memo-288.

Computation Structures Group

Academic Staff

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

Research Staff

G.A. Boughton P.R. Fenstermacher

Graduate Students

P.S. Barth
S.A. Brobst
D.E. Culler
B. Guha Roy
S.A. Gupta
S.K. Heller
J.E. Hicks
R.A. Iannucci
A. Iyengar
S. Jagannathan
C.F. Joerg
V. Kathail
G.K. Maa
J.S. Onanian
G.M. Papadopoulos
S. Sharma
R.M. Soley
K.R. Traub
E.W. Waldin

Undergraduate Students

V. Chaudhary
D. Chiou
C. Colby
A. DeHon
C. Fabian
F. Lam
E. Rothfus
S. Sanghani
J. Santoro
C. Seow
A. Shaw
V. Singal
S. Zamani

Support Staff

S.M. Hardy N.F. Tarbet

Technical Staff

J.P. Costanza R.F. Tiberio

Visitors

F. Hutner (Siemens, West Germany)
A. Konagaya (NEC, Japan)
T.S. Mohan (Indian Institute of Science, Bangalore, India)
L. Snyder (University of Washington, Seattle, Washington)

Computation Structures Group

1 Introduction and Overview

The main focus of the group continues to be in exploring the dataflow approach as a means to achieve the goal of high-performance computers that are general-purpose and easily programmable. Our research centers on

- the development of Id, a high-level, implicitly parallel language with a powerful functional subset;
- an optimizing compiler to translate Id programs to dataflow graphs, a parallel machine language, and
- multiprocessor architectures to execute dataflow graphs efficiently.

This year, we produced a reference manual for a major new version of Id (version 88.0). We have made substantial progress on the compiler for this new version— a major subset of the language has been implemented, and it is already in regular use within the group. We expect the complete language to be implemented by late summer 1988.

Work continues on the development of the Monsoon dataflow processor and the interconnection network. The designs for several key components have been tested and we expect to have a single-processor prototype running by the end of summer 1988.

We have been studying various systems issues (involving the language, the compiler and the architecture) such as resource management, delayed evaluation, persistence and input-output. We have also studied the coding of various applications in Id and their behavior under parallel execution.

In support of the primary thrust of the group, we have also studied other interpreters and compilers for functional languages, other parallel languages, and hybrid von Neumann/dataflow architectures.

We began work on Project Dataflow, in which we hope to build a real dataflow computer system (over 2000 MIPS) over the next three years, based on Id and the Monsoon processor architecture. Extensive experimentation in the last few years has convinced us that

- The basic dataflow instruction-scheduling mechanism is capable of utilizing hundreds of processors effectively, and a processor architecture based on this mechanism is viable.
- Resource management is a major issue, both in dataflow and other models, and cannot be studied until we can run programs of realistic size. Existing software simulators/emulators are incapable of handling adequately large problems.

The planned work on Id (and its compiler) involves extensions to make it a complete programming language in which all the software of the machine (including systems programs) will be written. We expect to devote substantial energy to the development of very powerful, systematic optimizations in the compiler. Though we will initially build single-node and 16-node machines, the ultimate goal is to build a 256 node machine (a node includes processor and I-structure memory). We are developing the multistage interconnection network for

the machine concurrently. The processor and network hardware will be built on the VME boards, using Sun workstations as hosts for maintenance, bootstrapping and diagnostics.

Preparation for Project Dataflow occupied much of our time. We gave several presentations to DARPA visitors, and our proposal for federal funding, which was rewritten several times (in part because of changes and uncertainties at DARPA) was finally submitted in March. We expect to build the 16-node machine within a year and, with the help of an industrial partner, to build the 256-node machine over the next three years. In March, we organized an Industrial Partners' Meeting to outline our approach. It was attended by representatives of about 20 companies. Follow-up negotiations continue, and we hope to come to some agreements by the end of summer 1988.

In the past year we also invested extra effort in presenting our work to the the research community. In addition to a one-week summer course at MIT in July 1987 (about 30 students), professors Arvind and Nikhil taught week-long courses at the Indian Institute of Science, Bangalore, India (over 50 students) and at Los Alamos National Laboratories, New Mexico (about 25 students).

2 Personnel

In the past twelve months, we were fortunate to have four visitors from diverse backgrounds and technical perspectives. Akihiko Konagaya spent a full year with us studying advanced computer architectures and developing his understanding of logic languages. He returned in January to NEC Corporation, where CHI-II, the machine in the design of which he had a major role, had just been launched as a commercial product.

In September, Professor Larry Snyder of the University of Washington joined the group to spend his sabbatical year exploring to what extent theoretical models of parallel computation, such as the P-RAM model, reflect real machines. He is also working on the development of a new taxonomy for parallel architectures.

At mid-year, T.S. Mohan arrived under the auspices of the United Nations Office of Program Execution to spend a six-month study leave. He is a member of the staff of the Computer Centre of the Indian Institute of Science, Bangalore, India.

Most recently, we have been joined by Franz Hutner, an engineer at Siemens, Munich, who will spend four months working with the Monsoon design team under the leadership of Gregory Papadopoulos.

After more than forty years as student, teacher and mentor at MIT, Professor Jack Dennis departed at the end of the Spring term to do further research in dataflow computation at RIACS in Moffett Field, California.

3 Work on Programming Languages

3.1 Id

In March, we released the reference manual for a major new version of Id (Version 88.0) [12], which we refer to as Id 88. The functional subset of Id is now comparable to other modern functional languages in that it has a polymorphic type system, user-defined algebraic types, pattern-matching on user-defined types, abstract types list comprehensions, *etc.*

A unique feature of Id 88 is the *array comprehension* which, we believe, is the first viable solution to the "array problem" in functional languages. Previous functional array primitives have been difficult to implement with adequate efficiency and parallelism, for which reason they have usually been omitted. For example, one such set of primitives is

```
new_array n v          and          update a j w
```

The first function creates an array of size `n` containing `v` everywhere, and the second creates a new array that differs from array `a` only at index `j`, where it contains `w`. These primitives usually result in far too many unnecessary copies of an array. Another popular primitive is

```
make_array n f
```

which makes a new array of size `n` containing `f(j)` at each index `j`. But even this primitive can result in awkward and/or inefficient programs, as we have argued in [3]. For these reasons, we had advocated the inclusion of I-structures into the language, despite the fact that they destroyed referential transparency (I-structures still retain determinacy and parallelism).

This study of arrays and I-structures led us to the design of array comprehensions, which remain functional (and therefore referentially transparent) while addressing all the objections raised about previous array primitives. The general form is an expression:

```
{k_nD_arrays (l1,u1), ..., (lN,uN)
| [eIndex1] = eValue1 || generator1
| ...
| [eIndexN] = eValueN || generatorN }
```

This simultaneously creates k arrays of the same shape. Each array is n-dimensional, with the j'th dimension having lower and upper index bounds `lJ` and `uJ`, respectively. Each clause in the expression specifies a *region* of the arrays. A `generator` specifies a sequence of environments; for each environment, `eIndex` computes an index value, and `eValue` computes a k-tuple, specifying the contents of the slot in each of the k arrays, at that index. For example, here is the definition of an array containing Fibonacci numbers:

```
A = {array (0,N)
    | [j] = 1                  || j <- 0 to 1
    | [j] = A[j-2] + A[j-1]    || j <- 2 to N }
```

Array comprehensions are compiled into very efficient code using loops and I-structures internally. A small extension to the array comprehension syntax also handles *accumulators*, such as histograms and hash tables.

In the future, we believe that Id will evolve into a *layered* language. The innermost layer is a functional language that is competitive with other modern functional languages. Being referentially transparent, programs in this subset of the language are subject to powerful program transformations (optimizations) and formal reasoning. They are also determinate, and can be compiled to highly parallel code.

While this innermost, functional layer of Id is syntactically similar to other modern functional languages, there are some essential differences. Unlike ML [11] (and Scheme [14]), Id has *non-strict* semantics, which gives tremendous additional expressive power and parallelism. Unlike Miranda [16], Id does not espouse normal-order evaluation, which is one way of implementing non-strict semantics, and which can have tremendous overhead and can seriously inhibit parallelism.

The next layer adds primitives that may not be refererentially transparent, though they retain the parallelism and determinacy of functional languages (*e.g.,* I-structures, general accumulators). Another layer introduces explicit non-determinism through the use of a construct called a *manager*. Using this layer, we can code shared resource management problems (*e.g.,* databases). Finally, the outermost layer will include primitives for fine control of the machine, including imperatives for I/O, to set memory, registers, *etc.* It is expected that we will be able to code most application programs within the innermost (functional) layer, thus revealing the most opportunities for optimization and parallel execution.

3.2 Haskell, a new "standard" functional programming language

For some time now, many researchers in functional programming have felt the need for a common language to facilitate communication and interchange amongst themselves. There are three widely-known functional languages— Scheme, ML and Miranda. However, Scheme is unacceptable because of its strict semantics, lack of a type-system, and lack of equational syntax. ML is unacceptable because of its strict semantics, and Miranda because it is a commercial product and cannot be used for language research.

In September 1987, at the Functional Programming and Computer Architecture conference at Portland, Oregon, a committee of about 12 researchers from the U.S. and Europe was formed to design a new, common functional language. Professors Arvind and Nikhil are on the committee and have spent substantial time participating in its discussions, particularly on the topics of arrays, parallel pattern-matching and overloading. The new language is to be called Haskell, after Haskell Curry, the logician. Though Haskell was originally intended to be merely a codification of "well understood" concepts in functional languages, our discussions have revealed that many of these concepts are not so well understood after all. The result is that Haskell will be as much a new, experimental language as any other. The Haskell report is expected in July 1988. Whether Haskell is successful or not, the effort has already paid off handsomely in that it has significantly deepened our own understanding of several

issues, and has had an effect on the current design of Id. Until the research community has had some experience with Haskell, we will continue to use Id as our own research vehicle.

3.3 Other language-related work

3.3.1 Optimal Interpreters for the λ-calculus

Vinod Kathail is investigating *optimal interpreters* for the λ-calculus and functional languages based on the λ-calculus. An optimal interpreter

1. always reduces an expression to normal form if the expression has a normal form;
2. does so in a minimum number of reduction steps.

Thus, an optimal interpreter respects the operational semantics of the language, and, moreover, is in some sense the most efficient implementation.

If we restrict ourselves to the usual string representation of expressions, then it is impossible to design an optimal interpreter for the λ-calculus [4]. We have strengthened this result to show that it is also impossible to design an interpreter that approximates the behavior of an optimal interpreter. Specifically, we have shown that there exists no reasonable interpreter that reduces an expression to normal form in $k \times m$ number of steps where k is a fixed constant and m is the minimum number of steps needed to reduce the expression to normal form.

However, practical implementations of programming languages rarely use the string representation of expressions. Graph reduction has now become a standard technique for implementing lazy functional languages. It is possible to design an optimal interpreter using a graph representation of expression, and a formal specification of such an interpreter was given by Lévy [10].

Most interpreters for the λ-calculus are based on one of the three models—substitution model, environment model, and the third model, which we call suspended substitution or functions on environments model. The first two are well known. The third model is a new development and occurs in various forms in the literature, the most elegant of which is Curien's *strong CCL* [5]. We have argued that these models are inherently inadequate to serve as the basis for an optimal interpreter.

We have designed an optimal interpreter for the λ-calculus. The interpreter is based on a new technique for evaluating expressions, which uses features from all the three models. The graph representation used by the interpreter has a special type of node, called *conditional* node. Conditional nodes are used to postpone the copying of shared sub-expressions. We have implemented the interpreter, and verified its correctness and optimality for a number of test cases. The details of the interpreter, as well as proofs of its correctness and optimality, appear in Kathail's forthcoming thesis [7].

3.3.2 Symmetric Lisp

Suresh Jagannathan has continued his investigation of Symmetric Lisp, a parallel programming language in which naming environments are first-class objects. The representation of programs is identical to the representation of data: to specify a computation, one writes a data structure. This structure can be examined and used as a component of other structures, yet it has the semantics of a naming environment— it defines a scope and can also be used to affect the evaluation of other expressions. Non-strict evaluation is a fundamental part of the language: a program's components may be examined even as its other elements continue to evaluate.

A logically concurrent interpreter has been implemented on a TI Explorer and a formal operational semantics for the language has been defined. Some of the applications that have been written include implementations of Simula and Smalltalk-style inheritance, hash-tables, various graph algorithms, priority-queues, resource managers, mutually recursive stream programs, and a digital-logic simulator. Besides the ability to model a diverse range of conventional sequential and parallel program structures, the language also appears useful in expressing more novel program constructs; among the applications currently under investigation are real-time monitors and simulators. These monitors are formulated as programs structured as lattices of communicating processes; the internal state of any process may be freely examined by any other process in the lattice. We are also currently investigating the design of a parallel monolingual computer system using Symmetric Lisp as a base language.

The viability of any implementation of the language depends upon the ability of the compiler to minimize the need to perform run-time name lookup. A major focus of our research is to develop compile-time optimizations and heuristics to lessen the burden on the run-time system in resolving symbolic names into base language addresses. Symmetric Lisp's support for fine-grained, expression-level parallelism makes it well suited for implementation on a dataflow system. We, therefore, intend to concentrate our efforts in the coming year on defining and implementing the extensions needed to a dataflow instruction set (*e.g.*, the instruction set of the TTDA) to support the language efficiently.

3.3.3 Functional Databases

Michael Heytens, a graduate student in Professor Penfield's CAF group, and Professor Nikhil have been investigating the synthesis of databases and functional languages. The basic approach is to treat a database as a persistent environment binding names to arbitrary values. Queries are expressions evaluated in this evironment, and an update is a declarative specification of a new environment, based on the old. Our work has focused on the design of the update language, and is inspired by the I-structure operations of Id. We are also building a prototype implementation.

4 Work on Compilers

4.1 Progress on the Id Compiler

In Spring 1988 James Hicks extended the Id compiler to handle Id 88. The compiler now handles algebraic types, abstract types, list and array comprehensions and pattern matching in procedures and case expressions. The general algorithm for compiling pattern matching, developed by Wadler and described in [17], was extended for use in a parallel language. The pattern matching expressions are compiled so that there is no top-to-bottom or right-to-left bias in the strictness of the expression and there is no nondeterminacy in the choice of clause to execute. The only extension that has not been implemented is the compilation of top-level constant definitions. This will require a great deal of change in the way GITA invokes code blocks as well as in the Id compiler.

Based on Professor Nikhil's type-inference system for Id 86, Shail Aditya developed the type inference system for Id 87 and installed it with help from James Hicks. The system provides full Milner style type inferencing of complete programs. Further work remains on upgrading the type system to Id 88, incremental type inferencing, and supporting user-defined overloading of procedure names.

4.2 Compiling Sequential Code from a Non-strict Language

We have recently become interested in developing good sequential or partially sequential implementations of Id. Such implementations would make Id available (and interesting) to a wider user community, since there are now many MIMD machines available that are based on von Neumann processors. This would also be relevant to new hybrid von Neumann/dataflow architectures, such as the one developed by Iannucci. Id's non-strict semantics makes this compilation problem interesting because it may not be possible to order instructions totally at compile time; the correct order can vary dramatically with the input data.

Sequential implementations of non-strict languages have been achieved by other researchers. They invariably rely on *lazy evaluation*, in which a subexpression is not evaluated until known (at run-time) to contribute to the final answer. By scheduling each subexpression separately, a lazy interpreter automatically deals with the varying orderings required by non-strictness, but with great overhead. Existing compilers optimize this by attempting to emulate the lazy interpreter.

Kenneth Traub has just completed a doctoral disseration on an alternative compilation strategy that separates non-strictness from laziness, through the analysis of data dependence. We achieve a much cleaner separation between considerations of correct object code behavior and code-generation for specific architectures. Non-strict object code is described abstractly as a set of sequential threads, each internally ordered but whose relative order with respect to other threads is determined at run-time. Translation of a source program into sequential threads involves first determining the constraints on thread construction imposed by the language semantics, and then partitioning the original program into threads based on these

constraints. From there, the abstract threads may be converted into concrete object code for a particular target architecture, given the execution mechanisms it provides.

The method is developed in the context of both sequential implementations and parallel implementations where the object code is partially sequentialized. It is also shown how lazy code can be generated from the same framework.

5 Work on Architectures

5.1 Monsoon

The Monsoon processor has moved closer to realization on several fronts. Gregory Papadopoulos completed the low-level microcontrol architecture reference on which the various versions of the processor will be based. Jack Costanza and Ralph Tiberio have implemented a subset of the specification in discrete TTL and CMOS logic that is presently being debugged. This single-processor version is targeted as a system software development accelerator for TI Explorer Lisp Machines, delivering approximately six dataflow MIPS[1], any mix of which may be 64-bit floating point operations. This is about 1000 times faster than GITA on an Explorer. The processor employs a scan register design to aid in debugging. The entire processor was simulated at the gate level under the MENTOR environment. We expect this prototype to be operational by the end of summer 1988. In a parallel effort, we have been investigating implementation technologies for a VMEbus printed circuit board version of the processor and a VLSI chip set architecture to be used in the construction of a 256 processor machine. Franz Hutner is nearing completion of the network interface unit gate array (NIU) that will be part of the chip set and used on the VMEbus version. The NIU acts the interface between the Monsoon processor and the network, performing clock domain crossing, packet buffering and routing table management.

In the software area, the Id compiler has been modified to produce code that is suitable for execution on Monsoon. While Monsoon is compatible with the TTDA, it does not directly execute TTDA machine graphs. The most obvious departures are

- Instruction fanout: TTDA instructions have arbitrary fanout, while Monsoon instructions may have at most two destinations;
- I-structure descriptors: on Monsoon, tokens can only carry pointers, not entire descriptors, and there is no dedicated hardware support for bounds checking and lower bounds compensation

We have observed, empirically, that Monsoon executes approximately 20–30% more instructions than the TTDA. Optimal register assignment algorithms for Monsoon have been explored but are not yet implemented.

[1] MIPS = Million Instructions Per Second

5.2 Interconnection Network for Monsoon

G.A. Boughton, Christopher Joerg, Robert Lester, and John Santoro have continued development of the network for Monsoon. This network is packet switched and is being designed to support a bandwidth of 800 MBits/sec/port. The two primary components of the network are the Packet-switched Routing Chip (PaRC) and the data link.

PaRC is a CMOS gate array being designed by Chris Joerg that will form the basis of the Monsoon network. A packet is received via one of its 4 input ports, stored in an on-chip buffer, and sent out through one of its 4 output ports. Each of the input and output ports is 16 bits wide and has a maximum throughput of 800 Mbits per second. Packets are 192 bits long, and are made up of 12 16-bit words (1 word of header, 9 words of data, and 2 words of CRC). An input port has 4 buffers, each of which can hold one packet. When a packet is received, the input port checks the packet for errors (using a CRC code) as it writes the packet into a buffer. The input port also determines to which output port the packet should go, and tells that port that a packet is waiting to use it.

Each output port keeps track of which buffers contain packets that want to use it. When an output port finishes sending a packet, it knows which buffer to read from next. This allows it to start sending the next packet immediately. PaRC is designed so that each output port can read a packet from any buffer at any time (regardless of the state of the other 15 buffers). This eliminates any unnecessary blocking and greatly increases the utilization of the network.

PaRC also allows a processor to get a fast acknowledgment that its message has been received. The mechanism for this is able to provide the acknowledgment without further burdening the network.

The PaRC chip will have 33,000 used gates and will operate at 50MHz. It will also have a low latency (as low as 100ns in light traffic), while making effective use of its bandwidth (95% utilization in heavy traffic). PaRC will be fabricated in LSI Logic's 1.5 micron compacted array series, and is being designed on a SUN workstation using CAD tools from LSI Logic. The design and testing of PaRC is nearly complete and the design will soon be handed over to LSI Logic for fabrication.

PaRC chips will be used to construct both 4 input 4 output network boards and 16 input 16 output network boards. The 4×4 board will contain a single PaRC and the 16×16 board will contain 8 PaRC's. A 256 input 256 output network can be constructed using 32 of the 16×16 boards.

While the output of a PaRC can be directly connected to the input of another PaRC on the same board, data links between boards will be implemented using separate data link transmitters and receivers. The data links between boards will have 4 bit wide data paths. The 4 bit wide data links have several advantages over 16 bit wide data links. They require a much smaller number of connector pins on each board and allow the use of more reliable connectors. They also require fewer wires and allow the use of cables with a higher noise immunity.

The data link transmitter is an ECL gate array that will multiplex a 16 bit wide PaRC output port into a 4 bit wide data link. The transmitter will differentially drive each of the

4 parallel pairs of the data link. The transmitter will use a balanced 4 into 6 code to encode the data sent on each of the parallel pairs. Thus 300 MBit/sec signals will be sent on each parallel pair.

The data link receiver is an ECL gate array that will demultiplex and decode the data from a data link and present it to a PaRC input port.

Robert Lester has completed a preliminary design of the data link transmitter and the data link receiver.

John Santoro and Choong Seow have completed the design and construction of a data link cable test circuit. This circuit uses GaAs circuitry to generate a 400 MBit/sec test pattern. It uses an ECL differential driver to send the pattern down the cable under test. It uses an ECL differential receiver and GaAs circuitry to check the received signal. This test setup is being used in conjunction with an electrostatic discharge noise generator to evaluate the noise immunity of proposed data link cables.

5.3 Hybrid von Neumann/Dataflow Architectures

Robert Iannucci has been investigating the space of architectures between pure von Neumann and pure dataflow machines. The overall goal has been to discover the critical hardware structures necessary for any scalable, general-purpose parallel processor to tolerate latency and synchronization costs effectively. The main conclusion is that any such machine must execute a *parallel machine language* (PML), having the following three characteristics:

- The execution time for any given instruction must be independent of latency. Traditional latency-sensitive operations, *e.g.,* LOADs from memory, must be re-phrased as split transactions which separately initiate an operation and later explicitly synchronize on the availability of the result.

- Each explicit synchronization event must be named. This implies efficient means for creating and re-using names as well as an efficient mechanism for enforcing synchronizing behavior based on the names. Names must be drawn from a large name space, and it must be possible to manipulate them as first-class hardware data types.

- Means for expressing both implicit and explicit synchronization must be provided. Implicit synchronization (*i.e.,* program counter based) provides the means for passing state between instructions within an unbroken thread. Explicit synchronization is necessary at the programming level in the exploitation of parallelism and at the machine level in the masking of latency.

In that neither von Neumann nor dataflow machines exhibit all three characteristics, a new hybrid architecture was synthesized and analyzed. It has been demonstrated through emulation experiments and analysis that this architecture, based on the principles of parallel machine language, has the ability to exploit the same classes of parallelism as a dataflow machine. Consequently, the hybrid architecture can control communication latency cost

through the exploitation of parallelism. Moreover, the hybrid architecture can execute sequential threads with the same efficiency as a von Neumann machine.

From the standpoint of pure von Neumann architectures, the hybrid is evolutionary in the addition of a synchronizing local memory, split transaction memory operations, and a large synchronization name space. Synchronizing local memories and register sets is not new. The most noteworthy previous machine in this regard is the HEP [6, 8]. The scheme proposed in the present work is more general than that of the HEP, however. Similarly, split transactions in and of themselves are not new, but the hybrid architecture shows the importance of inexpensive context switching as the primary means for making the most of split transactions.

The biggest departure from the traditional von Neumann architectural view is the introduction of large name spaces for synchronization purposes in the hybrid. In particular, the number of low-level synchronization names is limited only by the size of local memory. Further, the number of processes is limited only by the number of meaningful continuations. In contrast, the HEP architecture allows only 64 processes per processor to be named simultaneously. From a hardware point of view, 64 processes is a sizable number. From the compiler's point of view, however, the number is far too small and implies that processes are a precious resource to be carefully managed. In the hybrid, this artificial restriction is lifted, resulting in more generality.

From the standpoint of pure dataflow architectures, the hybrid is evolutionary in that it adds the means for the compiler to exercise some explicit control over the pipeline. Because a thread holds the pipeline until it executes a potentially suspensive instruction, the entire machine state can be brought to bear on the problem of efficiently communicating information between instructions. This class of compiler-directed pipeline control is absent in both the TTDA and in Monsoon [13]. The hybrid further takes the stance that synchronization should be explicit, as should forking of parallel activity. This simplification of the instruction set demonstrably does not drive the instruction count up in many cases because much of the forking and its attendant synchronization is superfluous. Even so, in the limiting case, the hybrid machine can still emulate instruction level dataflow with an instruction count expansion factor of no more than two. This leads to the observation that *explicit* synchronization instructions, used when necessary, may in some sense be cheaper than paying the full cost of synchronization at each instruction. This is, perhaps, the equivalent of the RISC argument applied to multiprocessing.

In [2], the question of the possibility of "modifying" a von Neumann processor to make it a suitable building block for a parallel machine was raised. It was believed that the salient characteristics of a dataflow machine which made it a suitable building block were split-phase memory operations and the ability to context-switch inexpensively. Given the addition of mechanisms like these, there was some lingering doubt as to what kind of synchronization efficiencies could be achieved and how much of the von Neumann architecture would be left. As presented in this work, engineering arguments regarding efficient implementation of PML's and the persistence of program counter based sequencing in the hybrid model have dispelled much of the doubt.

6 Work on Systems Issues

6.1 The price of dataflow parallelism

David Culler has made substantial progress in assessing the costs and benefits of the dataflow approach. This work has focused on three areas:

- Quantifying potential fine-grained parallelism in common programs;
- Assessing the overhead in terms of the number of instructions executed for fine-grained parallelism, and
- Controlling the resource requirements of dataflow programs.

To quantify parallelism in programs, we have developed the concept of an ideal parallelism profile, from which more common metrics, such as speed-up and utilization, can be estimated. Numerous scientific programs for which little parallelism is uncovered by traditional methods to accelerate loops [9] show ample parallelism under dataflow execution with I-structures. Our measurements show the cost of dataflow execution to be roughly a factor of two over sequential execution for programs representative of scientific applications. This is encouraging, since parallel execution will always involve some extra work due to distribution and synchronization. Preliminary measurements of "parallelized" sequential programs show a significant cost increase, making the dataflow approach very competitive when a large amount of parallelism is exploited. On the other hand, our measurements show that naive dataflow execution can incur inordinate resource requirements, in both token storage and I-structure storage. Judicious use of loop bounding has been very effective in controlling the token storage requirements of programs and imposes little performance penalty. It also provides the basis for extremely efficient I-structure storage management in scientific programs. A new bound-loop schema has been developed and implemented in the Id compiler.

6.2 Delayed evaluation

Steven Heller has been working on programming in Id using explicit annotations for laziness. Such annotations permit the use of potentially infinite data structures (such as streams). This is a compromise between the lazy-evaluation approach, which allows infinite structures but incurs much overhead even when laziness is not required, and the eager approach, which cannot handle infinite structures naturally but can be implemented efficiently.

This work involves language design, compiler development, and architectural enhancement. The architectural enhancements necessary to support this approach are well understood. Two preliminary implementations were completed last summer to demonstrate feasability at the compiler level.

Language work is now proceeding. Compiler support will be completed this summer, and the current Monsoon design includes support for this work.

6.3 Resource managers

Paul Barth has been working on *managers*, a paradigm for non-deterministic access to shared resources. A design was developed that eliminated several memory management and efficiency problems of earlier designs. The design was successfully implemented and tested on the GITA simulator. Several examples of managers were written, including a priority scheduler and a load-balancing scheduler. Managers were compared to traditional synchronization techniques (locks); the comparison showed traditional techniques appropriate for managing local resources that require serial access, while managers are preferable for global resources that may be accessed concurrently.

6.4 Stream input-output

Richard Soley finished the design and implementation of his scheme for automatic ordering of stream-based input/output requests, based on the order implicit in the input program. This addition to the Id language, described in [15], is available as a software upgrade to allow properly serialized input/output resource usage in Id programs.

During the analysis of this scheme, it was found that programs were "over-serialized"; in particular, all conditionals in all procedures became part of the critical path of the program. A scheme to avoid this problem by automatically discerning which procedures never needed to sychronize on I/O resource usage was added late in the year.

6.5 General persistence

Bhaskar Guha Roy has been working on extending the Tagged-token Dataflow Architecture to support a highly parallel disk I/O system that is well integrated into the programming model. The dataflow approach appears well suited for this by virtue of its tolerance of latency (important due to the slowness of I/O devices), and the fact that the model can naturally express the overlap of I/O activity with normal computation.

A first version of a file system on GITA has been implemented. The operations on disk storage units are similar to I-structure operations. Id has been extended for file operations and we are experimenting with various compilation strategies for file system operations. The file system has also been used to implement a persistent object system. Objects of any data type in Id (except closures) can be made persistent. Id has also been extended for persistence operations.

In the coming year, the plan is to examine issues related to locality issues and their impact on distribution of files across disks, and incremental I/O. We hope to evaluate the performance by comparing it with I/O system performance on other parallel machines.

6.6 Controlling speculative parallelism

Richard Soley has been investigating effective methods for controlling the explosive growth of speculative programs within the dataflow execution paradigm. Speculation, common in

symbolic processing tasks, is a potentially rich source of parallelism which was ignored previously, due to the expected high demand of speculative programs on memory and processor resources. In fact, such demands can quickly swamp an eager dataflow execution mechanism, causing deadlock. We have found some interventions for this problem, and have begun to implant them within standard symbolic processing codes to measure their success. Early results, based on controlled speculation in game tree searches, look promising.

7 The Id programming environment: Id World

P.R. Fenstermacher produced an alpha version of Id World for the Sun Workstation. In additon to running on a larger installed base of machines, the Sun version also offers better performance on large floating-point intensive computations. With Ken Traub, he worked on the port of Id World from Symbolics Genera 6 to Genera 7. Also with Traub, he worked on the port of Id World from Release 2.1 of the TI Explorer software to Release 3.2. This allows Id World to run on the new higher performance Explorer II processor as well as the Explorer I. Temporal garbage collection allows longer running times before the need to reinitialize the software. And, all I/O is faster.

Design work on the support software for the Monsoon Processor has begun. The very large amount of information necessary to specify the machine state of the 256 processor configuration has led to consideration of a graphic interface. Further, since the forest of support processors must present a coherent view to the Monsoon processors, network models are being considered. Because the X Window Sytem has both features, it is the leading candidate for a platform upon which the support software will be built.

Stephen Brobst has completed Version 1.0 of the C implementation of GITA With the assistance of Shail Gupta, he has successfully ported the C implementation of GITA to VAX and Sun Workstation environments. A beta version of the software has been released to Los Alamos National Laboratories for implementation on the Cray 2.

8 Applications

8.1 Simulated Annealing

Stephen Brobst is currently investigating the implementation of simulated annealing algorithms for the Tagged-Token Dataflow Architecture. Simulated annealing is an approach for solving certain integer optimization problems. We are working on a program to solve the quadratic assignment problem (QAP) that arises in the optimization of spatial layouts for buildings, factories, electronic circuits, and the location of firms and public facilities across regions. Since the QAP problem is NP-hard, and has been shown to lack efficient direct solutions, heuristic algorithms such as simulated annealing have been a focus for this application domain. We are working on an program in Id 88 for solving the QAP in the context of building layouts. We will also be looking into the resource requirements for execution of this algorithm on the TTDA and making comparisons to execution on a von Neumann machine.

8.2 Sparse-Matrix Techniques

During the Spring semester Ken Steele wrote a Sparse matrix LU decomposition program in Id 88, using the new array and list comprehensions. The goal is to investigate the applicability of Id for writing circuit simulators which currently use sparse matrix solvers. A study of the parallelism and alternative methods is now under way.

8.3 Signal Processing

In summer and fall 1987 James Hicks developed *D-PICT*, a diagrammatic language for signal processing, while completing his VI-A assignment at Lincoln Laboratory. *D-PICT* provides constructs for creating and combining ***signals*** in a notation similar to that used for describing signal processing algorithms. Because the execution model of *D-PICT* is dataflow based, a *D-PICT* program is an inherently parallel program. The *D-PICT* system consists of a graphical network editor, a compiler, based on Version II of the Id Compiler, and a module that converts the dataflow graphs generated by the compiler into sequential Lisp code. By writing signal processing programs in a single-assignment language, we are able to target compilation for both conventional sequential machines and parallel processors. In addition we hope to be able to compile code for special purpose signal processing hardware.

Janice Onanian developed a strategy for compiling signal processing dataflow graphs for more conventional medium-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 the network resources is optimized. We developed a high level language and intermediate graphical representation for signal processing applications. Special graphical constructs, called Data Routing Operators, support automatic partitioning of the application into tasks for static allocation on loosely-coupled, distributed multiprocessor architectures. The language is functional and consists of special iteration constructs which translate directly into graphical constructs upon which optimization is performed. Real-time throughput requirements of the application contribute to the complexity of the partition process. The language is used as a front end design tool; once the optimal partition is determined, coding proceeds in a conventional language. Implementation of the interpreter for the language is targeted for summer 1988.

8.4 DNA Sequence Algorithms

Arun Iyengar studied parallel sequence comparison algorithms used in molecular biology for comparing DNA and protein sequences. DNA sequence databases are expanding very rapidly, and parallel processing will become increasingly important as more data becomes available. Many of the computationally intensive problems in biological sequence comparison were found to contain a large amount of inherent parallelism. On the other hand, some of them require updates to aggregate data structure that require an excessive amount of copying in functional languages, which can drastically increase the running time of certain algorithms. Nonfunctional features of the language such as I-structures and accumulators do not always eliminate this overhead.

8.5 Computational Fluid Dynamics

P group, and has expressed a keen interest in exploring the suitability of

Id for CFD applications. One of his graduate students, Sandy Landsberg, is spending the summer of 1988 in our group, coding one of the CFD kernels in Id.

9 Project Dataflow

We believe that our dataflow research has reached a sufficient level of maturity to warrant the construction of a real dataflow machine.

First, our experiments to date have shown:

- Id is an effective tool for expressing algorithms from a diverse range of applications.
- The parallelism in dataflow object code obtained from Id programs is adequate to keep hundreds of processors busy, even in existing, conventional algorithms.
- The dataflow instruction-scheduling mechanism is capable of keeping processor pipelines utilized effectively, tolerating large memory and inter-processor latencies easily.

Still, since simulations always have numerous assumptions built in, the only convincing demonstration of these capabilities will be a real dataflow machine.

Second, resource management is still an open problem (both in dataflow and other models). Simulators are inherently slow, limiting the size of applications that can be run on them. Thus, we are not able to study the true resource management issues of large applications. This, again, argues for a real machine.

We have spent almost 9 months in preparing a proposal for Project Dataflow, in which we describe plans for a 256-processor machine based on the Monsoon architecture, programmed entirely in Id [1]. Our original plan was to do it entirely in-house, sub-contracting the manufacture of various components such as custom logic, pc boards, cables, *etc.* Following extensive discussions with DARPA, the current plan is to build it in three stages:

1. A single-processor machine that can be plugged into a workstation. Expected by 12/88.
2. A 16-processor machine. This will also be plugged into a workstation, though it will contain its own 16×16 interconnection network. Expected by 12/89.
3. A 256-processor machine. Expected by 2/91.

We will enter into a collaboration with an Industrial Partner who will do much of the detailed design, fabrication and packaging of the 256-PE machine, and who may also assist in the production 16-PE machine.

In Stage 1, we are first building a 6 MIPS, wire-wrapped, microprogrammable prototype with limited memory and no caches. The design is complete, and the fabrication of the wire-wrap boards is under way. We hope to have this running by the end of summer 1988. By the end of the year, we hope to have the pc-board version with caches and more memory. We plan to use a VME card, plugged into a Sun workstation. These boards will be distributed to external users from 2/89.

Stage 2 will result in VME cards each containing four PEs, and a VME card containing a 16 × 16 switch by 10/89. Each PE will run at 12-16 MIPS. As described earlier, work on the 4 × 4 routing chip has already progressed significantly. The 16-PE machine will be distributed externally from 2/90.

Stage 3 will be built by the Industrial Partner, by 2/91. We hope to achieve over 2000 MIPS peak, 1000 MIPS sustained.

While the hardware will be built in partnership with an Industrial Partner, the software for the machine will be our responsibility entirely. The work involves developing Id into a *complete* programming language in which all the machine's software, including systems software, will be written; further work on optimizations in the compiler; writing code-generators for Monsoon (given the fidelity of Monsoon to the TTDA, we anticipate no serious problem here); and, writing resource managers, primarily for activation-frame allocation and distribution, and heap allocation, deallocation and distribution.

We spent many months in tuning the proposal and hosting several visits from people at DARPA. Begun in June 1987, the proposal was finally submitted in March 1988. In March, we also held an Industrial Partners meeting in which we presented our plans to invited representatives from about 20 companies. It is likely that we will set up collaborations with more than one company, since many of them were interested also in implementing Id on some of their own multiprocessors.

Publications

Arvind, Brobst, S.A., and Maa, G.K. Evaluating the MIT Tagged-Token Dataflow Architecture. Computation Structures Group Memo 281, MIT Laboratory for Computer Science, Cambridge, MA, December 1987, the fourth in a sequence comprised of CSG Memos 278-281 (Not available.)

Arvind, and Culler, D.E. Resource Requirements of Dataflow Programs. In *Proceedings of the 15th IEEE/ACM Annual Symposium on Computer Architecture,* Honolulu, Hawaii, May 1988. Also Computation Structures Group Memo 280, MIT Laboratory for Computer Science, Cambridge, MA, the third in a sequence comprised of CSG Memos 278-81.

Arvind, Culler, D.E., and Maa, G.K. Assessing the Benefits of Fine-grained Parallelism in Dataflow Programs. To appear in *Supercomputing 88.* Also Computation Structures Group Memo 279, MIT Laboratory for Computer Science, Cambridge, MA, the second in a sequence comprised of CSG Memos 278-281.

Arvind, and Ekanadham, K. Future Scientific Programming on Parallel Machines. To appear in *Languages, Compilers, and Environments*, a special issue of *Journal of Parallel and Distributed Computing.* Also Computation Structures Group Memo 272, MIT Laboratory for Computer Science, Cambridge, MA, March 1987, revised February 1988.

Arvind, Dertouzos, M.L., Nikhil, R.S. and Papadopoulos, G.M. Project Dataflow: A Parallel Computing System Based on the Monsoon Architecture and the Id Programming Language (Extracts from March 1988 DARPA Proposal). Computation Structures Group Memo 285, MIT Laboratory for Computer Science, Cambridge, MA, March, 1988.

Arvind, Heller, S. and Nikhil, R.S. Programming Generality and Parallel Computers. To appear in *Fourth International Symposium on Biological and Artificial Intelligence Systems, Trento, Italy, September 18-22, 1988.* Also Computation Structures Group Memo 287, MIT Laboratory for Computer Science, Cambridge, MA, May 1988.

Barth, P.S. Managing Nondeterministic Access to Shared Resources in a Dataflow System. Unpublished, MIT Laboratory for Computer Science, Cambridge, MA, June 1988.

Brobst, S.A. Simulation Studies of Distributed Data Access in a Multicomputer Environment. ACS Memo 88-5, Advanced Commercial Systems Department, Hewlett-Packard Laboratories, Palo Alto, CA, January 1988.

Brobst, S.A. Organization of an Instruction Scheduling and Token Storage Unit in a Tagged-Token Dataflow Machine. In *Proceedings of the 16th International Conference on Parallel Processing,* St. Charles, IL, August 17-21, 1987.

Dennis, J.B. Data Flow Computer Architecture: Final Report. MIT/LCS/TR-385, MIT Laboratory for Computer Science, Cambridge, MA, November 1987.

Ekanadham, K., Arvind, and Culler, D.E. The Price of Parallelism. To appear in the *Proceedings of CONPAR 88.* Also Computation Structures Group Memo 278, MIT Laboratory for Computer Science, Cambridge, MA, December 1987, the first in a sequence comprised of CSG Memos 278-281.

Ekanadham, K., Arvind. SIMPLE: Part I–An Exercise in Future Scientific Programming. In *Proceedings of the International Conference on Supercomputing,* Athens, Greece, July 1987. Also Computation Structures Group Memo 273, MIT Laboratory for Computer Science, Cambridge, MA, and IBM T.J. Watson Research Center Report 12686.

Heytens, M.L. and Nikhil, R.S. GESTALT: An Expressive Database Programming System. MIT CAF project, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, December 1987. Submitted to *SIGMOD Record.*

Hicks, J.E., Jr. A High-level Signal Processing Programming Language. MIT/LCS/TR-414, MIT Laboratory for Computer Science, Cambridge, MA, June 1988.

Iannucci, R.A. A Dataflow/von Neumann Hybrid Architecture. MIT/LCS/TR-418, MIT Laboratory for Computer Science, Cambridge, MA, June 1988. (Ph.D. dissertation).

Iannucci, R.A. Toward a Dataflow/von Neumann Hybrid Architecture. In *Proceedings of 15th Annual Symposium on Computer Architecture*, IEEE/ACM, Honolulu, Hawaii, May-June, 1988.

Maa, G.K. Code-mapping Policies for the MIT Tagged-Token Dataflow Architecture. MIT/ LCS/ TR-425, MIT Laboratory for Computer Science, Cambridge, MA, May 1988.

Nikhil, R.S. The Semantics of Update in a Functional Database Programming Language. In *Proceedings of the Workshop on Database Programming Languages,* Roscoff, France, September 1987.

Nikhil, R.S. Id (Version 88.0) Reference Manual. Computation Structures Group Memo 284, MIT Laboratory for Computer Science, Cambridge, MA, March, 1988.

Papadopoulos, G.M. The Monsoon Processing Element Architecture Reference. Computation Structures Group Memo 283, MIT Laboratory for Computer Science, Cambridge, MA 02139, MIT Laboratory for Computer Science, Cambridge, MA, March 1988.

Soley, R.M. Implicit Serialization in Dataflow Programs. Computation Structures Group Memo 277, MIT Laboratory for Computer Science, Cambridge, MA, December 1987.

Traub, K.R. Sequential Implementation of Lenient Programming Languages. MIT/ LCS/ TR-417, MIT Laboratory for Computer Science, Cambridge, MA, June 1988. (Ph.D. Thesis.)

Theses Completed

Chaudhary, V. Implementing Parallel Functional Programming Languages Using Graph Reduction on the Dataflow Machine. S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1988.

Hicks, J.E., Jr. A High-level Signal Processing Programming Language. S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, February 1988.

Iannucci, R.A. A Dataflow/von Neumann Hybrid Architecture. Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1988.

Lester, R.M. Design of a High-speed Data Link for the Monsoon Dataflow Processor. S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1988.

Maa, G.K. Code-mapping Policies for the Tagged-token Dataflow Architecture. S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, February 1988.

Rothfuss, E. Query Optimization at the Expression Level. S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1988.

Traub, K.R. Sequential Implementation of Lenient Programming Languages. Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1988.

Theses in Progress

Culler, D.E. Effective Dataflow Execution of Scientific Applications. Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA. Expected December 1988.

Heller, S.K. Efficient Lazy Structures on a Dataflow Machine. Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA. Expected December 1988.

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

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

Kathail, V. Optimal Evaluators for Lambda-calculus Based Functional Languages. Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA. Expected August 1988.

Onanian, J.S. A Novel Language for Signal Processing Applications. S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA. Expected August 1988.

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

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, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA. Expected December 1988.

Talks

Arvind. The MIT Dataflow Project. DARPA Principal Investigators Meeting, Washington, D.C., September 15, 1988.

Arvind. MIT Tagged-Token Dataflow Project. University of California at Berkeley, Berkeley, CA, November 11, 1987.

Arvind. Project Dataflow. University of Utah, Salt Lake City, UT, January 16, 1988.

Arvind. Id: A Declarative Language for Future Scientific Programming. University of Utah, Salt Lake City, UT, January 16, 1988.

Arvind. Making it Fun to Program on Parallel Computers. Apple, Cupertino, CA, March 18, 1988.

Arvind. Making it Fun to Program on Parallel Computers. Sun Microsystems , Sunnyvale, CA, May 14, 1988.

Arvind. Making it Fun to Program on Parallel Computers. Computational Fluid Dynamics Meeting, MIT, Cambridge, MA, April 19, 1988.

Arvind. Future Scientific Programming. 1988 International Conference on Supercomputing, Boston, MA, May 19, 1988.

Arvind. Present Status of Dataflow Computers. National Conference on Knowledge-based Computer Systems, Bangalore, India, June 10, 1988.

Arvind. Making it Fun to Program on Parallel Computers. Computer Centre, Indian Institute of Technology, Madras, June 5, 1988.

Arvind. Making Highly Programmable Parallel Computers. Computer Science Research and Operations Conference, Bilkent University, Ankara, Turkey, June 22-23, 1988.

Brobst,S.A. Evaluation of the MIT Tagged-Token Dataflow Architecture. Hewlett-Packard Laboratories, Palo Alto, CA, January 7, 1988.

Brobst, S.A. Evaluation of the MIT Tagged-Token Dataflow Architecture. Sandia National Laboratories, Albuquerque, NM, December 14, 1987.

Brobst, S.A. Evaluation of the MIT Tagged-Token Dataflow Architecture. Manchester University, Manchester, England, November 18, 1987.

Brobst, S.A. Parallel Processing Architectures for Scientific Computing. Waltham, Massachusetts, November 4, 1987.

Brobst, S.A. Token Volatility and Resource Management in a Dynamic Dataflow Machine. Mitsubishi Electric Corporation, Kamakura City, Japan, October 9, 1987.

Brobst, S.A. Token Volatility and Resource Management in a Dynamic Dataflow Machine. MITI Electrotechnical Laboratory, Tsukuba Science City, Japan, October 6, 1987.

Brobst, S.A. Token Volatility and Resource Management in a Dynamic Dataflow Machine. NEC Electronics Inc., Kawasaki City, Japan, October 5, 1987.

Brobst, S.A. Organization of an Instruction Scheduling and Token Storage Unit in a Tagged-Token Dataflow Machine. 16th International Conference on Parallel Processing, St. Charles, Illinois, August 18, 1987.

Brobst, S.A. Program Transformations for Vectorizing Compilers. Systems Software Design Laboratory, Hewlett-Packard Company, Cupertino, California, August 14, 1987.

Brobst, S.A. Parallel Processing: A Survey of the Impending Revolution. Systems Architecture Laboratory, Hewlett-Packard Company, Cupertino, California, August 11-12, 1987.

Culler, D.E. Controlling Parallelism in Dataflow Programs. Workshop on Packaging Parallelism, Sponsored by the Supercomputing Research Center, April 25-27, 1988, Leesburg, VA.

Culler, D.E. Resource Requirements of Dataflow Computers. 15th Annual Symposium on Computer Architecture, IEEE/ACM, Honolulu, Hawaii, May 31, 1988.

Heytens, M. GESTALT: An Expressive Database Programming System. Panel on Multi-database systems, ACM International Conference on Management of Data (SIGMOD), June 2, 1988.

Iannucci, R.A. Towards a Dataflow/von Neumann Hybrid Architecture. 15th Annual Symposium on Computer Architecture, IEEE/ACM, Honolulu, Hawaii, May 31, 1988.

Iannucci, R.A. Towards a Dataflow/von Neumann Hybrid Architecture. IBM T.J. Watson Research Center, Hawthorne, NY, April 15, 1988.

Nikhil, R.S. Lectures in 6.83s, MIT 1-week summer course on Dataflow Architectures and Languages, August 1987.

Nikhil, R.S. Lectures in a 1-week course on Dataflow Architectures and Languages, Indian Institute of Science, Bangalore, India, August 1987.

Nikhil, R.S. Parallel Supercomputers. Frontier Technologies, Hyderabad, India, August 25, 1987.

Nikhil, R.S. The Semantics of Update in a Functional Database Programming Language. Workshop on Database Programming Languages, Roscoff, France, September 1987.

Nikhil, R.S. Lectures in a 1-week course on Dataflow Architectures and Languages. Los Alamos National Laboratories, October 1987.

Nikhil, R.S., Dataflow for Expressive, High-Performance Database Systems. MIT-Siemens Workshop on Highly Parallel Processing, Munich, W. Germany, November, 1987.

Nikhil, R.S. Id: A Declarative, General Purpose Language with Fine-Grained Parallelism. Unisys 1988 Parallel Processing Workshop, June 16, 1988.

Papadopoulos, G.M. The Monsoon Dataflow Processor. MIT-Siemens Workshop on Highly Parallel Processing, Munich, W. Germany, November, 1987.

Papadopoulos, G.M. The Future of the Engineering Workstation. MIT-IBM Workshop on Engineering Workstations, Austin, Texas, January, 1988.

Papadopoulos, G.M. Instrumentation for the Monsoon Dataflow Multiprocessor. Instrumention for Future Parallel Processors Workshop, Santa Fe, New Mexico, May, 1988.

Soley, R.M. Lisp Carries its Own Weight. Third Artificial Intelligence Applications Conference, IEEE, August 1987, Kissimmee, FL.

References

[1] Arvind, M. L. Dertouzos, R. S. Nikhil, and G. M. Papadopoulos. Project Dataflow: A Parallel Computing System based on the Monsoon Architecture and the Id Programming Language (Extracts from March 1988 DARPA Proposal). Technical Report CSG Memo 285, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, March 25 1988.

[2] Arvind and R. Iannucci. Two Fundamental Issues in Multiprocessing. In *Proceedings of DFVLR - Conference 1987 on Parallel Processing in Science and Engineering.* Bonn-Bad Godesberg, June 1987.

[3] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. In *Proceedings of the Workshop on Graph Reduction, Santa Fe, New Mexico, USA, (Springer-Verlag LNCS 279).*, pages 336–369, September/October 1986. (also Computation Structures Group Memo 269, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139).

[4] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1984.

[5] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Pitman, 1986.

[6] H. F. Jordan. Performance Measurement on HEP - A Pipelined MIMD Computer. In *Proceedings of the 10th Annual International Symposium On Computer Architecture*, pages 207–212, Stockholm, Sweden, June 1983. IEEE Computer Society.

[7] V. Kathail. *Optimal Interpreters for λ-calculus Based Functional Languages.* PhD thesis, MIT Department of Electrical Engineering and Computer Science, 1988. Expected August 1988.

[8] J. S. Kowalik. *Parallel MIMD Computation: HEP Supercomputer and Its Applications.* Scientific Computation Series. The MIT Press, 1985.

[9] D. J. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. In *Proc. 8th ACM Symp. on Principles of Programming Languages*, pages 207–218, January 1981.

[10] J.-J. Lévy. Optimal Reductions in the Lambda-calculus. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 160–191, London, 1980. Academic Press.

[11] R. Milner. A Proposal for Standard ML. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 184–197, August 1984.

[12] S. Nikhil, Rishiyur. Id (Version 88.0) Reference Manual. Technical Report CSG Memo 284, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, March 25 1988.

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

[14] J. Rees and W. Clinger. Revised^3 Report on the Algorithmic Language Scheme. Technical report, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, MA, 1986.

[15] R. Soley. Implicit Serialization in Dataflow Programs. CSG Memo 277, MIT Laboratory for Computer Science, CambMA, December 1987.

[16] D. A. Turner. Miranda: A Non-Strict Functional Language with Polymorphic Types. In *Proc. Functional Programming Languages and Computer Architecture, Nancy, France (Springer-Verlag LNCS 201)*, pages 1–16, September 1985.

[17] P. Wadler. Efficient Compilation of Pattern-Matching, 1987. (in *The Implementation of Functional Languages*, Simon L. Peyton Jones, Prentice-Hall, Englewood Cliffs, NJ).

Contents