LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Computation Structures Group
1985-86 Progress Report

Computation Structures Group Memo 263
August 1986

Source File = COVER.MSS.3, Last updated 22-SEP-87 at 4:13 PM

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

COMPUTATION STRUCTURES GROUP

Academic Staff

Arvind, *Group Leader*

J.B. Dennis

R.S. Nikhil

Research Staff

W. B. Ackerman

G. A. Boughton

R. A. Iannucci, Manager - Hardware Development

J. T. Pinkerton

Graduate Students

M.J. Beckerle
S.A. Brobst
A.A. Chien
T-A. Chu
D.E. Culler
G-R. Gao
B. Guharoy
S.K. Heller
R.A. Iannucci
S. Jagannathan
B.C. Kuszmaul
V.K. Kathail
G.K. Maa
G.M. Papadopoulos
K.K. Pingali
S.A. Plotkin
R.M. Soley
K. Theobald
K.R. Traub
B. Vafa
E. Waldin

Undergraduate Students

D. Anderson
A. Chang
S-W. Chen
D. Clarke
B. DeCleene
S. Desai
C. Goldman
E. Gornish
R. Griffith
R. Gruia
E. Hao
J. Hicks
F. Herrmann
S. Kaushik
S. Ma
D. Marcovitz
D. Morais
M. Ng
R. Rabines
P. Rosenblum
S. Sanghani
J. Soong
Y-M. Tan
S. Younis
C. Winters

Support Staff

S.M. Hardy

N. F. Tarbet

Visitors

M. D. Atkins (IBM)

B. Blaner (IBM)

M. Mack (IBM)

H. Nohmi (NEC, Japan)

N. Skoglund (Ellemtel, Sweden)

S. Truve (Chalmers University of Technology, Sweden)

Computation Structures Group

1. INTRODUCTION

The Functional Languages and Architectures Group (FLA) was started in January 1981, splitting off from the Computation Structures Group (CSG) to continue research in tagged-token dataflow architectures and functional languages. During this past year, the old CSG was disbanded when its group leader, Professor Jack Dennis, began a leave of absence. Professor Dennis and the remaining members of the old CSG joined the FLA group, led by Professor Arvind. FLA has taken on the name of its predecessor, and thus the group organization has been restored to what it was in 1980.

A major thrust of the new Computation Structures Group is in two interrelated projects--the Multiprocessor Emulation Facility (MEF) and the Tagged-Token Dataflow Machine. The goal of the MEF project is to construct a "sandbox" to facilitate research and development in parallel architectures and languages. The most exciting development of the year has been the execution of large dataflow programs on the 32-processor MEF. Though much work is continuing on the MEF front, the group is concentrating on dataflow research, now that a reliable MEF foundation is in operation.

The goal of the Tagged-Token Dataflow project is to demonstrate the feasibility of general-purpose parallel machines by simulation and elation. .

We have also implemented a preliminary version of a Parallel Graph Reduction Machine on the MEF. We hope to solidify the prototype and use it to study these architectures in the coming year.

As part of the Tagged-Token Dataflow project we have also been looking at several issues related to language design and compilation, such as data structures, data types and storage management, and the relationship and tradeoffs between data-driven and demand-driven computation.

Professor Nikhil has begun studying the integration of databases into functional languages, based on a model that has a rich type structure and explicitly manages histories of states.

In addition, several of Professor Dennis's students are pursuing different research goals. Bradley Kuszmaul has been exploring the simulation of applicative languages on the Connection machine [14]. Gao Guang-Rong continues to investigate compilation of VAL into efficient code for the static dataflow machine [8].

IBM has continued to support a small group of engineers assigned to CSG whose function is to assist in the design and development of the packet switch. This group has demonstrated the first working pieces of the packet switch (the crossbar and the scheduler/arbiter) using a cMOS gate array and MSI TTL components. The group has also refined and verified the serial link protocol for the packet switch, and is well on the way to releasing a 10,000 gate array which performs the serial receiver and transmitter functions, as well as FIFO buffer control.

As a consequence of the MIT-IBM technical review meeting in the summer of 1985, CSG initiated and hosted a series of seminars by prominent IBM people. The series, entitled "Large and Complex Computer Systems in the Commercial World," was designed to give members of the LCS community a view of issues and problems in the design of large-systems in the "real" world.

Irving Wladawsky-Berger, Vice-President of Development for IBM's Data Systems Division, began the series with a talk entitled "Trends in Large Computer Systems" in which he discussed technical problems in the growth of large systems. In December Thomas W. Scrutchin, technical assistant to the manager of the Transaction Processing Facility, discussed the application of this system to airline reservations and banking systems. Federal systems division representative Kyle Y. Rone, a twenty-year veteran of IBM's On-board Space Systems organization, came in February to present an overview of the computer systems research that has gone into the space transportation aspects of the nation's aerospace program. Concluding the series in April, Keith R. Milliken, a project manager at the IBM/T.J. Watson Research Center, talked about YES/MVS, a program which applies AI expert systems techniques to assist operators of large MVS installations. Judging from the size of the audiences and from the lively discussions that followed each talk, the series appealed to a wide cross-section of the Laboratory.

The group has decided to write all its new programs in Common Lisp, the new standard dialect of Lisp. Through the efforts of Richard M. Soley, the group continues to have strong ties to the Common Lisp standardization effort, both in the old Common Lisp committee, and the ANSI X3J13 and ISO TC97/SC22 Lisp standardization committees.

2. PERSONNEL

We were fortunate to have Dr. G.A. Boughton, a member of the old Computation Structures Group, join the new CSG as a research associate, assuming responsibility for the welfare of the MEF, in particular the circuit switch. Dr. Bill Ackerman, another member of the old CSG, came on board at the same time, but left for Apollo in February. He was a great help in getting all the new Texas Instrument Explorers (Lisp machines) up and running. Nils Skoglund, a visitor from Ellemtel, has returned to Sweden. During his stay, Nils developed the NuBus interface logic for the NuCA (NuBus Channel Adapter), and began construction of a wire-wrapped prototype. The bulk of the NuCA will be tested through incremental additions to this prototype.

The IBM group that has been working with us for the past several years has also changed. Mark Atkins, a Senior Associate Engineer from Endicott, joined the group in November. Mark has been working on the design of the 48 MHz clock subsystem and the FIFO control logic. At the end of April, Bart Blaner returned to Endicott. During his stay, he completed work on the serial protocol. He also completed detailed design on 80% of the serial receiver logic.

3. MULTIPROCESSOR EMULATION FACILITY

This past year has been one of successes for the Multiprocessor Emulation Facility (MEF). Thirty-eight Texas Instruments processors are now on site, with thirty-two fully operational as part of the facility, connected via the circuit switch as well as standard Ethernet. The eight existing Symbolic machines are now used primarily for software development. The old MEF software base, designed and implemented by Richard Soley, was retired this past year, making way for the new Id World MEF software supporting the new circuit switch.

3.1. The Current MEF Hardware and the Circuit Switch

During the past year, we have taken delivery of 38 Texas Instruments Explorer Lisp machines. Each explorer is equipped with 8 megabytes of physical memory, 140 megabytes of disk storage, ethernet, and bit-mapped display. Thirty-two of these machines are connected by a circuit switching network. The circuit switch is based on the Butterfly switch protocols of Bolt, Bernak, and Newman (BBN). The basic protocol has been robustified with the addition of routing header encoding, data parity, and error signalling.

Every TI Explorer hosts a circuit switch node. Each node consists of four input links and four output links. Any input can be routed to any free output through a 4 x 4 crossbar. One input link and one output link is dedicated to the host processor, while the remaining three input and output links can be connected to other nodes to form an arbitrary network topology. We have chosen a modified hypercube, called a *directed hypercube,* as the MEF network [10]. Developed by Steven Heller, the directed hypercube is similar to traditional hypercube networks except that edges are directed; data can only flow in the direction of the edge. This enables the network to accommodate sixty-four machines in a six-dimensional directed hypercube rather than the maximum of eight processors that could be achieved using a traditional hypercube.

Message routing is *source based.* The host processor appends a string of routing instructions to the head of each message. A routing instruction is an encoded designator of one of the four output links at each node. The node strips off the leading routing instruction and forwards the message along the requested link. If the requested link is already active with another message, then the request is denied and the message is rejected a condition which is detected by the source processor. The source processor is responsible for retransmission of a rejected message. Although the hypercube topology makes analytic derivation of routing strings straightforward, any real implementation is subject to failed links and nodes. Steven Heller and David Culler have developed distributed algorithms that establish the topology of the network and then derive shortest path routes without any preknowledge of the current topology. The algorithm is based first on a *network flooding,* in which each processor broadcasts to all its neighbors every new aspect of the network it has discovered until no new information is derived.

The links that are dedicated to the host processor (one input and one output) are interfaced through FIFO buffering to the NuBus. These FIFOs, 2K bytes in each direction, not only

perform synchronization but also word width and protocol translation to and from the 4-bit 6Mhz circuit switch and the 32-bit 10Mhz NuBus. Additional logic is supplied for message disposition reporting, retransmission, and instrumentation. The circuit switch node is implemented on a single NuBus card. The approximately 200 TTL integrated circuits are interconnected using Multiwire technology.

Each external link consists of seven signals in the forward direction: a four bit data path, a message frame signal, data parity, and data/address demarcation. The reverse path consists of two status signals, message reject and parity error, for a total of nine signals per link. Each signal is driven NRZ through twisted pair and is differentially received through optoisolators.

The three input and three output links from each machine are not directly connected to other machines to form the hypercube network. Instead, these links are bundled into twisted-pair flat cables, three links per cable, and are brought to *cluster centers* that form three dimensional hypercubes (eight processors). The cluster centers provide "dimension links" to hook together up to eight clusters, for a total of 64 machines. Developed by G.A. Boughton, the cluster centers also perform local clock distribution and supply unique node identifiers that can be sampled by any circuit switch card.

The system is globally synchronous at a rate up to six megahertz, yielding a raw per link bandwidth of three megabytes per second. The clocks are distributed by a network designed by Saed Younis [23]. The distribution system, implemented on a single NuBus card and hosted by a Lisp Machine, is autocalibrating and is able to hold a 10 nanosecond phase skew across the entire facility (approximately 10 meters).

The circuit switch is in active use for sending tokens between machines in the dataflow experiments being hosted on the facility. The network has also proven invaluable for facility maintenance. Dinarte Morais has implemented a disk transfer protocol that can simultaneously copy a 30 megabyte Lisp World file to all 32 machines in under four minutes.

Gregory Papadopoulos was primarily responsible for the design and prototype construction. G.A. Boughton was responsible for the production, test, and installation of 35 boards now in active use. Additional help was provided by Gregg Bromley and Shefali Sanghani. Mr. Bromley implemented and tested the modified BBN protocols. Ms. Sanghani aided in schematic capture and design verification.

3.2. The Packet Switch Development

The success of the circuit switch as an interim networking solution for the MEF has given us an opportunity to develop more rigorously and verify the design of our packet switching network card. The packet switch is intended to provide two orders of magnitude more usable bandwidth than the circuit switch while significantly increasing the reliability of the net itself. The packet switch provides, on a single card, an 8x8 crossbar, routing address translation logic, and serial $\leftrightarrow$ parallel conversion circuitry. Serial links interconnect sister packet switch cards. Using one

such card per Lisp machine, networks of various topologies can be composed. Robert Iannucci has been responsible for the high-level design of the packet switch [12] and has coordinated the implementation work as well. During the last year, we settled on a gate array technology for the switch and made significant progress on the design.

Our switchover to LSI Logic Corporation as our gate array foundry has been largely successful. We have fabricated our first chip---an 8x8 4 bit wide crossbar slice on a 2200 gate, 3μ m 2 layer metal array. Michael Mack designed and simulated this chip prior to manufacture on our Apollo/Mentor workstations. We received first silicon in February, and are just completing our acceptance testing. We are happy to report that the design is 100% operational and bug-free. Based on a lotsize of 10 chips (all from the same run), we find the AC performance meets the advertised worst-case specifications and, in some cases, exceeds the specs. The chip was designed with fairly wide margins and, consequently, can be used at speeds approaching 10 megabytes per second per port, while the current Packet Switch architecture only requires 4 megabytes per second per port.

Our second chip (called "PASS"--- a not-too-clever acronym for PAcket Switch Slice) is in detailed design now, and will be fabricated on a 10,000 gate 1.5μ m 2 layer metal array. The experience with the crossbar chip provides us with a data point which confirms the manufacturer's claim that if a design works under the timing simulator, it will also work in silicon. This has boosted our confidence in the PASS design.

The PASS chip is made up of three subsystems: the serial receiver, the serial transmitter, and the FIFO controller. Bart Blaner has made progress in partitioning the design along these lines, and has also taken the lead in developing the receiver section. He has proposed, developed, and analyzed a D.C. balanced serial transmission code which translates each four data bits into six baud, a so-called 4b/6b code. This is an improvement over the more traditional Manchester-style D.C. balanced codes which translate each bit into two baud (1b/2b code). He has also developed and verified the serial link protocol which will be used for flow control and error recovery on the serial links. The scheme is based on delimiting at message boundaries, and streaming *i.e.*, without flow control) within a message. Bart and Sumanth Kaushik designed and implemented a 32 bit (Ethernet polynomial) CRC generator / checker circuit for the Receiver.

Mark Atkins has designed and begun to implement the PASS logic for the FIFO Controller. The controller provides the next in / next out pointers for static RAM control in addition to (1) message delimitation allowing last message retransmission and (2) instrumentation for measuring FIFO occupancy. He has also completed the high-level design of the clock logic both on-card and within the PASS chip.

Michael Mack has begun design work on the transmitter. Michael and Andrew Chang have also completed the design and TTL prototyping of our scheduler subsystem as well. A mid-course redesign resulted in simplification of the clocking and interfacing of the scheduler. This resulted in a 20% reduction in chip count as well. Preliminary work has already been done to interface the crossbar chip to the scheduler prototype to test the interface.

We have come to an agreement in principle with IBM on the use of one of their internally developed high-speed bipolar phaselock loop chips as the serial line front end for each inter-switch link. James Pinkerton has studied the analog portions of this serial link and has developed (1) a Multiwire test card to study the impedance charateristics of on-card wire, (2) a printed circuit jig and test circuit to characterize the IBM phaselock loop chip and (3) a specification for our custom serial cable. Most noteworthy is the cable which represents the state of the art in shielded twisted pair interconnection. We are seeking price quotations now and will select a vendor shortly. One manufacturer has produced a sample of our cable which upon cursory analysis comes close to meeting our specifications.

We have also reached a decision regarding our strategic direction for circuit card fabrication. The circuit switch was designed for fabrication in Multiwire technology rather than in printed circuit for reasons of (1) simplicity in layout, (2) ease of change, (3) quick turnaround, and (4) economy for production quantities of thirty-five so. Our experience (and experience of others) has demonstrated to us that we were misguided in expecting quick turnaround and economy over printed circuit boards of equivalent complexity. These observations, coupled with the closing of the Multiwire fabrication plant in New Hampshire, forced us to reconsider the use of printed wiring.

3.3. The Hardware Laboratory and New Equipment

We approached IBM about our need for printed circuit design tools and asked that they donate to us a P.C. layout facility. They have agreed, and we are in the process of installing four IBM 5080 color graphics workstations which, when connected to the previously loaned 4381 host processor, will run the Circuit Board Design System (CBDS) developed by Bell Northern Research and marketed by IBM. We intend to use this tool to implement circuit cards as simple as two-layer, one-of-a-kind test fixtures up through an eight- or ten-layer high density card for the packet switch. This facility should be operational by August 1986.

We have installed and have started using the equipment donated by Apollo (reported last period) as schematic capture and simulation stations. We now have four workstations on our design LAN (interfaced to the DARPA Internet) along with 1.5 GB of file space and a 6250 bpi streaming tape drive.

To support our LSI Logic work, we have installed their design verifier and Software Data Book packages on these workstations. This provides us with a library of macrocells (NANDs, NORs, SSI-type structures, I/O pads, *etc.*,) and a set of tools for analyzing the manufacturability, timing, and loading characteristics of our gate array designs. The design verifier is usable with LSI Logic's LL3000, LL5000, LL7000, and LL9000 families, spanning the gap from 3μ m low density cMOS to 1.5μ m 10,000 gate cMOS.

We have made some progress in exercising our lab instruments through the IEEE 488 bus. We have developed a spectrum analyzer package for the IBM PC which interfaces to our Tektronix

7854 Digitizing Mainframe. When completed, this package will facilitate the analysis of the serial link section of the packet switch in addition to serving as a general-purpose spectrum analyzer.

4. TOOLS FOR DATAFLOW EXPERIMENTS

Our goal of investigating the feasiblity of general-purpose parallel machines has necessitated developing a variety of tools, including a compiler for generating dataflow graphs from programs written in Id, vehicles for executing these graphs, and an environment to facilitate application development (editing, compiling, debugging) and architectural study. The first version of the compiler is fully operational, as is a reasonably efficient graph interpreter. The Id World environment provides an Id mode within ZMACS, similar to Lisp mode, to allow interactive editing, compilation, execution, and debugging. Given a working dataflow program, we want to investigate its behavior when executed on a dataflow machine. To this end we have developed three vehicles for modeling the execution of dataflow programs on hypothetical dataflow architectures. The first provides an extremely abstract model, similar to the U-interpreter, with unit operation costs, unbounded processing power, zero communication delay, and ideal program distribution. This provides a notion of the *inherent* properties of the program. The second vehicle is a complex of thirty-two machines, each pretending to be a dataflow processing element and communicating by passing tokens. No statement is made about the internal structure of the processing element, however, resource management issues, such as how work and data are distributed over the complex, are addressed in earnest. An important ancilliary benefit is that we really get to indulge in the *multiprocessor experience*, keep a large number of machines alive and doing what they are supposed to. The third vehicle, a detailed simulator running on an IBM mainframe, allows us to look down inside the processor, in addition to observing system-level resource management concerns.

4.1. Id Compiler

Version 1 of the Id Compiler [11], which is being maintained by Ken Traub, continues to run. Several small modifications have been made in the past year, including a new schema for loop contructs which bounds the resources required to execute loops. While Version 1 has served the needs of the group adequately up to the present time, its limitations have become more and more apparent as work on the dataflow project progresses. Besides the issue of compiler robustness, the need for compiler-related experiments---program transformations, compile-time type checking, optimizations, architectural changes, and the like---has sharply increased.

To help accommodate experiments in program transformation, an alternative syntax for Id, called PID, has been developed. PID code strongly resembles Lisp, and as a result is easily manipulated by programs written in Lisp. An undergraduate, Eric Hao, has written a program which translates from PID to Id, so that PID programs may be compiled by the Id compiler and

subsequently run on the simulator or GITA [9]. Together with a modification to the Id compiler which permits translation from Id to PID, Eric's translator allows an Id program to be translated to PID, subjected to a source-to-source transformation by a LISP programs, and finally translated back into Id for compilation and execution. Steven Heller has demonstrated the power of this simple technique by implementing the program transformation described by Keshav Pingali which causes a program to be executed in a demand-driven manner on a data-driven machine [21].

While the PID-to-Id translator was a quick way of enhancing the power of Version 1 of the Id compiler, the need still exists for a compiler designed to accommodate all manner of compiler-related experiments. Version 2 of the Id compiler is currently under development, and is specifically designed to support experimentation. Its major design goals:

- Provide a common core of data structures and abstractions that is general and powerful enough to support all conceivable dataflow compilers. This common core is described in Ken Traub's "A Dataflow Compiler Substrate" [22].
- Provide a well-documented modular structure that allows the easy addition and removal of compiler phases, such as type-checking or common subexpression elimination. In this way, experimental compilation techniques can be introduced at each step of the compilation procedure.
- Make the major transformation phases of the compiler (parsing, program graph generation, machine graph generation, and assembly) as specification-driven as possible. This allows easy modifications to syntax, schemata, and machine architecture.
- Implementation in Common Lisp, to insure portability.

Development of Version 2 is well under way, and is expected to compile its first program in August 1986.

4.2. GITA

Last summer a new vehicle, GITA (for Graph Interpreter for the Tagged-Token Architecture) for executing dataflow programs was developed to provide an expedient means for executing and testing compiled Id programs. This facilitated final check-out of the compiler and preparation of programs for simulation experiments. The first version of GITA was designed by Ken Traub and Richard Soley, and implemented in Common Lisp by Traub, Soley, and Dinarte Morais to run on Symbolics and TI machines [17]. A second version of GITA was developed by David Culler, Greg Papadopoulos, Andrew Chien, Richard Soley, Steven Heller, Dinarte Morais, and Bhaskar Guharoy which runs in a distributed fashion on the MEF using the circuit switch.

In the abstract model, a configuration of a program is described by an assignment of tokens to arcs in the graph. A node is enabled to execute (or *fire*) when tokens with matching tags are present on its input arcs. When an enabled node fires, it consumes its inputs and generates result tokens on its output arcs. Dataflow programs are determinate, so any enabled node can be fired

at any time. Capturing this abstract model in a real interpreter is surprisingly straight-forward. GITA maintains a queue of unprocessed tokens and a collection of unmatched tokens which have been processed. The main loop dequeues an unprocessed tokens and processes it as follows:

1) If the token requires no partner, the instruction it is destined for is executed and the result tokens are added to the unprocessed queue.

2) If the token requires a partner, the collection of unmatched tokens is searched. If a partner is found, the destination instruction is processed as above.

3) Finally, if no match is found the token is added to the collection of unmatched tokens.

Of course, the name of the game is represention and access to the various data structures, and this has worked out very nicely. GITA currently processes about 2400 dataflow operations per second on a Symbolics 3600 and 1000 on a TI Explorer.

4.3. U-GITA

In addition to executing dataflow programs, GITA provides a basis for modeling certain abstract machines. In particular, it is possible to model the kind of greedy execution one associates with the U-interpreter. Given a configuration with tokens distributed throughout the graph, all enabled activities fire in a single time-step, producing tokens on result arcs. There is no bound place on the number of activities fired and no communication delay. This is realized within GITA by time-stamping the tokens themselves. While executing a dataflow program it is possible to construct *execution profiles* describing the behavior of the program on an ideal machine. Typically, we look at the parallelism (number of operations per time step) and resource requirements, *e.g.*, number of unmatched tokens. It is also possible to model the abstract behavior on a machine with a fixed number of processors.

4.4. MEF-GITA

A collection of Lisp machines with a high-speed interconnection network is made to act like a collection of (rather slow) dataflow machines by having each execute the GITA interpreter. Tags and I-structure addresses must be meaningful globally, as in a true datalow machine. Machines spawn work off to other processors and service and request structure operations. All communication is in the form of packets passed across the high-speed network. The resource management issues in this context are essentially as if we had real dataflow machines, except the time-scale is somewhat distorted and the primitives available within each processor are quite powerful. As with the abstract machine, it is possible to construct execution profiles to help understand the details of program behavior on the machine.

The three versions of GITA are one in the same source. Different statistics are collected in the different modes. In fact, we run U-GITA on MEF to construct profiles more rapidly than would

be possible with a single Lisp machine. Work has started on providing a uniform way of feeding statistics collected on these various abstract machines, and the simulator discussed next, into the Illustrate facility.

4.5. SITA: A Simulator for the Tagged-Token Machine

The simulator for the TTDA has been described in previous reports. It provides a detailed picture, essentially register transfer level, of the dataflow machine in operation. The simulator has continued to evolve in response to the needs of the project, and it finally has a catchy name. The changes reflect our growing understanding of the architecture. The simulator was modified to support the bounded loop schema developed by David Culler [7]. This allows the unfolding of loops to be controlled by a runtime constant in the simulator. Simulation results have made it clear that current structure allocation schemes fail to take advantage of the pairing of ALU pipelines and I-structure controllers. As a result, little benefit is derived from the local paths between the ALU and the I-structure controller in a PE. In response to this observation, the simulator is being restructured to allow different numbers of I-structure units and ALU pipelines. This does not preclude organizing them in tightly coupled pairs. The restructuring allows us more latitude in experimenting with different machine configurations.

Our computational power was increased considerably last fall by the addition of an IBM 4381. This faster processor allows us to simulate the TTDA at 2.5 times the rate that was previously possible. The simulator on the IBM 4381 now executes approximately 80 Dataflow Instructions/second.

4.6. Id World

The first version of Id World was completed this past year, integrating tools for editing, compiling, executing, debugging, and analyzing Id programs in an environment similar to what Lisp users enjoy. A new Id Mode within ZMACS allows dataflow programmers to live in the style to which Lisp hackers have become accustomed. From within the editor it is possible to compile meaningful pieces of dataflow programs. The resulting graph is automatically loaded into GITA. On the MEF, the graph is transparently broadcast to all the MEF machines over the circuit switch. When dataflow programs are invoked, GITA is brought into action, so your Lisp machine (or even 32 of them) look to you like a dataflow machine. The real *tour de force* is the Id debugger developed by Dinarte Morais. If your dataflow program breaks, you find yourself in the Id debugger, which is like the Lisp machine debugger with some important differences. Instead of a stack trace, you are presented with an invocation *tree*, since after all this is truly a parallel machine. The program structure is physically distributed over the entire MEF, but the debugger makes that entirely transparent.

The other role for Id World is to provide a uniform interface to the three abstract machines. Currently we can record and display profiles from U-GITA or MEF-GITA, but work remains in areas of statistics collection, storage, and presentation.

5. EXPERIMENTS ON MEF

5.1. Dataflow Experiments

The focus of our experiments has been resource management in the TTDA. The resources include token storage, ALU processing cycles, structure storage, and structure memory bandwidth. We have begun to examine how policies for managing these resources effect system performance. The experiments are being conducted on SITA or GITA, according to whichever provides the more powerful tool for highlighting the behavior of interest.

Simulation studies have exposed work distribution and data structure mapping as two crucial issues in constructing a scalable multiprocessor. If we hope to achieve speedup in regions of limited parallelism, work distribution is the critical factor. If we hope to fully utilize the machine in regions of high utilization, then structure contention becomes a very serious problem. Future experiments will further probe these crucial issues.

5.1.1 Token Storage Management

The token storage requirements of tagged-token machines will be quite large as parallelism unfolds and multiple contexts for procedures in the parallel invocation tree are accumulated. In his master's thesis [4], Stephen Brobst suggests that a hierarchical memory may be an effective way to organize the token storage unit of a tagged-token machine. A token cache can be used to take advantage of temporal locality in the token matching process to yield a high throughput of enabled instructions to the execution unit. For effective cache performance, however, prudent management of program parallelism must be exercised to avoid oversaturation of the token cache. The bounded loop schema developed by David Culler was shown to have a significant impact in this regard.

5.1.2 Structure Storage Management

In any parallel machine, data structure distribution and contention is an important problem. To this end, Andrew Chien [6] has examined structure referencing patterns for the SIMPLE code and several smaller programs. Despite the fact that GITA has limited fidelity in timing aspects of the TTDA, these measurements (done with one processor) have given us significant insight into the temporal patterns of structure references. Preliminary indications of "hot spots" have led us to reconsider seriously the benefits of spatial locality in a parallel processor. This has led to a restructuring of SITA, separating the I-structure units from the ALU pipelines and allowing structures to be allocated across structure memory units.

MEF GITA experiments have also given us some insight into the structure storage management problem. We have found that structure contention can be significant. Our initial allocation mechanism places entire structures on a single PE. This, not suprisingly, leads to structure reference contention when a large number of processors are employed. Future experiments will consider the effects of allocating structures consecutive elements are on consecutive processors.

5.1.3 Work Distribution

A number of SITA experiments have focused on the scalability of the architecture. Recent experiments conducted by Andrew Chien on highly parallel programs show that the architecture tolerates significant network and memory latency with only minimal increases in overall run time. Scalability studies have shown that linear speedups are achievable for programs with large amounts of parallelism. However, careful attention must be paid to work distribution in regimes of low parallelism if the linear speedup is to continue into larger numbers of processors. These results reflect the study of a very limited class of programs.

With each MEF GITA processor pretending to be a dataflow machine, we have strong motivation for developing truly distributed mechanisms for allocating work across the collection of machines. Our approach has been to start simple and look for shortcomings before developing complex mechanisms, such as proposed for the TTDA. Currently, each processor makes independent allocation decisions, without concern for decisions made by other processors. We have experimented with distributing work in a random manner and in ways that take advantage of program structure. So far we have observed good speed-up on most applications, but also certain limitations.

Distributing work on a code-block basis appears to be too coarse. In many cases, there is sufficient instruction level parallelism to keep many processors busy, but not a sufficient number of active code-blocks. This is exacerbated by inequities in the allocation of work. If most processors have one code block, a few have two, and a few have none, all may have to wait for the slowest. We plan to investigate finer-grained allocation mechanisms and simple load leveling techniques.

5.1.4 Limitations of MEF GITA

Our initial experiments with MEF GITA have had a dual purpose: (a) to understand the impact of certain resource allocation policies, and (b) understand the bias introduced by MEF GITA as an imperfect execution vehicle. Thus, we have run a collection of applications on a varying number of processors and have begun to cross-check our observations against results obtained on the simulator.

One must be somewhat cautious in drawing conclusions from results obtained on MEF GITA, since there are significant differences between that vehicle and what we expect a real dataflow machine to be. One difference is that in MEF GITA processing structure requests draws away from processing of dataflow instructions (since Lisp machine cpu cycles devoted to one subtract from those available to the other), whereas in a real machine these would proceed independently. Also, we expect a dataflow machine to be heavily pipelined, whereas MEF GITA is not.

In addition to resource management experiments, MEF GITA has provided a basis for architectural studies. These include (a) analysis of dynamic instruction mixes as compared to conventional machines, (b) token storage requirements of dataflow programs, (c) effects of loop bounding, and (d) impact of reference count storage reclamation.

5.2. DisCoRd: Parallel Graph Reduction on the MEF

Under the guidance of Rishiyur Nikhil, Patrick Lincoln has implemented an emulator for a parallel graph reduction machine on the MEF. It is called DisCoRd, and was part of his bachelor's thesis [16].

Graph reduction is another parallel model of computation for functional programs based directly on the abstract rewriting semantics of functional languages. (Interestingly, recent work by Keshav Pingali on demand-driven evaluation in dataflow architectures indicates that perhaps one can obtain a unified view of dataflow and graph reduction.)

The system consists of two independent parts: a) a compiler which translates programs in a functional language to a collection of combinator definitions (in ZetaLisp) and a table of strictness information for each combinator,[1] and b) the parallel graph reduction machine. We have concentrated mainly on b), leaving open compilation issues in a) for future research.

The parallel graph reduction machine consists of an interconnection of processing elements (PEs). Each PE consists of a reduction engine, a network manager, and a load manager. The reduction engine has a "ready queue" of tasks representing subgraphs to be reduced; for each task it may perform a reduction, or spawn more tasks to evaluate sub-expressions, based on strictness information. The load manager periodically broadcasts the size of the ready queue to neighboring PEs, and maintains an estimate of the sizes of ready queues in neighboring PEs. The network manager is responsible for all communication to other PEs, and uses the load information to decide which PE should receive spawned tasks.

The DisCoRd system allows the experimenter to specify the number of logical PEs and their logical interconnection topology, and the mapping onto the actual physical resources of the MEF. It starts up Lisp processes representing the logical PEs on the MEF Explorers, establishes the communication links, loads the ZetaLisp combinator code, and starts execution by placing the main program expression graph on one PE's ready queue. DisCoRd is instrumented to measure abstract computation steps and elapsed time.

The DisCoRd system is not yet very stable--there are several obvious coding improvements to be done, and it uses only Ethernet connections instead of the MEF's circuit switch. We have had time thus far to perform only rudimentary experiments, reported in Patrick Lincoln's thesis, which demonstrate the flexibility of the DisCoRd testbed. This summer (1986) Rishiyur Nikhil expects to make it more robust and then conduct meaningful experiments to evaluate the feasibility of parallel graph reduction.

---

[1] A function is said to be strict in an argument if, whenever the argument is undefined, the function application is also undefined.

6. LANGUAGE RESEARCH FOR THE TAGGED-TOKEN DATAFLOW ARCHITECTURE

Investigation of language-related issues for the Tagged-Token Dataflow Architecture continues under the direction of Rishiyur Nikhil, motivated by the belief that such research must be an integral part of any research into parallel architectures.

6.1. Id/83s

The base language we have been using for many years to program the TTDA machines is a functional language called Id. Originally proposed in [2], Id has evolved in a somewhat *ad hoc* and erratic manner over the last few years.

In June/July 1985, Rishiyur Nikhil and Arvind used their preparation for the summer course "6.83s: Functional Programming and Dataflow Architectures" as an opportunity to redesign Id completely, incorporating their latest understanding of issues in data structures and data types. The resulting language is called "Id/83s", and is reported in [2083s].

6.1.1 I-structures in Id/83s

A fundamental aspect of Id is a data-structuring facility called "I-structures", which attempt to solve what we believe are serious problems in purely functional models of data structures.

In a functional language, data-structures are never "updated". They are created as complete values, and "update" operations create new values. Thus the operation

```
update(a,i,v)
```

where `a,i` and `v` are an array, index and value respectively, produces a *new* array value that is just like `a` except at index `i` where it has value `v`. In general, the implementation may have to copy the rest of `a`. To minimize this copying, most functional languages provide linked structures (such as lists) built out of "smaller" units such as cons-cells, so that a updated value can *share* as much as possible of the original value. Arrays are then simulated by appropriate operations on linked structures. Unfortunately, for structures that logically behave like arrays, linked structures increase access times (to at least log(n)) , make some traversals extremely inefficient, and preclude some parallel accesses.

I-structures were first proposed by Arvind in [3] as a solution to this problem, affording parallel, constant-time access, and avoiding excessive copying. I-structures evolved out of very operational intuitions about data-structures in memory. The key idea was to separate the activity of allocating storage for a data-structure from the activity of filling in its components. Thus an array, when allocated, contains "empty" slots which are later filled by other concurrent operations. To preserve determinacy, only one value is allowed to be written into any slot, and any attempt to read a slot is delayed until it is non-empty.

The semantic implications of I-structures were not so easy to pin down. It was unclear as to

what were the appropriate linguistic constructs to manipulate I-structures. For a long time, I-structure constructs in Id clung to a conventional expression-oriented syntax, in keeping with the functional nature of the language.

In the last year, work by Keshav Pingali has greatly deepened our understanding of I-structure semantics. It is now apparent that with I-structures, Id is no longer in the class of purely functional languages, and can no longer be given classical λ-calculus-based semantics. In particular, Pingali has discovered a close connection between I-structures and so-called "logic variables" in logic programming languages. He has developed an elegant semantics for functional languages with I-structures, based on solving sets of equations involving so-called "closure" operators.

It was initially difficult for us to realize that we had gone beyond functional languages. I-structure operations do have the flavor of imperatives. However, the constraints on assignment and selection guarantee that programs remain determinate. Id/83s has been designed to reflect this new understanding of I-structures. There are three constructs relevant to I-structures:

- `array (n)` is an expression that allocates an "empty" array of size `n` and returns as value a descriptor to that array. This value is a first-class value.
- `x[i]` is an expression analogous to selecting a component of an array---it returns the `i`th value of the array `x`, *after* it becomes non-empty.
- `x[i] = v` is a statement analogous to updating an array location---t stores value `v` in location `i` of array `x`, provided the location is empty. Multiple writes into a location are run-time errors.

Because of the assignment statement, the language is no longer purely expression-oriented---conditionals, loops, and abstractions can also be statements.

Several programs have been written in Id/83s, including a version of SIMPLE (a moderately large program that is a classic benchmark for evaluating high-performance machines) written by Arvind, giving us some confidence that we have a robust language design.

6.1.2 Types in Id/83s

Rishiyur Nikhil has been studying the question of incorporating a data-type system and type-checking into Id/83s. We have become increasingly convinced of the necessity for the compiler to have detailed data-type information about a program. This conviction stems from various sources.

First, in studying the garbage collection problem on the dataflow machine, we have come to realize the necessity of the compiler to determine which arcs in dataflow graphs carry I-structure references and which do not. Without this knowledge, it is necessary to insert conditional garbage collection code at *every* fork point and at every array-select instruction---this is an unacceptable overhead.

Second, even code that manipulates scalars needs to perform rudimentary type-checking to avoid misintrepretation of scalar data. This is analogous to tag-checking in Lisp

implementations, and is also an unacceptable overhead.[2]

Third, it is our experience (corroborated by many other researchers) that compile-time type-checking is a powerful debugging tool. Many "silly" errors, such as applying CDR to a number, so typical in Lisp programs and so often discovered after months or years of use of a program, just do not occur in a statically typed system. It is a common folk theorem that when programming in languages like ML or Miranda, most errors are caught by the type-checker.

There are thus two separate but not unrelated questions: a) What data-type system, and b) What type-checking methodology should we adopt for Id/83s?

In answer to question a), we insist that the type system should not seriously constrain the programmer-- one should not have to abandon completely the expressive power of an untyped language.[3] A key to this objective is to adopt a type-system with sufficient *polymorphism*, such as that of CLU or ML. We are looking at the second-order typed λ-calculus as a possibility because we believe that it is a clean and well-understood system that subsumes the type systems of ML and CLU, the only other type systems we know that are practical and meet our needs for expressive power. This type system was proposed independently by Girard in 1971 and Reynolds in 1974.

In answer to question b), we would like the type-checker to perform as much *type-inference* as possible, *i.e.*, to be able to type-check a program without the programmer having to laboriously annotate the code with type declarations. We envisage a facility in Id World where the programmer can point to a sub-expression and ask for its type, and optionally have it automatically inserted. We expect the Id World system to keep track of dependencies between functions, and to redo incrementally the necessary type-checking when a function is edited.

We have been investigating several issues in the use of the Girard-Reynolds type system in Id. It is well known that this type system does not permit full type-inference---*i.e.*, if the programmer were to omit *all* type declarations, the problem of reconstructing those declarations automatically is undecidable. Thus, some annotation is necessary. We are investigating questions such as: How much type declaration is necessary, and is there a simple model for the programmer to decide where to insert such declarations? If essential declarations are omitted, is it acceptable for the type-checker to produce a (weaker) Milner-typing instead? If at times the programmer wishes to bypass the type system (and we believe there will always be such situations) is there a clean way of limiting the scope of this circumvention? The Girard-Reynolds type system is based on a purely functional language (the λ-calculus). Does it extend cleanly to Id, which also has I-structures?

---

[2]On Symbolics machines, ALU operations are done "optimistically" in parallel with tag checking which generates a fault if there is a type-error. Wearing our RISC hats, we believe that this adds unacceptable complexity to the hardware---most of this checking could and should be done at compile-time.

[3]The type system of Pascal is an example of one that does *not* meet this goal.

Because of the immediate need for a working language, we did not include a type system as part of Id/83s. However, using intuitions that Rishiyur Nikhil has gained from long experience with Milner-type systems, we have designed Id/83s with type-checking in mind. An example of this influence is seen in the separation of lists (homogeneous collections) from tuples (heterogeneous collections). We are confident that we can cleanly augment it with a modern, polymorphic type system.

To answer some of the type-checking questions, Shu-Wie Chen has completed a bachelor's thesis [5] in which he has built an experimental programming system with an incremental Milner type system, along the lines suggested in [19]. Currently, it uses a small functional language typical of many functional languages; we expect to move this system soon into Id World using the new compiler being developed by Ken Traub.

6.1.3 Garbage Collection Experiments on MEF

Now that we have started running large experiments on the MEF, the garbage collection problem for the Tagged Token Dataflow machine looms seriously.

Because I-structures are "first-class" objects in Id, they must be allocated dynamically, and reclaimed using some form of garbage collection. We do not think a conventional mark-sweep scheme is suitable for the TTDA, partly because it does not generalize well to the multi-processor situation, and partly because it is difficult, in the TTDA, to identify reliably the "roots" of reachable structures.

We are thus exploring garbage-collection based mainly on reference-counts.[4] The problem still to be solved is this: Because of the non-determinacy in the scheduling of events and in the unpredictability of network transit times, "decrement-refcount" messages and "increment-refcount" messages may arrive at an I-structure in any order. The reference count stored in the structure may thus spuriously drop to zero, and the structure may be reclaimed prematurely.

Bhaskar Guharoy is instrumenting the I-structure code in GITA to keep track of reference-count message histories for each structure. We hope to obtain a better idea of how often reference counts go to zero spuriously, and to develop a probabilistic model of how long reference counts stay spuriously at zero. With this understanding, we will be predict with high probability when a reference count is truly zero, and use that as a signal to move that structure to slower storage, reasonably assured that it will never be accessed again.

Rishiyur Nikhil has also been looking at program analysis to identify conditions under which we can reliably identify I-structures whose extent in time coincides with the lifetime of a code-block. We already have a method of detecting termination of code-blocks; the reclamation of such I-structures may then be triggered by the termination signals of their parent code-blocks, and we can elide their reference-counting instructions altogether.

---

[4]Reference counts cannot be used to reclaim cyclic structures, but we do anticipate cycles to be very common in Id programs.

6.2. Demand-driven Evaluation

Demand-driven evaluation is often described informally as "doing only those computations that are required to produce the output of the program." In his doctoral thesis, Keshav Pingali defined a functional language L and gave two semantics for it : an operational semantics using dataflow graphs and a denotational semantics using fixpoints on an abstract domain. The informal notion of "minimal computation required to produce the output of an L program" is characterized formally by the least prefix point of the program. He then showed how L programs can be transformed so that a data-driven evaluation of a transformed program performs only these computations. The target language for the transformation was a functional language that is a superset of L. These result can be extended to any functional language without non-sequential functions such as the parallel-or.

Pingali's thesis describes another transformation for achieving the effect of demand-driven evaluation, for which the target language is a functional language in which data structures behave like logical variables in a logic programming language such as Concurrent Prolog. This target language can be looked at as a variation of Id83/s in which I-structure cells can be written into more than once : if a write is attempted in any I-structure cell which already contains some value, the incoming value is unified with the contents of the cell. Steven Heller has taken up the task of implementing this transformation. Future work will include:

- defining the notion of demand-driven evaluation for a language with I-structures;
- performing various optimizations to reduce the overhead of the demand code;
- producing graphs which are self-cleaning, *i.e.*, which do not leave tokens behind when the program terminates.

6.3. Databases and Functional Languages

In the last year Rishiyur Nikhil has restarted research into databases in functional languages. In [18] he demonstrated the attractiveness and feasibility of the use of functional languages as front-end query languages for existing database systems. The semantic advantages included the availability of an expressive, computationally complete query language with a rich type system into which conventional data models (such as the relational model) have a simple and natural embedding. He demonstrated an efficient implementation by establishing a correspondence between conventional database operators and stream operators in a graph-reduction execution model for functional languages.

We are now studying the problem of including databases as an integral part of functional languages (rather than relying on an existing, external, conventional database system).

6.3.1 Data Models and Type Structure

In our view, a data model is nothing more than a data type system; a schema for a database is nothing more than a type declaration within that type system; and a database itself is nothing more than an object having the type declared in the schema. With this perspective, it is evident why conventional databases are not used for the hundreds of applications that could use them---mail systems, operating system tables, text-processor device and font databases, CAD/CAM engineering databases, *etc.*, Conventional data models (even the much-touted relational model) just do not offer the rich type systems that we are accustomed to (and find indispensible) in programming languages. Not only are the type-systems fundamentally inadequate, but they are riddled with *ad hoc* limitations (such as what types may be components of what other types, maximum field sizes expressed in bytes, *etc.*,) that have no semantic justification.

We therefore consider it axiomatic that the fact that an object is to be stored in a database should not be reason for any loss of allowable type structure in that object. Database objects (*i.e.*, those that are stored persistently) should be freely usable with transient objects. It is up to the system implementor to design methods that allow the programmer to store and retrieve *any* object in a type-safe and efficient manner.

6.3.2 Modelling State

At any given point in time, a database is supposed to model an abstracted state of the world. As the state of the world changes with time, the database is expected to keep track of these changed states.

We can ask: How should a database system model the evolving state of the world? Conventional database systems maintain only a "current" state. A change of state in the world is mimicked by erasing some information from, and adding other information to, the database so that once again it represents the "current" state of the world.

We believe that there are many semantic and pragmatic problems with this conventional view. First of all, it seems unlikely that humans model the world this way-- we are capable not only of having an estimate of the current state of the world, but also of previous states. We may "forget" information, perhaps because it is unnecessary, but we do not consciously erase information. In fact, it can be argued that this capability is necessary to make intelligent decisions about the future.

Second, we believe that erasure of information in databases is motivated mainly from efficiency considerations, and because we are so used to thinking in terms of constructs from imperative programming languages. In many database situations (*e.g.*, banks, personnel, billing) where it is necessary to keep a record of all past states of the database, one usually designs special, *ad hoc* methods to retain information that would normally be erased; these methods are not part of the data model, and are often designed on a per database basis.

When a current state database is shared by many users, it becomes an overly critical resource. Most users do not demand the current state-- a "recent" state is generally adequate. In CAD

databases, users often need access to a consistent state of the database for extended periods of time during which they can experiment with various designs. If the contention due to such undemanding users is to be avoided, additional mechanisms must be created to extract snapshots of the database for them.

A central feature of databases is that state transitions, no matter how large, are truly atomic. In a database that maintains only a "current" state, implementing this abstraction is extraordinarily difficult. Further, users who wish only to read the database (no state transition) must contend with those who want to write it (cause a state transition). In a database that models histories of states, these problems are greatly alleviated.

6.3.3 Functional Databases

Our view of databases in functional languages thus emerges. A state of the world is modeled as a persistently stored *environment*, binding names to values (and perhaps types). A query on this state is just an expression evaluated in this environment. A database is a history of states, and an update transaction is a meta-level expression that extends this history by creating a new environment (such as by redefining a function SALARY mapping PERSONs to NUMBERs) based on the existing history. Depending on the application, we may permit only linear history extensions, or we may permit branching extensions---both have their uses. Extensions occur atomically---a failed update transaction simply leaves the history unchanged. For linear history extensions, update transactions are serialized to preserve atomic state transitions.

Explicit environment-specification operators in the query and update languages allow queries and updates to depend not only on the most recent state, but on any previous state.

With this philosophical background we have been a) studying language issues to express this model of state, and b) possible implementation strategies. The major difficulty in a) is to have a view of types that is consistent across histories, *i.e.,* to have meaningful operations that may update a data-type definition from one state to the next.

For implementations it appears that histories of states can be superimposed nicely on graph-reduction models of memory. Each state is associated with its own private graph memory which can share large subgraphs with previous states via pointers to previous graph memories. Thus, a new state initially contains only the portions of the environment that are "updated" with respect to the previous state. Using standard copying garbage-collection techniques, other parts of the new state can be moved up to the new graph memory to improve locality transparently to the user, and old parts of the state history can be cleanly pruned or re-attached dynamically according to available storage capacity.

7. WORK UNDER PROFESSOR DENNIS'S SUPERVISION

7.1. The VIM Project

The VIM project aims to develop an experimental computer system based on principles of data-driven instruction execution and functional programming languages. The project is unique in striving for a system that will serve multiple users with a degree of semantic coherence well beyond what contemporary computer systems are able to offer. It also differs from other efforts to build systems to support functional programming in that the issues of efficient execution of functional programs over a hierarchical memory are addressed and solutions sought. The system uses a base language that is a form of acyclic dataflow graph, and a user language VIMVAL that is an extension and revision of the functional language VAL.

7.2. Accomplishments

In the past five years, substantial progress has been made on VIM. The resolution of successive issues has brought the project to the point where only a few problem areas need to be addressed before a major implementation effort can be mounted.

The project began in 1981 when we realized that our earlier proposals for a general purpose computer system based on dataflow principles could not be adequately evaluated due to the absence of experience in formulating and running functional programs. The nature of programming itself would be so changed by the kind of system we proposed that all past experience would be irrelevant to the detailed design decisions required to define a practical machine.

From the beginning, our plan has been to build an experimental system made of ordinary off-the-shelf components that can evaluate the functional programming style and particular mechanisms for supporting modular programming of a given user community. Once justified by our experience with this unique experimental programming environment, plans could be refined and developed for the kind of powerful and efficient computer systems envisioned in our earlier work.

The first step was the formulation of an operational computer system that encompassed all essential features of the proposed system. This operational model may be viewed as an early simplified specification of the *base language* of our system. Initially the plan of development was to define a series of operational models of increasing detail such that the most refined model was a specification of the VIM implementation-- a set of microcoded routines running on a suitable microprogrammable computer such as a Lisp machine.

In the academic year 1981-82 the attraction of having a working interpreter for VIMVAL prevailed and an implementation was written by Joseph Stoy, visiting scientist from Oxford University. Many issues concerning the efficient encoding of instructions and data for the

complete language were resolved at that time. In addition, our understanding of the VIMVAL language was refined and improved.

The next step was to develop and evaluate alternate approaches to representing and operating on data structure values stored in the VIM heap. Since the heap is intended to be implemented on a hierarchical memory comprising semiconductor and disk systems, and since efficient concurrent handling of many memory transactions is desired, we proposed using fixed-size *chunks* of memory as the unit of allocation to a data structure. The corresponding creation, augmentation, and access algorithms were designed and specified by Bhaskar Guharoy, who also developed suitable mechanisms, based on the reference count technique, for disposing of chunks no longer accessible to computations.

In a major study [13], Suresh Jagannathan showed how backup and recovery mechanisms may be built into VIM so that users suffer negligible loss of information in the event of a system crash due to any single failure. The technique is based on saving information that permits reconstruction of the results of all function evaluations performed from initiation of a user command to the time of the crash. A small online stable store is proposed to hold the backup data prior to writing it on backup tape.

One significant change in our plans is the switch from a strongly typed language in the spirit of CLU to the use of optional type declarations and a Milner-style type inference technique. The refinement of Milner's theory and an elegant statement of the type-inference algorithm for VIMVAL are given in the prize-winning bachelor's thesis of Bradley Kuszmaul [14].

Recently, a new operational model for the VIM base language has been formulated by Guharoy and Jagannathan. This work reflects our present thoughts on the elements of the VIM base language. It has been extended to help establish the correctness of the data structure access algorithms and to establish the correctness of the backup and recovery algorithms.

Several new studies have focused on the user programming environment supported by the VIM system. A proposal for the user command language has been drafted by Joseph Stoy. This proposal raises many questions about the role of environments (which in VIM serve the function of directories in other systems), and the impact of the command language on the VIM type system. These issues are the subject of current doctoral research by Earl Waldin.

Other topics for future study include evaluation of approaches to supporting nondeterminate computation using guardians, the application of non-determinacy to data base systems and backtrack programming, and study of language constructs for defining data structure values in general recursive data types.

7.3. Compiling for the Static Dataflow Machine

A compiler that produces efficient dataflow machine code from programs written in the applicative language VAL [1] is crucial to the success of the static architecture in large scale scientific applications. To keep the architecture simple and efficient for computations involving large arrays of numerical data, most of the decisions about resource assignment (allocation of data structures to space in array memory and of dataflow instructions to processing elements) have been entrusted to the compiler.

To accomplish an effective resource allocation, the compiler must transform the program so that its structure is a good match to the processing power and memory space of the target machine. Global program transformations are needed because the several sections of a large program must be capable of operating together in a way that allows full utilization of performance without requiring inordinately large amounts of memory for intermediate results. It is reasonable to attempt such global program transformations because VAL is a functional or applicative language, and therefore interactions among program parts occur only at points that are evident from the syntactic structure of the program---the impossibility of "side effects" removes the major difficulty that inhibits use of global optimizations in compilers for conventional languages.

A pipelined code mapping scheme as a conceptual basis for such a compiler has been developed in Gao Gaung-Rong's doctoral research, which will be completed soon. His thesis explores the transformations that can be made to achieve high performance for numerical programs when executed on a computer based on dataflow principles. We demonstrate how the massive parallelism of array operations in such programs can be effectively exploited by the fine-grain parallelism of static dataflow architecture.

The key is to organize the dataflow machine program graph such that array operations can be effectively pipelined. We introduce a simple value-oriented language to express user programs. Program transformation can be performed on the basis of both the global and local dataflow analysis to generate efficient pipelined dataflow machine code. A pipelined code mapping scheme for transforming array operations in high-level language programs into pipelined dataflow machine programs is described in Gao's doctoral thesis. The machine architecture support for efficient pipelining also is briefly addressed.

Certain results of the thesis research, as well as its application of the pipelined code mapping scheme to some numerical problems, are reported in the publications of the group.

7.4. Simulating Applicative Architectures on the Connection Machine

We have simulated applicative architectures on the connection machine. This work was done as a master's thesis by Bradley C. Kuszmaul [15].

The connection machine (CM) is a highly parallel single instruction multiple data (SIMD)

computer, which has been described as "a huge piece of hardware looking for a programming methodology." Applicative languages, on the other hand, can be described as a programming methodology looking for a parallel computing engine.

By simulating architectures that support applicative languages ("applicative architectures") (*e.g.*, dataflow and combinator reduction architectures) on the CM we can achieve the following goals:

- Quickly and easily experiment with the design and implementation of applicative architectures.
- Run large applicative programs effeciently enough to gain useful experience.
- Support programming environments that allow us to do general purpose computation on the CM.

Bradley Kuszmaul's thesis describes the techniques which we use to simulate applicative architectures on the CM, and the discuss implications for the generalized case of simulating multiple instruction multiple data (MIMD) systems on single instruction multiple data (SIMD) computers.

Publications

1. Arvind, and D.E. Culler. "Managing Resources in a Parallel Machine." *Proceedings of IFIP TC-10 Working Conference on Fifth Generation Computer Architecture*, Manchester, England, North-Holland Publishing Company, July 15-18, 1985.
2. -. "Dataflow Architectures." MIT/LCS/TM-294, MIT Laboratory for Computer Science, Cambridge, MA, February 1986. (To appear in *First Annual Review in Computer Science*).
3. Arvind, and R. A. Iannucci. "Two Fundamental Issues in Multiprocessing." Computation Structures Group Memo 226-4, MIT Laboratory for Computer Science, Cambridge, MA, January 1986. January 1986.
4. Brobst, S.A., T. Malone, K. Grant, and M. Cohen. "Toward Intelligent Message Routing Systems." *Proceedings of the 2nd International Symposium on Computer Message Systems*, Boston, MA, September 4-6, 1985. (Also available as CISR WP \#129 and Sloan SP \#1709-85.)
5. _. "The Dataflow model: An Alternative to von Neumann Architectures." *Proceedings of the 5th International Conference in Computer Science,* Santiago, Chile, July 15-17, 1985.
6. Gao, G-R. "A Maximally Pipelined Tridiagonal Linear Equation Solver." International Journal of Parallel and Distributed Computing), 1986 (to appear).
7. -. "A Pipelined Code Mapping Scheme for Tridiagonal Linear Systems." *Proceedings of the 1986 Working Conference on Highly Parallel Computer Architecture*, Nice, France, March 24-26, 1986.

8. -. "Massive Fine-Grain Parallelism in Array Computation: a Dataflow Solution." *Proceedings of Future Directions of Computer Architecture and Software Workshop*, Charleston, VA, May 5-7, 1986.

9. Heller, S.K. "Directed Cube Networks: A Practical Investigation." Computation Structures Group Memo 253, MIT Laboratory for Computer Science, Cambridge, MA, July 1985.

10. Iannucci, R.A. "Dataflow Computer Architecture: an Introduction." Lecture Notes for the International Summer School on Advanced Programming Technologies, Facultad de Informatica, San Sebastian, Spain, September 1985.

11. Nikhil, R.S. "Practical Polymorphism." *Proceedings of Functional Programming Languages and Computer Architecture.*, Springer-Verlag, LNCS 201, Nancy, France, September 1985.

12. _. "Functional Databases, Functional Languages." *Proceedings of the Workshop on Persistence and Data Types*, Appin, Scotland, August 1985.

13. Nikhil, R.S., and Arvind. "Id/83s." Computation Structures Group Memo 249, MIT Laboratory for Computer Science, Cambridge, MA, July 1985.

14. Papadopoulos, G.M. "Redundancy Management for Synchronous and Asynchronous Systems,"NATO AGARD Lecture Series No. 143, NASA Dreyden, CA; Copenhagen, Denmark; Athens, Greece; October 1985.

15. _. "Design Issues in Data Synchronous Systems." NATO AGARD Lecture Series No. 143, October 1985.

16. Pingali, K.K., and Arvind. "Efficient Demand-Driven Evaluation (I)." *ACM TOPLAS*, vol. 7, no. 2, April 1985. *Corrigendum*: *ACM TOPLAS*, vol. 8, no. 1, January 1986.

17. _. "Efficient Demand-Driven Evaluation (II)," *ACM TOPLAS*, vol. 8, no. 1, January 1986.

18. Pingali, K.K., and V.K. Kathail. "An Introduction to the λ-calculus." Computation Structures Group Memo 258, MIT Laboratory for Computer Science, Cambridge, MA, March 1986.

19. Soley, R.M. "Generic Software for the Emulation of Multiprocessor Architectures." MIT/LCS/TR-339, MIT Laboratory for Computer Science, Cambridge, MA, July 1986.

20. Traub, K.R. "A Dataflow Compiler Substrate." Computation Structures Group Memo 261, MIT Laboratory for Compiler Science, Cambridge, MA, August 1986.

Theses Completed

1. Beckerle, M.J. "Logical Structures for Functional Languages." S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, February 1986.

2. Bromley, G.F. "Waiting/Matching for Tagged-Token Dataflow Architectures." S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1986.

3. Brobst, S.A. "Instruction Scheduling and Token Storage Requirements in a Dataflow Supercomputer." S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1986.

4. Chen, S-W. "A Practical Polymorphic Type-inference Type-checking System." S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1986.

5. Gao, G-R. "A Pipeline Code Generation Scheme for Static Dataflow Computers." Ph.D. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, September 1986.

6. Gornish, E. "Loop Unfolding for a Static Dataflow Machine." S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1986.

7. Hughes, G.W. "A Unified View of Consistancy in Fault-Tolerant Computer Design." S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, November 1985.

8. Jagannathan, S. "Guaranteeing Data Security in a Static Dataflow Machine." S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, September 1985.

9. Kuszmaul, B.C. "Simulating Applicative Architectures on the Connection Machine." S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1986.

10. Lincoln, P.D. "DisCoRd: Distributed Combinator Reduction." S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1986.

11. Marcovitz, D. "A Comparison of Two Signal System Architectures for a Static Dataflow Machine." S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, February 1986.

12. Morais, D.R. "Id World: An Environment for the Development of Dataflow Programs Written in Id." S.B.thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1986.

13. Pingali, K.K. "Demand-Driven Evaluation on Dataflow Machines." Ph.D. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1986.

14. Pinkerton, J.T. "A Method for Translating from a Hierarchical Design System into a Flat Design Structure." S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1986.

15. Wanuga, T. "Routing Network Performance in a Static Dataflow Machine." S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, September 1985.

16. Younis, S. "The Clock Distribution System of the Multiprocessor Emulation Facility." S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1986.

Theses in Progress

1. Chien, A.A. "Congestion Control in Routing Networks." S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected August 1986.
2. Chu, T-A. "A Design Methodology for VLSI Self-timed Systems." Ph.D., MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected September 1986.
3. Gao, G-R."A Pipeline Code Generation Scheme for Static Dataflow Computers. Ph.D. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected September 1986.
4. Kathail, V.K. "Optimal Evaluators for Lambda-calculus Based Functional Languages." Ph. D. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, expected September 1986.
5. Kaushik, S. "A Hardware Error Checker for the Packet Switch." S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected May 1986.
6. Maa, G. K. "Scalability of the Tagged-token Dataflow Machine." S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected December 1986.

Talks

1. Arvind. "MIT Tagged-Token Dataflow Project." G.E.C., Penta Hotel, London, England. July 15, 1985.
2. _, "Managing Resources in a Parallel Machine." IFIP TC-10 Working Conference on Fifth Generation Computer Architecture, UMIST, Manchester, England, July 18, 1985.
3. _. "Dataflow: A Way of Doing Reduction"and 3 more lectures on Dataflow and Reduction, First Autumn Workshop on Reduction Machines, Ustica, Italy, September 3-13, 1985.
4. _. "Why Dataflow Architectures?." High Technology Futures, CDC, Riverwood Conference Center, Monticello, MN, September 19, 1985.
5. _. "Dataflow Research in Japan." High Technology Futures, CDC, Riverwood Conference Center, Monticello, MN, September 19, 1985.
6. _. "Why Dataflow Architectures?" Distinguished Lecture Series, Cornell University, Ithaca, NY, October 17, 1985.
7. _. "Why Dataflow Architectures?" Distinguished Lecture Series, Brown University, Providence, RI, October 31, 1985.
8. _. "Demand-Driven Evaluation on Dataflow Machines." Brown University, Providence, RI, October 31, 1985.

9. _. "Why Dataflow Architectures?" Oregon Graduate Center, Portland, OR, November 14, 1985.

10. _. "Why Dataflow Architectures?" Carnegie Mellon University, Pittsburg, PA, November 22, 1985.

11. _. "Demand-Driven Evaluation on Dataflow Machines." Invited talk, Fifth Conference on Foundations of Software Technology and Theoretical Computer Science, New Delhi, India. December 18, 1985.

12. _. "Why Dataflow Architectures?" Tata Institute of Fundamental Research, Bombay, India, December 20, 1985.

13. _. "Dataflow Architectures." Tata Institute of Fundamental Research, Bombay, India, December 20, 1985.

14. _. "I-Structures." Tata Institute of Fundamental Research, Bombay, India, December 23, 1985.

15. _. "Parallel Machines are Coming." Tata Consulting Services, Bombay, India, December 23, 1985.

16. _. "Why Dataflow Architectures?" Institute Lecture, I.I.T., Kanpur, India, January 1, 1986.

17. _. "Why Dataflow Architectures?" I.I.T., Delhi, India, January 3, 1986.

18. _. "The Dynamic Dataflow Architecture." Advanced Course on New Approaches to the Architecture and the Design of Embedded Systems, E.T.H., Zurich, Switzerland, March 6, 1986.

19. _. "Characteristics of a Processor for a General-purpose Parallel Machine." Workshop on Design and Performance Issues in Parallel Architectures, University of Maryland, College Park, MD, March 17, 1986.

20. Brobst, S.A. "Applying Microcomputers in a Small Organization." AIESEC Spring Regional Conference, Boston, MA, March 8, 1986.

21. _. "Toward Intelligent Message Routing Systems." 2nd International Symposium on Computer Message Systems, Washington D.C., September 6, 1985.

22. _. "Benchmark Analysis of High Performance MIMD Machines." Harris Corporation, Advanced Technology Division, Melbourne, Florida, June 20, 1985.

23. _. "Performance Evaluation of the Tagged-Token Dataflow Architecture." Workshop on Performance Evaluation of High-Speed Computers, Institute for Computer Sciences and Technology, National Bureau of Standards, Gaithersburg, Maryland, June 6, 1985.

24. Culler, D.E. "Parallel Processing." Parallel Processing Tutorial, Newport, R.I., June 10, 1986.

25. Dennis, J.B., "Dataflow Computing--An Inside View." Cornell University, Ithaca, NY, March 19, 1986.

26. _. "Dataflow Computing--An Inside View." New York University, New York, NY, March 20, 1986.

27. _. "Dataflow Computing--An Inside View." Yale University, New Haven, March 21, 1986.

28. _. "The VIM Project: Experimental Computer Design Using Dataflow Programming Principles, MIT, Laboratory for Computer Science, April 23, 1986.

29. Iannucci, R. A., "The MIT Multiprocessor Emulation Facility", MIT Summer Course 6.83s, MIT, August, 1985.

30. _. Six Lectures on Dataflow Architectures and MEF, International Summer School on Advanced Programming Technologies, Facultad de Informatica, San Sebastian, Spain, September, 1985.

31. _. "The MIT Multiprocessor Emulation Facility and Dataflow Projects." The University of Manchester, Department of Computer Science, Manchester, ENGLAND, September, 1985.

32. _. "The MIT Multiprocessor Emulation Facility and Dataflow Projects." IBM Cambridge Scientific Center, Cambridge, MA, November 1985.

33. _. "Dataflow Research at MIT." IBM System Technology Division Headquarters, Endicott, NY, May 1985.

34. _. "Dataflow Research at MIT." IBM Data System Division Headquarters, White Plains, NY, September 1985.

35. _. "Dataflow Research at MIT." IBM Cambridge Scientific Center, Cambridge, MA, December 1985.

36. Pingali, Keshav K., "Lazy Evaluation on Dataflow Machines." University of Pennyslvania, Philadelphia, PA, March 20, 1986.

37. _. "Lazy Evaluation on Dataflow Machines." Microelectronics and Computer Technology Corporation, Austin, TX, March 3, 1986.

38. _. "Lazy Evaluation on Dataflow Machines." University of Texas, Austin, TX, March 5, 1986.

39. _. "Lazy Evaluation on Dataflow Machines." Yale University, New Haven, CT, April 1, 1986.

40. _. "Lazy Evaluation on Dataflow Machines." University of Washington, Seattle, WA, April 8, 1986.

41. _. "Lazy Evaluation on Dataflow Machines." Stanford University, Stanford, CA, April 10, 1986.

42. _. "Lazy Evaluation on Dataflow Machines." Cornell University, Ithaca, NY, April 17, 1986.

43. _. "Lazy Evaluation on Dataflow Machines." University of Illinois, Urbana, IL, May 12, 1986.

44. Nikhil, Rishiyur S.,"Functional Databases, Functional Languages."

45. _. "Functional Databases, Functional Languages." Microelectronics Technology Corporation, Austin, TX, June 1985.

46. _. "Practical Polymorphism." University of St. Andrews, Scotland, August 1985.

47. _. "Functional Databases." Digital Equipment Corporation, Hudson, MA, February 1986.

48. "Functional Programming Languages." Three lectures in Professor Barbara Liskov's graduate course on programming languages, November 1985.

49. Papadopoulos, G.M. "Redundancy Management for Synchronous and Asynchronous Systems." NATO AGARD Lecture Series No. 143, NASA Dreyden, California; Copenhagen, Denmark; Athens, Greece; October 1985.

50. _. "Design Issues in Data Synchronous Systems." NATO AGARD Lecture Series No. 143, NASA Dreyden, California; Copenhagen, Denmark; Athens, Greece; October 1985.

References

[1] Ackerman, W.B. and J.B. Dennis.
VAL--A Value-oriented Algorithmic Language: Preliminary Reference Manual.
Technical Report MIT/LCS/TR-218, MIT Laboratory for Computer Science, Cambridge, MA, June, 1978.

[2] Arvind and K.P. Gostelow and W. Plouffe.
An Asynchronous Programming Language and Computing Machine.
Technical Report, Department of Computer Science, University of California, Irvine, Irvine, CA, December, 1984.

[3] Arvind and Thomas, R.E.
I-Structures: An Efficient Data Type for Functional Languages.
Technical Report Computation Structures Group Memo 178, MIT Laboratory for Computer Science, Cambridge, MA, October, 1981.

[4] S.A. Brobst.
Token Storage Requirements in a Dataflow Computer.
Master's thesis, MIT Department of Electrical Engineering and Computer Science, May, 1986.

[5] S-W. Chen.
A Practical Polymorphic Type-inference Type-checking System.
Technical Report S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May, 1986.

[6] A.A. Chien.
Structure Referencing in the Tagged-token Dataflow Architecture.
Technical Report Computation Structures Group Memo 268, MIT Laboratory for Computer Science, Cambridge, MA, October, 1986.

[7] D.E. Culler.
Resource Management for the Tagged-Token Dataflow Architecture.
Master's thesis, MIT Department of Electrical Engineering and Computer Science, January, 1985.

[8] G-R. Gao.
A Pipeline Code Generation Scheme for Static Data Flow Machine.
PhD thesis, MIT Department of Electrical Engineering and Computer Science, September, 1986.

[9] E. Hao.
PID Translator User's Manual.
August, 1985.
UROP Final Report.

[10] S.K. Heller.
Directed Cube Networks: A Practical Investigation.
Technical Report Computation Structures Group Memo 253, MIT Laboratory for Computer Science, Cambridge, MA, July, 1985.

[11] S.K. Heller and K.R. Traub.
Id Compiler User's Manual.
Technical Report Computation Structures Group Memo 248, MIT Laboratory for Computer Science, Cambridge, MA, May, 1985.

[12] R.A. Iannucci.
Packet Communication Switch for a Multiprocessor Computer Architecture Emulation Facility.
Technical Report Computation Structures Group Memo 220, MIT Laboratory for Computer Science, Cambridge, MA, October, 1982.

[13] S. Jagannathan.
Guaranteeing Data Security on a Static Data Flow Machine.
Master's thesis, MIT Department of Electrical Engineering and Computer Science, October, 1985.

[14] B.C. Kuszmaul.
Type Checking in V{\sc im V{\sc al
}.}
Technical Report S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May, 1984.

[15] B.C. Kuszmaul.
Simulating Applicative Architectures in the Connection Machine.
Master's thesis, MIT Department of Electrical Engineering and Computer Science, June, 1985.

[16] P.D. Lincoln.
DisCoRd: Distributed Combinator Reduction.
Technical Report S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May, 1986.

[17] D.R. Morais.
Id World: An Environment for the Development of Dataflow Programs Written in Id.
Technical Report S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May, 1986.

[18] Nikhil, Rishiyur S.
An Incremental, Strongly-Typed Database Query Language.
PhD thesis, Moore School, University of Pennsylvania, Philadelphia, August, 1984.
Available as Technical Report MS-CIS-85-02.

[19] R.S. Nikhil.
Practical Polymorphism.
In *Proceedings of Functional Languages and Computer Architecture.* Springer-Verlag, LNCS 201, Nancy, France, September, 1985.

[20] Nikhil, R.S. and Arvind.
Id/83s.
Technical Report Computation Structures Group Memo 249, MIT Laboratory for Computer Science, July, 1985.

[21] Pingali, K.K.
Demand-driven Evaluation on Dataflow Machines.
PhD thesis, MIT Department of Electrical Engineering and Computer Science, July, 1986.

[22] Traub, K.R.
A Dataflow Compiler Substrate.
Technical Report Computation Structures Group Memo 261, MIT Laboratory for Computer Science, Cambridge, MA, March, 1986.

[23] S.G.Younis.
The Clock Distribution System of the Multiprocessor Emulation Facility.
Technical Report S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June, 1986.