LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Computation Structures Group Progress Report 1988-89

Computation Structures Group Memo 300
June 15, 1989

This report describes research done at the Laboratory of Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-84-K-0099.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Computation Structures Group

Academic Staff

Arvind *(Group Leader)*
J.B. Dennis
R.S. Nikhil

Research Staff

G.A. Boughton
J. Young
G.M. Papadopoulos
R.P. Johnson

Graduate Students

S. Aditya
P.S. Barth
S.A. Brobst
D.E. Culler
B. Guha Roy
D.S. Henry
S.K. Heller
J.E. Hicks
A.K. Iyengar
S. Jagannathan
C.F. Joerg
V.K. Kathail
B.C. Kuszmaul
J.S. Onanian
S. Sharma
R.M. Soley
K.M. Steele

Undergraduate Students

D. Chiou
Y. Chery
C. Fabian
S. Furman
L. Muryanto
J. Santoro
I. Scharf
A. Shaw
D. Stetson
P. Tan
G. Wang

Support Staff

S.M. Hardy
N.F. Tarbet

Technical Staff

J.P. Costanza
R.F. Tiberio

Visitors and Adjunct Members

A. Altman (Texas Instruments)
Z. Ariola (Harvard University)
M. Heytens (Dept of EECS, MIT
F. Hutner (Siemens, Munich)
zenaS. Landsberg (Dept of Aeronautics and Astronautics, MIT)
M. Sadoune (Dept of Aeronautics and Astronautics, MIT)

Computation Structures Group

1 Introduction and overview

Our group is interested in general-purpose parallel computation. Our approach is centered on

- declarative, implicitly parallel languages.
- dataflow architectures, which are scalable because of their tolerance of increased memory latencies and support for frequent synchronization. Our vehicles for research include an abstract "Explicit Token Store" architecture (ETS), a hardware prototype implementation of ETS (Monsoon), various software emulators (GITA, MINT), a new proposed architecture called P-RISC, and a software emulator for it.
- sophisticated compiling and run-time systems for Id, both for dataflow and other architectures. We have also explored the use of dataflow compiling for an experimental persistent programming language to tolerate disk latencies by exploiting parallelism.
- applications programs to guide the language, compiler and architecture research.

Last year we reported that our research results in Project Dataflow had reached a level of maturity where we were ready to embark on the construction of a real dataflow machine, using the Monsoon processor architecture [11]. Towards that end, we held a meeting in March 1988 with prospective industrial partners. Since that meeting, Motorola, Inc. has emerged as our partner; it is setting up a research laboratory in Cambridge and will participate actively in the construction of the Monsoon system.

The most notable achievement of the past year has been the realization of a wire-wrap prototype of the Monsoon dataflow processor. It has been running small hand-coded programs since September 1988 and compiled code since December. It has been used to guide the design of the printed-circuit board for Monsoon. We continued to make progress on the design and implementation of the Monsoon interconnection network, consisting of PaRC switching chips and high-speed data links. We have begun work on the design of an I-structure memory board for Monsoon.

Our main research vehicle for programming languages is Id, a mostly functional programming language. We completed the basic type system and are exploring the use of a simplified version of a new overloading mechanism [14]. Id is a non-strict language designed for more parallelism, but non-strictness is not achieved *via* laziness, as is usually the case. Instead, we have explored the implications of using explicit constructs for lazy evaluation to deal with infinite structures. For nondeterministic access to shared state, we have developed a new construct called a "manager" that is similar to but more flexible than monitors, and also allows more concurrency. We have also explored a few other experimental language designs: a language with naming environments as first-class objects, and a language for signal processing. Our group is well represented in the international committee that is designing the new functional programming language Haskell.

On the more theoretical side, we have formalized Id's operational semantics using rewrite rules, and have been able to prove results about determinacy and to be more precise about

such concepts as termination, errors, *etc.* We have also studied optimal interpreters for the lambda-calculus.

We have ported a subset of Id World, our programming environment for Id, to the Unix environment. This should make Id available to a much larger audience. The Unix version lacks the graphics of the original Lisp-machine version; this work remains to be done.

We have incorporated more optimizations in the Id compiler, and are moving its target away from the Tagged-Token Dataflow Architecture (TTDA) to an Explicit Token Store model (ETS), of which Monsoon can be considered a specific implementation. We began to look very seriously at the run-time system and the control of parallelism in Id programs for better resource management, and have implemented several experimental mechanisms to that end.

Our repertoire of Id applications continues to grow and includes DNA sequence analysis, airport landing approach planning, computational fluid dynamics, image processing and simulated annealing.

Our architecture research has also moved further in the direction of achieving a synthesis between von Neumann and dataflow ideas. We proposed a new architecture called P-RISC (for "Parallel RISC"), and have begun simulation and compilation studies.

Based on the I-structure notation in Id, we have designed a "functional database language" in which data do not change— update transactions specify new versions of a database. We are implementing this database language, using ideas from P-RISC compilation to exploit parallelism to hide disk latencies.

2 Personnel

After finishing his doctorate in August 1988, Gregory Papadopoulos became a member of research staff, working as the chief architect for the Monsoon prototype processor in Project Dataflow.

Jonathan Young joined us as research associate after completing his Ph.D. with Professor Paul Hudak at Yale. He is working on the compiler back end and run-time system for Monsoon. His research is in compile-time semantic analysis and optimization of functional programs.

R. Paul Johnson joined our research staff in October and has been working on porting the existing Id World to Unix machines.

Franz Hutner, an engineer at Siemens in Munich, spent the summer months of 1988 working with our hardware team on the connection between the local bus of the Monsoon processor and the network.

Arthur Altman joined CSG in January 1989 as a visiting researcher from Texas Instruments, to study the dataflow approach to programming languages and architectures when applied to problems in image understanding.

After finishing Ph.D. in May 1988, Kenneth Traub stayed on as a research staff member. In early 1989 he joined the Cambridge Research Center of Motorola, the industrial partner on the Monsoon project.

It is with great sadness that we record the passing of Bhaskar Guha Roy on March 23, 1989. He worked first with Professor Dennis and later with Professor Nikhil. He fought an incredibly courageous, year-long battle against liver cancer, during which he managed to write his Ph.D. thesis proposal and set up his committee.

3 Programming languages

3.1 Id

In September, we released the reference manual for Version 88.1 of the Id programming language [9], which augmented the language with constructs for loop bounding.

3.2 Types and Overloading

During the summer and fall of 1988, Shail Aditya revised and upgraded the type checking system of the Id compiler to incorporate changes from the Id87 version to Id88. This involved the addition of several key features to type analysis, *viz.*, algebraic data types, constructor case analysis and abstract data types. Further, the type checker was made totally incremental at the procedural level. Thus, in the version currently installed, the user can compile individual procedures interactively from the editor, in any order. The type checker, installed as a module in the Id compiler, incrementally assembles enough information to check the type consistency of the accumulated program at each interactive step. Using this information, the run-time environment is able to double-check the type consistency of all the procedures in the invocation graph just before execution. The user is notified in case of any discrepancy and the appropriate section of the program can be corrected and re-compiled.

During the winter and spring of 1989, Aditya worked on a mechanism for the resolution and compilation of overloaded operators and general user-defined identifiers. The idea is a simplification of the system proposed by Wadler and Blott [14] which has been adopted in Haskell. Unlike previous overloading schemes, this one is not *ad hoc*. It is capable of expressing "recursive overloading", *e.g.*, if "+" is already overloaded on integers and floats, then it can also be overloaded to mean addition of lists of integers and floats and, inductively, on lists of lists of integers and floats, *etc.* There is a systematic way of resolving this overloading.

The type checker with overloading resolution is currently undergoing tests as regards efficiency of compilation and execution. We are also conducting experimental tests with existing Id programs including large scientific codes such as SIMPLE. It will be installed in the Id compiler in the near future. The proof of consistency of the incremental type system and the details of the overloading mechanism are due to appear in Aditya's forthcoming master's thesis.

The straightforward resolution of overloading results in some inefficiency because a procedure that uses the symbol "+" is implemented as one that receives an addition function as a

parameter, which is applied using a general function call. It remains to be seen how this can be optimized through a process called "specialization", where separate versions of the procedure are compiled, one for each implementation of "+" that is of interest.

3.3 Lazy evaluation

Id has non-strict semantics, which means that a procedure or data constructor application can produce a value before the value of its arguments is known. Traditionally, languages with non-strict semantics have been implemented using lazy evaluation, where nothing is evaluated until it is known that the value is needed for the result. Unfortunately, by the time an expression *is* needed, a lazy evaluator has already paid the overhead of building a closure for the expression and re-scheduling it. Further, it has lost the opportunity of evaluating it concurrently with other computations. For these reasons, we choose not to use lazy evaluation in Id.

However, lazy evaluation can be very useful for programming with infinite structures (*e.g.*, streams), and for large data structures of which only a small part is actually used. Steven Heller completed a doctoral thesis in January 1989 in which he investigated the design, use and implementation of explicitly designated lazy data structures in Id [5]. Heller and James Hicks implemented lazy data-structures in the graph intrepreter (GITA) based on some preliminary work of UROP student Charles Fabian. Fabian was able to show that of the numerous examples of applications using lazy evaluation in the literature, most of them needed only non-strictness and not laziness. The few instances where laziness was actually necessary were easy to identify, and it was quite simple to use the explicit lazy data structures in Id. Jonathan Young and Hicks implemented a restricted version of lazy data-structures on Monsoon (four states instead of sfive states in the state diagram, since Monsoon only has two status bits). Lazy data-structures are being used to implement global constants and for stream programming, and have also been used in system code for memory allocation.

3.4 Managers

Paul Barth continued his research on *managers*, a construct for supporting nondeterministic computation in Id. Nondeterministic constructs are needed for state-sensitive computation, including "application" programs, such as real-time systems and database systems that respond to multiple inputs according to their temporal order. They are also necessary for "systems" programs, such as run-time support for the implementation of a functional language, which are needed to manipulate the state of the machine.

The manager construct was redesigned to facilitate programming abstraction and efficient implementation. Rather than acting as stream functions, managers have been recast as abstract data types, with operators that access and update a shared state. This is beneficial from two standpoints. As a programming construct, this makes the nondeterminism explicit while encapsulating the state transformation. Each potentially nondeterminstic operator is easily identified, and can be written as a function from old state to new state.

Managers are similar to monitors but allow much more flexibility in scheduling the queues of waiting processes, and also allow much more concurrency between state-manipulating procedures.

From an efficiency point of view, the new paradigm allows mutual exclusion to be provided by hardware primitives rather than stream operations. These primitives, called *locks*, are an extension of I-structure operations that provide efficient mutual exclusion on individual memory cells. The design of locks (developed jointly by Barth, Soley, and Kenneth Steele) is currently being filed for patent. The new manager construct is fully described in CSG Memo 294.

Managers were incorporated into the compiler, and applications were developed, including the dining philosophers problem, a shared bank account (with deferred debits), a printer scheduler, a buddy system memory allocator, and a union-find set algorithm. These examples indicated that the new design was more perspicuous and efficient than stream-based managers.

3.5 P-TAC: formalization of Id's operational semantics

Arvind and Zena Ariola introduced a new notation aimed at proving the Church-Rosser property of Id and allowing formal reasoning about the correctness of the optimizations used in the Id-to-dataflow-graph compiler. We call the new notation P-TAC and define it as a programming language by giving its operational semantics in terms of an Abstract Reduction System [1]. The need for introducing yet a new formalism is due to the fact that current intermediate languages are not suitable candidates for supporting the current trend of unification between functional and logic languages. At issue is the sharing of computation. For example, the formalism must distinguish between the following two programs

$$(\ (Fa), (Fa)\ ) \text{ and } \{x = Fa;\ \textsf{in}\ (x, x)\ \}$$

which may arise as a consequence of evaluating $G\ (F\ a)$, where $G\ x = (x, x)$ and "," is the pairing operation. P-TAC captures such distinctions. Notice that by making the above distinction we are able, for example, to reason about the optimization called "common subexpression elimination", that would be impossible otherwise.

A characteristic of P-TAC is that every subexpression must have a name. This simplifies the analysis of the effect of performing a reduction on the surrounding context. The most novel aspect of P-TAC is the way it models data structure operations using a special class of identifiers called *Locations*. The *only* permissible operations on locations are "**I-fetch** l", for reading the contents of location l, and "**I-store** $l\ v$", for storing v, some *ground value*, in location l. Unlike variables, which are names for expressible values, locations are merely names of memory locations and as such they are *globally unique*, that is, the scope of a location identifier is the entire program.

The syntax and the associated rewrite rules do not allow any confusion between the name of a location and its contents. Thus, while a binding like "$x = y$" means that the variables x and y are names for the same value, the corresponding binding for locations "$l_i = l_j$" does

not make sense because two location identifiers can never be the same. No equality-test on locations is permitted; however, equality on location contents can be expressed by writing "{ x = I-fetch l_i; y = I-fetch l_j; in (Equal? x y) }".

Thus, one way to give a precise operational semantics for Id is to provide a translation of Id into P-TAC. The problem of proving the confluence of Id is then reduced to proving that P-TAC is indeed confluent. The proof of the confluence of P-TAC turns out to be fairly easy due to the simplicity of the language.

The compiler optimizations are then formalized as the hidden operational semantics of P-TAC. Thus, for example, we express the common subexpression elimination in terms of the rule

$$\frac{x = a + b; y = a + b;}{y = a + b; \underset{cse}{\longrightarrow} y = x;}$$

The correctness issue is then formulated in terms of observational congruence. In this setup we have been able to provide the sufficient conditions that the new rules have to satisfy in order to preserve the meaning of a program. All the rules currently applied by the compiler turn out to be partially correct but not totally. For example, consider the following rule

$$\frac{x = n + \underline{m}}{\mathsf{Less}\ n\ x \longrightarrow \mathsf{True}} \qquad \underline{m} > 0$$

which can create havoc in the presence of deadlocks, because its precondition can be satisfied by distinct P-TAC terms, resulting in both True and False as possible outcomes, destroying the confluence of the language.

3.6 Other language-related work

3.6.1 Sequential implementations of non-strictness

Ken Traub's work on sequential implementation of non-strict programming languages, reported last year, has continued, resulting in a paper presented at the Aspenäs Workshop on the Implementation of Lazy Functional Languages in Göteborg, Sweden. The paper is also to be presented at the 1989 Conference on Functional Programming Languages and Computer Architecture in London.

3.6.2 Symmetric Lisp

Suresh Jagannathan completed his doctoral thesis [6] on Symmetric Lisp, a novel parallel programming language in which naming environments (called *maps*) are first-class objects. Through numerous programming examples, he was able to show that many diverse programming paradigms and constructs from other languages can be expressed quite elegantly with just the map construct. Examples include records, LET and LETREC blocks, "object-oriented" programs, file systems and directories, *etc.*

Using a single construct (the map) both as a data structure as well as a control structure raises some interesting questions about formal properties of programs, because names are used both as program variables and as field selectors. For example, in the expression:

```
(with M e)
```

a free name `x` in `e` is looked up in `M`, if `M` is a map with a field `x`; otherwise, it is looked up in the surrounding lexical environment. Jagannathan developed an inference algorithm to produce statically a conservative approximation that predicted which environment a name would be looked up in. A compiler could use this information for efficient compiling name lookup efficiently. He also showed an implementation of Symmetric Lisp in terms of a translation to dataflow graphs.

3.6.3 Optimal Interpreters for the Lambda-Calculus

Vinod Kathail has continued his investigation of optimal interpreters for the λ-calculus and functional languages based on the λ-calculus. The work in the last year focused on two aspects of the interpreter we had developed: formally proving its correctness and optimality, and simplifying its exposition. To relate our interpreter to the λ-calculus, we developed a new term calculus which captures some of the essential features of the way the substitution operation of the λ-calculus is implemented in our interpreter. The term calculus is used as an intermediate step in proving the correctness of our interpreter; however, it may be of interest in its own right. We are in the process of completing the formal proofs [7].

3.6.4 PGL, a signal processing language

Janice Onanian completed a master's thesis in spring 1989 in which she developed a high-level, signal-processing language, called PGL, and a program graph representation for coarse-grain multiprocessors. Effective use of parallel processors requires dividing an application into concurrently executable tasks and assigning those tasks to processors such that their use of network resources is optimized. We plan to use the language and graph developed in the thesis to find an optimal partitioning of an application into parallel tasks for a given hardware configuration. This involves two efforts: the development of algorithms for evaluating a task partition denoted by the program graph, and finding the optimal partition by varying the parameters to the program graph. Implementation of the PGL compiler is targeted for summer 1989; development of the evaluation and optimization algorithms is to form the basis of subsequent doctoral research.

3.6.5 Haskell, a new functional programming language

Arvind, Nikhil, and Jonathan Young have continued to participate in the design of the new functional programming language Haskell. As reported last year, Haskell is being designed by a group of about twenty functional programming researchers from three continents. A draft of the report on the language was released to the FP mailing list for comments in December 1988. Extensive and lively discussion followed. The Haskell committee then met

again in Mystic, CT in May, 1989 where it charted the design decisions and actions to be taken before the final report is released in July 1989.

4 Id World, the Id programming environment

During the fall, Paul Johnson implemented a suite of interface functions designed by Soley for GITA, the graph interpreter. This suite of functions, known as the Id World Interface (IWI) will support a variety of Id World user interfaces. Id World Version 4.0 and its predecessors only provided a Lisp machine-specific graphical interface. Id World Version 4.1 includes a portable Common Lisp based command listener. An interface based on the X Windows System is under development. With the assistance of Hicks, Johnson released version 4.0 for internal testing in early December. Version 4.0 with support for Symbolics Genera 7.1/7.2 and TI Explorer 3.2/4.1 was shipped in January. Highlights of version 4.0 include an optimizing compiler for Id 88.1, Id Mode Zmacs editor support, and the GITA graph interpreter with support for top level constants. Version 4.1, which adds support for Lucid Common Lisp version 3 on Sun Workstations, was released externally for beta test in late March.

The next version of Id World will have greater separation between modules than in the current version, so that each piece may be run separately in a Unix environment as opposed to being tied to the Lisp Machine implementation. In addition, Hicks has been meticulously documenting the internals of the run-time managers and the compiler schemata used in the current system, as well as some of the desirable hacks on the new hardware.

5 Project Dataflow: The Monsoon Prototype System

5.1 The Monsoon Processing Element

A very exciting milestone was met in September 1988 when a single processor Monsoon prototype was made operational, able to execute incrementally compiled Id88 programs. The prototype implementation was engineered by Jack Costanza and Ralph Tiberio in compliance with the Monsoon microarchitecture specification developed by Greg Papadopoulos [10].

The Monsoon prototype is a 64-bit, fully pipelined (eight stages) dataflow processor. Constructed from off-the-shelf components on a single large wire- wrap panel (9U $\times$ 600mm), the processor processes a modest four million tokens per second, or approximately three dataflow MIPS of which any proportion can be double precision floating point. The processor board is enclosed in a custom cabinet with suitable power supply and cooling, and then connected via ribbon cables to a simple NuBus interface card hosted in a Texas Instruments Explorer Lisp Machine.

Hardware verification and debugging was facilitated by two design disciplines. First, we performed thorough timing simulations of the entire board using our Mentor design tools. During simulation we executed small dataflow graphs to verify overall operation and focused

specifically on various matching operations and token enqueuing sequences. The second design discipline was to employ *scan paths* for (almost) all internal state. In scan path design, each parallel register can have its contents read and written through a special serial path, and multiples of such registers have their serial paths concatenated and then looped back to form a large *scan ring.* Any bit of processor state can be accessed by shifting these serial registers. Finally, the scan rings can be read and written through NuBus operations performed by the host Lisp machine.

The prototype processor comprises over 800 bits of scannable state. Software on the host Lisp machine interprets and displays the processor state in a full screen format, with appropriate data conversions (*e.g.,* floating point) and mnemonics (*e.g.,* opcodes, field decodings). The prototype processor clock can also be single-stepped under host control, and by repeatedly stepping the clock and scanning state, a full suite of software breakpoint conditions can be established. In essence, we used the combination of scan-path design and host software to develop a sophisticated in-system logic analyzer. We found this to be a very effective debugging technique.

The Monsoon prototype forms the basis for the production Monsoon processor, a printed circuit board version to be manufactured by Motorola. Several improvements are incorporated in the production version:

- A network port based on the PaRC and link chips has been added to permit the construction of multiple processor systems.
- A set of exception mechanisms and more complete support for system programs (*e.g.,* loader, garbage collector) have been designed.
- The host interface has been changed from NuBus to VME, and a high bandwidth DMA path has been added from the host into the Monsoon frame store.
- The instruction format has been changed slightly to permit a wider opcode field (from 10 bits present to 12 bits) and variant formats are introduced that allow either two explicit destinations or a large absolute address displacement (20 bits).
- Much of the datapath has been byte sliced into 10,000 gate CMOS arrays (eight identical slices), and the specialized ALU functions that manipulates tags (the Pointer Increment Unit) has been cast by George Wang into a similar sized array.
- The pipeline rate has increased to ten million tokens per second, approximately seven million dataflow instructions per second.
- The board size has been reduced from 9U $\times$ 600mm to 9U $\times$ 400mm ("Sun size") through the use of gate arrays and surface mount assembly.

The production processor is in the final detailed design and simulation phase. We expect to hand off the design to Motorola by June 1989.

5.2 The interconnection network for Monsoon

G.A. Boughton, Christopher Joerg and John Santoro continued their work on the network for Monsoon. We have continued to develop the two chips that will be used in the network, the Packet Routing Chip (PaRC) and the Data Link Chip (DLC). PaRC is a four input four output packet router on a chip and is the primary component of the Monsoon network. DLC contains a data link transmitter and a data link receiver. The transmitter will allow a PaRC output port to be connected to an interboard cable and the receiver will allow an interboard cable to be connected to a PaRC input port.

Joerg has continued the development of PaRC. The design of PaRC has not changed significantly over the past year. Some work has been done to enhance its statistics collection abilities. Improvements were also made to the control port of PaRC. The control port is the section that allows a local controller to control several parameters of the chip's operation (such as how to do routing and what to do when errors are seen). Most of the work done on PaRC has involved creating test vectors. These vectors will be used to ensure that fabricated chips do not contain any functional defects.

Santoro has continued the development of DLC. During the past year we have used the preliminary logic design completed last year to develop a detailed design for DLC in Motorola's Mosaic II ECL gate array technology.

The top-level design of DLC has changed somewhat during the year. The primary change was the elimination of 4 into 6 encoding. Our original design called for the encoding of all data transmitted over interboard cables. The primary advantage of this encoding was the elimination of the DC component of the transmitted signal. However, encoding required that the DLC be designed to operate on a 50% faster clock. Designing DLC for such a clock turned out to be fairly difficult. Because of this, we ran a large number of tests on our proposed drivers, receivers and cable to determine whether data could be reliably transmitted without encoding. Our tests indicated that a data pattern containing an arbitrarily long sequence of 0's followed by a 1 and another long sequence of 0's could be transmitted over the cable with more than sufficient noise immunity. These tests also showed that the inverse pattern also worked. Based on these results, we elected to simplify design of the DLC by removing encoding.

The detailed design of DLC has been completed and simulated. Test vectors have been written which are sufficient for testing fabricated chips for faults. A preliminary version of the design has been transferred to Motorola. The final version of the design should be given to Motorola before June 30, 1989.

5.3 The I-structure memory board

Ken Steele officially joined the group in the summer of 1988. His first project was a joint paper with Richard Soley on an idea for integrating virtual memory address translation into the dataflow execution model, titled "Virtual Memory on a Dataflow Computer," [12].

Later in the summer, the first Monsoon prototype processor returned from wire-wrapping and was debugged and tested. During the fall, we wrote microcode for the prototype to

simulate I-Structures in the processors memory. New instructions were created to support the compiler and run-time environment. These included non-busy waiting locks and support for lazy evaluation. A patent has been applied for on the non-busy waiting locking mechanism. During the Spring, design was begun on hardware to implement I-structures and the new memory operations developed for the Monsoon prototype.

The I-structure controller will be built as the memory element of the Monsoon computer system. It will be a node on the PaRC network, so any configuration of processors and I-Structure controllers will be possible. Each controller will have 4M words of dynamic memory, with address space for 16M words. Each word has 3 presence bit, one 64-bit value, 8 bits of type, 24 bits of local pointer and error correction coding for single bit error correction and double bit error detection, a total of 50 MBytes per controller. The goal is to handle 5M tokens per second, which is faster than the network can deliver tokens. All operations will be microcoded in hardware. I-structure deferred lists will be maintained locally. The hardware will also use the I-structure presence bits and deferred list mechanisms to support non-busy waiting locks. Requests for ownership of a lock is deferred until the lock becomes available, exactly like I-structures, expect that the lock can only have one owner at any given time. These locks are used by manager code written in Id to control access to shared resources. Detailed design will begin during the summer of 1989.

5.4 MINT, a Monsoon simulator

Andrew Shaw and Jonathan Young implemented a simulator for the Monsoon architecture which proved to be an invaluable tool for debugging the hardware. For his bachelor's thesis, Shaw then extended this project into a complete interpreter that is capable of mimicing the hardware with great precision. The intent is that any object code that runs on Monsoon will run without modification on MINT. The design is highly modular and uses the Monsoon microde compiler described below.

Since we wish to simulate the processor accurately, regardless of its current microcode, a microcode to Common Lisp compiler was designed and coded by Derek Chiou. The compiler accepts Monsoon microcode and translates it into Common Lisp comparable to hand code in efficiency. Secondary considerations were human-readability and code size. Thus, the identical microcode specification used to drive the actual hardware is also compiled for the simulator, with the obvious advantage of hardware – simulator consistency. The compiler is flexible enough to adapt to any foreseeable microcode changes. The compiler is written in Common Lisp.

6 Implementations of Id

6.1 Id on Monsoon

Jonathan Young and Jamey Hicks spent much of the year porting the existing Id compiler to Monsoon (with some initial work by Bradley Kuszmaul). This enables us to run real

programs on the Monsoon wire-wrap prototype. We now have a working Monsoon compiler, as well as a loader, a run-time system, an execution manager, and a rudimentary debugger; the standard libraries have also been ported.

While much of the work of porting the compiler was easy because the Monsoon ETS architecture strongly resembles the previous TTDA architecture, the run-time system required substantial work. The GITA simulator relied on the storage management of the Lisp machines; on Monsoon we implemented hand-coded managers for free lists, frames for procedure calls, and two different heaps. We also implemented managers for I-structures and semaphores ("locks") to tide us over until we have a working I-structure memory board. In addition, special managers were needed to support particular language features such as delays and accumulators.

Using the execution manager, the user may now call any Id function which has been compiled and loaded into Monsoon with as many arguments as desired. Execution is currently limited either to "run until done" or "one step at a time," where all eight stages of the Monsoon processor pipeline are visible. After a program error has been detected, various tools allow the user to view waiting tokens, data structures and instruction memory.

Barth and Young developed a graph browser, and Douglas Stetson improved the display heuristics. The browser proved to be very useful for debugging both the compiler and Id programs, and it has been installed in the Monsoon system. When compiling, the graph of a procedure is optionally displayed; it is also possible to view the graph of a procedure with its waiting tokens after a partial execution on Monsoon.

6.2 Storage management

Ken Steele wrote microcode for the processor prototype to simulate I-structures in the processor's memory. Also, new instructions were created to support the run-time environment and compiler. These included non-busy waiting locks and support for lazy evaluation. A patent has been applied for on the non-busy waiting locking mechanism.

A storage management system was implemented in Id for dynamic allocation and deallocation of structure memory and frame memory. Stephen Brobst implemented a buddy system algorithm using nondeterministic lock and unlock primitives, which were provided as extensions of the Id language as a result of work done by Barth. Multiple instantiations of the allocation and deallocation routines can proceed in parallel, with suspension occurring only when two allocations attempt to allocate blocks of the same size. Fast path execution of the memory allocation routine requires less than 50 RISC-like instructions for its critical path. Young ported the buddy system to the Monsoon architecture and augmented the storage management system with stack-based allocation mechanisms for cons cell and fixed-size frame memory allocation. Brobst has also written an Id version of the first-fit algorithm and is experimenting with various granularities for free list management.

A first version of the Id run-time system has been specified and is now under implementation. The storage management system will leverage the work of Barth, Brobst and Young to provide dynamic allocation of structure storage and frames for large codeblocks using the

buddy system, and in-line stack allocation of memory for cons cells and fixed size frames. The I/O subsystem will provide a primitive interface to the file system using string objects and standard system-call interfaces for file open, close, read, and write. Extensions to the Id language for synchronizing multiple reads and writes to a single file are an active area of research.

6.3 Long-term software structure

The above retargeting of Id for Monsoon uses the TTDA code that is produced from the existing back end of the compiler. This is not an attractive route in the long term. Young has written a specification of the ETS abstract machine [15] for use in compiling to the Monsoon architecture as it evolves.

Traub has designed the architecture of the software system which will support Monsoon, to be implemented jointly at MIT and at Motorola Cambridge Research. The greatest difference between the new software system and the old TTDA/GITA system is one of modularity. Whereas the functions of loading, running and debugging Id programs, and displaying run-time statistics were previously all handled by the GITA program, in the architecture each of these functions will be handled by separate programs, with a top-level program provided to present the user with essentially the same programming environment as found in the current Id World. The resulting system will be much more robust and flexible, and will point the way for the eventual migration of these functions onto the dataflow processor itself. Perhaps even more importantly, these programs are designed to work both with Monsoon hardware and its software emulation (MINT). Local area networks are an important part of the new system, both in the use of X Windows as the framework for the user interface and in providing a network path to the Monsoon hardware or emulator. This will allow several users to share Monsoon with ease. The software architecture and all the interfaces between its components are thoroughly documented in [13].

Hicks and Traub designed the Monsoon Object Code (MOC) format. This is the format in which the Id compiler (and other programs) will write object files. MOC is based on CIOBL (Common Input/Output Base Language), redesigned by Traub.

Hicks had also made an initial design of the Id Object Format. The Id Object Format describes the data structures that will be loaded for an Id program. There will be structures for each procedure, global constant, and code block compiled and loaded. These structures will hold computed values and program code, and will support dynamic linking. They will also have source information for use in debugging. All program information needed at run-time will be structured using the Id Object Format. The actual object files will be encoded into MOC.

6.4 Experiments with structure-storage management

We are extremely interested in reducing the run-time overhead of garbage collection by reusing data structure storage whenever possible. To this end, we are exploring both user-supplied and automatically generated program annotations for the explicit deallocation of

storage. Clearly, a thorough programmer could insert annotations which suffice to enable the reuse of every data structure in a program. However, this process might prove too tedious for most programmers, and is prone to errors – in particular, the error of deallocating a structure before it is safe to do so.

Jamey Hicks extended the Id compiler to handle data structure release annotations. This allows us to deallocate data structures relatively painlessly, but it is not meant to be a language feature that users will employ. Rather, it is an experimental feature to compare the performance of hand-annotated programs with that of automatically-annotated programs.

The syntax of the annotation is:

```
@release IDENTIFIER;
```

or

```
@release IDENTIFIER_0, IDENTIFIER_1, ...  IDENTIFIER_n;
```

inside a block expression. This annotation specifies the release of the structure bound to IDENTIFIER$_i$ when all computation enclosed within the block expression has terminated. This only releases the storage corresponding to the top level of the structure; the compiler cannot determine how much sharing of substructures there is in the program, so it does not release them. The compiler inserts the synchronization code necessary to ensure that the object is not released until all of the code in the block has terminated computation.

Inside a loop, @RELEASE actually has two meanings: if the structure is not circulated, then it is released when the current iteration has terminated; otherwise, if the structure is circulated in the loop, then it releases all but the first and last values of the structure when the corresponding iteration has terminated. The release of circulating structures is accomplished by unrolling the loop once, and not releasing the structure in the initial execution of the body.

Here is an example of the @RELEASE annotation in the multiwave procedure:

```
%%% Run several iterations of the wavefront.
%%% Illustrates the arbitrary chaining achievable in dataflow.
%%% ***** Release the intermediate waves when through with them.
def multiwave edge_vector n =
  {m = initial_wave edge_vector;
   in
     {for i <- 1 to n do
        next m = wave m;
        @release m;
      finally m }};
```

Young has written a simple compile-time analysis program which determines when it is safe to deallocate structures in loops; deallocation annotations are then automatically added to the program.

6.5 Resource management in scientific programs

In his recently completed doctoral thesis, David Culler has made substantial progress toward effective management of parallelism and resources in dataflow programs. The problem

is that exploiting parallelism to achieve high performance invariably increases the resource requirements of a program. This phenomenon is not peculiar to dataflow; it can be observed to some degree in any form of parallel execution. However, it is particularly serious under dynamic dataflow execution, because all the potential parallelism in a program is exposed. This means that ample parallelism is available on a broad class of programs, but, unfortunately, the resource requirements of many programs are excessive, often leading to deadlock. Culler had documented both sides of this dilemma using parallelism and resource profiles of a variety of scientific programs derived under an ideal dataflow execution model (supported by GITA).

In 1985 Culler and Arvind developed a mechanism for controlling parallelism, called k-bounded loops [2]. Basically, loops are compiled into dataflow graphs in a manner that allows the maximum number of concurrent iterations to be set dynamically, when the loop is invoked. This approach is appealing for scientific programs, which are dominated by iterative computations over large, regular data structures. It has played a central role in the evolution of tagged-token dataflow architectures toward Explicit Token Store machines and hybrid machines, because it allows the tag-space to be used densely. Further, it provides a natural means of reusing resources within iterative computations. The question he has been exploring recently is how to assign the k-bounds automatically.

Culler's approach is to rely heavily on static analysis to characterize the dynamic behavior of programs. There are two aspects of this analysis: worst-case resource requirements and expected parallelism. A representation of the dynamic call structure of the program is constructed and annotated with symbolic *resource expressions* which are parametric in the k-bounds and in certain program variables. In addition, loops are classified as having limited useful unfolding, expensive unfolding, and efficient unfolding. Based on this analysis, the program is augmented with resource management code that computes the k-bounds by simple formulae derived from the resource expressions that capture a high-level policy, *e.g.*,favor the middle level in this triply-nested loop. A variety of policies have been examined analytically and empirically, and a particular policy has been effective in containing the resource requirements of scientific dataflow programs, while exposing adequate parallelism.

This work forms the beginning of a bridge between our research in dataflow execution and the work being done in the parallel execution of FORTRAN. In our case, the problem is to constrain potential parallelism that is not cost-effective to exploit. In the FORTRAN case, the problem is to determine where it is most cost-effective to uncover parallelism. We will never reach exactly the same place, because our analysis must err in the direction of assuming two computations cannot be serialized, while theirs must err in the direction of assuming two computations cannot execute in parallel. Still, we expect there will be a valuable cross-fertilization.

6.6 Speculative parallelism

Richard Soley completed his doctoral work this year on the control of speculative parallelism in Id programs under the abstract tagged-token dataflow execution model. Although resource control models for exploiting parallelism in large scientific codes have been explored recently,

no approach to exploiting speculative, searching parallelism has been explored, even though (or perhaps because) the potential parallelism of such applications is tremendous. Soley explores a view of speculation as a process which may proceed in parallel in a controlled fashion, using examples from actual symbolic processing situations.

The central issue of exploiting this parallelism is the dynamic containment of the resources necessary to execute large speculative codes. Efficient structures (graph schemata and architectural support) are shown for executing highly speculative programs (such as expert systems) under a dataflow execution paradigm. In order to control dynamic execution graph growth, controls are developed over cross-procedure parallelism in an extensible manner, with applications to the various current problems of dataflow computation. Approaches to scheduling, prioritization and search tree pruning were considered, evaluated and compared.

Soley's thesis fleshes out the details of primitive execution resource management (function application and memory allocation), giving implementations for general and primitive resource managers and other nondeterministic constructs at the Id language level. Dynamic binding of managers is also presented to give a meaning to the term "task"; the prioritization and termination of dynamically defined tasks are supported.

The underlying constructs used by Soley's speculation control features rely on an extended definition of I-structure storage. This new definition adds an uncontrolled I-structure WRITE (as opposed to STORE) instruction, which overwrites I-structure cell contents. This nondeterministic feature is useful for implementing higher-level control constructs, as shown.

More revolutionary, however, is the new cell-locking paradigm developed by Soley, Steele and Barth. The new scheme is detailed in Soley's doctoral thesis, Steele's upcoming master's thesis, and Barth and Nikhil's report on managers [3]. The new locking structure of I-structure memory, already implemented in the GITA simulator (by Soley and Barth) and the Monsoon prototype (by Steele), relies on the existence of structure presence bits and deferral lists to allow critical section coding of resource managers and the like. In addition to supporting busy-waiting-free lock primitives, these "dataphores" also allow the storage of data in the semaphore cell itself (hence the new name). The basic contract of the locking instructions[1] are the following:

- READ-AND-LOCK (cell): returns only when the cell has been locked, with the value written to the cell when it was allocated or last unlocked.
- WRITE-AND-UNLOCK (cell, value): unlocks the cell specified, writing the given value into the cell.

Recognizing that these instructions also support a primitive queueing mechanism (albeit of nondeterministic queue order), we have found several other uses for this new feature.

[1] Variously called lock/unlock, read-and-lock/write-and-unlock and take/put; here we shall use the most verbose forms.

6.7 Garbage Collection for Id on Monsoon

Arun Iyengar has begun looking at garbage collection on dataflow multiprocessors, and is implementing a copying garbage collector for Monsoon. Simultaneously, Young is looking at compile-time techniques for detecting when heap objects are no longer needed. We plan to do a quantitative study of the amount of storage which can be reclaimed by garbage collection and static program analysis. We are also interested in the increased execution time and additional support required by these two different approaches for reclaiming heap storage.

6.8 Parallel I/O

Bhaskar Guha Roy worked on the design of a parallel I/O system for a dataflow machine. In addition to Processing Elements and I-structure memories, he proposed that Disk Units be attached to the interconnection network. Processors would interact with the disk units using split-phase transaction in a manner similar to I-structures. To initiate a disk transfer, the processor sends a token to a disk unit, specifying the direction of transfer (read/write), the address of the disk block, the address of an I-structure for the data, and the continuation of a thread that awaits the completion of the transfer. The objective is to tolerate disk latencies in exactly the same way that the latency of I-structure accesses is currently tolerated by the processor. Guha Roy designed language constructs to express parallel I/O in the presence of non-strict data structures, and designed compilation techniques for them.

6.9 Other Monsoon-related work.

Lina Muryanto and Peter Tan wrote a compiler that takes ETS code from the Id compiler and produces MC68020 code, so that Id programs may be run on Sun workstations. It uses a MIPS-like RISC language as an intermediate form, to facilitate porting it to other machines. So far, the compiler only accepts a small subset of the full language, and much work remains to be done on optimization.

7 Applications

We are happy to report an increase in the number of large application programs being written in Id.

7.1 Simulated annealing

Stephen Brobst and Philip Kuhn implemented a number of different algorithms for simulated annealing. Simulated annealing is a heuristic that is commonly applied to a large class of optimization problems that are known to be NP-complete, such as scheduling and building layout. They found that although the purely functional subset of Id did not lend itself

well to an efficient implementation, accumulators provided an elegant paradigm for handling the nondeterministic aspects of the algorithm without sacrificing overall determinacy in the program. They also made use of Barth's lock and unlock primitives along with structure overwrites to implement a purely nondeterministic, non-functional version of the program. The ability to overwrite structure elements without copying the full structure provided a large reduction in the number of instructions during program execution. However, the synchronization required for correct implementation of the algorithm in the presence structure overwrites actually increased the critical path length of the program. Moreover, debugging and program design in the presence of the locking and structure overwite primitives became substatially more difficult. Issues of deadlock, nondeterminism, read-write races, *etc.* previously not present in the deterministic implementations, became major stumbling blocks in the parallel execution environment.

7.2 DNA sequence algorithms

DNA sequence data is accumulating very rapidly. If the genetic sequence of the entire human genome is determined, databases will grow by two to three orders of magnitude from their current sizes. Parallel processing is becoming increasingly important as biological sequence data increases. Iyengar implemented several different algorithms for comparing sequences using Id. Implicit parallelism makes Id a very easy language to use. One drawback is the extra copying required when an aggregate data structure needs to be updated.

7.3 Computational fluid dynamics

A computational fluid dynamics (CFD) application was implemented in Id by Sandy Landsberg, a graduate student in the Department of Aeronautics and Astronautics. The CFD application is a solution to the inviscid, compressible Euler equations for the flow past a circular arc bump in a channel. Convergence to the steady-state solution is achieved by an iterative process. In this application, a four-stage Runge-Kutta scheme is used to update the solution. Supersonic, subsonic, and transonic flow calculations can be performed. Supersonic flow requires approximately 400 iterations for convergence while subsonic flow requires approximately 3000 iterations. For all flows, each iteration requires over one million instructions. Due to the large number of data structures (primarily arrays) being created, explicit deallocation of data structures is necessary. Through explicit deallocation, only 2% of the old data structures remain after each iteration; however, this is 1K byte of memory! Currently, 75% of the code has been compiled for the Monsoon. The entire code will be compiled for the Monsoon by June 1989.

7.4 Flight path generation

For his doctoral thesis, Michel Sadoune of the Department of Aeronautics and Astronautics has implemented a Terminal Area Trajectory Planning System for air traffic control. A Flight Path Generator is defined as the module of an automated Air Traffic Control system

which plans aircraft trajectories in the terminal area with respect to operational constraints. The flight path plans have to be feasible and must not violate separation criteria.

The problem of terminal area trajectory planning is structured by putting the emphasis on knowledge representation and air-space organization. A well-defined and expressive semantics relying on the use of flexible patterns is designed to represent aircraft motion and flight paths. These patterns are defined so as to minimize the need for replanning and to accommodate operational deviations smoothly.

Flight paths are specified by an accumulation of constraints. A parallel, asynchronous implementation of a computational model based on the propagation of constraints provides mechanisms to build feasible flight path plans efficiently. A network of constraints is implemented as the superposition of dataflow graphs which are synchronized distributively.

A methodology for fast and robust conflict detection between flight path plans is introduced. It is based on a cascaded filtering of the stream of feasible flight paths and combines the benefits of a symbolic representation and of numerical computation with a high degree of parallelism.

The Flight Path Generator is designed with the goal of implementing a portable and evolving tool which could be inserted in controllers' routine with minimum disruption of present procedures. Flight path generation and conflict detection have been implemented in Id. The program which is run with various machine configurations is composed of six hundred procedures for a size of five thousand lines of Id code. It is used as a test program for the Monsoon compiler. The conflict-free feasible flight paths which are generated and tested in an Id environment can be translated into Lisp data structures by using an interface between Id and Common Lisp. They are then displayed on the screen and simulated in an interactive manner.

7.5 DARPA image-understanding benchmark

Arthur Altman, visiting from Texas Instruments, began implementing the DARPA Image Understanding benchmark as an Id application. This benchmark performs model-based recognition of a 2 1/2 D "mobile" of rectangles from two 512 X 512 pixel images, one containing intensity data (8-bit integers), the other depth data (32-bit IEEE floating point). As such, it performs extensive numeric (data-directed) and symbolic (knowledge-directed) processing. Once the benchmark has been converted to Id, he will evaluate its potential parallelism and related performance parameters on the simulated TTDA target machine provided by the GITA environment.

8 P-RISC

Our work has continued to bridge the once-wide gap between von Neumann and dataflow architectures. In 1988, Nikhil and Arvind proposed a new processor architecture called P-RISC (for Parallel RISC) that properly extends a conventional RISC processor in such a way

as to make it more suitable as a component for a parallel machine. The architecture will be presented at the 1989 International Symposium on Computer Architecture in Jerusalem [8].

We first organize the machine so that instruction and frame memory are local to a processor, while heap memory is global. Next, we identify a frame (activation record) as *the* register set for a thread. The control state of a thread can now be described succinctly as a token containing an instruction pointer and a frame pointer.

We now reorganize the processor so that it is multi-threaded. The first step is to introduce a token queue that can contain multiple tokens. On each clock a token is dequeued and sent through the processor pipeline. The instruction it points to is fetched and executed relative to the frame that it points to. Finally, a new token is produced that is reinserted into the token queue. Note that successive tokens can be from unrelated threads.

To deal with long memory latencies, we use the technique of I-structures. A `load` instruction sends a request to memory along with a return continuation. Meanwhile, the processor is free to execute other tokens. The response from memory comes back with the return continuation— the value is stored in the frame and the continuation is requeued.

For fine-grained parallel operation, we extend the instruction set with three new instructions:

- `fork`, which is like a jump, except that it also produces the token for the next instruction (*i.e.*, it is like a jump and continue).
- `join`, which specifies a frame offset containing a counter initialized to n, the number of threads that will execute this instruction. Each execution decrements the counter. Only the thread that decrements it to 0 continues— the other threads are discarded.
- `start`, which specifies a continuation in a different frame (which may be on a different processor), along with a value to be stored in that frame before the continuation is started.

With these instructions, it is possible to emulate the fine-grained parallelism of a dataflow graph. Since it is a superset of a conventional RISC instruction set, it is also possible to execute conventional compiled code, *e.g.*, from FORTRAN.

We have begun simulation and other studies to evaluate this architecture, described below.

8.1 Compiling for P-RISC

Bradley Kuszmaul specified an abstract P-RISC instruction set, complete with operational semantics (specified by a relation on machine states).

He is implementing a P-RISC code generator for the Id compiler. Code generation proceeds by transforming a data flow graph into a control flow graph, performing certain optimizations, and then transforming the control flow graph into machine-specific code. Examples of machine specific code which might be generated include

- the abstract P-RISC instruction set mentioned above,

- other specific P-RISC instruction sets (such as the P-RISC coprocessor for a RISC chip being worked on by Sharma, see below),
- a variant of Monsoon with registers,
- Eps'88 [4],
- a standard serial machine (such as a RISC computer, a VAX, a LISP machine, or a Cray supercomputer), or
- off-the-shelf parallel MIMD hardware.

It appears that the control flow graph intermediate format is well suited for the target architectures mentioned above. Currently, only parts of the Id language are correctly compiled to control flow graphs, and the only machine specific code generated by the compiler is the abstract P-RISC instruction set and the serial code for the Lisp machine. Preliminary results indicate that it may be possible to run Id programs almost as fast, *i.e.,* within a factor of four to ten, as LISP or C programs.

8.2 Simulator for P-RISC

Ira Scharf, as part of his bachelor's thesis, has been working on an interpreter for the abstract P-RISC instruction set developed by Kuszmaul. The objective is to build a tool like GITA, our graph interpreter for the TTDA, that has proved so invaluable in evaluating the TTDA. A first version of the interpreter is now running.

8.3 Implementation of P-RISC using ordinary RISC processors

Preliminary to the P-RISC work, Bradley Kuszmaul and Madhumitra Sharma surveyed commercial RISC chips, with an eye towards P-RISC implementation. We then did some design and back-of-the-envelope analysis of various strategies for implementing P-RISC on commercial RISC hardware (possibly using some sort of coprocessor to provide a hardware assist for the P-RISC specific operations). Those (very preliminary) results indicate that an unmodified commercial RISC computer might lose only a factor of ten to fifteen over a dedicated P-RISC processor. By adding an I-structure memory to the RISC computer, the performance degradation compared to a P-RISC processor drops down to four or five. (Steele has spent some effort at thinking about how to make I-structure memory work for a RISC processor.) By adding hardware assistance for context switching, that degradation goes down to two or three.

Sharma has been trying to identify a way of efficiently caching activation frames of tasks in the processor register set so as to minimize the penalty incurred on switching from one thread to another. We have developed a write-through caching scheme that caches activation frames in a set of register windows. We have also proposed a scheme which allows a very high degree of look-ahead in the instruction stream. In other words, a processor can easily

identify the next 15-20 instructions to be executed. We accomplish this by switching threads even on conditional branch instructions— which are nondeterministic in the sense that the flow of control beyond such instructions is not known until after the instruction is executed. Putting these two schemes together, we get an architecture which permits switching between threads with minimal (potentially zero) penalty. Further, the high degree of look-ahead in the instruction stream may offer several advantages that have alluded processor-pipeline designers in the past. We are currently examining these.

9 Functional Databases

Michael Heytens continued his investigation into the synthesis of databases and functional languages, treating an update transaction as a declarative specification of a new version of the database, inspired by the treatment of I-structures in Id. After completing the design of a kernel database language to express such updates, he has begun implementing a prototype, based on ideas from compiling Id to P-RISC machines.

Publications

Arvind, Culler, D.E., and Ekanadham, K. "The Price of Asynchronous Parallelism: an Analysis of Dataflow Architectures," *Proceedings of CONPAR 88*, British Computer Society — Parallel Processing Specialists, University of Manchester, Manchester, England, September 1988. Also Computation Strucures Group Memo 278, MIT Laboratory for Computer Science, Cambridge, MA, June 1988.

Arvind, Culler, D.E., and Maa, G.K. "Assessing the Benefits of Fine-grain Parallelism in Dataflow Programs," in *International Journal of Supercomputer Applications*, 2:3. November, 1988. Also appeared in *Proceedings of Supercomputing '88*, Orlando, FL, November 1988.

Arvind, Heller, S.K., and Nikhil, R.S. "Programming Generality and Parallel Computers" in *Proceedings of the Fourth International Symposium on Biological and Artificial Intelligence Systems*, Trento, Italy, September 1988;

Ariola, Z. and Arvind. "P-TAC: A Parallel Intermediate Language," MIT Computation Structures Group Memo 295, Cambridge, MA, March 1989. To appear in *Proceedings of the Conference on Functional Languages and Computer Architectures*, London, England, September 1989.

Barth, P.S. and Nikhil, R.S. "Supporting State-sensitive Computation in a Dataflow System," Computation Structures Group Memo 294, MIT Laboratory for Computer Science, Cambridge, MA, March 1989. Submitted to the Conference on Principles of Programming Languages, San Francisco, CA, January 1990.

Chiou, D.T. "A Monsoon Dataflow Microcode to Common Lisp Compiler," Computation Structures Group Memo 299, MIT Laboratory for Computer Science, Cambridge, MA, June 1989.

Culler, D.E. "Managing Parallelism and Resources in Scientific Dataflow Programs," MIT/LCS/TR-446, MIT Laboratory for Computer Science, Cambridge, MA, June 1989.

Heller, S.K. "Efficient Lazy Data-Structures on a Dataflow Machine," MIT/LCS/TR-438, MIT Laboratory for Computer Science, Cambridge, MA, January 1989.

Henry, D.S. and Barstow, D.R., "A Method for Specification and Verification of Real Time Constraints in a Statically Allocated Coarse Dataflow Program," Schlumberger Software Conference, 2, Ann Arbor, MI, November 1988.

Heytens, M.L. and Nikhil, R.S. "GESTALT: An Expressive Database Programming System," ACM SIGMOD Record, 18:1, March 1989, pp 54-67.

Hutner, F. "The Network Interface Unit (NIU) for the Monsoon Dataflow Processor," Computation Structures Group Memo 290, MIT Laboratory for Computer Science, Cambridge, MA, August 1988.

Iyengar, A.K. "Parallel DNA Sequence Analysis," MIT/LCS/TR-428, MIT Laboratory for Computer Science, Cambridge, MA, October 1988.

Jagannathan, S. "A Programming Language Supporting First-class Parallel Environments," MIT/LCS/TR-434, MIT Laboratory for Computer Science, Cambridge, MA, January 1989.

Nikhil, R.S., ed. "Id (Version 88.1) Reference Manual," Computation Structures Group Memo 284, MIT Laboratory for Computer Science, Cambridge, MA, August 1989.

Nikhil, R.S. and Arvind. "Can Dataflow Subsume von Neumann Computing," Computation Structures Group Memo 284, MIT Laboratory for Computer, Cambridge, MA, November 1988. In *Proceedings of the 16th International Symposium on Computer Architecture*, IEEE/ACM, Jerusalem, Israel, May-June 1988.

Onanian, J.S. "A Signal-processing Language for Coarse-grain Dataflow Processors," MIT/LCS/TR-449, MIT Laboratory for Computer Science, Cambridge, MA, June 1989.

Papadopoulos, G.M. "Implementation of a General Purpose Multiprocessor," MIT/LCS/TR-432, MIT Laboratory for Computer Science, Cambridge, MA, August 1988.

Sadoune, M. "Terminal Area Flight Path Generation Using Parallel Constraint Propagation," MIT/LCS/TR-451, MIT Laboratory for Computer Science, Cambridge, MA, June 1989.

Santoro, J. "Design and Implementation of a High-speed Data Link for a Dataflow Supercomputer," Computation Structures Group Memo 298, MIT Laboratory for Computer Science, Cambridge, MA, June 1989.

Shaw, A. "Design and Implementation of MINT: A Monsoon Dataflow Simulator," Computation Structures Group Memo 297, Laboratory for Computer Science, Cambridge, MA, June 1989.

Soley, R.M. "On the Efficient Exploitation of Speculation under the Dataflow Paradigms of Control," MIT/LCS/TR-443, MIT Laboratory for Computer Science, Cambridge, MA, June 1989.

Steele, K.M. and Soley, R.M. "Virtual Memory on a Dataflow Computer," Computation Structures Group Memo 289, MIT Laboratory for Computer, Cambridge, MA, July 1988.

Traub, K.R. "Compilation as Partitioning: A New Approach to Compiling Non-strict Functional Languages," Computation Structures Group Memo 291, MIT Laboratory for Computer Science, Cambridge, MA, October 1988. To appear in *Proceedings of the Conference on Functional Languages and Computer Architectures*, London, England, September 1989.

Traub, K.R. "Sequential Implementation of Lenient Programming Languages," MIT/LCS/TR-417, MIT Laboratory for Computer Science, Cambridge, MA, October 1988.

Young, J.H. and Arvind. "Instruction Set Definition for an Explicit Token Store Machine," Computation Structures Group Memo 293, MIT Laboratory for Computer, Cambridge, MA, February 1989.

Theses Completed

Chiou, D.T. "A Reverse Compiler: A Monsoon Dataflow Microcode to Common Lisp Compiler," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1989.

Culler, D.E. "Managing Parallelism and Resources in Dataflow Programs," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA. May 1989.

Heller, S.K. "Efficient Lazy Structures on a Dataflow Machine," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, January 1989.

Iyengar, A.K. "Parallel DNA Sequence Analysis," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, August 1988.

Jagannathan, S. "Programming Language Supporting First-class Parallel Environments," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, December 1988.

Kuhn, P.S. "Implementation of Simulated Annealing," S.B. degree, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1989.

Lee, M. "Interactions between the Query Processor and Buffer Manager of a Relational Database System," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1989.

Muryanto, L. "A Translator from ETS Dataflow Graphs into RISC Code," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1989.

Onanian, J.S. "A Signal Processing Language for Coarse-grain Dataflow Multiprocessors," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1989.

Papadopoulos, G.M. "Implementation of a General Purpose Dataflow Multiprocessor," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, August 1988.

Sadoune, M. "Terminal Area Flight Path Generation Using Parallel Constraint Propagation," Ph.D. dissertation, MIT Department of Aeronautics and Astronautics, Cambridge, MA, June 1988.

Santoro, J. "Design and Implementation of a High-speed Data Link for a Dataflow Supercomputer," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1989.

Scharf, I. "A Simulator for a Parallel RISC Processor," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1989.

Soley, R.M. "On the Efficient Exploitation of Speculation Under the Dataflow Paradigms of Control," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1989.

Shaw, A. "Design and Implementation of MINT: A Monsoon Simulator," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1989.

Tan, P.K.S. "A Translator from RISC Code into MC 68020 Code," S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, June 1989.

Theses in Progress

Aditya, S. "An Incremental Type Inference System for the Programming Language Id," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA. Expected December 1989.

Henry, D.S., "Analysis of Real-time Constraints in Stream Machine," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA. Expected December 1989.

Kathail, V.K., "Optimal Evaluators for Lambda-calculus Based Functional Languages," Ph.D. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA. Expected August 1989.

Steele, K.M., "Implementation of an I-Structure Memory Controller," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA. Expected August 1989.

Lectures

Arvind. "Making it Fun to Program Parallel Computers," Consorzio per la Ricerche e le Applicazione d'Informatica, Cosenza, Italy, July 1988.

Arvind. "Dataflow Approach to General-purpose Parallel Computing," Consorzio per la Ricerche e le Applicazione d'Informatica, Cosenza, Italy, July 1988; Summer Institute, Argonne National Laboratory, Argonne, IL, September 1988; Distinguished Lecturers Series, University of Washington, Digital Equipment Corporation, Maynard, MA, and LCS 25th Anniversary Celebration Symposium, October 1988; Distinguished Lecturers Series, University of Illinois, Urbana, IL, December 1988; Distinguished Lecturers Series, Princeton University, Princeton, NJ, February 1989; Harvard University, Cambridge, MA, March 1989; MITRE Corporation, Burlington, MA, April 1989.

Arvind. "Project Dataflow," Presentation at MIT to visitors from IBM T.J. Watson Research Center, Yorktown Heights, NY, July 1988.

Arvind. "Functional Programming on Parallel Machines," Workshop on Programming Languages and Compilers, Cornell University, Ithaca, NY, August 1988.

Arvind. "Future Scientific Programming," IFIPs WC2.5, Working Group on Aspects of Computation on Asynchronous Parallel Machines, Stanford, CA, August 1988; Seminario di Informatica, Italian Society for Computer Simulation, Rome, Italy, February 1989.

Arvind. "Programming Generality and Parallel Computers," Fourth International Symposium on Biological and Artificial Intelligence, Trento, Italy, September 1988; AT&T Bell Laboratories, Holmdel, NJ, October 1988; Presentation to computer students from Twente University, The Netherlands, May 1989; American Physical Society, Boston University, Boston, MA, June 1989.

Arvind. "Can Dataflow Subsume von Neumann Computing," ACM Student Chapter, MIT, November 1989.

Arvind. "Implicit Parallelism and Declarative Programming," Principal Investigators' Meeting, Houston, TX, November 1988; Distinguished Lecturers Series, University of California, Irvine, February 1989; Distinguished Lecturers Series, University of Oregon, Eugene, OR, and Oregon Graduate Center, Beaverton, OR, April 1989.

Arvind. "Algorithms for Scientific Computing," Istituto per lo studio delle metolodogie geofisiche ambientali, Consiglio Nazionale delle Ricerche, Modena, Italy, January 1989.

Arvind. "Evolution of Dataflow Architectures," Dataflow Workshop, Eilat, Israel, May 1989.

Brobst, S.A. "Performance Evaluation of the MIT Tagged-Token Dataflow Architecture," Adelaide University, Adelaide, Australia, Melbourne University, Melbourne, Australia, Monash University, Clayton, Australia, July 1988.

Brobst, S.A. "Architecture of a Tagged-Token Dataflow Machine," Royal Melbourne Institute of Technology, Melbourne, Australia, July 1988.

Brobst, S.A. "The Application of a High-level Parallel Programming Language with Functional Semantics to Simulated Annealing," Royal Melbourne Institute of Technology, Melbourne, Australia, July 1988.

Brobst, S.A. "A Parallel Graph Interpreter for the Tagged-Token Dataflow Architecture," Los Alamos National Laboratory, Los Alamos, NM, June 1988.

Culler, D.E. "Assessing the Benefits of Fine-grain Parallelism in Dataflow Programs," Supercomputing '88 Conference, Orlando, FL, November, 15, 1988.

Culler, D.E. "Controlling Parallelism and Resource Usage in Dataflow Execution of Scientific Programs," Oregon Graduate Center, Beaverton, OR, University of Washington, Seattle, WA, University of California, Berkeley, CA, Sun Microsystems Inc., Mountain View, CA, University of California at San Diego, La Jolla, CA, March 1989.

Culler, D.E. "Managing Parallelism and Resources in Scientific Dataflow Programs," University of California, Los Angeles, CA, University of California, Irvine, CA, April 1989.

Heller, S.K. "Copying and Real Time Garbage Collection," Boston University, Boston, MA, November 1988.

Heller, S.K. "Garbage Collection in Large Address Space Capability Systems," IBM Corporation, Rochester, MN, December 1988.

Heller, S.K. "Efficient Lazy Data Structures on a Dataflow Machine," IBM Corporation, Boulder, January 1989.

Heller, S.K. "Lifetime Based Garbage Collection and its Architectural Support," Apple Computer, January 1989.

Jagannathan, "First-class Parallel Environments," Stanford University, University of California, Los Angeles, Los Angeles, CA, University of Pennsylvania, March 1989; Columbia University, University of Arizona, Xerox Parc, Palo Alto, CA, Indiana University, Bloomington, IN, April 1989.

Jagannathan, S. "A Symmetric Language," Yale University, New Haven, CT, January 1989.

Joerg, C.F., "PaRC: A 50 MHz Packet Switched Routing Chip, VLSI Review, MIT, Cambridge, MA, May 1989.

Kathail, V.K. "An Optimal Interpreter for the λ-calculus," Hewlett Packard Research Laboratories, Palo Alto, CA, April 1989.

Kuszmaul, B.C. and Fried, J., "NAP: No ALU Processor," Frontiers '88 Conference on Massively Parallel Computation, George Mason University, Fairfax, VA, October 1988.

Nikhil, R.S. "Compiling Graph Reduction for a Dataflow Machine," Aspenäs Workshop on the Implementation of Lazy Functional Languages, Göteborg, Sweden, September 1988.

Nikhil, R.S. "Id, A Parallel Programming Language," Computer Science Day, Memorial University, St. John's, Newfoundland, Canada, October 1988.

Nikhil, R.S. "The Parallel Programming Language Id and its Dataflow Implementation," Boston University, Boston, MA, November 1988.

Nikhil, R.S. "Implicit Parallelism and Heap Storage are Necessary for Parallel Languages," Workshop on Opportunities and Constraints of Parallel Computing, IBM Almaden, Almaden, December 1988.

Nikhil, R.S. "Structure and Interpretation of Parallel Computer Programs," Apple Computer, Cupertino, CA, November 1988.

Nikhil, R.S. "A Construct for Expressing State and Nondeterminism in a Parallel Language," IFIP WG 2.8 Working Group (Functional Programming), Mystic, CT, May 1988.

Nikhil, R.S. "Project Dataflow: Status Report," Dataflow Workshop, Eilat, Israel, May 1989.

Nikhil, R.S. "Can Dataflow Subsume von Neumann Computing," 16th International Symposium on Computer Architecture, Jerusalem, Israel, May 1988.

Papadopoulos, G.M. "Monsoon: A General-purpose Dataflow Multiprocessor," Ultracomputer Research Laboratory, New York University, New York, NY, November 1988.

Traub, K.R. "Compilation as Partitioning: A New Approach to Compiling Non-strict Functional Languages," IBM T.J.Watson Research Center, Yorktown Heights, NY, July 1988; Aspenäs Workshop on the Implementation of Lazy Functional Languages, Göteborg, Sweden, September 1988.

References

[1] Z. Ariola and Arvind. P-TAC: A parallel intermediate language. Computation Structures Group Memo 295, MIT Laboratory for Computer Science, Cambridge, MA, March 1989. To appear in *Proceedings of the Conference on Functional Languages and Computer Architectures*, London, England, September 1989.

[2] Arvind and D. Culler. Managing resources in a parallel machine. Computation Structures Group Memo 257, MIT Laboratory for Computer Science, Cambridge, MA, March 1989. In *Proceedings of the Conference on Fifth Generation Computer Architecture*, Manchester, England, North-Holland, July 1985.

[3] P. Barth and R. Nikhil. Supporting state-sensitive computation in a dataflow system. Computation Structures Group Memo 294, MIT Laboratory for Computer Science, Cambridge, MA, March 1989.

[4] V. Grafe. Eps'88: Combining the best features of df and von Neumann computing. Technical Report Technical Report SAND 88-3182, Sandia National Laboratories, Albuquerque, NM, January 1989.

[5] S. Heller. *Efficient Lazy Data-structures on a Dataflow Machine.* PhD thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, January 1989.

[6] S. Jagannathan. *A Programming Language Supporting First-class Parallel Environments.* PhD thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, January 1989.

[7] V. Kathail. *Optimal Interpreters for λ-calculus Based Functional Languages.* PhD thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, September 1989 (expected).

[8] R. Nikhil. Can dataflow subsume von neumann computing. Computation Structures Group Memo 292, MIT Laboratory for Computer Science, Cambridge, MA, November 1988.

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

[10] G. Papadopoulos. The monsoon processing element architecture reference. Computation Structures Group Memo 283, MIT Laboratory for Computer Science, Cambridge, MA, March 1988.

[11] G. M. Papadopoulos. *Implementation of a General Purpose Dataflow Multiprocessor.* PhD thesis, MIT Department of Electrical Engineering and Computer Science, August 1988.

[12] K. Steele and R. Soley. Virtual memory on a dataflow computer. Computation Structures Group Memo 289, MIT Laboratory for Computer Science, Cambridge, MA, July 1988.

[13] K. Traub, S. Brobst, J. Hicks, G. Papadopoulos, A. Shaw, and J. Young. Monsoon software interface specifications. Computation Structures Group Memo 296, MIT Laboratory for Computer Science, Cambridge, MA, 1989. Also available as Motorola Cambridge Research Center Technical Report 1.

[14] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc.* In *Proceedings of the 16th. Annual ACM Symposium on Principles of Programming Languages*, 1989.

[15] J. Young and Arvind. Instruction set definition for an explicit token store machine. Computation Structures Group Memo 293, MIT Laboratory for Computer Science, Cambridge, MA, February 1989.

Contents