MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Laboratory for Computer Science

Computation Structures Group Memo 203

Computation Structures Group
Progress Report 1979-80

This research was supported in part by the Department of Energy under contract DE-AC02-79ER10473 and in part by the National Science Foundation under grants MCS75-04060-A01 and MCS79-15255.

February 1981

COMPUTATION STRUCTURES GROUP

Academic Staff

J.B. Dennis, Group Leader

Arvind, Group Leader

Research Staff

A. M. Feridun

E. S. Shaw

Graduate Students

W. B. Ackerman
D. J. Aoki
G. A. Boughton
J. D. Brock
R. E. Bryant
C. A. Cesari
D. L. Isaman
V. K. Kathail
J. W. Leth
C. K. C. Leung
L. B. Montz
K. K. Pingali
N. Singh
K. W. Todd

Undergraduate Students

A. J. Chien
S. V. Kwong
J. E. Lilienkamp
P. S. Ries
R. W. Tucker
T. L. Tung

Support Staff

A. Rubin

K. Warren

1. INTRODUCTION

The Computation Structures group is continuing its study and development of computer systems based on data-driven program execution. Our major accomplishments over the past year have been the development of a translator and interpreter for the functional programming language VAL (Value-oriented Algorithmic Language); continued study of the transformation of VAL programs into data flow graphs; the design of a translator for a subset of the data flow language Id; development of a scheme for decomposing programs for multiple processor systems; the mapping of data flow programs onto the U-interpreter machine; specification of the design of the processing unit and communication module to be used in evaluating prototype data flow computers; continued study of routing networks; and development of a MOS LSI logic simulator.

The Computation Structures group has also been active in bringing interested scientists together for exchange of ideas and knowledge in areas closely related to our research. In July 1979, with support from the Department of Energy, we sponsored a Workshop on Self-Timed Systems [7], which served to bring together experts in the field of asynchronous logic design for a unique opportunity to review and assess the state-of-the-art and to chart directions for future development. With the prospect of economical custom fabrication of LSI devices, there appears to be a new opportunity for profitable applications of self-timed principles. In particular, the workshop left one with the strong feeling that self-timed design methodologies will have an important, if not essential, role in the successful production of VLSI devices. A one-week mini-course, Data Flow Concepts in Computer Language and Architecture, was taught in June 1980, and the group is making preparations for the Workshop on Applicative Languages and Parallel Computation which it is hosting at the MIT Endicott House in July 6-9, 1980.

2. DATA FLOW LANGUAGE IMPLEMENTATION AND TRANSLATION

The programming language VAL [2, 13] is the product of cooperation between the MIT Computation Structures group and the Lawrence Livermore Laboratory to define a source language for expressing numerical computations for high performance execution on data flow computers. The programming language Id [3] was originally conceived by Professor Arvind and his colleagues at the University of California, Irvine, and is being used to explore implementation of the "unraveling interpreter" and applications of data flow computation at MIT. Both languages are functional programming languages and are free of non-function artifacts such as side effects or aliasing of arguments. Current work is concerned with implementation schemes and the mapping of programs expressed in VAL and Id onto data flow hardware architectures.

2.1. VAL Language Implementation

An interpretive implementation of the VAL data flow language has been constructed. This implementation consists of a translator and an interpreter written in CLU for the MIT-XX TOPS-20 computer system. The translator performs complete parsing and type checking of VAL programs; the interpreter is written to faithfully reflect the intended semantics of VAL without attempting to achieve high performance. Together, the translator and interpreter are a reference standard for the syntax and semantics of VAL. Other implementations of VAL will be tested against this standard. Also, the translator will serve as the first phase of transforming VAL programs into program graphs and thence to data flow machine code for execution on a prototype data flow machine.

The VAL interpreter has facilities to assist in debugging functional programs. In a VAL program, execution errors do not terminate execution, and operations are not strictly ordered in a sequence. Hence, there is no concept of a "next" instruction in the sequence, nor is there a concept of the "state" of the program. The common debugging technique of stopping a program at a certain instruction and examining the state does not apply. Instead, certain function calls or operations may be "traced." When a traced function call or operation occurs, diagnostic information is produced giving the operands of that operation. The VAL interpreter is capable of tracing any particular function or operator, or all operators, or all operator invocations that produce error values.

2.2. Data Flow Graph Generation

To be executed on a data flow computer, a VAL program must be transformed from the source language into a data flow graph and then into instruction cell codes. The existing VAL translator implements only the first step of this process. The design of programs to convert the output of the VAL translator to data flow graphs is underway. The best machine program structure depends on the capacity, instruction repertoire, and other properties of the target machine. Among the transformations to be considered in constructing data flow graphs are:

1) Direct expansion of function bodies wherever they are called, if the target machine does not support dynamic function instantiation as is the case for the prototype machines being evaluated. If a function is recursive, the recursion may be converted into an iteration. The exact nature of this conversion can be varied to achieve the desired execution time/space trade-off.

2) Explicit iterations can be partially converted into recursions to change the time/space trade-off.

3) Record and array data values can be expanded into sets of scalar values to achieve better execution speed. This is especially important in the case of arrays constructed by **forall** expressions.

Use of these transformations permits a trade-off between time and space consumed during program execution. We expect they will be applied with advice from the programmer or analyst to achieve the desired performance for a particular application.

Work is proceeding on a very preliminary version of the graph generating program for use in prototype evaluations. The goal is to be able to translate small to medium-size VAL programs for execution on a prototype machine.

In her graduate thesis [14], L. B. Montz explores the translation of a subset of VAL to data flow graphs. The major problem in performing this translation for the target machine, the Dennis-Misunas data flow computer [10], stems from the requirement that graph execution sequences place at most one value on a data flow arc at any time. Placing more than one token on an arc leads to nondeterminacy or possible deadlock as a result of values queueing up in the distribution network of the machine and blocking other values from reaching their destinations.

The *data/acknowledge arc pair transformation* is introduced as a means of implementing the required operational behavior. The transformation replaces graph arcs with initialized d/a arc pairs which hold either a data or acknowledge token depending respectively on the full or empty state of its corresponding arc. A formal argument in the thesis establishes that the *safe* operation resulting from the transformation is guaranteed, and that the *liveness* and functionality of the graph is not altered.

The more interesting part of this research focused on an examination of the d/a arc pair transformation to determine the cost of the scheme and the inefficiencies introduced. The obvious increase in overhead along with a potential loss of some concurrency inspired the development of two optimization techniques to be performed on transformed graphs.

An optimization to eliminate unneeded acknowledge arcs aims to decrease the acknowledge scheme overhead. By identifying situations in which particular arcs do not depend on an acknowledgment to prevent multiple token occurrences, the number of acknowledge arcs can be reduced. This is accomplished by analyzing the data flow graph implementation of each VAL construct to find arc pairs that may be subject to acknowledge arc removal, and applying rules which enable these situations to be recognized. The optimization to balance token flow aims to eliminate potential bottlenecks within a graph by buffering arcs with identity

operators so that all paths through the graph are of equal length. Analysis of performance shows that this approach maximizes throughput, but at a potentially high cost in terms of identity operations. Consequently, the possibility of decreasing the severity of bottlenecks through a limited buffering scheme is explored.

2.3. Id Language Implementation

To permit the development and testing of Id programs, a translator for a subset of Id, called Id0, has been developed. Id0 includes blocks, conditional and loop expressions, procedure definitions and applications, and structure operations. Streams, data flow managers, non-determinate computation, and programmer defined data types are not yet supported. The translator is written in MACLISP and translates Id0 programs into MACLISP programs. It uses LALR(1) parsing and syntax directed translation. Parsing tables for the translator were generated using YACC [11]. Compiled MACLISP code for the translator takes about 10K of memory on ITS (DEC-10).

A translator for the complete Id language is being developed by Keshav Pingali in his graduate thesis. It involves the implementation of streams and the nondeterministic merge operator. Id can also be looked upon as a language for distributed computation on a network of sequential processors, communicating by means of streams. The implementation of streams in this context is also being looked into.

2.4. I-structures

I-structures [4] are array-like data structures with certain constraints on their creation and use. Essentially, an element of an I-structure once generated can never be modified. An I-structure can be implemented efficiently using a storage system that permits writing into a cell only if it is empty and permits reading a cell only if it contains data. Such a storage can be built by associating a *presence bit* with every cell. An attempt to read an empty cell will cause the read operation to be deferred.

I-structures offer as much asynchrony as streams [3, 16] and at the same time preserve the ease of coding implicit in array manipulation. Since elements of I-structures are never modified, their decomposition and the mapping of programs using them is far easier than that of programs using generalized structures.

3. MAPPING OF ID PROGRAMS ONTO A DATA FLOW MULTIPROCESSOR

A machine comprising many processing elements must have a highly distributed and asynchronous control structure. We are designing a data flow computer in which each processing element contains part of an Id program, and processors communicate by sending information packets to each other. Our machine is a hardware realization of a novel way of interpreting data flow languages known as the U-interpreter.

3.1. Program Decomposition for a Multiple Processor System

We are studying a general scheme for decomposing programs for multiple processor systems. In a multiple processor system, each processor can be viewed as executing its own set of instructions and exchanging information with other processors as needed. The success of such systems depends upon the ability to decompose a program into small segments, each suitable for execution on one processor. It has been shown in [3] that high-level data flow programs can be mapped dynamically onto a set of asynchronously cooperating processors. In many cases, however, the cost and overhead of fully general dynamic mapping may be unwarranted. A static mapping of programs may prove to be more efficient and cost effective. Since programs written in applicative languages are based on the concept of values, they are easier to decompose than programs written in languages based on the concept of updating storage cells, e.g., FORTRAN.

Applicative programs that have loops as their primary control structure and that operate on bounded size data structures can be decomposed in three steps:

1) The nested loop structures are unrolled into the network of *computation cells*. A computation cell can be regarded as a virtual processor to which a program and local data have been assigned.

2) Data structure elements are assigned to cells. This assignment should be such that the distance between the PE (Processing Element) that creates the structure and the PE that uses the structure is small.

3) The network of computation cells is mapped onto the actual processors of the system according to the size and structure of both the network and the computer system.

3.2. Mapping Programs onto the U-interpreter Machine

The problem of mapping programs onto the U-interpreter machine [3] is that of assigning activities and data structures to PEs. A U-interpreter machine can be viewed as a collection of PEs, each having its own local memory and communicating with each other through a packet communication network. To reduce the communication overhead, instructions corresponding to an enabled activity should always be available in the local memory of a PE. This is achieved by statically distributing the code onto PEs. The U-interpreter assigns unique activity names of the form *u.c.s.i* to each activity. The set of tokens with the same *u.c* part can be thought of as belonging to a *logical domain*. A logical domain is assigned to a group of PEs in which intragroup communication distances are as short as possible. Such a group of PEs constitutes a *physical domain*. Several logical domains can be mapped onto a physical domain. Activities within a physical domain are mapped based upon either *s* or *i* or both. A physical domain can be divided into several physical *subdomains*, which are characterized by the fact that code is never duplicated inside a subdomain. A mapping based upon statement number *s* creates exactly one subdomain, whereas a mapping based upon initiation count *i* creates as many subdomains as the number of PEs in the physical domain. In a mapping based on *s* and *i* both, a subdomain is chosen on the basis of *i* while the PE within a subdomain is chosen on the basis of *s*. The distribution of code among processors for any of these mappings can be done statically.

4. TOWARD PRACTICAL DATA FLOW MACHINES

We are building a system for evaluating proposed data flow computers [9] as a basis for extrapolating the cost/performance of proposed architectures, and for developing a methodology for data flow program preparation. A form of data flow computer of particular interest is the cell block machine [8] which has evolved from the ideas of Dennis and Misunas [10].

Proposed data flow machines will be emulated using two types of hardware module: a processing unit (PU) and a communication module. The PU uses a standard hardware unit containing a microprocessor, which can be (micro)-programmed to emulate various units of data flow systems such as a cell block or a processing element. The communication element will be a 2 x 2 router from which routing networks as large as needed can be built. The 2 x 2 router performs such a basic function that a direct realization of its function will be used. A variety of data flow computer architectures may be realized by appropriate assembly of these modules. Our first engineering model will be a cell block machine realized using four PUs and four routing modules.

4.1. Processing Unit

G. A. Boughton has completed a detailed logic design for the PU. The primary data paths are 8 bits wide and are implemented using bit-sliced microprocessor chips. The design includes two input packet ports and two output packet ports. These asynchronous ports allow the PU to be connected to a packet routing network and to send and receive packets. The design has a writable control store and a separate data memory.

The design also includes a bus interface which allows the PU to be connected to the bus of an external supervisory minicomputer. The supervisor will be used to load programs and data into the PU, to control the operation of the PU, and to perform maintenance tests on the PU. The interface gives the supervisor access to the data memory and the control store of the PU, and gives the supervisor the capability to halt and single step the PU as well as directly access all the registers of the PU. All the PUs of a prototype machine can be interfaced to the same supervisor and be selectively addressed by it.

W. B. Ackerman has designed a symbolic assembly language for the PU microcode, and has also written a programming manual [1] for the PU which describes the operations of the machine at a level appropriate for assembly level programmers. The assembly language permits programming in a manner similar to assembly language programming on ordinary computers. S. V. Kwong has written an assembler for the PU assembly language. A simulator of the PU has been written by T. L. Tung. The assembler and simulator are written in CLU and run on the MIT-XX DEC-20 computer.

4.2. 2 x 2 Router

A 2 x 2 router receives packets at its two input ports and delivers each received packet at one of two output ports according to the destination address carried by the packet. Each packet is transmitted byte-serially. Packet bytes are delivered and received using an asynchronous packet communication protocol.

J. E. Lilienkamp has completed his undergraduate thesis [12] on the implementation of a 2 x 2 router using SSI and MSI components, giving a complete logic design for the router and developing tests for verifying the correct behavior of the router. The design is based on the earlier work of T. L. Tung and uses seven component modules: two master modules, two FIFO modules, two arbiter modules, and one multiplexer module. The FIFO modules are sixteen word first-in first-out buffers. Each master module is an input port controller which examines the first byte of the packet and generates a request to the appropriate output buffer. Each arbiter grants mutually exclusive use of an output port. The multiplexor module is responsible for linking input ports to output ports.

The router design sketched above is an asynchronous self-timed system and requires verification procedures which are significantly different from those used in synchronous systems. Lilienkamp's thesis develops special verification procedures for each of the modules of the design. In addition, his thesis describes a verification procedure for the router as whole.

4.3. Routing Networks

G. A. Boughton has continued the study of routing networks. One issue that has been examined is the impact of anticipated advances in integrated circuit technologies on the design of routing networks. Two models of integrated circuit technologies have been developed. These models correspond to the anticipated state of available circuit technologies at two points in the future. The first model is similar to the VLSI model of Thompson [15]. It assigns a cost to a wire which is proportional to the wire's length, but it assumes that the propagation delay of a wire is independent of the wire's length. This model corresponds to the anticipated behavior of VLSI technologies of the near future. The second model uses the same cost function for wires, but it assumes that the propagation delay of a wire is proportional to the square of the wire's length. This model corresponds to the anticipated behavior of extremely dense VLSI technologies where the propagation of a signal down a wire is limited by a phenomenon similar to diffusion.

The characteristics of the models imply certain limits on the cost and performance of routing networks. For example, the cost of wires in the models implies certain minimal costs for networks. Present research indicates that there are particular applications for N-input routing networks which cannot be supported in the models by any network with less than $O(N^2)$ area. Similarly, the propagation delay of wires in the second model implies certain constraints on the average delays of networks. Present research indicates that there are particular applications for N-input networks which cannot be supported in the second model by any network with an average delay for packets of less than O(N).

Further study of these topics will be done. In each of the two models, the relation between the characteristics of an application and the cost and performance of a network to support that application will be examined in more detail. The implication of such relations on the design of routing networks for technologies which correspond to the models and on the overall design of packet communication systems for such technologies will be examined.

5. LOGIC SIMULATION OF MOS LSI

A logic simulator, MOSSIM [6], has been developed for modeling metal-oxide semiconductor, large-scale integrated circuit designs. In contrast to conventional logic simulators which model a system in terms of Boolean logic gates, this new simulator directly models the network of field-effect transistors. This approach provides a much more accurate and consistent simulation for those portions of a system which do not follow the Boolean gate model, such as pass transistor logic, dynamic memory, and bus structures. Furthermore, the logic network for this simulator can be extracted directly from the layout specification by a relatively straightforward computer program such as the one written by C. Baker [5]. Hence, the simulator catches errors in the layout as well as in the logic design. With these design tools debugging an LSI design becomes much like debugging a computer program: the layout analyzer "compiles" the design into a transistor network, and then the simulator allows an interactive testing and monitoring of the design. MOSSIM and its offspring have been tested on a variety of MOS designs including a LISP microprocessor chip with over 10,000 transistors. Recent improvements will allow even larger projects to be simulated quickly and efficiently as well as allow the user to describe portions of the design at a more functional level. The algorithms used in MOSSIM have been improved in generality, accuracy, and speed. In addition, techniques for verifying the correctness of the algorithms in terms of an abstract MOS logic model are being developed.

References

1. Ackerman, W. B. "Processing unit programming manual," Computation Structures Group Memo 192, MIT, Laboratory for Computer Science, Cambridge, Ma., April 1980.

2. Ackerman, W. B. and Dennis, J. B. "VAL—A Value-oriented Algorithmic Language: Preliminary reference manual," MIT/LCS/TR-218, MIT, Laboratory for Computer Science, Cambridge, Ma., June 1979.

3. Arvind, Gostelow, K. P., and Plouffe, W. E. "An asynchronous programming language and computing machine," TR114a, Dept. of Information and Computer Science, University of California, Irvine, Ca., December 1978.

4. Arvind and Thomas, R. E. "I-Structures: An efficient data type for functional languages," to appear as MIT/LCS/TM, MIT, Laboratory for Computer Science, Cambridge, Ma.

5. Baker, C. "Artwork analysis tools for VLSI circuits," MIT/LCS/TR-239, MIT, Laboratory for Computer Science, Cambridge, Ma., June 1980.

6. Bryant, R. "MOSSIM: A logic-level simulator for MOS LSI, user's manual," IC Memo 80-7, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., January 1980.

7. Bryant, R. "Report on the Workshop on Self-Timed Systems," MIT/LCS/TM-166, MIT, Laboratory for Computer Science, Cambridge, Ma., May 1980.

8. Dennis, J. B. "The varieties of data flow computers," Proceedings of the First International Conference on Distributed Computing Systems, Huntsville, Al., October 1979, 430-439. Also Computation Structures Group Memo 183-1, MIT, Laboratory for Computer Science, Cambridge, Ma., December 1979.

9. Dennis, J. B., Boughton, G. A., and Leung, C. K. C. "Building blocks for data flow prototypes," Proceedings of the 1980 Symposium on Computer Architecture, La Baule, France, May 1980. Also Computation Structures Group Memo 191, MIT, Laboratory for Computer Science, Cambridge, Ma., February 1980.

10. Dennis, J. B. and Misunas, D. P. "A preliminary architecture for a basic data-flow processor," Proceedings of the Second Annual Symposium on Computer Architecture, January 1975, 126-132. Also Computation Structures Group Memo 102, MIT, Laboratory for Computer Science, Cambridge, Ma., August 1974.

11. Johnson, S. C. "YACC—yet another compiler compiler," CSTR 32, Bell Labs, Murray Hill, N.J., 1975.

12. Lilienkamp, J. "The development of a prototype router: Design, implementation, and test patterns," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma. May 1980. Also Computation Structures Group Memo 199, MIT, Laboratory for Computer Science, Cambridge, Ma., September, 1980.

13. McGraw, J. R. "Data flow computing: The VAL language," Computation Structures Group Memo 188, MIT, Laboratory for Computer Science, Cambridge, Ma., January 1980.

14. Montz, L. B. "Safety and optimization transformations for data flow programs," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., January 1980. Also MIT/LCS/TR-240, MIT, Laboratory for Computer Science, Cambridge, Ma., January, 1980.

15. Thompson, C. D. "A complexity theory for VLSI," Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, Pa., 1980.

16. Weng, K.-S. "Stream-oriented computation in recursive data flow schemas," MIT/LCS-TM-68, MIT, Laboratory for Computer Science, Cambridge, Ma., October 1975.

Publications

1. Ackerman, W. B. "Processing unit programming manual," Computation Structures Group Memo 192, MIT, Laboratory for Computer Science, Cambridge, Ma, April 1980.

2. Brock, J. D. and Montz, L. B. "Translation and optimization of data flow programs," Proceedings of the 1979 International Conference on Parallel Processing, August 1979, 46-54. Also Computation Structures Group Memo 181, MIT, Laboratory for Computer Science, Cambridge, Ma., July 1979.

3. Bryant, R. E. "Simulation on a distributed system," Proceedings of the First International Conference on Distributed Computing Systems, Huntsville, Al., October 1979, 544-552. Also Computation Structures Group Memo 182, MIT, Laboratory for Computer Science, Cambridge, Ma., July 1979.

4. Dennis, J. B. "The varieties of data flow computers," Proceedings of the First International Conference on Distributed Computing Systems, Huntsville, Al., October 1979, 430-439. Also Computation Structures Group Memo 183-1, MIT, Laboratory for Computer Science, Cambridge, Ma., December 1979.

5. Dennis, J. B., Boughton, G. A., and Leung, C. K. C. "Building blocks for data flow prototypes," Proceedings of the 1980 Symposium on Computer Architecture, La Baule, France, May 1980. Also Computation Structures Group Memo 191, MIT, Laboratory for Computer Science, Cambridge, Ma., February 1980.

6. Dennis, J. B. and Weng, K.-S. "An abstract implementation for concurrent computation with streams," Proceedings of the 1979 International Conference on Parallel Processing, August 1979, 35-45. Also Computation Structures Group Memo 180, MIT, Laboratory for Computer Science, Cambridge, Ma., July 1979.

7. Isaman, D. L. "Data-structuring operations in concurrent computations," MIT/LCS-TR-224, MIT, Laboratory for Computer Science, Cambridge, Ma., October 1979.

8. Kosinski, P. "Denotational semantics of determinate and non-determinate data flow programs," MIT/LCS/TR-220, MIT, Laboratory for Computer Science, Cambridge, Ma., July 1979.

9. Leung, C. K. C. "ADL: An architecture description language for packet communication systems," Proceedings of the 1979 International Symposium on Computer Hardware Description Languages and Their Applications, Palo Alto, Ca., October 1979, 6-13. Also Computation Structures Group Memo 185, MIT, Laboratory for Computer Science, Cambridge, Ma., October 1979.

10. McGraw, J. R. "Data flow computing: The VAL language," Computation Structures Group Memo 188, MIT, Laboratory for Computer Science, Cambridge, Ma., January 1980.

Theses Completed

1. Chien, A. "Structuring the fast Fourier transform for data flow computation," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., May 1980.

2. Isaman, David L. "Systems of data structuring operations for parallel processors," Ph.D. dissertation, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., July 1979.

3. Lilienkamp, J. "The development of a prototype router: Design, implementation, and test patterns," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., May 1980.

4. Montz, L. B. "Safety and optimization transformations for data flow programs," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., February 1980.

5. Ries, P. S. "A VLSI implementation of a two by two packet router," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., May 1980.

6. Tucker, R. "Implementation of arithmetic for the data flow processing unit," S.B. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., June 1980.

Theses in Progress

1. Bryant, R. "A switch-level simulation model of integrated logic circuits," Ph.D. dissertation, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., expected February 1981.

2. Leung, C. K. C. "Fault tolerance in packet communication computer architecture," Ph.D. dissertation, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., expected September 1980.

3. Pingali, K. "Streams and data flow: Implementation issues," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., expected February 1981.

4. Singh, N. "A design methodology for self-timed systems," S.M. thesis, MIT, Department of Electrical Engineering and Computer Science, Cambridge, Ma., expected February 1981.

Talks

1. Arvind. "Data flow: An alternative to von Neumann language and architecture,"

   Dartmouth College, Hanover, N. H., July 30, 1979;
   Chalmers University of Technology, Gothenburg, Sweden, January 7, 1980;
   Imperial College, University of London, London, England, January 14, 1980.

2. Arvind. "Consideration for the design of a partial differential equation machine," University of Utah, Salt Lake City, Ut., September 14, 1979.

3. Arvind. A series of five lectures delivered in Tokyo, Japan at the invitation of Japan Electronic Industry Development Association.

   "Data flow: An alternative to von Neumann language and architecture," December 10, 1979;
   "Data flow languages," December 11, 1979;
   "The U-interpreter and its implications for computer architectures," December 12, 1979;
   "Streams and database problems," December 13, 1979;
   "Data flow concepts applied to partial differential equation simulations," December 14, 1979.

4. Arvind. "A suggestion for the design of a VLSI processor for a data flow machine,"

   Nippon Electronic Company, Tokyo, Japan, December 12, 1980;
   Fujitsu Limited, Tokyo, Japan, December 13, 1980.

5. Arvind. "Streams and database problems," Chalmers University of Technology, Gothenburg, Sweden, January 8, 1980.

6. Arvind. "A multiple processor data flow architecture,"

   Workshop on Implications of VLSI, Lumley Castle, U.K., April 16, 1980;
   MIT, Cambridge, Ma.. May 19, 1980.

7. Arvind. "Design considerations for a partial differential equation machine," Annual meeting of Society for Industrial and Applied Mathematics, Alexandria, Va., June 5, 1980.

8. Dennis, J. B. "Methodologies for design," Workshop on Self-Timed Systems, MIT Endicott House, Dedham, Ma., July 8-12, 1979.

9. Dennis, J. B. "Data flow computer architecture," IBM Corporation, Cambridge, Ma., August 10, 1979.

10. Dennis, J. B. "The varieties of data flow computers," First International Conference of Distributed Computer Systems, Huntsville, Al., October 4, 1979.

11. Dennis, J. B. "Data flow computer architecture," IBM Corporation, Rochester, Mn., March 10, 1980.

12. Dennis, J. B. "Research in data flow computing," Harvard University, Cambridge, Ma., March 19, 1980.

13. Dennis, J. B. "Mapping PDE computations onto data flow computers," ICASE Workshop on Array Architecture for Computing in the 80's and 90's, Hampton, Va., April 29, 1980.

14. Dennis, J. B. Panel session: "Future impact of high-level architecture," International Workshop on High-Level Language Computer Architecture, Fort Lauderdale, Fl., May 27, 1980.

15. Leung, C. K. C. "Fault tolerance in self-timed hardware systems," Workshop on Self-Timed Systems, MIT Endicott House, Dedham, Ma., July 8-12, 1979.

16. Leung, C. K. C. "An approach to the design and implementation of self-timed hardware systems," Workshop on Self-Timed Systems, MIT Endicott House, Dedham, Ma., July 8-12, 1979.

17. Montz, L. B. "Translation and optimization of data flow programs," International Conference on Parallel Processing, Bellaire, Mi., August 21, 1979.