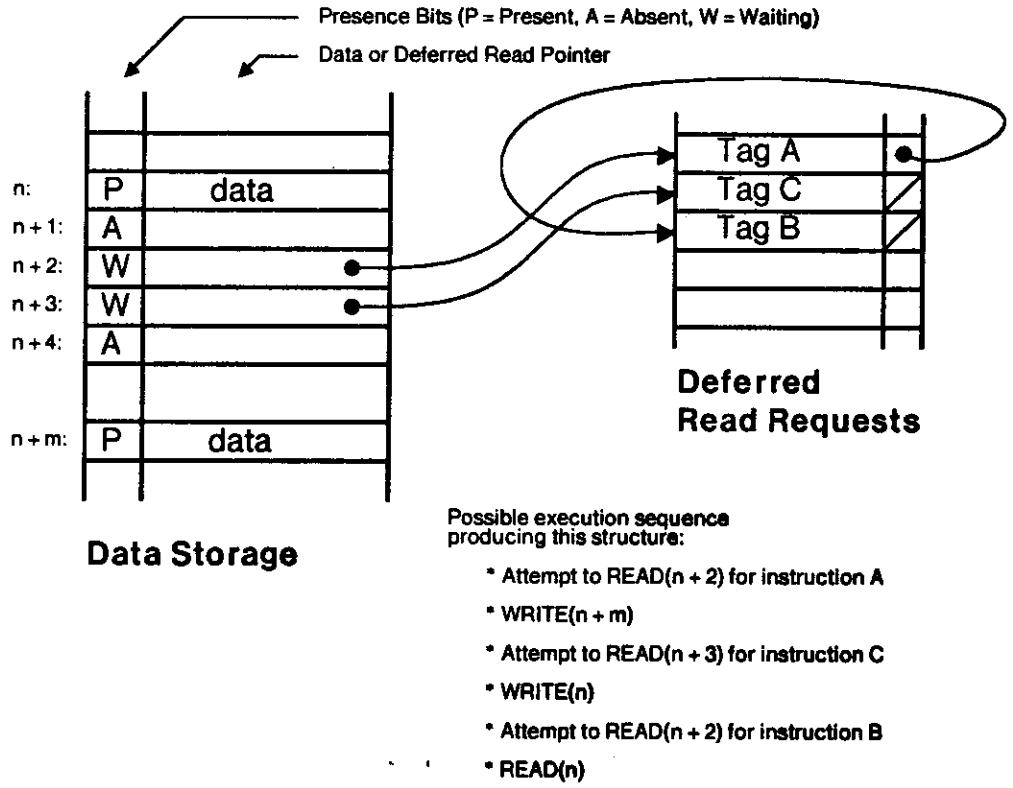

Figure 4-1: I-Structure Storage

Now, when a write request for that location arrives at some time in the future, the memory module notices the pending read request, and forwards the newly-arrived datum to instruction x (as

well as writing it into memory and setting the presence bits accordingly). Note that the memory module must maintain a list of deferred read requests (see Figure 4-1) as there may be more than one *read* of a particular address before the corresponding *write*. We call this type of memory *I-Structure Storage*. The issues involved with building such a memory, and the design for an I-Structure memory controller are discussed extensively elsewhere[9].

This mechanism, when coupled with a processor which is able to issue multiple, overlapped memory requests and which can tolerate out-of-order responses, allows the uncoupling of memory latency from the performance of a multiprocessor. The penalty of such a scheme in terms of the demands placed on memory elements is not excessive. A read operation is as efficient as in a traditional memory. Write operations take twice as long, however, due to the prefetching of presence bits. Many different implementation strategies are possible which can largely eliminate this penalty.

5. The Tagged Token Dataflow Machine

Our machine is a hardware implementation of the U-interpreter for a graphical data flow base language[8]. Programs written in any functional language which can be compiled into this base language may be executed on our machine. Such a compiler for the high-level data flow language Id is in use at M.I.T. The U-interpreter uncovers parallelism in programs during execution by uniquely labeling independent activities as they are generated. Each instance of execution of an operator is called an *activity*, and is given a unique *activity name*. Activities that have all their input values available can execute provided a processor is available.

An abstract U-interpreter machine has 5 essential subsystems:

1. a *waiting-matching* section,

2. an *instruction-fetch* section which is connected to a program memory,

3. an *arithmetic logic* unit,

4. an *output* section, and

5. a *data structure* storage.

These subsystems are connected as shown in Figure 5-1. The first four subsystems form a ring in which many tokens may circulate in a pipelined fashion. Data structure storage provides an alternative path to the other four sections. Every token carries, in addition to an activity name, data and port number, the number of partners needed to enable the destination instruction. The *waiting-matching* section matches input tokens with their partners. If a match occurs, the corresponding activity is *enabled* and matched tokens are forwarded to the *instruction fetch* section; otherwise the incoming token is stored in the *waiting-matching* section. All code blocks reside in the program memory. The *instruction fetch* section retrieves the instruction specified by the code block name and statement number part of the activity name from the *program memory*. Each instruction contains an *opcode* and the addresses of the instructions to which the results are to be sent. The opcode from the instruction and the two operands are passed to the ALU which produces

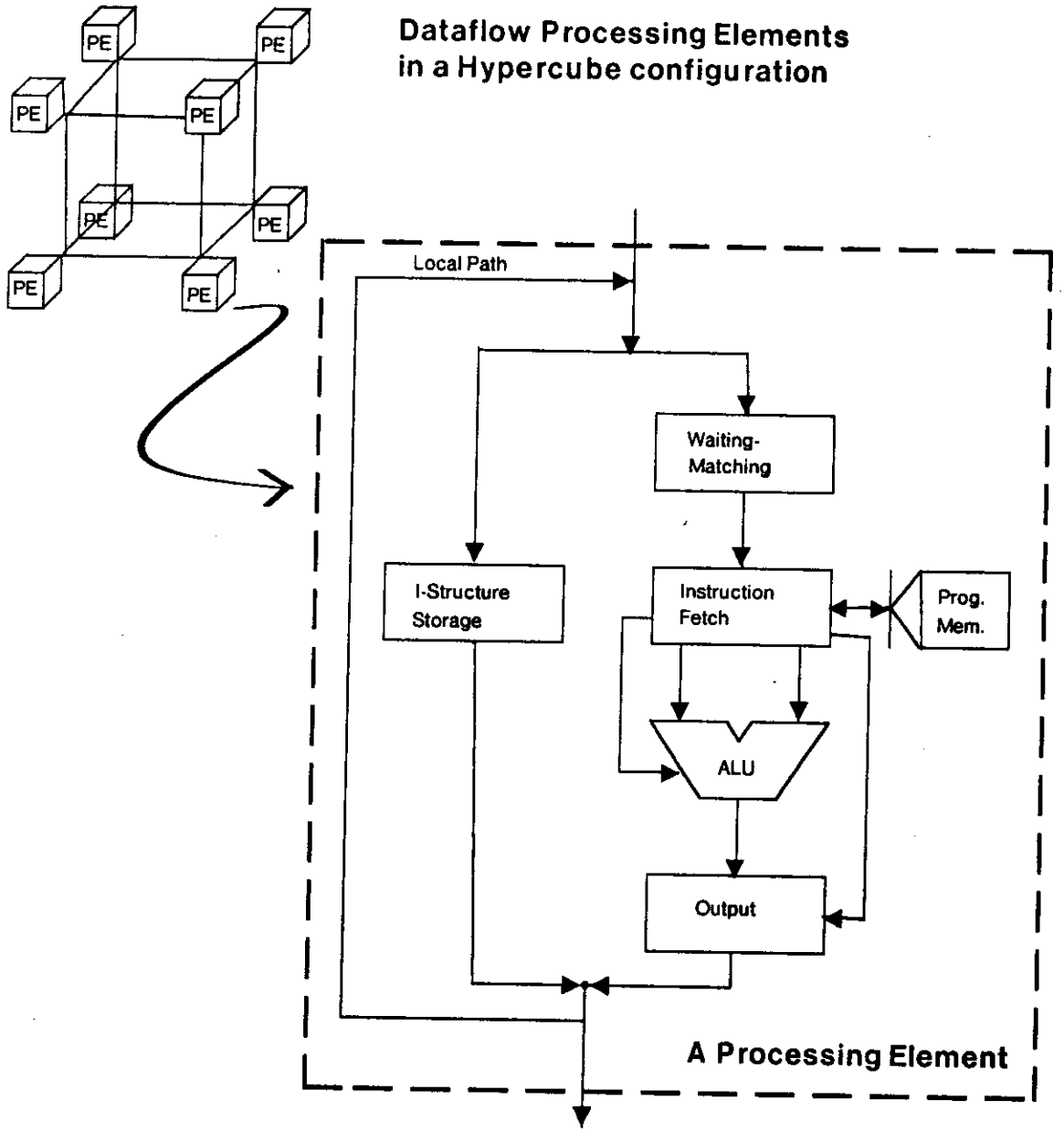

Figure 5-1: A Block Diagram of the Tagged Token Dataflow System

the result data value and passes it to the *output* section. The output section computes the new activity name for the result data value, in accordance with the U-interpreter rules, using the input activity name and the destination instruction address stored in the program memory. An output token is produced for each of the destination instructions specified in the current instruction. If the ALU executes a data structure operation (e.g., *append* or *select*) , a token for the data *structure storage* is generated and routed appropriately. The tokens for the *data structure* storage are different from the tokens that enter the *waiting-matching* section because they carry an opcode (e.g.,

read. write) and do not require matching of activity names or instruction fetching.

A multiprocessor U-interpreter machine is formed by replicating the abstract machine of Figure 5-1 and connecting these machines by a packet communication network. The system is (logically) configured into a collection of *physical domains*; these will generally be a group of PE's in which intragroup communication distances are as short as possible. Loops and procedures are compiled into individual code blocks which are loaded dynamically into physical domains, just prior to execution. The mapping of programs onto this system is divided into two phases: code-blocks are assigned to physical domains at the time of invocation by a semi-centralized scheduler, and all activities comprising the invocation of a code-block are distributed over processors in the domain by a hardware hashing mechanism which incorporates parameters set by the compiler. The mapping attempts to assign a code-block that uses a data structure in the locality of the code-block that produces the structure, and to allocate the data structure also in that locality.

The architecture of the PE for the Tagged Token Dataflow machine is closely related to the architecture of the abstract U-interpreter machine. The PE has 8 asynchronously functioning subsystems which are connected by finite size buffers and communicate with each other using a send-acknowledge protocol. A central *PE control* is included for general operations such as I/O and diagnostics. Although the activity names generated by the U-interpreter may become arbitrarily large, in the Tagged Token Machine all activity names are represented by finite size *tags*; tags must be reused as computation progresses. This is achieved by manipulating tags in a manner which is isomorphic to the way the U-interpreter manipulates activity names. The instruction set of the machine is presented by Arvind and Iannucci[10]. For details of the architecture, including address translation and tag generation, the reader is referred to Arvind, *et. al.*[11].

6. Current Status

The Functional Languages and Architecture group at M.I.T. is refining the Tagged Token Dataflow Architecture through a series of 'soft' prototypes. We have developed a detailed simulation of the architecture, in conjunction with the IBM T. J. Watson Research Laboratory, to study the relationships of each of the components in the system and to determine the performance specifications for each component. This simulation is being run stand-alone on an IBM 4341. We expect to simulate approximately 20 million dataflow instructions in 24 CPU hours.

The second project is a large scale emulation of the machine, with somewhat less internal detail. A Multiprocessor Emulation Facility is being developed at M.I.T. to support this and similar applications. It will comprise 64 Symbolics 3600 machines connected by a high speed packet switched network. The network is built out of 8x8 routers being developed by our group. These routers will provide 2000 bytes of buffering per port and a bandwidth of 4 megabytes per port per second[12]. This emulation experiment will allow us to test the viability of the dataflow approach on applications of substantial size.

The final goal is to migrate the architecture onto a small collection of custom VLSI chips. Select portions of the machine have already been implemented in VLSI as preliminary studies.

Acknowledgments

The development of the Tagged Token Dataflow Architecture has been a concerted effort of the members of the Functional Languages and Architecture Group at M.I.T. We wish to thank Robert Iannucci for his contributions and his fine graphics.

References

1. Bouknight, W.J., Denenberg, S.A., Mcintyre, D.E., Randall, J.M., Sameh, A.H. and Slotnick, D.L., " The ILLIAC IV System," *Proc. of the IEEE*, Vol. 60, No. 4, April 1972, .

2. Wulf, W., Levin, A.R., and Harbison, S., *Hydra/C.mmp: An Experimental Computer System* McGraw-Hill Book Company, New York, 1981.

3. Swan, R.J., Fuller, S.H., and Siewiorek, D.P., "Cm* - A Modular Multiprocessor," *Proceedings of the National Computer Conference*, 1977, .

4. Swan, R.J., Bechtolsheim, A., Lai, Kwok-Woon, and Ousterhout, John, " The Implementation of the Cm* Multi-microprocessor," *Proceedings of the National Computer Conference*, 1977, .

5. Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM*, Vol. 21, No. 8, August 1978, pp. 613-641.

6. Arvind, and R. A. Iannucci, "A Critique of Multiprocessing von Neumann Style," *Proc. of the 10th International Symposium on Computer Architecture*, June 1983, .

7. Kuck, D. J., Kuhn, R. H. Padua, D. A. Leasure, B. and Wolfe M., "Dependence Graphs and Compiler Optimizations," *Proceeding of ACM Symposium on Principles of Programming Languages*, January 1981, .

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

9. Heller, S. K., "An I-Structure Memory Controller," Master's thesis, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., June 1983.

10. Arvind, and R. A. Iannucci, "Instruction Set definition for a Tagged-token Dataflow Machine," Tech. report 212, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., December 1981.

11. Arvind, et. al., "The Tagged Token Dataflow Architecture", to be published as an MIT Technical Report, August 1983

12. Iannucci, R. A., "Packet Communication Switch for a Multiprocessor Computer Architecture Emulation Facility," Memo 220, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., October 1982.