Computation Structures Group Progress Report 1986-87

Computation Structures Group Memo 274

June 1987

COMPUTATION STRUCTURES GROUP

Academic Staff

Arvind, *Group Leader*

J.B. Dennis

R.S. Nikhil

Research Staff

G.A. Boughton

Graduate Students

P.S. Barth
S.A. Brobst
A.A. Chien
T-A. Chu
D.E. Culler
G-R. Gao
B. Guha Roy
S.K. Heller
R.A. Iannucci
S. Jagannathan
V.K. Kathail

G.K. Maa
D.R. Morais
G.M. Papadopoulos
K.K. Pingali
S.A. Plotkin
R.M. Soley
I. Taylor
K.R. Traub
E.W. Waldin
S. Younis

Undergraduate Students

H. Chan
V. Chaudary
J. Cheng
S. Desai
C. Fabian
J. Hom
J. Hicks
C. Joerg
H. Krishnan
F. Lam

T. Leung
S. Malinak
S. Manadhar
J. Nieh
T. Olkin
M. Penniston
S. Sanghani
C. Seow
S. Zamani

Support Staff

S.M. Hardy
N.F. Tarbet

Technical Staff

J.P. Costanza
R.F. Tiberio

Visitors

M. D. Atkins (IBM)
A. Konagaya (NEC, Japan)
J. Lewis (Boeing)
M. Mack (IBM)
H. Nohmi (NEC, Japan)
S. Truve (Chalmers University of Technology, Sweden)

Computation Structures Group

1. INTRODUCTION AND OVERVIEW

The Computation Structures Group (CSG) made significant progress from July 1986 through June 1987. The primary thrust continues to be towards general-purpose parallel machines, focusing on functional and declarative languages, and on the Tagged-Token Dataflow Architecture (TTDA). In addition, we have been looking at parallel graph reduction, persistence in functional languages and the TTDA, and functional databases.

Our dataflow programming language Id Nouveau (or Id, for short) was redesigned and reached some degree of stability, based on vastly improved understanding of the denotational and abstract operational semantics of I-structures, our parallel data-structuring primitive. With experience in Id, we are beginning to develop a programming methodology for languages with I-structures. We are also looking beyond I-structures to other parallel, determinate object-oriented structures.

The Id compiler was completely rewritten in Common Lisp, and is now used quite heavily. Work continues in developing a type-system for Id and incorporating a type-checking module in the compiler. Because of its extremely modular structure, the compiler is very amenable to modifications, and is being used thus for research into other parallel architectures and languages.

A major activity in our group was the development of Id World, an integrated programming environment for experimenting with parallel programs. The components of Id World include editor customizations for Id, the Id compiler, GITA---an extensively instrumented emulation of the Tagged-Token Dataflow Architecture, and the Id debugger. These are packaged in an integrated user interface that makes it extremely easy to prepare, run, debug, and study parallel programs quickly. We believe that with its flexibility and power, Id World is a unique tool for studying parallelism in programs.

Various members of our group have begun using Id World to study a variety of application programs. Id World was publicly released in April 1987, and is available to anyone with a Symbolics or TI Explorer Lisp Machine. Based on the interest it has generated so far, we expect that Id World will soon be used in several research projects in the U.S. and abroad.

In the past year we have conducted many experiments to study instruction counts, instruction mixes, the effects of memory latency, granularity of parallelism, networks, program mapping, *i.e.* These experiments have significantly improved our understanding of fundamental issues in parallel architectures, and thrown much light on the TTDA and its relation to conventional von Neumann machines. This improved understanding of dataflow and parallel von Neumann machines has sparked two exciting new research efforts.

One new research effort is "Monsoon," a concrete architecture for the thus-far relatively abstract Tagged-Token Dataflow Architecture. Monsoon solves many of the open questions about the TTDA, such as the implementation of the Waiting-Matching store, and for the first time we have reached a point where we are ready to build a hardware dataflow machine. We are seeking funding to build the machine in the next three years. We have already begun building and studying prototypes of some components of the Monsoon architecture.

The second outcome of this deeper understanding has been research into hybrid von Neumann/dataflow architectures. Here we are exploring changes to the classic von Neumann architecture, borrowing ideas from dataflow. These changes address the fundamental latency and synchronization problems that we believe make it difficult, if not infeasible to use conventional von Neumann architectures in a general-purpose parallel machine.

The study of resource management issues continues to be a major effort. We have also begun to study various other topics that concern both language design and architectural support thereof: lazy evaluation, input-output and general persistence of arbitrary objects. We have also continued work on DisCoRd, an emulator for a parallel graph-reduction machine on the MEF.

In April 1987 we participated in the third MIT/IBM Workshop on parallel computing at Essex, Connecticut (held every other year). This is a useful forum for the exchange of ideas---we learned about the status of IBM's RP3 architecture and hardware construction, and about the EPEX programming environment for studying parallel programs, in use by IBM and several university partners. We continue to have strong and productive connections with IBM, particularly with Dr. K. Ekanadham. We now also have strong connections with Cornell University---Dr. Pingali joined the faculty there after graduating in August 1986.

Professor Nikhil is also working on the database problem in the Computer-Aided Fabrication project (Professor Penfield). Under his guidance, they have implemented a database system interface using ideas from functional languages. It is unique because with its clean functional formalism, non-computer experts have been able to learn its use rapidly; it is accessible both from C and from Common Lisp; and it encompasses data transparently from three distinct and different real database management systems.

The Multi-Processor Emulation Facility has stabilized to a large extent. It currently consists of thirty-two TI Explorer Lisp Machines on our circuit-switch network, with no current plans for enhancements or improvements. It is currently used for two kinds of emulations: the TTDA and DisCoRd. The generic MEF software to support arbitrary parallel emulators has been rewritten and cleaned up significantly as a result of our experience with the dataflow and graph-reduction emulators. We have given several public demonstrations of the MEF running parallel programs. These and various related matters are discussed in greater detail in sections below.

2. PERSONNEL

Over the course of the year, Computation Structures had six visiting researchers. Hitoshi Nohmi, a hardware specialist from Nippon Electronics Corporation, spent his year-long study leave strengthening his knowledge of computer languages, and participating in aspects of MEF design work. His colleague Akihiko Konagaya arrived in January to study dataflow and reduction architectures, and to further his interest in logic languages with constraints.

Staffan Truve, a Fulbright fellow from the Chalmers University of Technology in Sweden, joined the group to continue his study of logic languages. John Lewis, taking a year off from his post at Boeing in Seattle, has been studying the parallel execution of scientific appplications.

1986 saw the termination of IBM/Endicott's participation in CSG design work. In late September, senior associate engineers Michael Mack and Mark Atkins returned to Endicott. Mack had successfully designed the group's first chip, an 8 x 8 4-bit crossbar, and made a sizeable contribution to the design of the MEF transmitter. Atkins had been working on the FIFO logic and 48Mhz clock subsystem of the MEF.

The departure of the Endicott team did not mean the end of IBM's interest in the group's work, however. Although not formally a visitor, Dr. K. Ekanadham, a researcher in parallel processing at IBM/Yorktown Heights, has worked very closely with us over the past year. His work includes studies comparing the instruction set of the TTDA with von Neumann machines, and a significant rewrite of the SIMPLE application to take advantage of Id's high-level features.

On March 1, Professor Dennis took early retirement to devote more time to his new company, Dataflow Technologies. He now holds the rank of Senior Lecturer in Computer Science, and in that capacity will see his remaining graduate students through to the completion of their degrees. He will also continue to pursue his long-standing VIMVAL interests.

3. ID WORLD

Last year we reported the completion, by Dinarte Morais, of a first version of Id World, an integrated programming environment for preparing, running, debugging and analyzing Id programs and their behavior on the Tagged-Token Dataflow Architecture. Based on that experience, we invested much effort this year in a major redesign to make it stable and flexible, with the objective not only of making it easier to use internally, but also to release it publicly for use elsewhere. Originally aiming for January 1987, we released Id World in April 1987. Id World is implemented mostly in Common Lisp and runs on Symbolics and TI Explorer Lisp Machines. It is accompanied with extensive documentation that minimizes prerequisite knowledge of Lisp machines [10]. We have plans to port it to other popular workstations, such as Suns and MicroVaxes. Several research projects in the U.S. and abroad have expressed interest in acquiring and using Id World.

The program development process in Id World is patterned after the analogous process for Lisp. The editor (currently only Zmacs) is customized for Id--- it knows about Id syntax and has commands that facilitate entry and formatting of Id programs. The editor also has commands to invoke the Id compiler on segments of Id source code---the resulting object code (dataflow graphs) is normally loaded automatically into GITA, the "Graph Interpreter for the Tagged-Token Architecture." In case of compilation errors, the editor can immediately display the relevant parts of the source for correction.

To run compiled Id programs, one switches to the GITA window. Here one can compile and load Id files, and invoke any compiled Id function on GITA, the graph interpreter. Id objects for function arguments may be entered using a special "Lisp-ified" syntax.

A significant and unique part of Id World is the Id debugger, designed and implemented by Dinarte Morais. In case of run-time errors, GITA enters the debugger, which has commands similar to the Lisp debugger except that they are with respect to a tree of contexts instead of a stack, because this is a truly parallel emulator. Using the debugger, the programmer can examine the state of the machine, mostly using source-language names and constructs. A unique feature of Id is that the set of run-time errors (a parallel machine may not have a *single* run-time error!) does not change with machine configuration or run-time scheduling, and so debugging is easy and may be performed on a single processor.

After debugging, programmers can study the parallel behavior of Id programs: they can choose the number of processors, the network latency, and the kinds of run-time statistics that should be collected. After running the program, they can immediately display and plot the statistics on the three graphics panes in the GITA window. There are facilities to save, restore, and hardcopy statistics.

Id World is exciting not only for dataflow research, but also for other approaches to parallelism. Because the parallelism in Id Nouveau and the TTDA is limited *only* by data dependencies, Id World can supply a reference point for the *maximum* parallelism in a given algorithm---a calibration point against which one can compare the actual parallelism obtained in an encoding of the algorithm in, say, parallel FORTRAN running on a parallel von Neumann machine.

The development and implementation of Id World involved a significant cooperative effort by many members of the group, notably Dinarte Morais, Richard Soley, Ken Traub, Ian Taylor, and David Culler.

In order to pursue experimentation with scientific code requiring significantly more computational cycles than can be delivered from minicomputer Lisp machines, Stephen Brobst has begun an effort which will allow GITA to be ported to high-end mainframe machines. A version of the GITA experimentation tool is being developed in the C programming language for portability to a wide variety of high-end machines. Initial machines targeted for the port are Vax computers, the IBM 4381, and the Cray 2. We expect that it will be a simple matter to migrate the experimentation tools to any machine supporting a standard C compiler (with the *caveat* that sufficient memory resources exist on the machine for the large token storage requirement of the program). Completion of a first prototype is expected by the end of 1987.

4. LANGUAGES AND SYSTEMS

This year saw significant effort on several language and systems issues in Id Nouveau and the TTDA.

4.1. I-structure Semantics

The concept of I-structures as an architectural idea for parallel data structures is not new to the dataflow project, but it had long been unclear how to incorporate them into a programming language. Two significant steps this year cleared the way. First, Keshav Pingali established a connection between I-structures and logic variables---variables whose values are incrementally refined by unification---and thus showed a fixpoint denotational semantics for a language with I-structures. Based on this, Professors Arvind and Nikhil and Pingali, assisted by Vinod Kathail, developed an abstract Plotkin-style operational semantics for Id Nouveau. The semantics are given as rewrite rules that transform an Id program to its result, and capture exactly the parallel dataflow behavior of the program [1].

Abstractly, the machine state is modeled as a number of components *executing in parallel*---an expression and zero or more statements:

```
E ; S ; ... ; S
```

The result of the program is the ultimate value of the expression. The rewrite rule for function applications looks like this:

```
(f Earg1 ... Eargn)
E ; S ; ... ; S
--------------------------------
(Ebody')
E ; S ; ... ; S ; x1 = Earg1 ; ... ; xn = Eargn
```

i.e., the upper machine state which somewhere contains the expression `(f...)` can be rewritten to the lower machine state with the expression replaced by `(Ebody')`, where `Ebody'` is the body of function `f` with the formal parameters given new names `x1` through `xn`. This captures exactly the behavior in the TTDA where a function can begin executing while its arguments are still being computed. We call this dataflow behavior the *parallel call-by-value* computation rule. Similarly, there is a rule which says that an identifier can be substituted only when there is a statement in the machine state that binds the identifier to a reduced *value*---this corresponds exactly to the arrival of a token on an arc in the dataflow machine.

The rule for I-structure allocation is:

```
(array (vl,vu))
E ; S ; ... ; S
--------------------------------
(<Xvl,...,Xvu>)
E ; S ; ... ; S
```

Where Xv1 through Xvu are new variables. This rule illustrates a difference from functional languages--- rules for functional languages never introduce new variables on the right-hand sides.

An exciting aspect of these rewrite rules is that, for the first time, one can now understand the parallelism of the dataflow machine purely in source-language terms, without any appeal to dataflow graphs or the Tagged-Token Dataflow Architecture. In addition to facilitating the dissemination of dataflow ideas to a wider audience, this simplification also gives us a much better perspective on the relation of dataflow to other approaches to parallel execution of functional languages such as parallel graph reduction.

4.2. Id Nouveau

The advances in understanding the semantics of I-structures gave us insight into the semantic categories to be supported in the language Id Nouveau. The difficulties were in integrating cleanly the expression-oriented constructs of the functional subset with the refinement-oriented constructs of the I-structure subset. The syntax was frozen in January 1987, and the compiler and Id World upgraded accordingly.

One consequence of the introduction of I-structures is that the language loses "referential transparency", thus making it more difficult to reason about programs. For this reason, many in the functional programming community are still skeptical about I-structures and advocate the use of "bulk" functional array operators. For example,

```
make-array n f
```

returns an array of size n such that the i'th component contains f i. Thus the returned array can be considered a "cache" for a finite part of f.

We have argued in [2] that any fixed set of functional primitives will result in inefficient programs. The programmer must be allowed to invent and code new array abstractions, and for this, I-structures are essential in the language. We have thus gradually evolved a programming methodology in which one part of the program contains the definitions of program-specific array abstractions using I-structures, and the remaining, major part of the program is a purely function program that uses these abstractions and does not mention I-structures at all. We are still experimenting with this programming methodology.

To encourage this programming style, we have defined a large library of standard functional array, list and set operators [11] in the hope that programmers will use this common library and train themselves to think along those lines. Paul Barth wrote the code for these libraries which are now loaded automatically as part of Id World.

4.3. Id Compiler

Version II of the Id Compiler, written by Kenneth Traub, compiled its first program one month ahead of schedule in July 1986. Besides simply accepting the latest version of the Id programming language, Version II has several features which set it apart from most other compilers, including its predecessor, Version I:

- It is founded on a common core of data structures and abstractions that is general and powerful enough to support all conceivable dataflow compilers. This common core is described in Kenneth Traub's "A Dataflow Compiler Substrate" [14].
- It has a highly modular structure, and includes a facility (known as `defcompiler`) which permits modules to be incorporated into the compiler with very little effort.
- It includes a novel attribute grammar evaluator which incrementally computes parse tree attributes on demand, and automatically adjusts to changes in parse tree structure made by source-to-source transformation modules. The evaluator is designed to work from grammatical specifications developed with PAGEN, a parser generator program also written by Traub.
- It supports incremental compilation through a sophisticated database mechanism for recording properties of Id procedures. Assumptions about separately compiled procedures are recorded in object code, allowing the consistency of a collection of procedures to be verified at load time.
- It includes a number of code optimization modules, including such well-known transformations as common subexpression elimination, loop invariant code motion, and procedure integration (these were implemented by Ian Taylor). There is also a peephole optimizer for dataflow machine code, believed to be the first use of peephole optimization within a compiler for dataflow architectures. The peephole optimizer is also noteworthy as it is completely specification-driven.
- The compiler and PAGEN are written entirely in Common Lisp, ensuring their portability.

Many of these features reflect the Id Compiler's special nature as a *research* compiler; it is specifically designed to support experiments at all phases of the compilation process. Already, the compiler has proved itself adaptable enough to be used in two projects for which it was not originally designed: Bob Iannucci has modified the back end of the compiler to support his von Neuman Dataflow architecture (VNDF), resulting in a compiler from Id to VNDF object code, while John Lucassen of LCS' Programming Systems Research Group has replaced the front end with one for his FX language, resulting in a compiler from FX to TTDA object code.

So far, Version II of the Id Compiler has proved to be an overwhelming success. Future plans include a type-checking module to be written by Professor Nikhil, support of pattern-matching syntax, and code-generation for the Monsoon architecture.

4.4. Types and Type-checking in Id Nouveau

In Fall 1986 Professor Nikhil implemented a first version of a Milner-style polymorphic type-checker for Id. One problem was to devise the type-checking rules for I-structure constructs, which are analogous to side-effects in a functional language such as ML, and which normally make the Milner-style rules unsound. A solution exists in implementations of ML, but these implementations have never been published, and so we had to re-invent it for Id Nouveau.

Preliminary use of the type-checker was very encouraging---it promises to be a major aid in debugging and compiling. Our plans for the type-checker are:

- Upgrade it for the current Id Nouveau syntax,
- Fix a major limitation, which is the lack of user-defined union types and the associated pattern-matching syntax,
- Design a limited inheritance capability which is another kind of polymorphism that also simplifies programs,
- Use the type information to improve compiled code,
- Permit incremental type-checking, and the coexistence of typed and untyped code.

4.5. Lazy Structures

Eager interpreters are able to exploit vast parallelism, yet lazy interpreters have more desirable termination properties. Pingali proposed a source-to-source program transformation for achieving lazy behavior within an eager interpreter [13]. Pingali's approach offers the power of a lazy interpreter within the framework of dataflow, but is difficult in practice. When some values are not demanded, cleanup problems occur. For example, forks are not self-cleaning---if one arm does not demand a value that is demanded by the other arm, the value sits at the fork forever. This cleanup problem is quite difficult in the context of the TTDA, and we cannot ignore it. If values are always demanded by all possible consumers, lazy evaluation buys us nothing.

Steven Heller is considering another approach. An eager interpreter evaluates expressions as soon as the inputs are available, and a lazy interpreter evaluates an expression if and only if its value is needed to produce an answer. These extreme positions span a spectrum of possibilities, and he is studying some of these mixed evaluation strategies. If we delay only those expressions that sit in array slots, an interesting compromise is achieved. The TTDA already synchronizes array producers and consumers in hardware using I-structure Memory [2] [4] [7]. A similar synchronization mechanism is required to support demand propagation for delayed expressions that sit in array slots. By generalizing I-structures to *L-structures* ("lazy" structures) we can support both producer/consumer synchronization and demand propagation in hardware.

4.6. Accumulators

In studying various applications, we have repeatedly encountered a paradigm that cannot efficiently be handled by functional data structures or I-structures. One initializes an object, performs numerous "accumulations" on that object, and finally reads the value of the object. Because the accumulations are commutative, the order of accumulations is immaterial. An example would be to compute a histogram of 10,000 values into 10 intervals. In an imperative (and sequential) language, one would start with an array with 10 zeroes, and repeatedly increment the components. This cannot be done with functional data structures or with I-structures without excessive copying because one cannot update an array element in place. On the other hand, it is safe to do them in parallel because the increments may be done in any order.

Professors Arvind and Nikhil and Messrs Pingali and Traub have produced an initial proposal---both linguistic and architectural---to solve this accumulation problem in Id Nouveau on the TTDA. For example, to allocate an array for the histogram, one says:

```
xa,xr = 1D_accumulator (1,10) 10000 (+)
```

This allocates a vector with bounds 1 and 10, where each cell can accumulate values by addition, and where a total of 10000 accumulations are allowed. The expression returns two descriptors---an accumulate-only descriptor `xa` and a read-only descriptor `xr`. One can increment bucket (cell) `j` by saying:

```
1D_accumulate (xa, j, 1)
```

One can read bucket `j` as if it were an ordinary I-structure: `xr[j]`. However, the token for `xr` is not released until `10000` accumulations are done, thus ensuring that there are no read-write races. The function `1D_accumulator` is non-strict in `n`, the number of accumulations, so that the number of allowed accumulations need not be known beforehand.

We know how to compile these constructs into dataflow graphs for the TTDA and plan to implement it and start using it immediately. We certainly do not expect this to be the final word on accumulators---the hope is that experience in using it will allow us to understand the problem better and to produce a better solution.

4.7. Serialization and Serial Input-output

Richard Soley has begun exploring explicit and implicit serialization methods, with the aim of controlling machine resources efficiently during the execution of combinatorially explosive expert system programs. This serialization would be carried out in a completely distributed fashion, without any centralized control over the system. This methodology has carried over to a serialization scheme to enable serious input/output facilities within the Id language. In Soley's approach, serialization of calls to I/O primitives is carried out by the Id compiler, which adds static and dynamic program arcs to order the run-time execution of I/O primitives as the programmer has implicitly specified.

4.8. DisCoRd: Parallel Graph Reduction

Ian Taylor has been working on modifying the Id Compiler to generate combinator code for DisCoRd, a parallel graph reduction architecture. In contrast to other parallel graph reduction machine projects, we assume eager evaluation wherever possible, using this assumption to minimize message traffic. Ted Leung has been working on implementing an emulator on the MEF for the parallel graph reducer. This work is a redesign of the initial version of DisCoRd that we reported last year.

4.9. Persistence in Id/TTDA

Bhaskar Guha Roy and Professor Nikhil have been investigating the design and implementation of databases on the Tagged-Token Dataflow Architecture. Many database applications contain a high degree of inter-transaction parallelism and performance is often limited by disk latency. We believe the TTDA provides an excellent substrate for a high-performance database machine because of its ability to tolerate high latencies. Dataflow allows us also to take advantage of intra-transaction parallelism. I-structure memory allows synchronization among tasks to be expressed naturally, and the synchronization is achieved in hardware.

We are developing extensions to Id Nouveau for experimenting with functional databases. The main extension is the *bag* data structure for modelling large, homogeneous collections of objects. Bags have parallel, I-structure-like semantics. We are currently examining how operations in this language can be implemented to take advantage of features of the Monsooon architecture.

We have also been working on architectural extensions to the TTDA to support persistent store (disks). Objects of any data type in the language can be made persistent. A persistent object is initially referred to by a *logicalname*, and can be associated with a name in the program. The actual movement of data from primary to persistent store is transparent and incremental. We are currently extending the Id Nouveau compiler to support a variety of operations related to persistent objects. In the coming year, our goal is to complete the design of the persistent storage system and design and implement a complete transaction processing system.

4.10. Environments as First-Class Objects

This past year, Suresh Jagannathan (with Professor David Gelernter of Yale University and Professor Nikhil) has been examining the ramifications of incorporating *environments* as first-class values into a programming language. We have produced a new programming language with several novel features. Symmetric Lisp is built around an environment-building structure, the ALPHA form. The semantics of an ALPHA is derived by (conceptually) transposing the familiar Lisp PROGN or Algol compound-statement symmetrically around a time-space axis. Where the elements of a PROGN are evaluated during sequential lifetimes in a fixed temporal order, the elements of an ALPHA form are evaluated during concurrent lifetimes in a fixed spatial order. Concurrent evaluation lifetimes mean that the ALPHA form is a concurrency creating structure, and that Symmetric Lisp is a parallel language. Because elements of an ALPHA can refer to one another, they all have a shared evaluation lifetime. Shared lifetimes mean that the elements of an ALPHA-form may be taken to define a *scope*; all name-binding, program-building and scope-defining mechanisms in the language are based on this form. The semantics of Symmetric Lisp is defined by a collection of rewrite rules that preserve the structure of the source program (that is, the number and order of its elements). Unlike other languages, the "shape" of a program is invariant over the transformation process. Thus, ALPHA forms evaluate to new ALPHA forms and, consequently, Symmetric Lisp has no notion of a data structure: a data structure is simply any program that evaluates to itself.

"First-class environments" means that Symmetric Lisp allows programmers to write expressions that evaluate to environments and to create and denote variables and constants of type environment. One consequence is that the roles filled in other languages by a variety of limited, special-purpose environment forms like records, structures, closures, modules, and classes are filled instead by the ALPHA. In addition to being the fundamental structuring tool in the language, environments also allow us to treat function application as syntactic sugar for environment building: LAMBDA-forms become constants and are no longer constructs in their own right. Because the elements of an environment are evaluated in parallel, Symmetric Lisp is a parallel programming language intended for implementation on fine-grained architectures such as a dataflow or graph-reduction machine. Because environments may be constructed statically as well as dynamically, Symmetric Lisp accommodates an unusually flexible and simple parallel interpreter that is well-suited as an interface to a concurrent, language-based Symmetric Lisp computer system. Our goal for the coming year is to refine the design of the language and build an implementation on an available multi-processor architecture such as the MEF.

4.11. Functional Databases

Professor Nikhil has been guiding the database effort in the Computer-Aided Fabrication (CAF) project run by Professor Penfield. The problem here is to provide a single on-line information facility that encompasses not only traditional data-processing mainstays such as personnel and accounts, but also highly complex data such as IC masks, process-flow programs, intermediate states of process-flow programs, wafer states, *etc.* In addition, the facility must be able to access data from other existing software packages such as IC simulation packages.

With our suggestions and guidance, Michael Heytens, a graduate student in the CAF project, has designed and implemented GESTALT, a Functional Data Model interface, in which one views information as a collection of database types and functions that map between those types. These functions are embedded in a full functional language. The database is unique in its power and flexibility. The interface uses three separate commercial database management systems underneath for data storage; however, users see a single, integrated model of all the data. Accessible from C and Common Lisp, users have been able to learn to use it very quickly, and it is in daily use. We expect to continue this collaboration with the CAF project. Now that immediate operational needs have been met, we are exploring several enhancements to the type system, and to the data model so that it incorporates a notion of history. That is to say, data is never updated, only appended to.

5. ARCHITECTURES

5.1. TTDA Experiments

We are continuing our experiments toward assessment of the token storage requirements in a tagged-token architecture. Stephen Brobst has conducted and simulated studies with a variey of scientific codes to help develop understanding of temporal locality for tokens in the TTDA. Keeping track of active versus inactive (suspended) contexts in the machine is being examined as a possible means of implementing a migration strategy for tokens between memory and a fast local store.

In any parallel machine, data structure distribution and contention is an important problem. The recent attention given to "hot spots" in the parallel processing community is a reflection of this importance. Andrew Chien has done an in depth study of the effects of "hot spots" in the context of the TTDA [5] [6]. His results show that hot spots may in fact be a significant concern in the TTDA. He has also developed a simple network enhancement that greatly reduces the severity of performance degradation due to "hot spots." It is important to note that the TTDA processors' ability to tolerate memory latency is crucial to the effectiveness of this scheme. The ability to tolerate latency in a dataflow machine allows us to address the "hot spot" problem with much less hardware than a combining network.

Gino Maa conducted extensive experiments using the existing emulation tools to study the effects of the various code-mapping strategies and grain sizes, the presence of significant latencies in the communications network, and interleaved memory allocation on the performance of relatively large systems (hundreds of processors) executing a large-scale scientific application kernel. The results have all shown that a dataflow machine, when running large programs with sufficient parallelism, is indeed very tolerant of extreme communications latencies: system performance degrades very gracefully even with an almost order-of-magnitude increase in such latencies.

The resource allocation experiments provided evidence that relatively simple run-time strategies such as round-robin and randomized code mapping produce surprisingly good results consistently. They also showed that by choosing the grain size of the code-mapping unit to be around the iteration level, we get the scalability characteristics of mapping at the instruction level while still maintaining much of the locality property of mapping at the code-block level. Past studies on data structure reference patterns have indicated that contention for specific memory locations may hinder scalability in large systems, but recent experiments showed that a modest degree of interleaving in memory allocation yields much improvement over a non-interleaved memory system, although some contention for constant data structures can only be eliminated by altering the source program.

5.2. Storage Usage

GITA and the new Id compiler have been put to extensive use by David Culler in studying the resource requirements of dataflow programs under a variety of conditions. Our expectation that storage requirements grow in proportion to the amount of unfolding under idealized execution with unrestricted parallelism was confirmed; this implies cubic growth for triply nested loops, for example. Moreover, restricting the amount of parallelism exploited in executing a program does not alleviate the problem, rather, it is necessary to constrain the unfolding of the program itself. Loop-bounding techniques developed by Culler have proved effective in this; under restricted parallel execution, it is possible to reduce resource requirements dramatically without increasing the running time of the program appreciably. We are continuing to explore this direction.

5.3. Instruction Counts

An important metric in assessing the effectiveness of the dataflow approach is the total number of instructions executed. TTDA instructions are roughly comparable in power to those of a load/store architecture - memory access operations are disjoint from arithmetic operations. We expect instruction counts of dataflow programs to be somewhat higher than a good sequential implementation, as there is a certain amount of work required to initiate and synchronize concurrent computations. Nonetheless, for dataflow machines to be viable, they must be comparable to sequential machines in this regard. Comparative studies performed by David Culler in conjunction with K. Ekanadham at IBM/Yorktown indicate that Id programs with little optimization typically require 2-3 times as many instructions as highly optimized Fortran. This comparison is encouraging in light of the number of additional instructions that would be performed in a "parallelized" Fortran version. However, it is clear that without relatively sophisticated program graph generation, as in the current Id compiler, the gap would be much worse. Recent work with common subexpression elimination has narrowed the gap dramatically.

5.4. Towards Real Implementations

While it has been well understood for some time that dataflow offers a framework for thinking about parallel computation, it is only within the last year that we have been able to make statements about the *essence* of dataflow architecture which may be meaningfully applied in the von Neumann multiprocessor domain. It has become clear through analysis [3] and experiments by Gino Maa that any scalable architecture must be able to tolerate basic, machine-induced latencies and must provide, at the hardware level, a synchronization mechanism that is inexpensive to use. The former is necessitated by the physical partitioning of a machine into cooperating processing and memory elements separated by nontrivial communication delays. The latter is a direct result of decomposition of the program into communicating pieces or *tasks*.

Dataflow by its very nature allows parallelism in the program to be traded off against latency---given sufficient parallelism in the program (on the order of the processor-memory-processor pipeline depth *times* the number of such parallel pipelines), latency cost as measured by induced processor idle time can be controlled. Dataflow also offers a uniform synchronization paradigm through the tagging and matching of data. Each enabled instruction represents a *task* which can execute independent of all other such tasks. Tags serve to identify these tasks. Dataflow machines provide the means for bringing together identically tagged values; this is the necessary and sufficient condition for the task's execution.

The essential features of the dataflow mechanism are a large namespace for identifying "meeting places" (synchronization events), provision at the hardware level for multiple, concurrent tasks, and the ability to switch between these tasks as necessary with speeds approaching single instruction times.

5.5. Von Neumann Dataflow Machine

One proposal which has grown out of this work is the construction of a hybrid dataflow/von Neumann machine by extending von Neumann architecture with some embodiment of the essential features of dataflow. The proposal is made and discussed by Iannucci [8]. The goal of this work is to refine further the notion of *essential features*, and to explore compiler-directed, pipelineable sequential code sections as a tool for implementing resource management primitives and for exploiting vector-type instructions.

One possible approach is to recognize the relationship between arcs in a compiled dataflow graph and slots in a traditional invocation stack frame. Both are used for holding temporaries local to the invocation of the associated procedure. As such, they embody the intra-process communication mechanism. Augmented with a basic synchronization mechanism, stack frame slots allocated out of a relatively large address space would provide two of the three above-mentioned features considered essential for a scalable multiprocessor. Local memory with I-structure-like synchronization bits on each slot (indicating *slot-empty*, *slot-full*, or *deferred-read*) [7]provides such a synchronization mechanism; deferred reads cause suspension of the current process and storage of the current program counter (PC) into the empty slot. Subsequent writing to the deferred slot reawakens the suspended process by extracting the deferred PC and making it a candidate for execution once again.

The third feature, fast task switching, implies sufficient high-speed storage to hold the computation state for a large number of such PCs (similar to the requirement for a large, fast waiting-matching memory and token buffer in a dataflow machine), and the ability to interleave instructions from different tasks on a per-instruction (or nearly so) basis. Note that it is neither essential nor always desirable to switch tasks at each instruction dispatch---it is often the case that small groups of instructions may be statically scheduled for execution as a unit given the satisfaction of only a few input data dependencies. Thus, the quanta of execution may be bigger than single instructions. It is, however, essential that the task switching be done to the *resolution* of individual instructions.

Progress to date includes definition of a simple machine model, complete definition of the syntax and semantics of a suitable machine language, design, coding, and testing of a von Neumann/dataflow back end for the Id compiler (previously described), and preliminary work on an emulator for the architecture.

5.6. Monsoon

We have been sufficiently encouraged by our research results to contemplate and evaluate critically a hardware implementation of a multiprocessor based on the Tagged-token Dataflow Architecture. Central to this goal is the ability to translate the dataflow execution mechanism, specifically waiting-matching and I-structure operations, into practical and efficient hardware. In [12], Greg Papadopoulos has described a novel instruction execution mechanism that implements both the I-structure storage and the waiting-matching section in the same *explicitly addressed* storage. He has also given the outline for generating code for such a machine from TTDA-style dataflow graphs.

The processing element is somewhat more general than a TTDA graph interpreter. The design draws heavily on traditional pipelined von Neumann techniques as popularized by the "RISC" methodology. A processing element is really a *multi-threaded* non-blocking RISC pipeline, where a join of two threads, an operation, and a fork of two threads can all occur within a single pass through the pipe. Threads are interleaved each cycle, without switching overhead, from a hardware managed task queue. We address the two fundamental multiprocessing issues by (1) providing non-blocking split-transaction global memory references, and (2) providing very efficient hardware synchronization on an instruction-by-instruction basis. We believe that this architecture brings the dataflow machine a step closer to von Neumann machines, exploiting the efficiencies of pipelined designs while reducing the overhead of fine-grained data-driven evaluation.

We intend to construct a 256 PE multiprocessor prototype called "Monsoon." Because we believe the processor pipeline to be well balanced and technologically scalable, we are initially employing fairly conservative TTL and CMOS gate array technologies. Our initial implementation calls for each PE to have a 100ns. cycle time, 64-bit floating point, and 2 Megawords of local storage. The network will be a packet switched 256-way two-stage exchange with 800 Mbits/sec/port. This will yield a machine with a peak performance of over two GigaFLOPS. We believe it will *sustain* between 100-300 MegaFLOPS on a wide variety of scientific codes, making it competitive with the fastest general purpose von Neumann machines presently available. A discrete logic laboratory prototype of a processor (125ns. cycle time) is now under construction.

5.7. Compiling for Sequential Architectures

Many of the programming languages devised for fine-grained parallel architectures are *non-sequential.* Non-sequential languages, which include Id as well as most lazy functional langauges, cannot be directly compiled into ordinary sequential code (as for a von Neumann machine). Instead, they must be compiled into fine-grain parallel code (as for a dataflow machine) or into many sequential threads, executed concurrently. In his PhD research, Kenneth Traub is examining the problem of compiling non-sequential languages into multi-thread code. Multi-thread code can be run on von Neumann machines by simulating parallel scheduling, and so this work will address the problem of efficient execution of non-sequential languages on von Neumann machines. More importantly, this work will have direct application to machines whose architecture is based on a multi-thread model, such as Iannucci's VNDF.

As part of his dissertation work, Iannucci [8] has constructed a new code generator for the Id compiler. While the target machine architecture is markedly different *c.f.* TTDA, the relative ease with which new modules were integrated to the existing compiler was significant. This new compiler retains parse tree and program graph generation and substitutes a new machine graph generator, an altered peephole optimizer, and a new assembler for a von Neumann style (i.e., program counter based) architecture.

6. APPLICATIONS

K. Ekanadham of IBM/Yorktown, working closely with Professor Arvind, has been rewriting the SIMPLE code in Id Nouveau. With extensive use of higher-order functions and array abstractions, Ekanadham's masterful SIMPLE code in Id has provided the best example to date of the high level that scientific programming can reach. His SIMPLE code has become the standard against which we compare the quality of codes written here and elsewhere.

During the fall, three scientists from Los Alamos National Laboratory visited the group to learn about our programming environment and to start work on several large dataflow applications. The discussions prompted a valuable review of Id. These visitors became a beta-site for the Id World release. One of their applications, a Particle-In-Cell (PIC) electrodynamics code, had been implemented on a variety of parallel machines. Culler wrote a version of it in Id, and we are now comparing various implementations. The PIC code involved more sophisticated data structures than most scientific applications and stressed the expressiveness of the I-structure paradigm. One of the Monte Carlo codes involving neutron transport proved very difficult to express efficiently with I-structures and added to the on-going discussion of accumulators.

Paul Barth wrote a signal-processing application in Id Nouveau based on his experience at Schlumberger with software for oil-exploration. James Hicks also wrote a general electronic signal-processing application in Id Nouveau. Both these experiments shed light on the need for streams and stream-processing operators in the langauge.

Paul Barth wrote several algorithms for the single-source, shortest path problem. These algorithms highlight several methods of applying dataflow to graph traversal problems. One algorithm uses I-structures for synchronizing the traversal of several parallel paths; another encodes the graph as a dataflow program that can be executed directly. Two algorithms were written that used nondeterministic constructs for marking the graph. These nondeterministic constructs are not currently part of Id Nouveau; their addition would support these algorithms, as well as many others, such as dynamic programming and search problems in AI.

Serge Plotkin studied the problem of writing a symbolic polynomial arithmetic package in Id Nouveau. This exercise again reinforced the need for "accumulators" in the language. For example, when multiplying two polynomials with coefficients a_0, a_1, ... and b_0, b_1 for x^0, x^1, ... each coefficient c_j is sum of products of the form a_i*b_{j-i}. The summing can be done in any order, and so can ideally be expressed as an "accumulation" of product terms.

Richard Mark Soley has been performing experiments with Id World on pattern-matching systems, both as an approach to debugging Id World and a study of the potential sources of parallelism within "expert" production systems. He has identified various combinatorially explosive aspects of these computations, leading to his current work in efficiently serializing such highly parallel, and highly "speculative" programs.

Arun Iyengar joined our group late in the Spring Term, 1987. He will be studying the implementation of graph algorithms in Id Nouveau/TTDA based on his experience in writing applications in molecular biology.

In the fall, James Hicks wrote an interface that enhances the use of the Quicksim circuit simulator, part of our Mentor logic CAD system. A designer can use an ordinary text editor to enter specifications of circuit inputs and expected circuit outputs, and then run a driver that performs the circuit simulation, automatically applying the inputs at the right (simulated) times and comparing the simulated outputs with the expected outputs. We expect this to be of use in our future hardware design efforts.

7. MULTIPROCESSOR EMULATION FACILITY

Andy Boughton, Jack Costanza, and Ralph Tiberio have been responsible for ensuring the reliability of the MEF hardware. During the past year the circuit switch and the other MEF hardware have stabilized. The high infant mortality rate among certain active components on the circuit switch accounted for the greatest number of failures. For example, many optoisolator failures were recorded in the first few thousand hours of service. As the number of hours on the circuit switch boards has gone up, the failure rate has gone down. Currently, the average time between circuit switch hardware failures is a few months. The reliability of the circuit switch has been more than adequate to support large experiments.

During the same period we have also seen a decrease in the failure rate of the MEF processors.We have kept detailed records of all MEF hardware failures in an IBM SQL database designed by Jack Costanza. The failure rate that we have observed on the MEF processors is consistent with industry averages. The failure rate of the circuit switch boards has been substantially less.

7.1. Network Development

We have redirected our development effort from the MEF to Monsoon. The departure of the IBM team of Atkins and Mack, coupled with the emergence of a design for an extremely practical hardware implementation of the Tagged-Token Dataflow Architecture has caused this change in emphasis. The MEF packet switch design of the IBM team promised greatly improved robustness, flexibility, and reliability over the existing MEF circuit switch. The team's departure, however, prompted the shift to the circuit switch. While the circuit switch has proven to be sufficient for the current 32 processor MEF configuration, the MEF packet switch would have provided the robustness and reliability necessary to support larger and more flexible MEF configurations.

The key characteristics required of a network for Monsoon are bandwidth and reliability. Monsoon requires a network capable of supporting 800 megabits per second of bandwidth on each network input. We believe that such a network can be constructed using concepts similar to those developed for the MEF packet switch.

Andy Boughton, Chris Joerg, and Greg Papadopoulos have developed an overall structure appropriate for the Monsoon network. The proposed structure is a staged packet switched network. The proposed network is composed of two stages of 16-input 16-output switch boards. The maximum size of the network is 256 inputs and 256 outputs. Each switch board is composed of eight four-input four-output Packet Switched Routing Chips (PaRC's). Each PaRC will be a complete switching node with a crossbar, control circuitry, and packet buffering. The data paths of the network are assumed to be 16 bits wide and capable of running on a 50 Mhz clock. The network also supports circuit switched connections between network inputs and network outputs. This facility is required by the Monsoon architecture in order to allow a processor to maintain, if necessary, a strict order among the arrival times of its messages at other processors.

A preliminary logic design for one possible implementation of PaRC has been completed by Chris Joerg. The proposed implementation is based on a number of the concepts developed for the MEF packet switch [9]. The implementation uses LSI Logic compacted gate array technology. The proposed PaRC is composed of four major subcomponent types; fifo input controller, scheduler, transmitter, and crossbar. The overall structure is similar to that of the MEF packet switch board but there are some differences. PaRC uses a distributed scheduling scheme with a scheduler associated with each output. While buffering a packet received on a given input and destined for a blocked output, PaRC is capable of transferring a subsequent packet from the same input to a different output if that output is not blocked. PaRC is also capable of supporting circuit switched connections. The proposed PaRC was designed and simulated using LSI Logic 7000 series logic. At the time our CAD system only supported the 7000 series and we decided to proceed with a preliminary design for PaRC rather than wait for the upgrade of our CAD system. Since the 10000 series array is required to fabricate a chip of the size of PaRC, we must now transfer the design to the 10000 series and simulate it in more detail.

Jack Costanza and Ralph Tiberio have explored potential link technologies for the Monsoon network. The size of Monsoon may require some of its network links to be 30 to 40 feet long. We have tested a link technology that uses a 16 bit wide data path. This link is based on earlier work done by Mark Atkins, using coaxial cables and CMOS drivers and receivers. While our initial results have been encouraging, much more work is required to develop a link technology with the reliability that is needed for the Monsoon network.

7.2. Evolution of MEF Software into a General Emulation Model

Two years ago, we reported on Tanglewood, a powerful, general substrate for MEF experiments built on top of the EtherNet. Unfortunately, Tanglewood proved to be rather slow. Last year we put Tanglewood aside in favor of CSWITCH, an extremely lean, specialized interface to the MEF circuit switch network. This network interface was tightly integrated with the dataflow emulator MEF-GITA. In a sense, this year we have come full circle: the circuit switch interface has been abstracted from MEF-GITA and generalized to provide a simple, efficient substrate for a broad variety of MEF experiments. The new CSWITCH abstraction has facilitated many extensions to GITA and has been used as a substrate for other MEF experiments, including DisCoRd, a graph-reduction architecture, and the game of MultiLife.

7.3. Demonstrations on the MEF

MEF made a number of public appearances this year. The demonstration to participants in the Lisp and Functional Languages conferences in August drew a large crowd. All thirty-two TI Explorers in the MEF were used in executing a variety of dataflow programs. In December we showed various aspects of MEF-GITA and GITA to the executive director of DARPA. This included system utilities of the MEF, a large hydrodynamics code written in Id and running on thirty-two machines, and a non-dataflow MEF application, MultiLife, employing the new CSWITCH interface. In addition, we demonstrated many facilities in GITA for studying the behavior of parallel programs.

8. WORK UNDER PROFESSOR DENNIS'S SUPERVISION

Tam-Anh Chu has completed his doctoral dissertation entitled "Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications" under the supervision of Professor Jack Dennis. In this thesis, he presents an approach for direct and efficient synthesis of self-timed (asynchronous) control circuits from formal specifications called Signal Transition Graphs (STGs). Control circuits synthesized from this graph model are speed-independent and capable of performing concurrent operation. The property of speed-independence means that the circuit operates correctly regardless of variations in delays of logic gates, thus implying that the circuit is hazard-free under any combination of gate delays. The capability of STGs for explicitly specifying concurrent operations internal to a control circuit is unique to this model, unlike other approaches based on Finite State Machines.

STGs are a form of interpreted Petri nets, in which transitions in a net are interpreted as transitions of signals in a control circuit. While other synthesis approaches based on Petri nets have not been very successful, we have developed a number of analytical results which establish the equivalence between the static structure of nets (their syntax) and their underlying firing sequence semantics--an analytical approach called structure theory of Petri nets. This equivalence permits the characterization of the low-level properties of control circuits in terms of STG syntax: the deadlock-free and hazard-free properties of circuits are characterized as syntactic properties of liveness and persistency of STGs. A preliminary STG specification of a control circuit can be modified into one which is live and persistent, from which a deadlock-free and hazard-free logic implementation can be derived mechanically.

STGs allow efficient synthesis of control circuits by using a method of decomposition based on a graph-theoretic technique called contraction. Instead of implementing a logic circuit from a STG directly, it can first be decomposed into a number of contracted nets, one for each signal generated by the control circuit. A logic element can then be determined from each contracted net, and the composition of logic elements produces the final circuit implementation.

Publications

1. Arvind, and D.E. Culler. "Dataflow Architectures." *Annual Review of Computer Science*, I, 1986.

2. Arvind, and K. Ekanadham. "Future Scientific Programming on Parallel Machines." To appear in *Proceedings of the International Conference on Supercomputing (ICS)*, Athens, Greece, June 1987.

3. Arvind, and R.A. Iannucci. "Two Fundamental Issues in Multiprocessing." To appear in *Proceedings of DFVLR - Conference 1987 on Parallel Processing in Science and Engineering*, Bonn-Bad Godesberg, Germany, June 1987. Also MIT/LCS/TM-330 and Computation Structures Group Memo 226-6, MIT Laboratory for Computer Science, Cambridge, MA, May 1987.

4. _. "Two Fundamental Issues in Multiprocessing." Computation Structures Group Memo 226-5, MIT Laboratory for Computer Science, Cambridge, MA, July 1986.

5. Arvind, and R.S. Nikhil. "Executing a Program on the MIT Tagged-token Dataflow Architecture." To appear in *Proceedings of the PARLE Conference*, Eindhoven, The Netherlands, June 1987. (Also Computation Structures Group Memo 271, March 1987.)

6. Arvind, and R.S. Nikhil, and K.K. Pingali. "Id Nouveau, Reference Manual Part II: Operational Semantics." Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, MA, April 1987.

7. _. "I-structures: Data Structures for Parallel Computing." Computation Structures Group Memo 269, MIT Laboratory for Computer Science, Cambridge, MA, February 1987. Also to appear in *Proceedings of the Graph Reduction Workshop*, Santa Fe, NM, October 1986.

8. Chien, A.A. "Hot Spots in Routing Networks: A Collection of Studies." Computation Structures Group Memo 267, MIT Laboratory for Computer Science, Cambridge, MA, October 1986.

9. _. "Structure Referencing in the Tagged-token Dataflow Architecture." Computation Structures Group Memo 268, MIT Laboratory for Computer Science, Cambridge, MA, October 1986.

10. Chu, T-A. "Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specificiations," MIT/LCS/TR-393, MIT Laboratory for Computer Science, Cambridge, MA, June 1987.

11. _. "Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications." *Proceedings of the International Conference on Computer Design*, IEEE, New York, October 1987.

12. Chu, T.-A., and L.A. Glasser, "Synthesis of Self-timed Control Circuits from Graphs: An Example." *Proceedings of the International Conference on Computer Design*, IEEE, New York, October 1986.

13. Chu, T.-A., and C. K. C. Leung, "Design of High Performance FIFO Queues for Packet Communication Architectures." *Proceedings of the International Conference on Parallel Processing*, IEEE, Chicago, August 1986.

14. Culler, D.E. "Amdahl's Law Revisited: Measurements of Dataflow Programs," Workshop on Performance-Efficient Parallel Processing," Seven Springs, PA, September 1986.

15. Gelernter, D., S. Jagannathan, and T. London. "Environments as First-class Objects." 14th Conference on Principles of Programming Languages, Munich, Germany, January 1987.

16. _. "Parallelism, Persistence and Meta-cleanliness in the Symmetric Lisp Interpreter." 1987 SIGPLAN Conference on Interpreters and Interpretive Techniques, St. Paul, MN, June 1987.

17. Malone, T.W., K-R. Grant, E.A. Turbak, S.A. Brobst, M.D. Cohen. "Intelligent Information Sharing Systems." Communications Of The ACM, March 1987, vol. 30, no. 5.

18. Nikhil, R.S., K.K. Pingali, and Arvind. "Id Nouveau." Computation Structures Group Memo 265, MIT Laboratory for Computer Science, Cambridge, MA, July 1986.

19. Nikhil, R.S. "Id Nouveau Quick Reference Guide." Computation Structures Group internal document, MIT Laboratory for Computer Science, Cambridge, MA, February 1987.
20. _. "Id Nouveau, Reference Manual Part I: Syntax." Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, MA, April 1987.
21. _. "Id World Reference Manual." Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, MA, April 1987.
22. Traub, K.R. "A Compiler for the Tagged-token Dataflow Architecture." MIT/LCS/TR-370, MIT Laboratory for Computer Science, Cambridge, MA, August 1986.

Theses Completed

1. Brown, D.A. "Concurrent Synchronous Simulation." S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1987.
2. Chien, A.A. "Congestion Control in Routing Networks." S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, October 1986.
3. Chu, T.-A., "Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications." Doctoral dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge MA, May 1987.
4. Joerg, C.F. "Design of a Circuit Switched Routing Chip for a Dataflow Supercomputer." S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1987.
5. Kaushik, S. "Design of a Cyclic Redundancy Code Generator Circuit for a Packet Switch." S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1987.
6. Traub, K.R. "A Compiler for the MIT Tagged-token Dataflow Architecture." S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, August 1986.

Theses in Progress

1. Culler, D.E. "Effective Dataflow Execution of Scientific Applications." Doctoral dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected December 1987.
2. Guha Roy, B. "Transaction Processing on Dataflow Computers." Doctoral dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected December 1988.
3. Heller, S.K. "Efficient Streams on a Dataflow Machine." Doctoral dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected December 1987.

4. Iannucci, R. A. "A Dataflow/von Neumann Hybrid Architecture." Doctoral dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected August 1987.
5. Jagannathan, S. "The Design and Implementation of a Symmetric Programming Language." Doctoral dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected August 1988.
6. Kathail, V.K. "Optimal Evaluators for Functional Languages." Doctoral dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected December 1987.
7. Maa, G. "Scalability of the Tagged-token Dataflow Machine." S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected September 1987.
8. Papadopoulos, G.M. "Implementation of a General-purpose Dataflow Multiprocessor." Doctoral thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected August 1987.
9. Soley, R.M. "On the Efficient Exploitation of Speculation Under Dataflow Paradigms of Control." Doctoral thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected December 1987.
10. Traub, K.T. "Sequential Implementation of Non-sequential Programming Languages." Doctoral dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, expected June 1988.

Talks

1. Arvind. "Dataflow and the Multiprocessor Emulation facility." DARPA/JASON Workshop on Advanced Computer Architecture Research, La Jolla, CA. July 10, 1986.
2. _. "A von Neumann-Dataflow Machine." IBM Research, Hawthorne, NY, August 26, 1986.
3. _. "Data Structures for Parallel Computing." Graph Reduction Workshop, Santa Fe, NM, October 1, 1986.
4. _. "Quantifying Parallelism in Programs." Los Alamos National Laboratory, Los Alamos, NM, October 2, 1986.
5. _. "Dataflow Architectures." Keynote Talk, ICCD 86, Rye Town Hilton, Port Chester, NY, October 6, 1986.
6. _. "Dataflow Architectures." Michigan State University, E. Lansing, MI, October 16, 1986.
7. _. "Parallel Computing: New Directions in Dataflow." MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, November 17, 1986.
8. _. "Quantifying Parallelism In Programs." IBM-ACES 11th University Study Conference, Bonaventura Hotel, Ft. Lauderdale, FL, November 19, 1986.

9. _. "Dataflow Architectures." NCR Workshop on Distributed Systems Architecture, San Diego, CA, December 9, 1986.

10. _. "Dataflow and Scientific Programming." Lawrence Livermore Laboratory, Livermore, CA, December 10, 1986.

11. _. "Future Scientific Programming: Ek's SIMPLE Code." Parallel Processing: Matching Execution Models with Program Classes, A High-Speed Computing Conference, sponsored by Los Alamos and Lawrence Livermore National Laboratories, Gleneden Beach, OR, March 17, 1987.

12. _. "Parallel Computing: The need to move away from the von Neumann Model." A keynote talk, SEAS meeting, Montpellier, France, April 7, 1987.

13. _. "Monsoon: A realization of the MIT Tagged-Token Dataflow Architecture." The 3rd MIT-IBM workshop on Parallel Processing, Essex, CT, April 20, 1987.

14. _. "Future Scientific Programming: Ek's SIMPLE Code", The 3rd MIT-IBM Workshop on Parallel Processing, Essex, CT, April 21, 1987.

15. _. "Future Scientific Programming." A keynote talk, International Conference on Supercomputing, Athens, Greece, June 11, 1987.

16. _. "Executing a Program on the MIT Tagged-Token Dataflow Architecture." A keynote talk, PARLE Conference, Eindhoven, The Netherlands, June 15, 1987.

17. _. "Two Fundamental Issues in Multiprocessing." DFVLR - Conference 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, W. Germany, June 25, 1987.

18. Brobst, A.A. "Dataflow Computing." Given to group from Chalmers University of Technology (Stockholm, Sweden) at MIT Laboratory for Computer Science, July 2, 1986.

19. _. "Design Considerations for a Closely Coupled Multicomputer System." Systems Architecture Laboratory, Hewlett-Packard, Cupertino, CA, August 12, 1986.

20. _. "An Approach to HP Shared-disc Clusters." Information Technology Group, Special Interest Group on Systems Clusters, Hewlett-Packard, Cupertino, CA, August 21, 1986.

21. _. "Dataflow Computing." Boston University, Boston, MA, November 19, 1986.

22. Culler, D.E. "Current Commercial Parallel Machines. Computer Architectures Course, Boston University, December 3, 1986.

23. Iannucci, R. A. "Multiprocessor Emulation Facility: Retrospective." DARPA/IPTO meeting held at LCS, December 18, 1986.

24. _. "Dataflow Computer Architecture: an Introduction." Bolt, Beranek and Newman Laboratories, Cambridge, MA, March 2, 1987.

25. _. "Dataflow Computer Architecture." DSD Kingston Laboratory, IBM Corporation, Kingston, NY, March 2, 1987.

26. Jagannathan, S. "Environments as First-class Objects." 14th Conference on Principles of Programming Languages, Munich, Germany, January 1987.

27. Jagannathan, S. "Parallelism, Persistence and Meta-cleanliness in the Symmetric Lisp Interpreter." 1987 SIGPLAN Conference on Interpreters and Interpretive Techniques, St. Paul, MN, June 24-26, 1987.

28. Nikhil, R.S. "Functional Database System." IEEE Hyderabad Chapter, Hyderabad, India, July 1986.

29. _. "I-structures: Data Structures for Parallel Computing." Department of Computer and Information Sciences, University of Pennsylvania, Philadelphia, PA, November 1986.

30. _. "Functional Databases." Boston SIGMOD Seminar, Cambridge, MA, November 1986.

31. _. "Dataflow and Databases." DARPA/IPTO meeting held at LCS, December 18, 1986.

32. Soley, R.M. "Lisp Carries Its Own Weight." 3rd Artificial Intelligence Applications Conference, IEEE, Orlando, FL, February 25, 1987.

References

[1] Arvind and Nikhil, R.S. and Pingali, K.K.
Id Nouveau Reference Manual, Part II: Operational Semantics.
Technical Report Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, MA, April, 1987.

[2] Arvind and Nikhil, R.S. and Pingali, K.K.
I-Structures: Data Structures for Parallel Computing..
Technical Report Computation Structures Group Memo 269, MIT Laboratory for Computer Science, Cambridge, MA, February, 1987.

[3] Arvind and Iannucci, R.A.
Two Fundamental Issues in Multiprocessing: the Data Flow Solution.
Technical Report Computation Structures Group Memo 226-6, MIT Laboratory for Computer Science, Cambridge, MA, June, 1987.

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

[5] Chien, A.A.
Congestion Control in Routing Networks.
Master's thesis, MIT Department of Electrical Engineering and Computer Science, October, 1986.

[6] Chien, A.A.
Hot Spots in Routing Networks: A Collection of Studies.
Technical Report Computation Structures Group Memo 267, MIT Laboratory for Computer Science, Cambridge, MA, October, 1986.

[7] Heller, S.K.
An I-Structure Memory Controller (ISMC).
Master's thesis, MIT Department of Electrical Engineering and Computer Science, June, 1983.

[8] Iannucci, R.A.
A Dataflow / von Neumann Hybrid Architecture.
PhD thesis, MIT Department of Electrical Engineering and Computer Science, 1987.
(in preparation).

[9] C.F. Joerg.
Design of a Packet Switched Routing Chip for the Dataflow Supercomputer.
Technical Report S.B. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May, 1987.

[10] Nikhil, R.S.
Id World Reference Manual.
Technical Report Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, MA, April, 1987.

[11] Nikhil, R.S.
Id Nouveau, Reference Manual Part I: Syntax.
Technical Report Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, MA, April, 1987.

[12] Papadopoulos, G.M.
An Engineering Implementation of the Tagged-Token Dataflow Machine.
Technical Report Computation Structures Group Memo 270, MIT Laboratory for Computer Science, Cambridge, MA, 1986.
(for internal use only).

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

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