MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE

Computation Structures Group Memo 173

Computation Structures Group Progress Report
July 1, 1977 - June 31, 1978

This research was supported in part by the University of California Lawrence Livermore Laboratory under contract No. 8545403 and in part by the National Science Foundation under grant DCR75-04060.

February 1979

COMPUTATION STRUCTURES GROUP

COMPUTATION STRUCTURES

Academic Staff

J. B. Dennis, Group Leader

Research Staff

W. B. Ackerman

D. P. Misunas

Graduate Students

W. B. Ackerman
K. Amikura
S. A. Borkin
G. A. Boughton
J. D. Brock
R. E. Bryant
D. J. Ellis

D. L. Isaman
P. R. Kosinski
C. K. C. Leung
G. S. Miranker
L. B. Montz
K. S. Weng

Undergraduate Students

A. M. Feridun
D. Hirschman

M. E. McNally
D. R. Nadler

Support Staff

A. L. Rubin

COMPUTATION STRUCTURES

A. INTRODUCTION

Research in the past year has been directed toward developing a high level language compatible with the concepts of data flow, studying the types of programs which will ultimately run on a data flow machine, and analyzing data flow architectures in light of these application studies.

B. PROGRAMMING LANGUAGE DEVELOPMENT

A major effort of the Computation Structures Group over the past year has been the development of the programming language VAL (for Value-oriented Algorithmic Language). VAL is both a high level user language and a "base language" for the Form 2 data flow computer which supports high speed, concurrent execution of programs, including the storage and manipulation of data structures.

The structure of VAL reflects the characteristics of the Form 2 data flow computer. Its value oriented nature and freedom from side effects meet the requirements of data flow, so VAL programs can be directly translated for efficient execution. The array and record operations of VAL are in correspondence with the elementary data structure operations of the form 2 machine, and the forall construct of VAL allows efficient implementation of parallel computations on arrays.

Unlike most conventional languages, VAL is value-oriented rather than object-oriented. That is, each argument or result of an operation is a mathematical value; understanding the meaning of a VAL program requires no reference to the state of any conceptual "global memory." All computation takes place through functional application on these values. The only effect of a functional application is the computation of result values from argument values--there are no "side effects." As a result, there is no need to impose sequencing constraints in evaluating VAL expressions other than those implied by data dependencies.

The design of an applicative, side-effect free language that can handle arrays and records requires a redefinition of array and record manipulations in terms of operations (functions) instead of the more traditional statements. The append operator in VAL returns an array or record similar to its argument, but with a specified element changed to a specified value. Because there is no conceptual global memory in which the array or record resides, the append operation has no side-effects.

The properties of VAL that support efficient data flow computation are precisely those that are considered desirable for structured programming: freedom from side-effects, absence of "GOTO" statements, complete compile-time type checking, and clean interfaces between program modules. While VAL is not the first value-oriented programming language (pure LISP [2] and LUCID [1] are earlier examples), we believe

it to be the first which is seriously intended for writing large scale programs for efficient numerical computation on high performance machines.

PROGRAM EXAMPLES

Array elements are accessed in a manner similar to conventional languages.

A(J) selects element J of array A

The "subscripted assignment" operation of conventional languages is replaced by the append operator in VAL:

A[J : S] returns an array similar to A, but with element J changed to S

Instead of the conditional command:

```
if A > 0 then B := A
         else B := -A
```

that is used in many conventional languages, VAL uses a conditional expression:

```
B := if A > 0 then A
             else -A
```

Functions are defined by a header giving the function name, formal paramenter names and types, and returned value types, and a body, which is a multi-expression defining the tuple of values to be returned. Because of the applicative nature of VAL, functions have no side effects and no "own" variables. A function invocation has access only to its arguments and other functions.

For sequential iterations, VAL uses the for ... iter block shown in the example below. The word for is followed by declarations and initial values of the loop variables. The iteration body follows. It is a conditional expression with ordinary arms and iter arms. If evaluation of an ordinary arm is called for the iteration terminates with evaluation of that arm (a multi-expression) yielding the result of the entire iteration. If evaluation of an iter arm is called for, the redefinitions following the word iter are performed and the body is evaluated again.

Example (Euclidean Algorithm for greatest common divisor):

```
function gcd(x, y : int returns int)
    for m, n, : int := x, y
    do if m = n then m
          elseif m < n then iter n := mod(n, m)
          else iter m := mod(m, n)
          end
    end
```

```
        end
```

For parallel iterations, the forall block evaluates an expression for each value of an index and either constructs an array containing the results or applies an associative operator such as addition to the results. The forall block is similar to, but more general than, the "vector" operations provided on some supercomputers.

Examples

```
function vector_sum(V, W : array[real] returns array[real])
    forall J in [1, N]
        construct V(J)+W(J)
        end
    end

function inner_product(V, W : array[real] returns real)
    forall J in [1, N]
        eval plus V(J)*W(J)
        end
    end
```

C. DATA FLOW COMPUTER ARCHITECTURE

Our concepts for the design of a computer capable of high performance execution of data flow programs have evolved over a period of five years [3,4,5]. Current plans envision four "forms" of data flow machines with different levels of capability:

Form 1:

Statically-loaded programs; scalar values only

Form 2:

Statically-loaded programs; scalar values and data structures

Form 3:

Dynamically-loaded programs (i.e. caching of instructions); scalar values and data structures

Form 4:

Dynamically-loaded programs; scalar values, data structures, and streams.

1. Form 4 Processor Architecture

As a part of continuing research on the design of generalized forms of data flow processors, K. Weng has studied the feasibility of extending the Form 2 architecture to

support procedure activations, streams of data [6], and forall constructs as part of his Ph.D. research [7].

Mechanisms for procedure activation are essential for supporting general purpose computations and are the basis for generalizing data flow architecture concepts to apply to computer systems that serve communities of users. The notion of streams extends the data flow semantics to include the class of computations which are history sensitive--in the sense that the behavior of a computation is characterizable as a function from sequences of input values to sequences of output values. This form of computation conventionally has been expressed in languages which either have side-effects or requires explicit synchronization primitives. One important characteristic of computations expressed with streams is that the inherent concurrency is not lost, yet they are guaranteed determinate if no explicit non-determinate primitives are used. The forall constructs are intended for expressing concurrent operations on data structures. Since this form of concurrency is very often found in many numerical applications, they are useful language features.

For one form of such a data flow processor, Weng has chosen recursive data flow schemas [6] (represented by acyclic directed graphs) as the basic model of computation. This processor has several advantages over a processor that supports cyclic data flow schemas: the absence of cycles makes the acknowledge signals required in our Form 2 architecture unnecessary, and the use of recursive data flow schemas enhances the asynchrony of computation. The implementation of stream data types and forall-based computation on this form of data flow processor is under investigation.

Two important subsystems of the generalized form of data flow processor require further study. The Instruction Memory must support a logical address space which is much larger than that of the Form 2 architecture, and the design of the Packet Memory must support efficient storage of large data structures some substructures of which may be accessed at much higher rate than others.

2. Fault-Tolerant Data Flow Processor Design

Consider a Form 1 data flow processor consisting of an instruction memory and a processing section connected by two routing networks. Instructions are stored in the instruction memory, packets are routed through the networks and data operations are performed by functional units in the processing section. All these modules must be capable of carrying out their specified logical functions to execute a data flow program successfully. Work on fault-tolerance investigates methods of designing and implementing these modules so that a data flow processor will not be disabled due to the occurrence of a small number of isolated physical failures distributed among its modules.

Experience in fault-tolerant computer design indicates that a cost-effective

approach would probably apply coding techniques in the routing networks and in the memory modules of the instruction memory, and apply modular redundancy in the control section of the instruction memory. Design of the functional units has not been specified in sufficient detail to favor any specific technique at this point.

In his doctoral research [8], C. Leung is studying the problems of applying modular redundancy and coding techniques to design a fault-tolerant data flow processor, assuming a packet communication architecture for the hardware. A packet communication architecture organizes the hardware into modules which communicate only by sending packets to each other using an asynchronous hand-shake protocol. A data flow processor so organized is an interconnection of self-timed modules to which fault-tolerant mechanisms which assume system events are coordinated by global timing pulses, are not directly applicable.

By studying the failure modes of packet communication modules and techniques for constructing and connecting them, a fault model for these modules has been constructed. In this model a failed module appears to other modules as either generating erroneous packets or having halted its packet processing activity and having stopped generating packets or acknowledging inputs. A methodology to construct modular redundancy configurations under this fault model is being developed. In this methodology module copies are assumed to be *performance compatible*, i.e., their difference in processing speed can be bounded. Using performance compatible module copies, modular redundancy configurations are designed to be *consistent* and to guarantee *in-phase* operation, in the absence of failures. Consistency is achieved when all copies of a module receive the same input packet sequence. During in-phase operation, all copies of a module receive the same input packet within a fixed time interval. When failures occur, data errors are masked by voting. Techniques are being developed to support consistent, in-phase operation, even in the presence of control faults that directly affect the packet transmission protocol.

Leung's work on fault-tolerance does not place any restriction on failure modes at the logic gate level, assumes an asynchronous operation mode and also attempts to deal with failures in non-determinate packet communication modules, such as arbitration units.

3. Data Structure Processing

W. Ackerman has developed the design of a data structure processing facility for a data flow computer [9,10]. The handling of arrays is completely "pure" or value-oriented, as required by the data flow concept. Implementation of such structures requires that they sometimes must be partially copied when an update takes place. The proposed facility is designed to reduce this copying to a minimum, and proper design and translation of programs should make it possible to handle structures efficiently. The structures are stored in a *packet memory*, which operates in accordance with the principles of packet communication used throughout the computer. The

memory and processing units are designed so there are no bottlenecks anywhere in the system, hence the processing rate can be increased virtually without limit by increasing the size of the computer.

4. Routing Network Design

G. A. Boughton has continued his work on the design of packet routing networks [11]. A routing network is a packet communication system with designated inputs and outputs. Each tagged packet placed on an input is routed to the output corresponding to its tag. The design of an arbitrary routing network can be achieved with the use of two simpler networks which we call concentration and connection networks.

A concentration network has more inputs than outputs and has the property that each accepted packet is eventually placed on an output but is not necessarily routed to a particular output. Our work has shown that a concentration network which will accept packets distributed arbitrarily over its inputs at a rate equal to its total output capability can be constructed with O(N log N) modules where N is the number of inputs and N/a for some positive integer a is the number of outputs. A further result indicates that this complexity is within a constant factor of the optimum. The key to this construction is a network which evenly distributes over its outputs all packets placed on its inputs. Such a distribution network can be constructed recursively from alternator modules. An alternator module divides the packets available on its two input ports between its two output ports. The first stage of the distribution network consists of N/2 alternator modules connected to pairs of system input ports. This stage divides the flow of packets equally between two sets of outputs. Each of these sets of outputs then becomes a set of inputs to a smaller distribution network.

A connection network is a routing network with the same number of inputs as outputs. Our work has shown that connection networks which have probabilistically high throughput can be constructed using switches and concentration networks. The present goal is the discovery of simpler connection network structures with equivalent performance.

D. HARDWARE IMPLEMENTATION

1. Introduction

To further our goal of implementing a data flow computer, we have studied the feasibility of building some of the key components. In earlier work K. Amikura [12] described an implementation of an instruction cell block. This work has now been supplemented by studies of other parts of the system.

2. Functional Units

In his bachelor's thesis [13], A. Feridun studied the design of floating-point

arithmetic processors for operands transmitted byte-serially. By using *on-line* arithmetic methods [14,15], designs of pipelined processors are possible in which bytes are processed from the most significant byte to the least, with each result byte being generated after a fixed number of input bytes have been received. Functional units capable of byte-serial, on-line processing are very attractive for a data flow processor in which all packets are transmitted byte-serially, especially if they can support multiple precision arithmetic.

On-line methods require numbers to be encoded in a redundant number representation where the size of the digit set is greater than the radix. Algorithms have been developed to add two numbers such that any carry digit generated during the addition of two digits only propagates one digit to the left. Similar algorithms exist for multiplication and division [16].

Based on these algorithms, architectures for a floating-point adder-subtractor and a floating-point multiplier have been developed. A floating-point number is represented by a number of bytes, each consisting of several digits. Digits of the operands are processed in order of decreasing significance. Result digits can be generated, most significant digits first, before all operand digits are available. Each arithmetic unit is decomposed into submodules. Functions implemented in these submodules include exponent adjustment and manipulation, addition or negation followed by addition for subtraction, multiplication and normalization. Data and control flow between submodules of each unit are specified using Petri nets.

3. Routing Network Design

In a recently completed senior thesis, M. McNally [17] designed an arbitration network (one of the two routing networks of a data flow computer). The design is at the level of individual integrated circuits and other components, and uses commercially available components throughout. Asynchronous arbiters [18] are used to ensure hazard-free operation. Serial-to-parallel conversion is performed in several stages to maintain high circuit utilization through the network. The 128-bit packets initially enter the network in 8-bit parallel/16-byte serial format, to minimize the number of data wires required. This is later converted to 32-bit parallel/4-byte serial format and finally to 128-bit fully parallel format.

The number of integrated circuits required to implement an arbitration network with currently available components is quite large, and custom-made LSI devices appear to be required for construction of moderately large data flow computers following this architectural plan.

E. APPLICATION STUDIES

1. Introduction

We have continued to study the properties of programs for which a data flow processor would seem attractive. We have found that many large numerical programs

have a great deal of concurrency which could be exploited by a Form 2 data flow computer.

2. Weather Simulation

In a recently completed senior thesis, D. Nadler has analyzed in detail the execution of the GISS atmospheric simulation algorithm [19,5]. The program requires a large amount of memory for the data structures and is to be executed on a Form 2 data flow computer with data structure facilities. The analysis shows that a data flow computer with about 370K instruction cells and 900K data structure cells, using reasonably fast contemporary component technology, can execute the algorithm 100 times faster than an IBM 360/195, achieving an arithmetic processing rate of 150 million floating point operations per second.

3. Hydrodynamics Simulation

The Lawrence Livermore Laboratory has furnished a program called SIMPLE to be used as a "benchmark" for data flow language and computer evaluation. SIMPLE is a two-dimensional hydrodynamics program, written in Fortran, similar to programs that are regularly run at Lawrence Livermore Laboratory on such machines as the CDC 7600. D. Hirschman has translated SIMPLE into a preliminary version of VAL. The translated program is significantly more concise in many places than the original Fortran code and demonstrates that VAL is capable of expressing serious application programs.

F. THEORETICAL STUDIES

1. Data Flow Language Semantics

The master's thesis of J. D. Brock [20] presents two semantic specifications for a simple, determinate data flow language and proves their equivalence. The operational semantics of a data flow program is obtained by a two-step process. First, the data flow program is translated into a data flow graph. Rules for this translation are given in the thesis. Next, the data flow graph is executed, or rather, the result of executing the data flow graph is formally derived. Because deterministic data flow programs may be translated into determinate data flow graphs, the fixed point methods of Kahn [21] may be used to determine the result of graph execution. The denotational semantics of a data flow program is defined using Scott's theory by directly mapping program language elements into a mathematical domain. The proof of equivalence for the two semantic methods is also a "proof of correctness" for the data flow translation algorithm.

An elegant semantic characterization of determinate systems as mapping input histories into output histories has been defined by Kahn. A theory of similar elegance for non-determinate systems has not yet been found. However, we have demonstrated that non-determinate systems may not be characterized by a naive extension of Kahn's

theory in which systems are represented by mappings from input histories into sets of output histories [22]. Presently, we are defining a theory of non-determinate computation in which systems are characterized by sets of scenarios--pairs of input histories and output histories ordered by a causality relation. Eventually, this theory will be used to define the semantics of non-determinate data flow languages.

2. Program Translation and Data Flow Graph Generation

The master's thesis research of L. Montz [23] deals with safety and optimization transformations for data flow programs. The basic problem encountered occurs in translating data flow graphs to a machine language representation, called an instruction cell configuration. The result of naively translating a properly functioning data flow graph to its corresponding instruction cell configuration can lead to a representation which can deadlock. This situation arises because the firing rules developed for data flow graphs are not enforced in the instruction cell configurations. The problem will be corrected by replacing the firing rules with a more explicit mechanism in the data flow graphs that will extend to the instruction cell configurations.

We approach the solution of this problem from a language based perspective by developing a translation algorithm $\mathcal{T}$ which maps data flow programs written in the high level language VAL, into data flow graphs. The basic algorithm $\mathcal{T}$ which can be found in the master's thesis of J. D. Brock [20] will be accepted as the "standard" and is thus assumed to produce correct data flow graphs. This algorithm will then be modified to $\mathcal{T}_{Q1}$ which produces the same overall graphs but replaces the infinite Q arcs of $\mathcal{T}$ with arcs of length 1, and specifies operation according to the firing rules. $\mathcal{T}_{Q1}$ is then modified to $\mathcal{T}_{d/a}$ which replaces the arcs of length 1 with data/acknowledge pairs of arcs, making explicit the implicit solution of $\mathcal{T}_{Q1}$. The formal tasks are to prove:

1. $\mathcal{T}_{Q1}$ functionally equivalent to $\mathcal{T}$, establishing that while the firing rule provides a theoretical solution to the problem it doesn't change the correctness of the translation.

2. $\mathcal{T}_{d/a}$ functionally equivalent to $\mathcal{T}_{Q1}$, establishing that the d/a mechanism correctly implements the theoretical firing rule solution.

The remainder of the thesis develops methods for optimizing this basic solution.

3. Properties of Data Models for Database Systems

S. Borkin has continued research on the formal equivalence properties of data models for database systems [24,25]. The research is motivated by the desire for database systems which support several different data models.

The past year has seen the completion of his master's thesis which presents a

formal framework for the investigation of such problems. The thesis presents formal definitions of terms such as *database, operation, operation type, application model* and *data model.* Using this formal framework, *database state equivalence, operation equivalence, application model equivalence* and *data model equivalence* are distinguished. Three types of application and data model equivalence are defined--*isomorphic, composed operation* and *state dependent.* Possibilities for *partial equivalences* are mentioned. Implementation implications of these different equivalences are discussed.

In applying this framework to specific data models, it seems desirable to deal with data models which allow the user to reflect more of the semantic features of the application being modeled than those data models currently in popular use. Doctoral research currently in progress defines two *semantic data models* and studies their equivalence properties. The data models defined, the *semantic relation data model* and the *semantic graph data model*, are "semantic versions" of the relational and network data models used in many current database systems.

REFERENCES

1. Ashcroft, E. A., and Wadge, W. W. "Lucid, a Nonprocedural Language with Iteration." Communications of the ACM, Vol. 20 No. 7 (July 1977), 519-526.

2. McCarthy, J., "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I." Communications of the ACM, Vol. 3 No. 4 (April 1960), 184-195.

3. Dennis, J. B., and Misunas, D. P. "A Computer Architecture for Highly Parallel Signal Processing." Proceedings of the ACM 1974 National Conference. New York: ACM, 1974.

4. Dennis, Jack B.; Misunas, David P.; and Leung, Clement K. C. A Highly Parallel Processor Using a Data Flow Machine Language. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 134. Cambridge, Ma., 1976.

5. Dennis, Jack B., and Weng, Kung-Song. "Application of Data Flow Computation to the Weather Problem." Proceedings of the Symposium of High Speed Computer and Algorithm Organization. New York: Institute of Electrical and Electronics Engineers, 1977.

6. Weng, Kung-Song. "Stream-Oriented Computation in Recursive Data Flow Schemas." M.I.T., Laboratory for Computer Science, LCS/TM-68. Cambridge, Ma., 1975.

7. Weng, Kung-Song. "An Abstract Implementation of a Generalized Data Flow Processor." Ph.D Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, January 1979.

8. Leung, Clement. "Fault Tolerance in Packet Communication Computer Architecture." Ph.D Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, January 1979.

9. Ackerman, William B. A Structure Memory for Data Flow Computers. M.I.T., Laboratory for Computer Science, LCS/TR-186. Cambridge, Ma., September 1977.

10. Ackerman, William B. "Structure Processing Facility for Data Flow Computers." To be published in Proceedings of the 1978 International Conference on Parallel Processing. New York: Institute of Electrical and Electronics Engineers.

11. Boughton, George A. "Routing Networks in Packet Communication Architectures." unpublished S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, June 1978.

12. Amikura, Katsuhiko. A Logic Design for the Cell Block of a Data-Flow Processor. M.I.T., Laboratory for Computer Science, LCS/TM-93. Cambridge, Ma., December 1977.

13. Feridun, Arif M. "Design of an On-Line Byte-Level Pipelined Arithmetic Processor." unpublished S.B. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, June 1978.

14. Avizienis, A. "Signed-Digit Number Representations for Fast Parallel Arithmetic." IRE Transactions on Electronic Computers, Vol. EC-10 No. 3 (September 1961), 389-400.

15. Avizienis, A. "Binary Compatible Signed-Digit Arithmetic." Proceedings of the 1964 Fall Joint Computer Conference. Santa Monica, Ca.: AFIPS, 1964, 663-671.

16. Trivedi, K., and Ercegovac, M. "On-Line Algorithms for Division and Multiplication." IEEE Transactions on Computers, Vol. C-26 No. 7 (July 1977), 681-687.

17. McNally, Mary E. "The Design of an Arbitration Network for a Data-Flow Processor." unpublished S.B. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, May 1978.

18. Plummer, W. W. "Asynchronous Arbiters." IEEE Transactions on Computers, Vol. C-21 No. 1 (January 1972), 37-42.

19. Nadler, David R. Data Flow Computer Performance for the GISS Weather Model. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 159. Cambridge, Ma., March 1978.

20. Brock J. Dean. "Operational Semantics of a Data Flow Language." S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, September 1978.

21. Kahn, G., "The Semantics of a Simple Language for Parallel Programming." Information Processing 74. New York: American Elsevier, 1974.

22. Brock, J. and Ackerman, W. An Anomaly in the Specification of Nondeterminate Packet Systems. M.I.T., Laboratory for Computer Science, Computation Structures Group Note 33-1. Cambridge, Ma., January, 1978.

23. Montz, Lynn B. "Safety and Optimization Transformations for Data Flow Programs." S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, January 1979.

24. Borkin, Sheldon A. Data Model Equivalence. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 158. Cambridge, Ma., February 1978.

25. Borkin, Sheldon A. "Data Model Equivalence." To be published in Proceedings of the Fourth International Conference on Very Large Data Bases. New York: ACM.

Publications

1. Ackerman, William B. A Structure Memory for Data Flow Computers. M.I.T., Laboratory for Computer Science, LCS/TR-186. Cambridge, Ma., September 1977.

2. Ackerman, William B. A Structure Controller for Data Flow Computers. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 156. Cambridge, Ma., January 1978.

3. Amikura, Katsuhiko. A Logic Design for the Cell Block of a Data-Flow Processor. M.I.T., Laboratory for Computer Science, LCS/TM-93. Cambridge, Ma., December 1977.

4. Borkin, Sheldon A. Data Model Equivalence. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 158. Cambridge, Ma., February 1978.

5. Bryant, Randal E. Simulation of Packet Communication Architecture Computer Systems. M.I.T., Laboratory for Computer Science. LCS/TR-188. Cambridge, Ma. November 1977.

6. Bryant, Randal E., and Dennis, Jack B. Concurrent Programming. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 148-2. Cambridge, Ma., June 1978.

7. Ellis, David J. Formal Specifications for Packet Communication Systems. M.I.T., Laboratory for Computer Science, LCS/TR-189. Cambridge, Ma., November 1977.

8. Jacobsen, Robert G., and Misunas, David P. "Analysis of Structures for Packet Communication." Proceedings of the 1977 International Conference on Parallel Processing. New York: Institute of Electrical and Electronics Engineers, August 1977, 38-43. Also M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 151. Cambridge, Ma., August 1977.

9. Kosinski, Paul R. "A Straightforward Denotational Semantics for Non-Determinate Data Flow Programs." Proceedings of the 5th Annual Symposium on Principles of Programming Languages. New York: ACM, 1978, 214-221. Also M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 157. Cambridge, Ma., December 1977.

10. Miranker, Glen S. "Implementation of Procedures on a Class of Data Flow Processors." Proceedings of the 1977 International Conference on Parallel Processing. New York: Institute of Electrical and Electronics Engineers, August

1977, 77-86.

11. Misunas, David P. "Report on the Workshop on Data Flow Computer and Program Organization." Computer Architecture News. New York: ACM SIGARCH, Vol. 6 No. 4 (October 1977), 6-22. Also M.I.T., Laboratory for Computer Science, LCS/TM-92. Cambridge, Ma., November 1977.

12. Misunas, David P. A Computer Architecture for Data-Flow Computation. M.I.T., Laboratory for Computer Science, LCS/TM-100. Cambridge, Ma., March 1978.

13. Nadler, David R. Data Flow Computer Performance for the GISS Weather Model. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 159. Cambridge, Ma., March 1978.

Accepted for Publication

1. Ackerman, William B. "Structure Processing Facility for Data Flow Computers." To be published in Proceedings of the 1978 International Conference on Parallel Processing. New York: Institute of Electrical and Electronics Engineers.

2. Borkin, Sheldon A. "Data Model Equivalence." To be published in Proceedings of the Fourth International Conference on Very Large Data Bases. New York: ACM.

3. Bryant, Randal E., and Dennis, Jack B. "Concurrent Programming." To be published in Research Directions in Software Technology. Edited by P. Wegner. Cambridge, Ma., M.I.T. Press.

4. Dennis, Jack B.; Fuller, Samuel H.; Ackerman, William B.; Swan, Richard J.; and Weng, Kung-Song. "Research Directions in Computer Architecture." To be published in Research Directions in Software Technology. Edited by P. Wegner. Cambridge, Ma., M.I.T. Press.

Theses Completed

1. Ackerman, William B. A Structure Memory for Data Flow Computers. S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, September 1977.

2. Amikura, Katsuhiko. A Logic Design for the Cell Block of a Data-Flow Processor. S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, September 1977.

3. Borkin, Sheldon A. Data Model Equivalence. S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, June 1978.

4. Boughton, George A. "Routing Networks in Packet Communication Architectures." unpublished S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, June 1978.

5. Ellis, David J. Formal Specifications for Packet Communication Systems. Ph.D Thesis, M.I.T., Department of Electrical Engineering and Computer Science, February 1978.

6. Feridun, Arif M. "Design of an On-Line Byte-Level Pipelined Arithmetic Processor." unpublished S.B. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, June 1978.

7. Grossman, Steven. "Data Flow Machine Performance for an Airplane Collision Avoidance Algorithm." unpublished S.B. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, August 1977.

8. Jacobsen, Robert G. Analysis of Structures for Packet Sorting Networks. S.B. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, May 1978

9. McNally, Mary E. "The Design of an Arbitration Network for a Data-Flow Processor." unpublished S.B. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, May 1978.

10. Nadler, David R. Data Flow Computer Performance for the GISS Weather Model. S.B. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, January 1978.

Theses in Progress

1. Borkin, Sheldon A. "Equivalence Properties of Semantic Data Models for Database Systems." Ph.D Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, January 1979.

2. Brock J. Dean. "Operational Semantics of a Data Flow Language." S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, September 1978.

3. Isaman, David L. "Systems of Data Structuring Operations for Parallel Processors." Ph.D. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, January 1979.

4. Kosinski, Paul R. "Data Flow Programs and Implementations: A Mathematical Semantics." Ph.D Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, January 1979.

5. Leung, Clement. "Fault Tolerance in Packet Communication Computer Architecture." Ph.D Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, January 1979.

6. Montz, Lynn B. "Safety and Optimization Transformations for Data Flow Programs." S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, January 1979.

7. Weng, Kung-Song. "An Abstract Implementation of a Generalized Data Flow Processor." Ph.D Thesis, M.I.T., Department of Electrical Engineering and Computer Science, expected date of completion, January 1979.

Talks

1. Ackerman, William B. Speaker, Session on Architecture. Workshop on Data Flow Computer and Program Organization, M.I.T., Dedham, Ma., July 1977.

2. Amikura, Katsuhiko. Speaker, Session on Implementation. Workshop on Data Flow Computer and Program Organization, M.I.T., Dedham, Ma., July 1977.

3. Borkin, Sheldon. "Equivalence Properties of Semantic Data Models for Database Systems." M.I.T. Laboratory for Computer Science, Cambridge, Ma., June 29, 1978.

4. Boughton, George A. Speaker, Session on Architecture. Workshop on Data Flow Computer and Program Organization, M.I.T., Dedham, Ma., July 1977.

5. Brock, J. Dean. Speaker, Session on Specification and Verification. Workshop on Data Flow Computer and Program Organization, M.I.T., Dedham, Ma., October 1977.

6. Bryant, Randal E. Speaker, Session on Language Issues and Session on Performance and Simulation. Workshop on Data Flow Computer and Program Organization, M.I.T., Dedham, Ma., July 1977.

7. Dennis, Jack B. Speaker, Welcoming Session. Workshop on Data Flow Computer and Program Organization, M.I.T., Dedham, Ma., July 1977.

8. Dennis, Jack B. Chairperson and Speaker, Session on Research Status and Objectives. Workshop on Data Flow Computer and Program Organization, M.I.T., Dedham, Ma., July 1977.

9. Dennis, Jack B. Speaker, Session on Applications and Session on Language Issues. Workshop on Data Flow Computer and Program Organization, M.I.T., Dedham, Ma., July 1977.

10. Dennis, Jack B. "Data Flow Architecture." Digital Equipment Corporation, Maynard Ma., July 19, 1977.

11. Dennis, Jack B. "Opening Remarks." IFIP Working Conference on Formal Description of Programming Concepts, Saint Andrews, New Brunswick, Canada, August 1, 1977.

12. Dennis, Jack B. "Program Structure and Computer Architecture." ACM Boston Chapter SIGPLAN Meeting, Boston, Ma., October 6, 1977.

13. Dennis, Jack B. Panel Discussion on Application-Directed Research. Conference on Research Directions in Software Technology, Brown University, Providence, R. I., October 12, 1977.

14. Dennis, Jack B. "Data Flow Computer Architecture." Carnegie-Mellon University, Pittsburgh, Pa., December 7, 1977; Computer Science Division, University of California, Berkeley, Berkeley, Ca., March 15, 1978; Purdue University, Lafayette, In., April 4, 1978.

15. Dennis, Jack B. "Language Design for Data Flow Computation." Digital System Laboratory, Stanford University, Stanford, Ca., March 16, 1978.

16. Dennis, Jack B. Panel Discussion on Privacy and Security. Computer Science Division, University of California, Berkeley, Berkeley, Ca., March 16, 1978.

17. Dennis, Jack B. "The Data Flow Concept." Lawrence Livermore Laboratory, Livermore, Ca., March 17, 1978.

18. Ellis, David J. Speaker, Session on Specification and Verification and Session on Specification and Verification. Workshop on Data Flow Computer and Program Organization, M.I.T., Dedham, Ma., July 1977.

19. Ellis, David J. "Formal Specifications for Packet Communication Systems." M.I.T. Laboratory for Computer Science, Cambridge, Ma., November 1977.

20. Kosinski, Paul. Speaker, Session on Language Issues and Session on Specification and Verification. Workshop on Data Flow Computer and Program Organization, M.I.T., Dedham, Ma., July 1977.

21. Kosinski, Paul. "A Straightforward Denotational Semantics for Non-Determinate Data Flow Programs." Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Az., January 23, 1977.

22. Leung, Clement. Speaker, Session on Implementation and Session on Specification and Verification. Workshop on Data Flow Computer and Program Organization,

M.I.T., Dedham, Ma., July 1977.

23. Miranker, Glen S. Speaker, Session on Architecture. Workshop on Data Flow Computer and Program Organization, M.I.T., Dedham, Ma., July 1977.

24. Miranker, Glen S. "Implementation of Procedures on a Class of Data Flow Processors." 1977 International Conference on Parallel Processing, Stony Creek, Mi., August 1977.

25. Misunas, David P. Chairperson, Session on Performance and Simulation. Workshop on Data Flow Computer and Program Organization, M.I.T., Dedham, Ma., July 1977.

26. Misunas, David P. "Analysis of Structures for Packet Communication." 1977 International Conference on Parallel Processing, Stony Creek, Mi., August 1977.