LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Functional Languages and Architecture Progress Report for 1983-84

Computation Structures Group Memo 245
December 1984

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

FUNCTIONAL LANGUAGES AND ARCHITECTURES

Academic Staff

Arvind, Group Leader

Research Staff

R. Iannucci

J. Pinkerton

Graduate Students

M. Beckerle
S. Brobst
C. Chiang
D. Culler
P. Fuqua
S. Heller
R. Iannucci
V. Kathail
P. Lim
G. Papadopoulos
K. Pingali
M. St. Pierre
R. Soley
B. Vafa

Undergraduate Students

R. Adamjee
S. Viqar Ali
M. Bucci
J. Buonora
J. Cernada
T. Chambers
E. Desai
S. Douglass
S. Gahni
W. Hamdy
T. Im
C. Li
D. Morais
G. Ng
J. Ngai
C. Ozveren
J. Picciotto
K. Rahmat
R. Sathyanandan
K. Traub
W. Tsang
C.-S. Wei
A. Wang
M. Wong
C. Wu
J. Ying
S. Younis

Support Staff

P. Sedell

Visitors

E. Hagersten

I. Jacobson

1. INTRODUCTION

The primary direction of the Functional Languages and Architectures Group continues to be the study of new computer structures to exploit parallelism in application programs. Our approach in studying parallelism is based on functional languages and dynamic dataflow machines. We believe the success of a general-purpose multiprocessor computer depends on its effective programmability and efficient utilization of resources. Thus, in addition to the hardware architecture, we are concerned with high level language support, communications requirements, and efficient distribution of workload over the machine. We feel the development of novel parallel architectures will require several iterations. Hence, the group is pursuing a variety of interrelated projects, all aimed at developing our understanding of the problems and moving us closer to a final implementation. We have organized this report in two major sections: The first describes the Tagged Token Dataflow Project and the other describes the Multiprocessor Emulation Facility (MEF) for experimenting with parallel machines. The Tagged Token Dataflow architecture is going to be the first large scale emulation experiment on the MEF.

The Tagged Token Dataflow Project is a major effort to realize the model of dataflow computation embodied in the U-interpreter [3] The architecture continues to evolve as our understanding of the issues related to realizing this model deepens. The development of a detailed simulation of the architecture on the IBM 4341 in cooperation with IBM Yorktown Heights over the past year has provided invaluable experience. In order to build the simulation it was necessary to give detailed specifications of every major component of the architecture. This process uncovered a number of oversights in the early design. Moreover, in order to run programs on the simulated machine, it was necessary to develop a run-time resource management system to control the allocation of resources and distribute work over the collection of processors. This has considerably sharpened our attention on resource management issues in the past one year.

The simulator is too slow to be used as a vehicle for experimenting with large dataflow applications. Thus, a large scale multiprocessor emulation of the Tagged Token Dataflow Machine is being developed to meet this need. We expect the emulation to bring about a depth of understanding of dataflow applications far beyond the current state of the art. Currently, the emulated dataflow machine runs on five high-performance Lisp Machines which are connected by an Ethernet. programming for both the simulated and the emulated machines is done in the high-level dataflow language Id. We have an operational Id-to-graph compiler which is used to drive both prototype dataflow machines. The Id definition has been revised to permit a more elegant use of higher-order functions. We plan to implement a new version of Id in the next two years.

The goal of the Multiprocessor Emulation Facility is to develop it as a useful tool for research on new parallel architectures and associated languages. The facility will consist of 64 Lisp Machines and a high bandwidth interconnection network. Generic software for inter-processor communication and for emulating other architectures is also to be provided. In the past year, significant progress toward these goals was made. Two parallel efforts to design the communication network have been put in place. One design uses circuit switching and communication on four bit parallel data links while the other uses packet switching and bit-serial communication. The former is conservative in the use of technology and thus represents lower risk than the latter. A detailed design for the circuit switch card, excluding the Lisp Machine interface, has been completed.

We have been able to attract significant industrial support in the form of three circuit designers from IBM Endicott and one from Ericsson, to do the hardware development for the emulation facility. However, the infrastructure in the Laboratory for Computer Science for hardware development is not adequate to support the MEF. Thus, a detailed plan for a hardware laboratory was prepared and partially executed. Further construction in the hardware laboratory is contingent upon receiving more research funds which are believed to be available during the coming year.

2. TAGGED TOKEN DATAFLOW PROJECT

2.1. Architectural Background

The Tagged Token Dataflow Architecture is composed of a number of Processing Elements (PEs), connected by a packet switched communications network. Each PE is a complete dataflow computer. The basic organization is shown in Figure 7-1. The PE consists of a number of asynchronous pipeline stages, connected by FIFO buffers. The various stages form three subsystems. The subsystem shown toward the right in Figure 7-1, performs the basic instruction processing. The stages of this pipeline reflect the essential steps in the processing of a dataflow instruction: detect when data has arrived to enable an instruction, fetch the instruction, compute the result, generate result tags, and finally dispense the result tokens. The subsystem to the left provides storage for data structures. This structure store incorporates a number of innovative ideas to allow for sharing of information without constraining parallelism. The benefits of this approach are presented in [4] A detailed design of the controller for the structure store is presented in [9]. The center subsystem includes a PE controller, which provides a variety of support operations, including input/output, block transfers, and access to the resource management system.

Tokens and Tags: In the Tagged Token Dataflow Architecture, values are carried

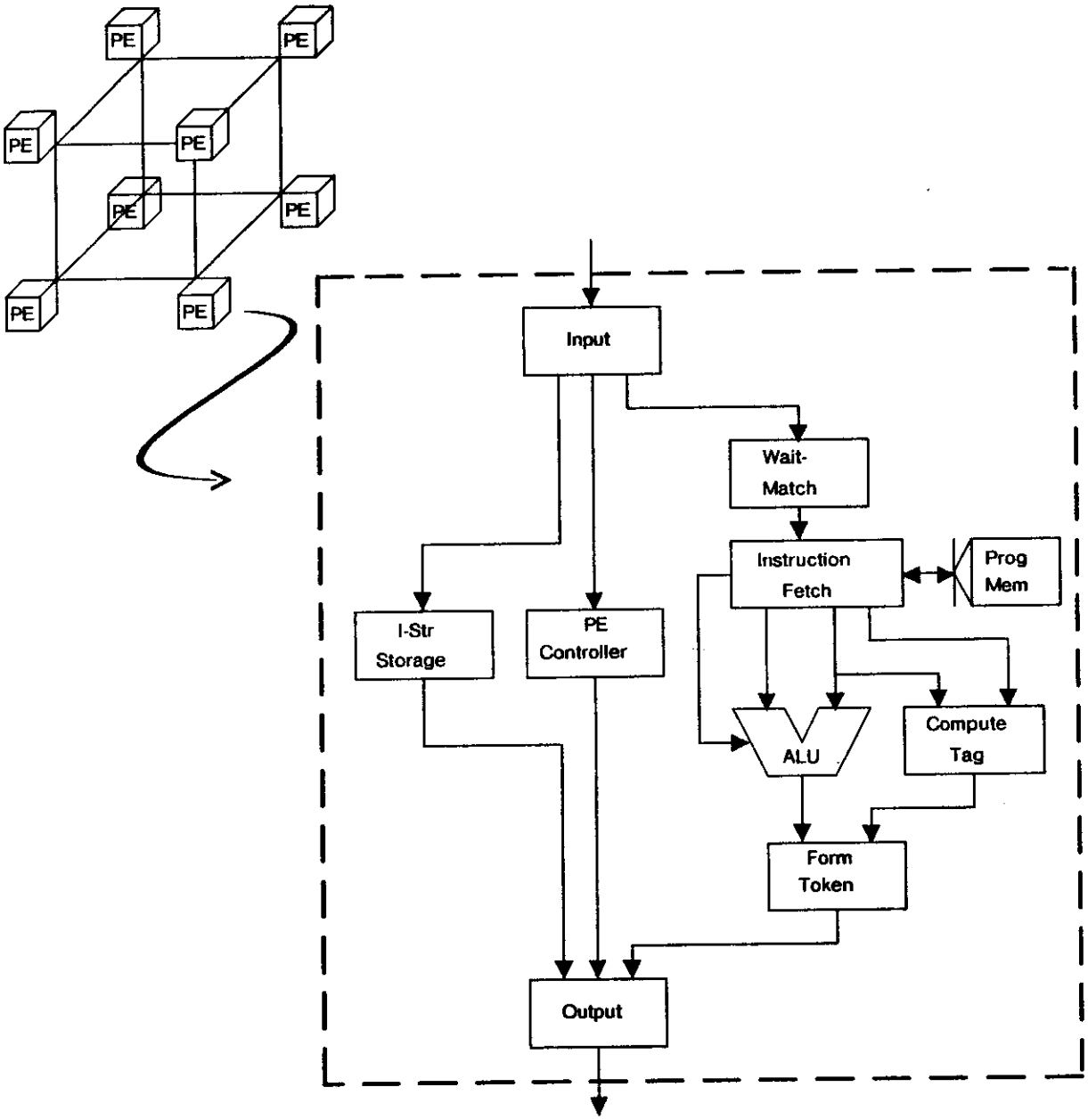

Figure 7-1: A Block Diagram of the Abstract Machine

on *tokens*, which are passed from one instruction to the next. The arrival of data causes the corresponding instruction to be fetched, unlike a conventional computer in which the execution of an instruction causes data to be fetched. There is no program counter in this machine. Each token carries a *tag*, in addition to a data value, which specifies the instruction to be executed. The tag contains essentially three items of information: the address of the PE which is responsible for executing the instruction, the address of the instruction to execute within that PE, and the *context* in which the instruction is to be executed. The PE address is required because a code-block may be spread over many PEs, and tokens must be freely transferred between PEs. The contextual information is required because many logically distinct activations of a given code-block may be in execution simultaneously. There must be a way to distinguish the various activations so that tokens belonging to different activations do not interact. All tokens belonging to a given activation carry the same context identifier or *color*. Thus, two tokens are destined for the same instance of an instruction if and only if their tags match.

Instruction Processing: Upon arriving at a processing element, a token enters the waiting-matching section. The tag it carries is compared against the tags of all the tokens resident in the waiting-matching store. Instructions are limited to two operands, so if a match is found, the corresponding instruction is enabled for execution. The two matching tokens are purged from the waiting-matching store and forwarded to the *instruction fetch* section. The instruction specified in the tag is fetched from program memory, along with any required constants. The data values are aligned, and an operation packet is sent to the *ALU* for processing. In parallel with the ALU, the *compute-tag* section forms tags for result tokens, based on the destination list of the instruction and contextual information on the input tag. The result values and tags are merged to form tokens and passed on to the communication network, whereupon each is delivered to the PE specified in its tag.

Tolerance to Communication Latency: In many respects, a multiprocessor setting presents a fundamental architectural challenge. Communication latency between processors is generally large and unpredictable. Thus, for a multiprocessor architecture to be successful, the individual processing elements must be extremely tolerant to communication latency [4]. The PEs which comprise the Tagged Token Dataflow Architecture meet this challenge. Note that once an instruction is enabled, it may be processed to completion without further communication with other PEs. The pipeline is never held up by communication latency. Waiting only takes place in the waiting-matching section. Completion detection for external communication is provided naturally by the basic instruction scheduling mechanism; when data arrives an instruction is scheduled. This dynamic scheduling, coupled with the ability to interlace many independent threads of computation, allows for overlapped requests to the communication system, tolerates long latencies, and tolerates unordered responses.

2.2. Resource Management

The description of the architecture presented above assumes that the program graph is in place, distributed appropriately over the machine. This section looks at the process of executing programs on the Tagged Token Dataflow Architecture at a somewhat higher level. It focuses on how resources are managed and how work is distributed over the machine.

Static vs. Dynamic Resource Allocation: There are basically two approaches to resource allocation, each offering advantages and disadvantages. On one extreme is the static approach where the compiler assigns processors and storage to a program. This is a common strategy for scientific programs written in Fortran. If the decomposition is just right, the performance can be extremely good, with little overhead for the distribution of work at run-time. However, developing a good static decomposition appears to be a very difficult problem. In any case, functional languages require dynamic resource allocation and thus preclude a purely static approach. A purely dynamic approach, on the other hand, implies allocating each computational activity on the least busy processor. Dynamic allocation on the Tagged Token machine is undertaken only at the level of procedure calls, (i.e., code-block invocation), each of which is assigned to a group of PEs at run-time. David Culler has developed a dynamic resource management system for the simulation along these lines. This resource management system has also been adopted for the emulated dataflow machine. We are currently experimenting with various allocation algorithms and load balancing techniques. The present architecture provides an efficient and flexible way to map loops and recursive procedures on a group of processors. As static analysis techniques develop, we will consider more aggressive optimizations for mapping special program structures.

Hierarchy of Resources: A variety of resources are required to support a code-block activation. These include PEs, program memory, code-block registers, colors, etc. Coordinating resource allocations between a variety of PEs can be extremely cumbersome, if PEs are allowed to cooperate in arbitrary ways. In order to keep the complexity of resource management tractable, a hierarchy is imposed on the set of system resources. In effect, the resource manager deals with blocks of resources and allows the hardware to distribute them at a finer grain automatically. This also allows the number of requests to the manager to be reduced.

The notions of *physical domain* and *physical subdomain* are introduced to facilitate selecting a set of PEs for an invocation. The collection of PEs in the system is divided into disjoint *physical domains* (PD), each being a set of consecutively addressed PEs. This partition is not allowed to change while programs are running. Each PD may be further partitioned into a set of disjoint *physical subdomains* (PSD), again each is a set of contiguously address PEs. The size of the PSD is dynamic and will vary for different code-blocks in a given PD. A code-block activation is assigned to a domain, and a complete copy of the code is placed in each subdomain.

To simplify memory management, we require that memory allocation be identical in every PE in a given domain. The code-block is split into as many sub-blocks as there are PEs in a subdomain. Each PE receives one sub-block, all starting at the same base address. This allocation is replicated for each subdomain in the domain. For computational efficiency we require that all PD and PSD be a power of two in size.

Each copy of the code-block may be used by many concurrent activations and each of these may require mapping information to be stored locally to the PEs, so that result tags can be generated. Again, by grouping activations together this mapping information can be kept within reason. Each PE contains a set of 256 *code-block registers* (CBR). Each code-block register contains a code-base address and information pertaining to the mapping of the code-block onto the domain. Each code-block register can support 16 *colors*; additional information can be associated with each color. Code-block registers, like program memory, are allocated uniformly throughout each domain. With this partitioning, the manager views the system resources in terms of domains, code-block registers, and colors.

To understand the role of the manager, consider the process of code-block activation. A request is sent to the resource manager. It first selects a domain with sufficient resources to support the activation. If the code-block is not present in this domain, it will have to be loaded. In this case, the subdomain size may be set as suggested by the compiler, or as determined by the resource manager. A code-block register must be allocated, and a set of colors, relative to this CBR, is assigned to this activation. Finally, program memory is allocated throughout the domain and the code is loaded. Mapping information is established in each PE to describe the disposition of the code-block, as part of the CBR. Further information pertaining to each color can also be provided. The code-block is now ready to execute. The manager allocates argument and result structures and sends descriptors to both the caller and the newly activated code-block. The code-block may now execute on its own except for invocation and resource allocation requests. Subsequent invocations of the same code-block may share the CBR, if colors are available. If not, a new CBR must be allocated, but it may share the copy of the code. In either case the mapping parameters will have to be the same as the original, since the code has the same disposition. New color information may be provided.

These resource management concerns essentially dictate the structure of the tag carried on tokens. It consists of five fixed size fields <PE #, CBR, Color, Initiation #, Relative Instruction Address>. The code-block register gives the base address of the code and the base address of the color information. This allows instructions and constants to be fetched from program memory. The CBR also describes the disposition of the code-block across the domain, so the hardware can generate tags for result tokens.

Efficient Internal Invocation: The resource manager provides a mechanism for initiating activity in a possibly distant region of the machine. However, in many cases this generality is not required and is overly expensive. In order to allow highly repetitive program structures to execute without involving the resource manager, 256 initiation numbers are associated with each color. Loops and recursive procedures make use of the initiation numbers to distinguish tokens belonging to different iterations, or different recursive invocations. A code-block activation is allotted a color and has the range of initiation numbers at its disposal. Loops utilize the D operator, which increments the initiation number. Recursive procedures make use of the R operator, which computes a new initiation number of the form A * I + B. Many loops may exceed 256 iterations, hence facilities are provided for allocating blocks of colors and for using the next color in the case of initiation number overflow. The compiler generates code to test whether all colors and initiation numbers have been exhausted and to issue a manager request for more colors if necessary.

The hardware provides a mechanism to distribute internal activations over processors efficiently. This is the rationale for partitioning domains into subdomains. Recall, each subdomain contains a complete copy of the code. The hardware distributes activations within a domain based on the initiation number. An additional parameter is provided to support block distribution (i.e., k iterations on this PSD, k on the next, and so on). This follows along the lines discussed in Arvind, et al. [2].

The results of static analysis are expected to guide the resource manager in making allocations. In particular, it enters into the process of choosing among domains, choosing subdomain size, choosing k (the number of iterations to keep together), and for deciding where to allocate data structures. Conveying this information is somewhat subtle, however. Completely static information, such as nesting relationships, can be included in the compiler output along with the code. Execution dependent information is supplied by augmenting the graph slightly and sending information along with the request. Currently, only simple information is conveyed in this way, such as the expected number of iterations. But we are investigating more powerful aids of this form.

Compiler Responsibilities: Introducing a resource management system into a dataflow model raised a number of interesting compilation issues. The U-interpreter is entirely independent of resources; it assumes unbounded computational resources. In order to execute dataflow programs on a practical machine, resources must be explicitly requested. They must also be released in some manner. The basic dataflow graphs must be augmented so that the resource manager request is generated when resources are required. Also, the compiler must generate code to make proper use of blocks of resources. For example, to use initiation numbers

properly, it is necessary to handle overflows. The compiler must generate code to determine when resources can be released. This involves detecting completion of code-block activations and proper handling of reference counts for structures.

Detecting completion is non-trivial in a dataflow system because a code-block may continue to execute after all interesting results have been produced. Many iterations of a loop may be active simultaneously, so the compiler must detect that all have completed. For acyclic graphs there is little trouble. A signal token is generated for any operators which have no destinations. These and all the output arcs of the graph form the inputs to a binary reduction tree. The token that falls out the bottom of the tree is the last token of the activation. This technique can be extended to loops, but care must be taken to avoid serializing loops that would otherwise provide parallelism. The special signal token of one iteration is part of the completion tree for the next. In effect completion detection is serial, even though the loop may be parallel. The detection simply lags behind the loop execution.

One important implication of this incremental completion detection is that it makes it possible to run loops of an arbitrary number of iterations using only a few colors. The loop is given multiple colors. As it exhausts one color it goes on to the next or requests more. The completion follows behind detecting and releasing colors that are no longer needed and, hence may be recycled.

The compiler must also assist in the detection of parallelism in the construction of data structures. When certain restrictions apply, arrays can be modeled as I-structures which allows for a more parallel implementation than for ordinary structures. Detecting those arrays which can be implemented as I-structures is difficult in general; it involves analysis similar to that done by vectorizing compilers during code vectorization. There are, however, several important special cases where I-structures can be detected easily. We have also found several situations in which the techniques of vectorizing compilers can be borrowed and applied. The compiler generates instructions for the run-time system to allocate I-structure storage. For storage reclamation, reference counting can be implemented because I-structures are acyclic. The overhead of maintaining reference counts can be substantially reduced if, with compiler assistance, they are incremented and decremented at procedure call and exit only.

2.3. Simulation Experiments

The primary purpose of the simulation facility is to provide a prototype and test bed for the Tagged Token Dataflow Architecture. The basic requirement in designing the simulator was that all aspects of the architecture be modeled in a manner that could be cast directly into hardware. Deficiencies in the architecture were to be exposed and remedied, rather than disguised by software tricks. Working through the

implementation of the machine in this manner did indeed uncover shortcomings and caused the specification of the architecture to become much more precise. The architecture is basically an asynchronous pipeline, with stages connected via finite sized buffers. This modularity is reflected in the design of the simulator; the functional behavior of each stage is modeled by a Pascal procedure which receives an input packet and the state of a station and produces a list of result packets with a new state. Thus, the specifications of the major hardware components are easily discernible from the simulator software. Also, the modeling of buffer interactions is divorced from the modeling of the functional behavior of the components. This design has greatly simplified the check-out task and allows the architectural specifications to be easily modified.

The first requirement of this 'soft' prototype is that it support all operations defined in the instruction set [1] and provide enough resource management support to run substantial Id programs. This requirement has been met. Most of the code has been written by Culler, Brobst, Vafa and Wei. In the coming months, the simulator will provide a vehicle for analyzing the architecture. The initial set of experiments are directed at identifying potential architectural failures: buffer deadlock, insufficient waiting-matching facilities, and inordinate overhead. A larger body of experiments will focus on determining the proper balance of various components (e.g., buffer sizes and processing speeds) and how various factors affect the performance of the machine.

The simulation facility has provided valuable experience in programming, debugging, and diagnosis in a multiprocessor setting. The simulator pursues many computations in parallel. Thus, debugging a program being run on the simulator involves all the subtleties of debugging on a true multiprocessor: what does a breakpoint mean? what would be a meaningful trace? etc. We have developed a debugging tool which will allow the user to examine the invocation tree (i.e., the parallel counterpart to a conventional procedure trace) and to observe the execution of a specific code-block at the instruction level. Debugging the simulator itself has proved much like diagnosing hardware; bugs have to be isolated by running diagnostic programs on the simulator and inferring from their misbehavior. We are developing a significant diagnostic package which will carry over to hardware implementations of the Tagged Token Dataflow Architecture.

2.4. Emulation Experiments

As a further proving ground for the Tagged Token Dataflow Architecture on large applications, we are developing a high speed emulation of the architecture on the MEF. The first version of the software for this emulator has been completed, allowing us to run graphs generated by the Id compiler. The emulated dataflow machine can be configured to run several dataflow PEs on each Lisp Machine which

communicates with other Lisp Machines over a ten megabit Ethernet. Higher-speed networks are currently under development and the software is structured so as to allow machines to easily integrate with such networks when they become available.

Currently, enhancements to support debugging in an asynchronous environment and a user interface consistent with the simulator are being developed. The performance of the Lisp version of the emulator is roughly 200 dataflow instructions per second on a single Lisp Machine. If scaled up to 64 processors, this would be approximately 13,000 dataflow instructions per second. However, we have not yet begun optimizing performance by finding bottlenecks in the Lisp code or moving critical pieces of the Lisp code into micro-code, so we are confident that we can obtain our ultimate performance goal of 64,000 to 640,000 dataflow instructions per second after some fine tuning. These issues are discussed further in Section 3.

2.5. Compiler Development

Two compilers for the dataflow language Id have been developed. One of them translates Id into Maclisp, and provides a complete run-time system to execute the object code; the other one translates Id into dataflow graphs where each node in the graph is a Tagged Token Dataflow Machine instruction. The Id-to-graph compiler is also responsible for generating instructions that ask for allocations and deallocation of resources. Both compilers are written in Maclisp, and run on DEC-20 as well as on Lisp Machines. The first compiler was written by Kathail and Pingali, while the second one was written by Kathail alone. Work, in cooperation with IBM Yorktown Heights, is underway to run these compilers on IBM machines using YKTLISP.

Further work on compilers can be classified into the following three categories:

1) Develop and implement algorithms for the Id-to-graph compiler for doing static analysis of the programs to acquire information that may be useful to the system manager. This includes finding producer-consumer relationship among code-blocks and finding sizes of various data structures to name a few.

2) Develop and implement optimization techniques to improve the efficiency of the code generated by the Id-to-graph compiler. This includes optimizations like constant subexpression elimination, code-block merging to eliminate the setup time associated with a procedure call, as well as reduce the number of calls to the manager and constant propagation.

3) Incorporate streams, managers, and a declarative or deductive typing system in both compilers.

2.6. Higher-order Functions and Reduction

It has often been remarked that much of the power and elegance of functional languages stems from higher-order functions i.e., functions that can take functions as arguments or return functions as the result of application. The use of higher-order functions has been stressed by researchers interested in *reduction* and reduction languages; in particular, by Turner [17], Backus [6] and Burge [8] As part of our design effort, Arvind, Kathail and Pingali [5] re-examined the primitives provided in Id for programming with higher-order functions and found them deficient in several respects. In particular, we were dissatisfied with the *compose* operator of Id since we found that it was clumsy to use and led to contorted programs. Our research into languages based on the reduction (in particular, SASL [16] model of implementation has led us to revise Id in order to permit efficient implementation and ease of use of higher-order functions. In addition, this research has lead one of our group members, Kenneth Traub, to design a novel architecture for performing parallel reduction [15] The proposed architecture has several advantages over current proposals in the literature for performing parallel reduction.

Given an expression in a functional language, reduction interpreters attempt to simplify the expression by applying a "rewrite rule" to generate another expression which can be simplified in turn. By doing this repeatedly, the interpreter generates a sequence of expressions; if, at some point, an expression is generated which cannot be simplified further, the interpreter returns that expression as the answer. Expressions that cannot be simplified further are called *normal forms* and the number of reduction steps taken by an interpreter to produce the normal form of an expression is usually taken be be a measure of the work performed by the interpreter in reducing the expression. Examples of reduction-based interpreters are the normal-order and applicative-order interpreters for the λ-calculus.

Since most functional languages can be considered syntactic sugar for the λ-calculus, our research concentrated on interpreters for the λ-calculus. Our goal was to determine the factors which affected the number of times identical sub-expressions were evaluated. We believed intuitively that if an interpreter shared larger sub-expressions, it would take fewer steps to find the normal form. To our surprise, we found that this was not always true. We have shown in [5] that Wadsworth's *fully lazy* interpreter [18] shares more sub-expressions than Henderson and Morris' *lazy* interpreter [10] at each reduction step. Furthermore, we have given a λ-calculus expression for which the *fully lazy* interpreter takes more steps than the lazy interpreter. This led us to define the notion of *weak normal forms* for which it is in fact true that an interpreter that does more sharing takes fewer steps to find the answer. The importance of weak normal forms is that practical interpreters return weak normal forms as answers. We then showed that it is possible to transform any expression so that a lazy interpreter reducing the transformed expression would

share the same sub-computations as a fully lazy interpreter reducing the original expression. Finally, the effect of representing λ-expressions as combinatory forms was explored. It was found that sharing was affected not by the alternative representation itself but by the abstraction algorithm used to transform the λ-expression into the combinatory form.

3. THE MULTIPROCESSOR EMULATION FACILITY

3.1. The Emulator as a Prerequisite to the Dataflow Machine

The Tagged Token Dataflow Machine and associated programming environment represent a radical departure from both the hardware and software for parallel processing based on von Neumann processors communicating via shared memory. A real challenge in building the first Tagged Token Dataflow Machine is to solve two highly interrelated problems. Firstly, there are few large dataflow programs whose dynamic behavior is well understood; estimates of dynamic behavior must be based exclusively on the analysis of the source code expressed in dataflow languages because of a lack of facilities for executing these programs. This method of gaining insights into the behavior of dataflow programs can be applied to a very limited number of application programs because it requires close cooperation of dataflow and application experts. Application experts have little motivation to spend time to understand the behavior of their applications on "hypothetical computers". Secondly, the architecture of the Tagged Token machine is so different from conventional processors that it is difficult to estimate some architectural parameters (e.g., size of buffers) whose effect on the overall performance of the machine may be critical, without some concrete assumptions about the dynamic behavior of programs. One way we are trying to resolve this paradoxical situation is by implementing the Tagged Token Dataflow Machine on a multiprocessor emulation facility.

3.2. The Emulator as an Interesting Parallel Processor

In addition to providing a highly productive vehicle for the direct emulation of large scale dataflow programs, the Emulator is an interesting and innovative multiprocessor in its own right. It is the first MIMD machine which can support general, distributed, asynchronous processes that require relatively large amounts of interprocess communication *and* can do so much more rapidly and be more cost-effective than the equivalent simulation on a single very large SISD machine. This unique characteristic is a result of two fundamental design decisions.

First, the Lisp workstations being used to build the facility are themselves powerful machines with sophisticated programming environments. The power lies in the

supermini class internal architecture of a high-performance micro-engine, wide data paths, and a high-speed memory subsystem. The machine also has an I/O structure capable of supporting a high-performance interprocessor communication mechanism. The associated Lisp system provides a good environment for the development of cooperating asynchronous processes, and thus also a good vehicle for architectural exploration, assuming that each processing element (or each of its subsystems) is viewed as a process.

Second, the interprocessor communication networks provide an instrumented, flexible, and inherently fault-tolerant data transport mechanism that is well-matched to the processor's throughput. The network is easily reconfigured and allows evaluation of a variety of communication strategies including perfect n-cube, planar, token ring(s), and shuffle exchanges. This permits empirical exploration of the effects of interconnect topology and bandwidth on the overall machine performance. The communication controller design is matched to the high-performance of the Lisp Machine. It provides its own horizontal microcontrollers to permit transmission and reception of messages concurrently with the normal program execution. Each switch node supports an aggressive instrumentation, test, and maintenance subsystem as well as managing the routing tables for the currently supported network topologies.

In a sense, these two features are not new - at least when taken independently. It is precisely the fact that mature, high-performance von Neumann machines like the Lisp Machine and the theory of n-dimensional packet switching *exist* that reduces the risk, and thus the time to completion, of the Emulator. What makes this interesting is that this level of power, a necessary level to run meaningful programs, has never been assembled in such a flexible and balanced fashion. Previous implementations have either lacked processor performance, productive programming environments or, the common problem, a well-matched communication system that does not impose excessive processor overhead. Compromises in design either due to cost or narrow intent haven taken their toll. This really marks the first time that a parallel machine can directly execute and instrument a reasonably broad class of distributed programs *significantly faster than* an equivalent simulation on the largest uniprocessor systems.

3.3. Hardware Development

We view the Emulator as being an evolutionary step toward our goal of building a VLSI Tagged Token Dataflow processor. It will allow us to properly study this radically different architecture without the usual constraints and commitment of resources that are required for a typical hardware project. Also, the component of the Emulator which is being *invented*, namely the communications network, is crucial to the realization of a Tagged Token Dataflow Machine. The network is being

designed in three phases, all oriented toward developing VLSI for packet communication over high-speed serial point to point circuits. The *circuit switch* is the first phase, while the *packet switch* and its VLSI version are the final two phases.

The Circuit Switch: In an effort to get a network operational, we decided to adapt the network used by Bolt, Beranek, and Newman in their Butterfly multiprocessor [14]. The switching node in this network is a 4x4 crossbar with 4 bit wide data paths. These nodes are interconnected and centrally clocked to form a synchronous network of arbitrary topology. Messages traversing the network carry an encoding of the network path they are to take. The adaptation of the BBN design required re-engineering the switch so that the network will be built solely out of cards plugged directly into the backplanes of our Lisp Machines. As a consequence, the design had to allow for longer inter-switch cables. More importantly, we are designing the necessary buffering and control logic so that the network presents as little processing load on the associated processors as possible. Greg Papadopoulos and Eric Hagersten have been doing this design.

We believe there is a high probability that a system based on the circuit switch will be operational within a year, as opposed to a system based on the packet switch which may not be available for two years. In a move to reduce duplication of effort, we have constrained both switch designs to share major subsystems. Our re-implementation of the BBN circuit switch has proceeded smoothly since September. We now have three of the four major subsystems designed and entered into our Computer Aided Engineering station. An extensive discussion of the development plan for the circuit switch can be found in [13].

Packet Switch: We think that a good network structure for the Emulator is a hypercube of high speed, bidirectional, serial interconnections with full *store and forward* logic at each switching node [12]. The exploratory design work by Robert Iannucci has shown the approach to be within the capability of modern engineering techniques [11]. Nevertheless, to achieve the target speed and reliability, the serial link drivers and receivers within the switch will be implemented with a mixture of analog and digital circuits. Further, the internal logic of the switch places strong demands on the circuit technology for speed and, more critically, predictability. Off the shelf logic typically has a very wide spread of "acceptable" performance specified by the manufacturer. This large spread, when combined with the desire to produce a robust design, forces the use of worst-case performance figures. This typically translates directly into a bloating of the design in terms of chip count. It should be noted that we are interested in medium sized volumes of assembled and *tested* switch cards because of our own needs and anticipated need for those who wish to replicate the Emulator.

To solve these problems, we have reached an agreement with IBM under which

they have installed a department of four full-time engineers at MIT for three years. These engineers will help in the design, prototyping construction, and testing of this packet network. We are studying the possibility of using an IBM serial data communications circuit of their own design which more than meets our performance goals. In addition, IBM will give us access (indirectly through their engineers) to a medium capacity bipolar gate array technology and to their Engineering Design System. This should solve our circuit testing problem because IBM will provide us with tested gate array chips.

During the spring of 1983 a project to study the feasibility of a VLSI version of the bit-serial transciever was undertaken in the advanced VLSI subject at MIT. It resulted in the preliminary design of a 100 MHz serial data communications circuit based on the MOSIS 1.2 μm two layer metal cMOS process [7]. The transmitter section develops a Manchester coded data stream using asynchronous finite state techniques. The receiver re-extracts data and clock from the Manchester waveform using an on-chip phaselock loop.

We are continuing this VLSI effort because the bit-serial transreciever will be a retrofitable part of this switch and has applications far beyond the Packet Switch. In VLSI chips, the transistor to I/O pin ratio will continue to increase and thus, as the use of pins for inter-chip communication will rely more heavily on time multiplexing techniques in the future. Our VLSI version of the bit serial transreciever set is very robust and implements full flow control. It will allow a multichip system to be designed according to the *synchronous on-chip, a synchronous across-chip* methodology. Thus, traditional chip design techniques can be employed and the need for centralized clocking eliminated.

3.4. Emulation Software

The Multiprocessor Emulation Facility, developed by Richard Soley, is a large body of Lisp software which allows its user to quickly and easily prototype a single- or multiprocessor computer architecture, and to then execute the proposed machine in emulation. In order to achieve high emulation speed without excessive cost, and to force systems architects to begin thinking of computers in a distributed, multiprocessing way, the Emulation Facility is designed to execute on many parallel Lisp processors, linked via a variety of communications media.

The general case of a multiprocessor is a group of various asynchronous independent *processing elements* with no shared state, connected by a packet communications network of arbitrary connectivity. MEF supports this type of architecture directly, by providing a set of software abstractions which facilitate the modeling of such an architecture in the Lisp language. The abstractions include programs to define *experiments* and *logical processors*. An *experiment* consists of

the number and type of processors being emulated, their interconnection topology, and other information pertinent to running the architecture in emulation. A *logical processor* is a Lisp program which exhibits the behavior of a physical processor in the multiprocessor architecture by modifying local data representing the state of the processor, and communicating with other processors by calling MEF's message sending primitives. The MEF will also provide support for dealing with the lower-level issues of handling networks and protocols used on them, configuring emulated topologies on top of different physical interconnections of the Lisp Machines, etc.

In addition to these tools, the system includes more mundane subsystems, such as the *Control Panel*, which acts as the bootstrap-load processor of the system, configuring the Lisp Machines taking part in a particular emulation experiment, broadcasting the experiment to be executed, and setting up the communications media for the desired interconnection topology. The control panel subsystem also provides primitives for collection and display of various statistics, either during execution, or afterward.

The MEF system automatically distributes logical processors in the experiment over the physical Lisp Machines currently configured into the facility. In this way, an experiment that requires sixteen logical processors can be run on the facility using one, two, or more physical Lisp Machines. In addition, the Lisp Machines may be partitioned into separate sub-facilities by the control panel, allowing multiple emulation experiments to run at the same time.

Although the abstract structure outlined above was chosen for its generality in emulating multiprocessor configurations, the skeptic will immediately note that it does not directly support such multiprocessor designs as synchronous processors (like the Connection Machine, or Illiac IV) or the shared memory model (e.g., C.mmp). In fact, this model is abstract enough to allow prototypical implementation of even these models of parallel computation; generally, another virtual processing element is added to the emulation experiment definition to model this shared resource (e.g., clock or memory) of the system. This is actually quite close to the real hardware implementation of such a system, in which a central clock for synchronous operation or a central shared memory will actually be a separate subsystem of the architecture. In addition, interconnection schemes other than packet-switching networks can be simulated on top of our packet switch by using the packets to simulate individual items in a continuous stream communication mechanism.

The implementation of the Multiprocessor Emulation Facility is proceeding smoothly. We currently have an emulation system, written entirely in ZetaLisp, running on five Symbolics 3670 processors. We continue to utilize the ten megabit Ether network, executing Chaos protocols, for our communications medium. As

work progresses on the circuit and packet switch hardware, we are in the midst of preparation for use of those new and faster media.

We currently have two architectures in emulation. The first, written by Richard Soley, is a simple von Neumann style machine, emulated as two logical processors (one *CPU* and one *MEMORY* box) connected via a *bus* (all communications actually pass through the Emulation Facility software, as described above). This simple emulation provided a vehicle for debugging parts of the system.

We also have an emulated version of the Tagged Token Dataflow Machine running on the Emulation Facility. This emulation, developed by Richard Soley, Paul Fuqua, and Poh Lim, fully supports our current definition of the Tagged Token Dataflow Architecture. We are already using the dataflow emulation experiment to tune the design of the Tagged Token Dataflow Architecture. We are executing Id programs, compiled into machine code, at the rate of one hundred dataflow machine instructions per second.

We have an immediate pressing need for more speed in the dataflow emulation; therefore, selected parts of the emulation facility and the dataflow experiment itself are being hand-tuned to reach the goal of one thousand emulated dataflow machine instructions per second.

4. RELATED TOPICS

4.1. Logic Programming

A great deal of attention has recently been given to the Fifth Generation Computing project in Japan, mostly because of its focus on Logic Programming as the programming method of choice. Logic Programming shares the "single assignment" characteristic of Functional Programming, as well as the declarative style; hence, it is of great relevance to our group. In order to learn more about Logic Programming, during the IAP in January '84, an informal, week long workshop was held. Talks were presented by Jan Komorowski, who was invited from Harvard, Michael Beckerle, Gary Lindstrom (a visiting scientist from the University of Utah), Arvind, and Gordon Robinson (from the AI Laboratory). The talks covered topics ranging from the fundamentals of logic programming to current efforts in the parallel processing of logic programs.

4.2. New Equipment

Five Symbolics 3600s were acquired as the first of 64 machines for the Multiprocessor Emulation Facility. These machines are in the process of being

upgraded to 3670s. Each machine has 6 megabytes of main memory and one machine is equipped with 500 megabytes of disk storage for file service. An additional 1.6 gigabytes of disk memory was installed on the group's VAX-750, also for file server use by the 3670s. A total of sixteen 3670s are expected by the end of this year. Several IBM PCs, networked onto TCP/IP-speaking Ethernet, were also obtained. The PCs are used for local editing, software development, and laboratory instrumentation control.

4.3. Support Tools

As his first UROP project, Dinarte R. Morais, an undergraduate member of the group, implemented an interactive illustrator program, which is now being used by members of both the LCS and AI labs. The program, called ILLUSTRATE, is an interactive illustrator for creating pictures composed of lines, curves and text captions, and was modeled after the DRAW program available on Alto computers, as described in the Alto User's Handbook.

The problems with using the existing DRAW program on the Alto computers include the fact that pictures could not be very complicated due to the relatively small amount of memory available. In addition, DRAW came *as is* and consequently could not be customized. Finally, the few Alto computers left are getting older and less reliable, and it was hoped that ILLUSTRATE could allow our group to finally do away with the Altos.

ILLUSTRATE was implemented in ZetaLisp on a Symbolics 3600 Lisp Machine. The advantages of ILLUSTRATE over DRAW are many. One advantage is that there is no practical limit to the complexity of a picture created with ILLUSTRATE. More importantly, we now have the ability to customize the program to suit our needs.

References

1. Arvind and Iannucci, R. A. "Instruction Set Definition for a Tagged-Token Dataflow Machine," Massachusetts Institute Technology Laboratory for Computer Science, Computation Structures Group Memo 212-3, Cambridge, MA, February 1983.

2. Arvind, Culler, D.E., Iannucci, R.A., Kathail, V.. and Pingali, K. "The Tagged Token Dataflow Architecture," to appear as an MIT/LCS/TM.

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

4. Arvind, and Iannucci, R.A. "A Critique of Multiprocessing von Neumann Style," *Proceedings of the Tenth International Symposium on Computer Architecture*, Stockholm, Sweden, June 1983.

5. Arvind, Kathail, V. and Pingali, K. "Sharing of Computation in Functional Language Architectures," *Proceedings of the Workshop on High-level Language Architectures*, Los Angeles, CA, May 1984.

6. Backus, J. "Can Programming be Liberated from the von Neumann Style?" *Communications of the ACM* 21, 8 (August 1978) 613-641.

7. Bassett, P.D, Geldens, P.M., Goodhue, J.T. and Iannucci, R. "Design of a 100 MHz cMOS Phaselocked Manchester Encoder/Decoder Circuit," Term Paper in VLSI Subject 6.372, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, 1982.

8. Burge, W.H. *Recursive Programming Techniques*, Reading, MA, Addison-Wesley Publishing Company, 1975.

9. Heller, S. and Arvind "Design of a Memory Controller for the the Massachusetts Institute of Technology Tagged Token Dataflow Machine," Computation Structures Group Memo-230, MIT Laboratory for Computer Science, Cambridge, MA, October 1983.

10. Henderson, P. and Morris, J.H. "A Lazy Evaluator," *Conference Record of the Third ACM Symposium on Principles of Programming Languages*, 95-103, Atlanta, GA, January 1976.

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

12. Ng, G.W. "Design of a Packet Communication Switch for a Multiprocessor Computer Architecture Emulation Facility Part 1: Clock Subsystem, Functional Languages and Architectures Group Design Note 2, MIT Laboratory for Computer Science, Cambridge, MA, November 1983.

13. Papadopoulos, G.M., Iannucci, R.A. and Chiang, C.J. "Preliminary Design for MEF Near-Term Communication Switch," Functional Languages and Architectures Group Design Note 4, MIT Laboratory for Computer Science, Cambridge, MA, January 1984.

14. Rettberg, R., Wyman, C., Hunt, D., Hoffman, M., Carvey, P., Hyde, B., Clark, W. and Kraley, M. "Development of a Voice Funnel System: Design Report," Bolt Beranek and Newman, Inc. Technical Report 4098, Cambridge, MA, August 1979.

15. Traub, K.R. "An Abstract Architecture for Parallel Graph Reduction," MIT/LCS/TR-317, MIT Laboratory for Computer Science, Cambridge, MA, September 1984.

16. Turner, D.A. "A New Implementation Technique for Applicative Languages," *Software -- Practice and Experience* 8 (1979) 31-49.

17. Turner, D.A. "The Semantic Elegance of Applicative Languages," *Functional Programming Languages and Computer Architecture* 85-92, October 1981.

18. Wadsworth, C.P. *Semantics and Pragmatics of the Lambda-Calculus*, University of Oxford Press, Oxford England, 1971.

Publications

1. Arvind, Dertouzos, M.L. and Iannucci, R.A. "Multiprocessor Emulation Facility," MIT/LCS/TR-302, MIT Laboratory for Computer Science, Cambridge, MA, September 1983.

2. Arvind and Iannucci, R.A. "Two Fundamental Issues in Multiprocessing: The Dataflow Solution," MIT/LCS/TM-241 MIT Laboratory for Computer Science, Cambridge, MA, September 1983.

3. Arvind, Kathail, V. and Pingali, K.K. "Sharing of Computation in Functional Language Implementations," *Proceedings of the Workshop on High-Level Language Architectures*, Los Angeles, CA, May 1984.

4. Arvind and Culler, D.E. "Why Dataflow Architectures?" *Proceedings of the Fourth Jerusalem Conference on Information Technology*, Jerusalem, Israel, May 1984. (Also CSG Memo 229-1)

5. Bassett, P. D., Geldens, P., Goodhue, J.T. and Iannucci, R.A. "A 100 MHz. cMOS Manchester Encoder/Decoder Circuit" MEF Design Note #6, MIT Laboratory for Computer Science, Cambridge, MA, May 1983.

6. Brobst, S.A. "Simulation Techniques in the Design of a Data Flow Supercomputer," *Proceedings of the 1984 Summer Computer Simulation Conference*, Boston, MA, July 1984.

7. Desai, E. D. and Pinkerton, J.T. "Design of a Packet Communication Switch for a Multiprocessor Computer Architecture Emulation Facility *Part 2: Input FIFO, Output Buffer, Sequencer, and Scheduler*, MEF Design Note #3, MIT Laboratory for Computer Science, Cambridge MA, September 1983.

8. Desai, E. D. "Specification for a High Speed Point to Point Serial Data Communication Circuit," MEF Design Note #8, MIT Laboratory for Computer Science, Cambridge, MA, May 1984.

9. Heller, S.K. and Arvind, "Design of a Memory Controller for the MIT Tagged-Token Dataflow Machine," *Proceedings of IEEE ICCD 83*, Portchester, NY, October 1983. (Also CSG Memo 230)

10. Ng, G., "Design Of A Packet Communication Switch For A Multiprocessor Computer Architecture Emulation Facility *Part 1: Clock Subsystem*" MEF Design Note #2, MIT Laboratory for Computer Science, Cambridge, MA, September 1983.

11. Papadopoulos, G.M. and Arvind "Dataflow Models for Fault-Tolerant Control Systems," *American Control Conference Proceedings*, San Diego, CA, June 1984.

12. Pingali, K. and Arvind, "Efficient Demand-Driven Evaluation (I)," MIT/LCS/TM-242, Laboratory for Computer Science, Cambridge, MA, September 1983.

13. Pingali, K. and Arvind, "Efficient Demand-Driven Evaluation (II)," MIT/LCS/TM-243, Laboratory for Computer Science, Cambridge, MA September 1983.

14. Pinkerton, J. T., Iannucci, R.A. and Papadopoulos, G. M. "A Comprehensive Hardware Laboratory for the Multiprocessor Emulation Facility," MEF Design Note #4, MIT Laboratory for Computer Science, November 1983.

15. Soley, R. M. "A Third Opinion on Dataflow Machines and Languages," to appear in *IEEE Computer.*

Theses Completed

1. Fuqua, P.C. "Emulating the I-Structure Memory for the Tagged-Token Dataflow Machine," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1984.

2. Muriph, S. W. "Optimized Execution of the APL Structured Functions," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1984.

3. Chambers, T. B. "A Database System for Parameters of Data Flow Machine Simulation," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1984.

4. Desai, E. D. "Specification for a High Speed Point to Point Serial Data Communication Circuit," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1984.

5. Douglass, S. A. "Demand-driven Efficiency on Dataflow Machines," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1984.

6. Traub, K. R. "An Abstract Architecture for Parallel Graph Reduction," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1984.

7. Ying, J. J. "Instrumentation to Collect Statistics for a Multiprocessor Emulation Facility," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1984.

Theses in Progress

1. Beckerle, M. J. "The Graph Resolution Method for Logic Programming," MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected April 1985.

2. Brobst, S. A. "Token Storage Requirements in a Dataflow Supercomputer," MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected May 1985.

3. Culler, D.E. "Resource Management for the Tagged-Token Dataflow Architecture," MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected December 1984.

4. Soley, R. M. "Generic Software for the Emulation of Multiprocessor Architectures," MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected December 1984.

5. Vafa, B. "A Resource Management Policy for the Tagged-Token Data Flow Machine," MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected December 1984.

Talks

1. Arvind, "The Tagged-Token Dataflow Machine," Honeywell Avionics Division, Minneapolis, MN, August 19, 1983.

2. Arvind "The Tagged-Token Dataflow Machine," The Sixteenth IEEE EASCON, Washington DC, September 19, 1983.

3. Arvind "Two Fundamental Issues in Multiprocessing," The 1983 IFIP's Congress, Paris, France, September 23, 1983.

4. Arvind "The Tagged-Token Dataflow Machine," IBM-Research, Zurich, Switzerland, September 26, 1983.

5. Arvind "The Tagged-Token Dataflow Machine," General Electric, Schenectady, NY October 18, 1983.

6. Arvind "The Tagged-Token Dataflow Machine," ICOT, The Institute for New Generation Computers, Tokyo, Japan, January 18, 1984.

7. Arvind "The Tagged-Token Dataflow Machine," Yale, New Haven, CN March 7, 1984.

8. Arvind "The Tagged-Token Dataflow Machine," University of Delaware, DE, March 19, 1984.

9. Arvind "Fundamental Issues in the Design of Multiprocessor Computers," University of Texas, Austin, TX, March 26, 1984.

10. Arvind "The Dataflow Solution," University of Texas, Austin, TX, March 27, 1984.

11. Arvind "Sharing of Computation in Functional Language Implementations," International Workshop on High-Level Computer Architectures, Los Angeles, CA, May 24, 1984.

12. Arvind "Sharing of Computation in Functional Language Implementations," IFIP Working Group 2.2 Meeting, MIT Endicott House, Needham, MA, June 12, 1984.

13. Brobst, S. A. "Simulation Techniques in the Design of a Data Flow Supercomputer," 1984 Summer Computer Simulation Conference, Boston, MA, July 24, 1984.

14. Culler, D. E. "Why Dataflow Architectures?," Fourth Jerusalem Conference on Information Technology, Jerusalem, Israel, May 1984.

15. Heller, S. K. "Design of a Memory Controller for the MIT Tagged Token Dataflow Machine," IEEE International Conference on Computer Design, Portchester, NY, October 31, 1983.

16. Iannucci, R.A. "VLSI: The Next Cottage Industry?," IBM Glendale Laboratory, Endicott, NY, September 14, 1983.

17. Iannucci, R.A. "Phaselocking for Fun and Profit," MIT VLSI Design Review, Cambridge, MA, December 1983.

18. Iannucci, R.A. "Dataflow Architecture: an Introduction," IBM Glendale Laboratory, Endicott, NY, December 1983.

19. Iannucci, R.A. "Dataflow Architecture: an Exercise in Top-Down Design," IBM Glendale Laboratory, Endicott, NY, December 1983.

20. Papadopoulos, G.M. "Dataflow Models for Fault-Tolerant Control Systems," American Control Conference, San Diego, CA, June 1984.

21. Pingali, K.K. "Demand Driven Evaluation on Dataflow Machines," University of California, Irvine, CA, May 27, 1984.

22. Pingali, K.K. "Demand Driven Evaluation on Dataflow Machines," IFIP Working Group 2.2 Meeting, MIT Endicott House, Needham, MA, June 12, 1984.

23. Soley, R. M. "A General Multiprocessor Emulation Facility," First Annual ACM Northeast Regional Conference, University of Lowell, Lowell, MA, March 20, 1984.

TABLE OF CONTENTS

LIST OF FIGURES