MIT/LCS/TM-136

REPORT ON THE SECOND WORKSHOP ON DATA FLOW COMPUTER AND PROGRAM ORGANIZATION

David P. Misunas

June 1979

Report on the

Second Workshop on Data Flow Computer and Program Organization

David P. Misunas
M.I.T. Laboratory for Computer Science

The following report comprises an edited transcription of presentations made at the Second Workshop on Data Flow Computer and Program Organization, held at M.I.T. on July 9-13, 1978, and co-sponsored by the Lawrence Livermore Laboratory (LLL) and the Department of Energy, Mathematical Sciences Branch. These informal transcriptions are only intended to provide a general picture of ongoing work in the area, and to that end, have been heavily edited and often summarized

The efforts of a number of people greatly aided the generation of this report. In particular, the original version of the bibliography was compiled by Dean Brock and drafts of the report were read and criticized by Dean Brock, Jack Dennis, and Ken Weng.

Any opinions expressed in the transcriptions are those of the speakers and not necessarily those of their institutions or of the sponsoring institutions. The speakers have not had a chance to review the report and to correct any mistranscriptions which may have occurred. We hope that the contributions of the participants have been represented accurately and fairly.

KEY WORDS: Computer Architecture
Computer Systems
Programming Language
Data Flow Computation

TABLE OF CONTENTS

Session 0. Welcome
Jack Dennis, M.I.T.

Welcome to the second Workshop on Data Flow Computer and Program Organization. The first workshop in July 1977 served to acquaint workers in data flow with each other and with the extent of research in this exciting field, both in the United States and in Europe. This year we have 48 participants representing 18 institutions -- a sizable growth from last year. It is good to see the initiation of new projects at several universities in England and at the University of Utah. The next several years will be fascinating to witness as our concepts are refined and realized in practical machines. In the present workshop I expect to see a constructive comparison of the great variety of architectural approaches proposed by different research groups, and there should be many opportunities for discussion and debate of critical technical issues. Let us proceed!

Session 1. Research Status and Goals
Chairperson: Jack Dennis, M.I.T.

I. Jack Dennis, M.I.T.

The M.I.T. research group has been funded for the year 1978 by the Lawrence Livermore Laboratory (LLL) for the purpose of seeing how the concepts of data flow computers and programming languages will mesh with their needs for high performance machines.

We are interested in four forms of data flow machine. The Form 1 machine supports the language concepts of scalar variables, conditionals, and iteration, and is applicable to signal processing problems. Such problems are characterized by relatively small programs and the presence of only a small amount of data at any point in time.

For the LLL applications, we are interested in studying an extension of the Form 1 machine to support data structures. Hence, the main focus of our proposed effort is to demonstrate the feasibility of this Form 2 data flow machine and also to develop a corresponding level of user programming language.

The Form 3 machine is intended to overcome the program size limitations of Forms 1 and 2. To build such a machine to hold large programs, we need to incorporate some concept similar to a cache memory for the instructions of a program. We would like to arrange the machine so the active instructions of a program are held in the cache and the remaining instructions are in some kind of backup memory.

The Form 4 machine is our dream of the ultimate general purpose data flow computer. We would like this machine to stand alone, compile its own programs, run a time-sharing service for many users, and so on. But that's something considerably far in the future.

We have divided our work for the immediate future into six projects. The most important one at the moment is to develop a complete description of a user level programming language. We need this description to allow the Livermore people to apply the language to expressing their computations. This will be a first step in determining how fast our proposed data flow machine will perform in practical applications.

The second project is to build an engineering model. Such a model would be a small-scale Form 1 machine. The purpose of this project would be to check out concepts and to gain experience in the construction of large asynchronous systems.

Project three is the study of data flow machine languages and the problems of translating a high level source language into the machine language. So far, we have only scratched the surface of these translation, optimization, and code generation problems.

Project four is the extension of the Form 1 machine to a Form 2 machine. The issues concern the design of representations for structured data and the mechanisms for the structure processor of the machine.

Project five is the development of an approach to architecture description for use in the design of large, asynchronous computer systems, including methodology for formal specification and verification.

Project six is to develop a specification for a full-scale Form 2 machine.

We hope during the next year to have a reference manual on our programming language. Project two, the engineering model, is a project we hope to complete during the next two years. The rest will follow as it fits in place.

II. Dominique Comte, C.E.R.T. - Toulouse, France

The LAU project began in 1973 at C.E.R.T., Toulouse, France, Department of Computer Science. The project was inspired by the Tesler and Enea paper on single assignment.

In 1973-74, we studied single assignment in a formal way, examining single assignment program schemata and the degree of parallelism achievable in single assignment programs. We also studied different levels of single assignment: the task level, the block level, and the statement level.

In 1975-76, we designed the LAU system which is currently under construction. First, we designed a high level programming language based on single assignment. The goals were that it be easy to use by non-specialists, that it naturally expose the parallelism in an algorithm, and that it be readable and debuggable. We then designed a single assignment machine based on data driven mechanisms. There is no program counter in the machine, an instruction is ready to execute as soon as data is available. We built a simulator in order to evaluate the performance of the LAU multi-processor architecture and have performed significant simulation work on the performance of the machine.

From 1977 to the present, we have been building the prototype machine. Today, some parts of the machine are separately operational. The control unit (approximately 400 ICs), one block of memory (approximately 250 ICs), and one

execution processor based on the AMD 2900 series bit slice processor (approximately 280 ICs) are working. We are in the process of increasing the number of memory blocks to eight and the number of processors to ten.

Machine instructions are three-address instructions, specifying two input operand addresses and one result address. A tag bit field controls the execution of each instruction, specifying the availability of the operands and if the instruction is in an environment which allows its execution. An instruction is executable when all tag bits are set.

The data driven sequencing of the LAU architecture can exploit parallelism at several levels: among jobs (multiprogramming), among tasks (concurrency), among instructions in a task (parallelism), and within an instruction (pipelining).

The elementary parallel processor which we are currently building in Toulouse is composed of three blocks: the control memory block, the execution block, and the control unit block. The control memory is divided into eight independent banks managed by a memory control unit. The execution block is a set of elementary processors which execute instructions of the single assignment machine language. The processors are independent and identical, and their number is limited only by the bandwidth of the central memory.

The high level language incorporates five statement types: assignment statements, CASE statements, LOOP statements, EXPAND statements (equivalent to a parallel DO), and CALL and RETURN procedure statements. In parallel with the building of this machine, we are extending the high level language to incorporate synchronization expressions and type definition facilities.

III. Al Davis, University of Utah

Our project has been relocated from a Burroughs location in San Diego to the University of Utah, where the work is being continued with Burroughs support.

We have constructed a machine, DDM1, which is an element, called a processor-store element, of a larger architecture. The fundamental characteristic of our architecture is that it is a recursive architecture and, if viewed non-recursively, looks like a tree, the nodes of which are processor-store elements.

One of the fundamental differences between our architectural approach and some of the other efforts is that we're very interested in being extensible physically. We hope to organize the machine so that some small performance gain will always be made as more money is spent on the machine, but, we don't want any electronic tuning to be required as more and more hardware modules are added. Hence, everything in

DDM1 is self-timed to the card level.

The store of the machine is a file system which manages its own free space in hardware. The processor communicates with the memory system through such commands as delete, append, create, assign, and insert, rather than physical addresses. In accessing a particular target in the store, there are two possibilities. One is that the file structure at some level can be either ordered or unordered. If it is ordered, then the target can be reached through a node index vector. There's absolutely no correlation with any absolute addresses. If the file is unordered, then the item is found by using either associative techniques or some form of scanning.

The main change that has occurred in Utah is that we have connected DDM1 to a DEC System 20. This allows us to do program manipulation, microcode development, and so forth on the 20. We now have a symbolic assembler for two versions of the microcode and can download DDM1 from this machine.

We are using the 20 to obtain a feel for how many processors can exploit parallelism and also to allow the use of larger and larger programs, both for programming experiments and execution experience. Our approach is to develop some software tools on the 20 which allow us to do complex experiments on DDM1. When we go to the physical multi-processor system, we intend to do it by custom integration and are currently cooperating with people at Cal Tech on integration studies.

We're also performing some architectural studies. The fixed wired tree architecture has some limitations. One is that it's possible to have a process not use a whole subtree when something else may drastically need those resources. A pool of resources idea where one could modify the physical and logical structure of the network will give users two degrees of freedom, although we don't currently know how to do the allocation.

We are also working in the language area. Data driven nets are a low-level machine representation, nobody in their right mind would want to program in them. Other people are taking steps in the direction of high level data-driven languages -- the LAU system, ID at Irvine, Lucid at the University of Waterloo. Our approach at Utah is to develop a high level data-driven graphical language and to develop the tools to make it easy to use.

IV. Kim Gostelow, UC - Irvine

Our goal has always been to design a general-purpose computer composed of a large number of small processors and, in so doing, to achieve increased capabilities in reliable computation and to realize smaller system cost due to semantic cleanliness and regularized interconnection of similar components.

Raw speed is not explicitly a goal, nor do we believe that it should be a goal at this early point in our work; but we do expect speed to be a natural consequence (eventually) of a well-balanced design. Intuitively, we expect the machine to behave as a box of (processor) resources, such that as more processors are thrown into the box, the computation rate increases. In effect, we want a highly asynchronous language, and data flow (2-1/2 years ago) looked like a very attractive starting point, after observing how impossible it was to achieve decent results with von Neumann languages.

We have devised a data flow base language that generates large numbers of tasks to be executed, called activities. An activity is a named instance of execution of a data flow operator. As an example, consider a recursive procedure in which some operator x at recursion level i is independent of that same operator x at level j of the recursion. Our system allows these two executions to be as independent as possible and to be executed on the same or different processors in the machine. The logical entity of computation is independent of the physical processor that executes it -- just like a "task" on a standard computer, but smaller.

Arguments against data flow in the past have included the problem that there is no way to handle resources and database problems (operating system needs), that there is a lack of a higher-level language (for coding operating systems, as well), that programming in data flow is difficult (both for applications programs as well as operating systems), and that there exists no machine to run these programs.

Over the last two years we have solved for ourselves the first two problems through the development of our high-level data flow language Id (Irvine data flow). The third point will take a long time to prove to others, although we are convinced that it is no harder to write programs in Id that in, say, Fortran; and that better programs result. Also, we can surpass the capabilities of Fortran, PL/1, etc. in Id in many areas - e.g., in abstract data types and error recovery and reliability capabilities, and in operating systems programming (resource managers and protection). On this latter point, Lubomir Bic will be discussing the work he has recently completed which allows a programmer to define the protection policies he wishes to have by utilizing mechanisms devised for Id. Bic's work clearly demonstrates another area in which data flow can be shown to be superior to von Neumann languages and machines. The mechanisms he has incorporated into Id are very simple (four operators and a data tag), yet a programmer can implement protection policies in his programs more complex than those possible on Multics, Hydra, etc. Id is now capable of being used to write operating systems (file systems, databases, resource managers), and our machine design is capable of running them.

On the fourth point, we have been working on and off for about a year and a half, but in just the last six months we have made significant progress (measured according to our understanding of our architecture, program behavior, and performance

improvements). In the session on architecture, I will describe our machine, which has changed from some of our earlier ideas.

We intend to pursue work in the areas of language design and architecture. Our goal in developing Id was simply to have a complete higher-level language; we explicitly did not want to develop any new concepts or facilities over and above what was necessary or easy. Now that the basic goal has been met, it is clear that the restriction to conventional programming is debilitating. A new approach is called for, and we intend to investigate the recent work of Backus on FFP Systems. Also, we will complete other work on error recovery and handling in Id with potentially significant influence on reliability and software construction.

We will further develop the architecture and remove bottlenecks in the design; expanding the design to areas not yet understood (interconnection of physical domains). Also, we are waiting on some theoretical work now in progress on time complexity taking machine communication into account; standard time complexity measures are explicitly von Neumann based and are inappropriate to data flow. Further studies will be directed toward the development of a prototype system which will provide performance analysis of the various design alternatives. Finally, we intend to investigate the topic of fault tolerance in the context of our architecture.

V. Arch Oldehoeft, Iowa State University

We have been in the data flow business for a year and a half now and the last year has been one of intensive activity in developing some software packages that allow us to study the concepts of data flow and to look at some translations of high level languages into data flow.

The project has concentrated on performance evaluation, simulating the execution of a data flow machine. We are not only interested in simulating its execution, but also in providing packages to report certain statistics. To date we have looked at 28 programs drawn from textbooks, from projects around campus, and from three of the Livermore programs.

There were some difficulties with such simulation work. First of all, simulation is a very expensive process, not only in processor time, but also in terms of the demands for memory space of the machine on which you're simulating. We found that we couldn't handle recursion to any significant depth without running out of memory.

That led us to take an alternative approach which is a program graph analysis approach borrowing from some of the existing ideas in the literature. It turns out that we can get at the same information we would have obtained from a simulation

within the limited context of the processor we're simulating.

We've considered a data flow machine language including recursive procedure calls and a structure memory. We assume a feedback constraint which requires that both the operands be ready and an acknowledge signal be received from the successor instructions for an operation to execute, enforcing the single token per arc concept.

In the 28 programs run to date, we've found parallelism allowing speedup factors of 1.5 to 7.5. We see the possibility of immense speedup in some of the programs if we add a stream or vector capability to the language and simulator. Other programs may exploit a stream capability, but only after a complete restructuring of the program.

We begin with a program in language X and do a hand conversion to our own high level language. This involves primarily getting rid of the GOTO's, handling COMMON (passing COMMON references as arguments to a procedure), and eliminating any EQUIVALENCE statements. We hand the resulting program to our compiler which performs program analysis and code generation, producing a symbolic assembly language program which is used as input to the simulator and is assembled into machine code.

The simulator accepts machine parameters specifying the number of functional units of each type and their timing. The result is a statistical report on the simulated execution in terms of the maximum and average memory and processor use and the utilization of data paths between the memories of the processors.

The alternative approach is to enter the full analysis stage with statistics amounting to the timing of the particular operations and to then generate a program graph and the timing equations which are associated with it. An advantage of this approach is that in looking at these timing equations, one can relate the speedup to particular program characteristics. If we go through the simulation technique, we can generate the statistics, but we know nothing about the program structure and can't relate the speedup and the parallelism to characteristics of the program.

Our planned activities for the second year and beyond include extension of the language, compiler, and simulator. We also intend to study what kind of support is needed for general operating system functions in a processor of this nature. Thus, we would like to extend our language to include data types, a high level monitor construct, and general facilities for communicating processes. In terms of the simulation, we'd like to put in stream data operations and then whatever other support we might need for monitors and communicating processes. In the performance studies, we'd like to look at real-time applications and the divide-and-conquer type programs.

We're also concerned with a memory hierarchy model. Because we can't currently support recursion due to memory demands, it appears that some cache mechanism is necessary. The idea on this is to bring in to the processor just those instructions which are active. Our interest here is in the organization of the program so information is brought into the cache at the right time.

VI. Robert Keller, University of Utah

I'd like to report on some work by Gary Lindstrom, Suhas Patil, and myself. This is a fairly new project as is evident by the fact that we weren't here last year. We're developing an architecture and an evaluation model for what we call loosely coupled parallel processors.

We're studying a highly parallel architecture which supports a variant of pure Lisp as its main language. We feel that Lisp is well-suited for data flow-like computation, although a few minor modifications are required, for instance converting the cons operator to a lenient cons, an idea publicized by Friedman and Wise of Indiana University.

Lisp is a single assignment language, at least if you don't include the PROG feature, although we are looking at the possibility of including that feature in some limited form. It already has a data structure capability, which allows concurrent operation on many sites throughout a large tree. And its use is already established within a substantial user community.

Our processor has a tree-structured organization. The leaves of the tree are processor-memory combinations and the interior nodes serve a communication function. One novel feature is that the frontier of this tree, taken collectively, forms a universally addressable memory. There are no murky problems with how you get information. Everyone can get to the same information as everyone else, at least at the machine level.

One advantage of this organization is that inherent in this tree structured communication scheme is a way of taking advantage of locality of communication, i.e. reducing communication costs for parallel sub-computations which are logically related to each other. Also, implicit here in the locality consideration is what amounts to an automatic cache memory; that is, we don't have to explicitly put in another layer of memory and call it a cache. The leaves of the tree are acting like caches themselves, keeping information close to where it's going to be utilized.

The current status of the project is that we have defined the basic topology of the processor and designed a basic evaluation model and simulator for it. Programs in the machine language are linearized representations of a graph version of pure Lisp

programs. The parallelism is displayed in the graphical representation, and recursion comes almost free.

Immediate plans include looking into the scheduling and balancing problems associated with allocation of tasks in this tree topology. Also, we have to extend the language to what might be called a programmer's Lisp rather than simply a pure Lisp. For example, including the Prog feature, assignment statements (or certain limited variations thereof), arrays, and so forth.

VII. Rob Witty, SRC - United Kingdom

I'm from the Science Research Council (SRC) which is the U.K. equivalent of the National Science Foundation. The SRC is one of the major funding bodies of scientific research in the U.K.

Around mid 1976, the SRC decided to create a program specifically to fund distributed computing, where distributed computing was defined as several autonomous, but interacting, computers cooperating on a common problem. We then created a coordinated program that really got off the ground when the coordinators were appointed in January 1978. Mrs. G. Ringland, a senior member of a software house in the U.K., has the task of linking the program into non-university institutions and government research departments in industry. Professor Bob Hopgood of the SRC is the academic supervisor. I was appointed technical secretary, whose job it is to ask the nasty questions whenever people apply for funds.

The first problem that arose was to define a coordinated program. We invited proposals against what we saw as distributed computing and interacted with the various university people to see if we could get some sort of a coherent research program going. The normal way that research is funded through the SRC is that a university group applies for money, and we either do or do not give them the money. They then go away for approximately three years, somewhat on their own with no formal feedback. However, with a coordinated program we require cooperation with other groups working in the area, the production of reports, and attendance at workshops such as this one.

Distributed computing, as we see it, covers quite a bit. Categories in the field cover such topics as theory, measurement, reliability, data flow, architecture, applications, software, and communication. Research on data flow computation is considered one of the major areas of the distributed computing program.

The four main data flow groups in the U.K. are John Gurd and Ian Watson at Manchester University, Peter Osmon at Westfield College of London University, Brian Randell and Philip Treleaven at Newcastle, and Roland Sleep who has just started at

Brunel University.

The main interest of the Newcastle group is general purpose systems. They see problems with conventional data flow and see the possibility of architectures built as a synthesis of data flow ideas and von Neumann ideas, incorporating the best of both. Their main interests involve architecture and language; they're currently designing languages and building simulators.

Westfield College is interested in the language area. They are currently designing a single assignment language which is near completion and have recently been funded to develop compilers and simulators for that language.

The motivation of Manchester University is to design a high speed engine which is modular, extensible, and asynchronous. They have a single assignment language called Lapse, with a compiler and two simulators. We just funded this group to build their processor, and we expect to have an operational machine by September 1981.

Session 2. Language Issues
Chairperson: Bill Ackerman, M.I.T.

I. Karl Boekelheide, University of Utah

Our research program began with the development of a data driven net model. We then designed and built a machine to execute these nets. Now that we have the machine, we'd like something a little better to use in programming it. We'd like to have some data structures, to provide some order to our nets, and to allow some type of high level constructs.

We're doing a few things differently in the language we're developing. One of the most radical is that we're going to use an interactive graphical representation rather than a textual representation. We want to use graphs as both the display representation and the input representation, keeping the style of variables or data items represented by arcs connecting boxes. However, we want to use constructs such as iterate blocks to reduce the amount of spaghetti we now have in programming data driven nets. This trades off with the ability to see the influence any data item can have when all the arrows are right there in front of you.

We'd like to retain in our language a recursive quality similar to that displayed by the architecture; that is, at any level, everything looks the same as at any other level. At any one level, you are in a certain context, there may be context above you in that arrows are coming into you. And you may be able to detail down to any depth, until reaching a primitive operation. thing.

The language should be a design tool, rather than just a programming language. The representation should be close to the manner in which programmers organize their thought process when they design something. Thus, we hope to eliminate the phase of thinking about a problem and writing down flowcharts or state diagrams by creating a process in some graphical form. We feel this is a very natural way of representing programs.

The language will be composed of a set of box shapes and arrows. Each construct will probably have a different shape. We want to handle such things as calls, iteration, serial data flow to parallel data flow constructs for shared resources, parallel data flow to serial data flow access, data structures, some sort of monitor facility, pipelining, synchronization, and coroutines.

Our major concerns are the data input/output problem and how to deal with shared data. Without the implicit sequential access, it's very hard to talk about sequential files. Yet, we want to allow sharing of files for data base applications. Finding solutions to these problems will entail considerable work, and how they are

implemented will have some effect on our language.

II. Jack B. Dennis, M.I.T.

We are developing a language which will be a user language for a Form 2 data flow machine and will be suitable for expressing Livermore type problems. We've done some work on this, and we're trying to put together before the end of this year a reference manual that will be a zeroth order approximation to the language.

In our design, we've drawn heavily on the language CLU designed by Barbara Liskov at M.I.T., particularly in the area of data types and type specification. Basically, we are taking CLU and eliminating all the side effects, leaving an applicative programming language. The entities on which the programs in the language operate are strictly values. There's no concept of object in the language -- no concept of something that can be changed by actions of the program.

The data types in the language are the classical scalar types (integer, real, Boolean), records, and arrays as they exist in CLU, although in our language they are fixed values and not subject to being updated by an operation. The only operations available on arrays and records are operations which accept an array or record and produce a new array or record which might differ from the given one in some way.

Arrays and records are user-defined types. When an array is declared, the type of its elements must be specified and all the elements must be of the same type. An array is regarded as a mapping from integers to values of the element type. Records are similar to Pascal records, there are a number of fields, each field may be of a different fixed type.

The language is expression-oriented. There is only one expression which resembles an assignment. However, instead of calling it assignment, we do what many applicative language designers do and call it a definition. An expression in the language consists of a BEGIN, a sequence of definitions, the key work RESULT, and a further expression. The definition creates a number of bindings of identifiers to values, and that set of bindings defines the environment in which the expression is evaluated.

Our approach to iteration is based on the position that recursion is more natural and understandable than iteration. There is a special kind of recursion, called tail recursion, which can always be translated into iteration, and we are seeking a syntax for expressing iteration that is in some sense isomorphic to tail recursion.

In conventional languages, there are two forms of iteration. In one, the instances of execution of the body depend on one another. That kind of iteration is

naturally expressed as recursion. In the other form, the instances of the body are really independent. I feel those are more naturally expressed by a Forall construct.

There are two kinds of Forall constructs in our language. The first one finds the value of an expression for a set of input values and lets these values be the elements of a newly constructed array. This is a major source of opportunities for concurrency. The second form of Forall is one in which an expression is evaluated for a set of input values and the result is formed through combination of the resulting values using an associative operation such as addition or multiplication.

Applicative languages are side-effect free, which brings up the question of how we deal with those kinds of computation which make use of side effects in conventional programming languages. For that purpose, the language supports two kinds of program units. The first one is the procedure. Procedures are very much like conventional procedures, except they are free of side effects, which means that input and output parameters are separate and distinguished.

The second form of program is the module. A module receives a stream of values and generates a stream of values. The module may also receive input scalar values. One cycle of operation of a module uses the values of the input stream and a single scalar value to produce a single scalar value and a stream of output values. Our model of what occurs inside a module consists of some state variables which have scalar values. The module looks at the state information, reads some fixed number of inputs from an input stream, and looks at the scalar input values. Based on that information, it redefines the state and sends out values which are appended to the output stream.

In our model of data flow programs, a stream is terminated by an end-of-stream token. The end-of-stream token need never arrive, but if it does, its arrival signals the end of one cycle of the operation of the module. The module will then reinitialize itself in preparation for the next set of input values. Similarly, when its cycle of operation terminates, an end-of-stream token will terminate the output stream.

III. Kim Gostelow, UC - Irvine

Our goal is to design a general-purpose computer that can utilize a large number of processors. To do this, we devised a data flow base language that is as asynchronous as possible, that can create demands for a large number of small computations. The resulting system is called the "unraveling interpreter."

The basic principle of the language, Id, is the creation of a large number of activities, where an activity is a uniquely named instance of execution of a data flow

operator. Each token in the base language carries not only a data item, but a control field. The control field consists of four subfields, u.p.s.i., where u is the context in which the current procedure p is executing, s is the data flow operator destination of the token, and i is the particular iteration or initiation of s for which this token is an input.

Consider as an example the matrix multiply of two n x n matrices. The activity name manipulation mechanism is such that if the Id program is written as three nested loops, where the innermost loop is a dot product, the net effect is that the two outer loops "unravel" and create n^2 inner loop (dot product) instances, all potentially in execution at the same time.

Id is a complete language, incorporating procedures (with recursion), loops, conditional expressions, streams, and several special loop constructs on streams (For Each and Return All), abstract data types, protection, and data flow resource managers. The following discussion concerns resource managers.

Our goal is to build a general-purpose machine, and Id is to be its language for both applications and operating system coding. For this reason, resource managers were incorporated into Id. As opposed to a procedure, where each instance of execution creates a new logical instance of the applied procedure, each use of the same resource manager object reuses that same manager. A manager is a separate process to which inputs are passed, and, as opposed to the monitors of Brinch Hansen and Hoare, control always resides within the manager itself. When another process uses a manager, it actually sends the data to the manager and relinquishes all control of that data. A manager may have memory in the form of a loop and thus exhibit history-sensitive behavior. We have programmed the readers-writers problem, distributed databases, and other such applications with this mechanism.

Our intention is to show that data flow can do anything any other language can do. We are not trying to develop any new theories or ideas other than those which are easy or necessary. Developing managers was one new idea which was necessary. This goal, now must change. Data flow is being stifled by the old ways, and the semantics are getting too bulky.

Backus observed that "Programming languages have too much framework, and not enough observable parts." Too many similar ideas get expressed/implemented in distinct ways: given appropriate primitives, we can combine them by if-statements, while-statements, etc. Each is implemented separately, but all are really simply examples of <u>combining forms</u>, where expressions/functions are the parameters to those forms.

In response to this, Backus developed FFP Systems. Over the past two years, the changes we have made to Id keep moving it further in the direction of FFP.

I think it needs to go all the way in one jump to benefit from it, everything good about data flow is in FFP, plus much more. I feel that FFP is a tremendous advance in machine languages and I hope the data flow will simply be absorbed and become a consequence of a more encompassing theory.

IV. Chris Hankin, Westfield College, University of London

I have been working in data flow for about a year with Peter Osmon, who leads the group, and four research students. Two of us have been looking at architectures, and two of us have been looking at language implications. We haven't actually come up with any answers on the architecture side yet, so I'll just speak on the language.

We have a very primitive graphical notation, the operators operate on single bits and one bootstraps from there to more complex operations. A data flow graph in our language is a directed graph in which the nodes are used to represent one of the following seven primitive operations: constant generator, copy, union, primitive transformation, data sink, true/false gate, or application of a data flow graph. The arcs connecting nodes of the graph represent the data dependencies between the nodes

We haven't committed ourselves yet to data driven or demand driven systems, both concepts exist in our system. A data driven program terminates when there are no elements enabled, no unused input data, and no element is currently being interpreted. The alternative model, a demand driven system, terminates when a result has been produced.

There is no cyclic graph structure in our language. A well-formed graph is an acyclic graph of functional nodes. Such a well-formed graph represents a well-formed program in the language. This has a number of implications for languages. First, we can consider programs of any meaningful complexity to be networks of function applications. This suggests a functional language, such as Lisp. Because all the nodes are side-effect free, it doesn't really matter what order you write them down, the order of execution is apparent from the various data dependencies between the statements.

The absence of cycles in the graph tends to suggest a single assignment rule. We have opted for a semantic single assignment rule which states that the interpretation of a program must not cause any name in the program to be assigned more than one value. This is similar to Chamberlain's single assignment rule and different from the syntactic single assignment rule. A name may appear on the left hand side any number of times, as long as only one of those statements is executed.

By ruling out cycles in our primitive graphs, we've ruled out the use of iterative processes. We don't necessarily have to follow this through at the high level language level, but recursion seems to us a more elegant way of representing iterative processes. Hence, all of our iteration is represented by tail recursion.

Our programming language, CAJOLE, is similar to other nonprocedural languages such as Lucid. Many of the basic features of CAJOLE arose directly from a desire to develop a high level language based on data flow principles which programmers would find as natural to use as current block structured languages such as Algol 68.

V. John Gurd, University of Manchester

Like most of the groups in data flow, we started off with an architecture, not a language. I think the only exception to that was the LAU system in Toulouse which started with a language and then developed an architecture. It seemed to occur to everyone at once that single assignment was desirable, although for highly different reasons. Our language Lapse is another single assignment programming language for a data flow machine. We attempted to keep its syntactic form as much like Pascal as we could, but it's semantics are completely different.

We approached the language design by finding problems and developing mechanisms to solve the problems. We then simulated in order to evaluate the mechanisms and made decisions as to the usefulness of the features. In this sense, Lapse was both experimental and educational.

There are two issues involved in the design of data and control structures for a language. The first is the accuracy of the notation, its ability to express the problem under consideration. That clearly must be paramount in any language. And the sort of things I think are important are the accuracy, the conciseness, and the ability to prove that the program does what you expect it to.

But then comes the very real and nasty problem of what happens when the program is translated into a machine runnable form. At this point the efficiency of translation, the efficiency of compilation, and the efficiency of execution become very important. These are the problems that we're a long way from solving, but everybody else seems to be in the same boat.

Data types in Lapse are scalars, corresponding to one token on one arc. Structured data is carried as a number of tokens that travel about, conceptually simultaneously, on the same arc. The tokens are not necessarily in sequence, and can be distributed around the machine. And they're not organized like a stream, they can all sit on the arc at the same time, with the last one coming first. This creates a lot of

problems and is the major reason why we talk about control structures and data structures separately.

The language is block structured and functional. Every statement appears to be a function which takes in some input type and produces some output type. In the case of primitive instructions, a statement is a machine defined operation which converts between the two types. But, we can also have other types of blocks which correspond to primitive expressions, compound expressions, conditional expressions, iterative expressions, and recursive expressions.

Conditional activation in the language is performed by a construct called a test block. This construct is never seen by the programmer, he doesn't have access to test blocks, so he can't have conditional statements, he can only use conditional expressions which are tied together for him by the compiler. The test block performs some test on the input data, and if the result of the test is true, the data is sent out one output of the expression; if false, down the other. So we can ensure mutual exclusion purely by using IF THEN ELSE expressions.

A Repeat or While block performs iteration through translation into into a sequence of test blocks, one branch of which reexecutes the block, and the other of which drops out of the iteration through a common merge operator which has an input from one side of each test block.

Recursion in the language is of the form test, if at the dropoutpoint, then compute some partial nonrecursive function. Else, go into some block which calls the function again, called a token block. The three forms of a token block are the iterative form, in which the last action is to call the function that is being defined; the linear form, in which some operation remains to be performed after the function is called; and the nonlinear form, incorporating a number of parallel calls to the same function.

We want the implementation to be simple, compact, and efficient at runtime. To achieve these objectives requires keeping the number of tokens small, so an operator accepts only two input tokens at any one time. This causes problems in the case of arrays where there are a lot of tokens, but there is way in the language to define a function which acts directly on the array. In some sense we don't want this, since if we had a function which acted directly on arrays, it would have to collect all the tokens together at one point, and that's the very thing we're trying to avoid. Since we don't want to duplicate code for iteration or recursion to handle components of an array, we use the same sort of labeling mechanism as Arvind and Gostelow. We split this label into a number of fields, including an iteration level, an identifier which designates the instantiation, and an index which designates the part of the array.

VI. Gary Lindstrom, University of Utah

My purpose here is to ask the following question, "Although a lot of references have been made to Lisp as a potential language for the kinds of machines we're talking about today, nobody seems to be targeting their machine toward Lisp. Why not?" My thesis is that Lisp should be used as a programming language for data flow machines.

Lisp already exists. It has an applicative kernel, hence, it is side effect free if viewed as an isolated evaluation of a function call. It's fundamentally built upon recursion, so the problem of how to incorporate recursion is already taken care of. Also, reduction evaluation is natural for the language.

There is a need for more computing power on Lisp machines in such areas as symbolic computation and artificial intelligence. Also, the Lisp user community is sophisticated and non-entrenched. Such a community would tolerate changes in recommended programming styles for suitable payoffs.

There is already a wealth of implementation ideas for Lisp in existence, revealing that it's a very supple language in terms of implementation. Lisp is implemented interpretively using S expressions and a tree structured evaluation model, in compiled form using activation records and pure code, and in hardware, as is done at M.I.T. on the Lisp machine.

Lisp is linguistically rich in the sense that its proven over the period of its existence to be a very convenient language for experimentation. It has a base for language extension which partly interests us, but more importantly, it serves as a trial horse for different machine designs.

The architecture we have in mind should support a hybrid style of both data flow and demand evaluation. Also, the concept of "lenient" cons should allow us to view cons as a non-operator, as simply a currying construct. Particularly from the standpoint of demand evaluation, it should allow demands to be directed simply to car and cdr fields independently.

All problems of "standard" languages can be studied within Lisp to avoid developing a completely new language for their evaluation and solution. Lisp is a more manageable vehicle for those studies right now. Lisp has encapsulated in it the key problems that have been brought up in this session; that is, cells/assignments, sequential control, I/O, and files.

VII. Philip Treleaven, University of Newcastle-Upon-Tyne

The work at Newcastle can best be described as dissatisfaction with what I'd like to call pure data flow. The problems that we see in pure data flow are that it has a very elegant, although primitive, activation by availability control model, it has the concept of passive storage like the von Neumann model which leads you to regenerating data occasionally, and there seems to be difficulty in representing notions such as files and input/output.

We tried to take the best features of pure data flow and combine these with the von Neumann model. What we like in data flow is the concept of activation by availability of data and the underlying directed graph representation. What we like in conventional control flow is the passive memory and the ability to be able to provide additional sequencing constraints.

We've based our work upon the concept of activating computations with control tokens. We've built concepts of memory into the models. So we're moving away from pure data flow, although we feel we can support data flow concepts.

We've based the project upon two forms of program representation which are basically independent attacks on the problem. One, which we call the structured directed graph language, we like to think of as data flow plus. The other representation is called the generalized control flow architecture. We're developing this architecture as a better form of simulator to represent this spectrum of languages, to give us a machine for trying out our language concepts to deal with structured directed graphs.

At the moment, the project involves seven or so people and has three simulators implemented for the architecture. Two are being used to study the architecture and the other interprets a target machine language into which our various translators compile. We have implemented our own variant of data flow language and have a translator which translates a subset of Fortran to run on our concurrent architecture.

We're hoping to implement the language and architecture and, with the experience we've gained from that, move across to some new form of language and architecture. We tend to think of the structured directed graph language as being something like Algol, and the computer architecture as being something like a 360. You have to actually remove all the structure from the language to run it on the architecture. Our longer term goal is to design an extended version of the architecture which would enable the structure of the program to be retained explicitly at run time.

Our model of computation has the notion of unique assignment in which there are never multiple tokens upon one arc. Only a single object is created whenever a

statement is executed, and that object acts as a memory that can be referenced by any number of destinations.

We activate computations in the language through use of control tokens. We also allow two mechanisms that seem to be complementary: activation by availability, which seems very nice for partial results, and activation by need, which solves the problems we see in implementing files.

Objects are read only, to be accessed any number of times by any destination. We also allow these data objects to be structured in that they can contain subelements which can be accessed either by name or by position. An object is only accessed from within the function in which it is located. Hence, when that process has finished and generated its results, that is an appropriate time to get rid of the object.

Session 3. Applications
Chairperson: George Michael, Lawrence Livermore Laboratory

I. Jeffrey Jaffe, M.I.T.

I'm going to speak about two data flow programs which solve the Laplace equation. In developing these programs, we had the goal of trying out the data flow language under development at M.I.T. and seeing if the language constructs were able to express the ideas in the program. A related goal was to investigate the development of translation rules to translate source programs into data flow graphs, to see if a program expressed in the source code could be successfully translated into the highly parallel representation we would have expected if we had coded the program at the graph level.

We wanted to study a real problem whose solution would be of use to people. We also wanted to examine potential solution methods for the Laplace equation that exist in the literature and have certain nice features about them, but require too much computation time on a sequential computer, and to see if these solutions could be used on a data flow computer. The final goal was to perform some sort of performance analysis to get an idea of the sort of computation speed that could be expected from a data flow machine.

The problem can be viewed as taking place in a two-dimensional region on the boundary of which there exist temperature or potential values which are fixed for all time. The problem is to find the steady state temperature for each point in the region. The usual way of solving this problem is a relaxation method in which initial values are assigned each point, the temperature is recomputed by averaging the neighbors, and the termination condition is tested to see if the process is converging.

The second method for solving this problem is the Monte-Carlo or probabilistic method. From each point in the region, we take a random walk to the boundary, evaluate the temperature at the boundary and use that as the initial estimate of the temperature of the point at which we began the walk. This process is performed many times and the temperature values are averaged to obtain the estimate for the steady state temperature at the point.

Both the relaxation and Monte-Carlo methods can be readily expressed in the data flow language through use of Forall constructs to apply the required computation to each point.

Both methods achieve significant computational speedup in data flow due to the capability to compute all points in parallel and perform the random walks in parallel. The advantage of each depends on the particular configuration of data flow

machine on which they are to be executed.

II. David Hirschman, M.I.T.

I translated a Fortran program to solve the Lagrangian formulation into data flow. Although a number of problems arose from the different natures of the languages, I was able to achieve a significant decrease in the size of the program by the translation process.

The basic assumption of the Lagrangian formulation is that a gas exists in a three-dimensional space which is divided into small boxes. The assumption is then made that all the atoms or molecules in a box will stay there forever, although the shape of the boxes may change as the temperature and pressure of the gas changes.

Since a box always has the same neighbors, and the contents of a box remain the same, the state of the system can be depicted by an fixed size array that holds all the state variables for the system. Some variables are associated with each box, and others are associated with the corners and ends of the boxes.

The relationship between temperature and pressure is described by many differential equations. The program is an approximation of those equations, turning the partial differential equations into difference equations and solving them iteratively.

The original Fortran program which I translated into data flow was very clean in its control structure, but created quite a problem with global variables in COMMON blocks. Hence, a good portion of the translation effort involved determining which programs used which data.

I ran into a few problems that arose from the different natures of the two languages. The Fortran program is concerned primarily with array structures, and the implementation of such structures is an important problem in data flow machines. In a standard von Neumann machine, one just carves out a hunk of memory for the array, and each element has a specific cell. However, a data flow processor has no discrete memory where the array can be placed.

An array is created in a data flow machine by use of a Forall construct to perform some operation on the elements of one array, creating a new array, or by use of an Append operator to add an element to an existing array. It is very difficult to just change one element of an array. For example, it is very difficult to clear the main diagonal of the array, since the whole array must be copied to perform this, using nested Forall constructs.

Such operations on arrays were common in the Livermore code I was translating, primarily due to boundary conditions. These operations, which are relatively easy in Fortran, involved a good deal of copying of arrays, just adding elements in the right places.

Strangely enough, these parts of the code were the parts where I achieved the greatest reduction in the size of the code. In one part of the program, I reduced the size of the code by a factor of seven, since I was forced to do the operation as one whole, whereas the Fortran code had special pieces to take care of the right edge, the left edge, etc.

We aren't finished with the translation -- there are some tricky parts on which other people in the project are working -- yet enough has been done to convince me that a "real world" program can be written in data flow.

III. Bob Meyer, Clarkson College

I'm not going to speak about a specific application, but rather about the interaction between applications that one would like to solve and the architectures and languages used in the solutions. I am driven by applications, and entered this field from the point of view of studying algorithms and systems for solving certain kinds of problems, not from the point of view of designing machines.

The application areas that primarily interest me are image processing, automated cartography, and scene analysis. Our research goals are to characterize these problem classes in some manner which is mathematically precise, defining the class of problems to which a particular architecture or language is well suited.

The next step is to look for a common descriptive framework for describing algorithms or programs for solving these problems. Then one must synthesize an architecture which is well-matched to the problem class, where well-matched implies good performance in terms of speed. Finally, there must be methods of evaluating the performance of the proposed architecture, where this in combination with the previous step forms a loop through which one travels.

Image processing tends to deal with an intermediate size data base, somewhere between 256 thousand to about four million bytes of data. One of the other important characteristics is that there are many instructions of a few types. In a typical application, one might have ten operations on a point, with six to ten instruction types. Third, the computations are very much locally interdependent, depending only on a few other computations within some small neighborhood. Also, the data dependency graph for the problem is very highly structured.

The automated cartography problem usually utilizes a much larger data base, anywhere from one million to ten million or more bytes. The processing required tends to be fairly significant, there are a number of N-squared algorithms, where N is the number of bytes. Again, the computations tend to be local, but the data dependencies are very involved and complex.

We wish to match the structure of a processor to the structure of the problem we're interested in solving. To do this, we utilize a collection of processor-memory modules, where each module consists of a processor, some program memory, some data memory, and some communications interface. We would like to be able to vary the processor capability to match the functional level of each of the nodes in the computation.

The inter-processor communication in the system should be flexible and dynamic, allowing the machine to readily appear as though it were connected in different structures at different points in time. The program should determine the way in which the interprocessor communication takes place. We don't need complete interprocessor communication, each processor need not communicate with all others, however, the communication paths should not be arbitrary.

IV. Tim Rudy, Lawrence Livermore Laboratory

Livermore's interest in data flow is for high speed computation. We consider it one of several alternatives that we are currently investigating. One of these alternatives is the S-1 project which involves the development of more conventional multiprocessing structures. We're also looking at Cal Tech/Ivan Sutherland sort of Solomon reincarnation.

When we talked to M.I.T. initially last June, we thought 100 Megaflops (million floating point operations per second) was a pretty good number. A 7600 runs anywhere from four to zero, the Star up to fifteen. After some thought it appears that one hundred fifty, two hundred is a more reasonable request of a data flow machine.

I think the bottleneck in a problem such as a one-dimensional hydrodynamics arises in the memory interface. Examining this on the Star at 13 megaflops, we see that a memory request is made every 19 nanoseconds. For the Cray, with double the performance, the memory access rate is almost half as much, due to the vector registers which act as a cache. If we extend this to 100 megaflops, we see that on a Star there would be a memory access request every 3-1/2 nanoseconds, and for the Cray, about every 10-1/4 nanoseconds. I suggest that it's difficult to get memory that runs this fast, and that more attention should be paid to this problem.

The codes that M.I.T. has been examining are relatively small. If we examine a large computation, such as a two-dimensional heat conduction problem, we see that a Cray performs each time step in 7.05 milliseconds, a speed of 27.1 megaflops, or 11.6 nanoseconds per memory request.

In terms of data flow, assuming we can process each zone independently, there would be approximately 204,642 operations, 29 per microsecond or 34.5 nanoseconds per operation. This requires 1,588,739 memory references, 22.5 per microsecond, or 44.4 nanoseconds per reference. Just to match the Cray, requires 19.4 nanoseconds per operation or memory reference, and we want to go much faster.

It's not clear that Livermore can use pure data flow. The real value of data flow to Livermore is to open us up to other possibilities for getting high performance. We know how to do vector computation, but at some point we're going to have to go to multiprocessing. What we learn from data flow is going to help us in that regard.

Session 4. Translation
Chairperson: Kim Gostelow, UC - Irvine

I. Lubomir Bic, UC - Irvine

The long term goal of our project at UC - Irvine is to develop a general purpose sharable data flow machine. The sharable aspect implies that we are entering the domain of operating systems. One of subproblems of this is to develop a mechanism that allows the implementation of a variety of protection policies to perform such services as guaranteeing the privacy of information, permitting controlled sharing and exchange of information and services, and preventing possible sabotage acts.

When I first looked at this area about a year ago, I discovered that there are some common problems that are automatically eliminated by use of a data flow representation. The modification of data automatically implies the creation of a new copy, so there is no need for write protection. Also, use of a procedure written by someone else (the Trojan Horse problem) will not allow that procedure access to any data not passed to it as an argument.

There are still some other problems left. The monitor concept is an essential part of sharing and exchange of information in our system. The monitor is an execution domain consisting of two actors, the Entry actor and the Exit actor. The Entry actor collects requests and and forms a stream which goes into the body of the monitor. The monitor also contains a feedback loop which acts as a memory.

In contrast, a procedure call, which has a similar syntax, passes the procedure definition and argument to an activate actor, a domain is created between a Begin and End statement, and the execution is carried on between these two actors. The major distinction between monitors and procedures is that the execution domain of a procedure is created only once. Each execution of a procedure has a different execution domain. Whereas for a monitor, there exists one domain which may be accessed several times, as many times as you wish during the life of the monitor. Thus, a procedure is memoryless, whereas a monitor is history sensitive.

We envision a model in which the boundary of the system is a sphere. Inside is a collection of monitors, and the users sit outside the sphere and access the monitors inside. Every user initially obtains a job control monitor and can dynamically create any number of monitors which can interact with each other, exchange information, and communicate.

One major departure from conventional systems is that we don't equate the human user with the user process. In a conventional system, if a process executed within the system can gain access to secret information, it is considered a protection

violation. This is not the case in our system, we allow information to go anywhere within the sphere. Any information can propagate to any monitor within the system, and a protection violation occurs only for information which leaves the sphere and reaches a user outside the sphere.

A mechanism is provided which allows the attachment of a unique protection key to any value within the system. A value may leave the system only if no key is attached to it. Two primitives operate upon keys, attaching and detaching keys to values.

These constructs allow us to solve the selective confinement problem. We can send two kinds of information to a monitor simultaneously, protected and unprotected. For example, an income tax monitor could receive income information protected and other information unprotected. In such a case, any returned information, such as a bill is unprotected only if no protected information was used to generate it.

II. Susan Conry, Clarkson College

The work I'm going to speak about is related to translation and implementation of data flow programs. My feeling is that although we want to follow a data flow approach, we must be intelligent about the manner in which we do it. We'd liked to make sure that things are done in a fashion that doesn't cost us much, undisciplined data flow can be very expensive.

The problem I'd like to address is how to execute data flow programs fast. Clearly, when we get down to the nitty gritty, there will be problems that are machine specific and problems that are language specific. The kind of problems I'd like to address today are those which are neither machine specific nor language specific.

It's been mentioned several times that there are a number of costs associated with the execution of data flow programs. One of them is of course the time it takes to perform operations. Another is the number of processors required. And, particularly in data flow computation, it's rather clear that the communications overhead will substantially affect the cost of implementing a program.

I'd like to revise the problem a bit and ask how can we assign operations in a data flow program to functional units in the machine in a manner which allows three things to occur. First, no concurrency should be lost, if at all possible. Second, the solution should use a minimum number of processors. Third, the communications overhead should be reduced as much as possible. It's improbable that all three can be optimized simultaneously. However, we'd like to try.

Given these goals, I'd like to make a number of assumptions. First of all, we assume that a data flow graph of the program is available. Second, we have to assume that flow of data is comparable with flow of control. We also assume that there are enough processors to carry out the computation.

The first pass at an algorithm to perform the assignment of processors is to take a static program graph and find a maximum length cycle-free path through the data flow graph. We assign all the operations on this path to the same functional unit. We then delete all "assigned" nodes from the graph, forming a reduced graph. If this reduced graph is not empty, we perform the same operation on the new reduced graph. This yields an assignment of operators to functional units which is optimal in the sense that it provides a way of assigning operators which uses a minimum number of operators and in which the communications overhead is reduced.

This naive approach has the problems that the algorithm is quadratic in the number of nodes and the algorithm provides no information about loops in the graph. This information is necessary to exploit locality within any loops. So our secondary goal is to obtain a reasonable assignment, information about loops, and to do less work.

A first approach to this new problem is to partition the graph so that loops are easy to identify and the cost of finding assignments is lower. It seemed to me that the body of work on global flow analysis as used in compiler optimization was appropriate to this problem.

An interval in a control flow graph is a maximal single entry subgraph in which all cycles pass through the entry node. It is possible to choose head nodes of the graph so a flow graph is uniquely partitioned into intervals. Furthermore, the algorithm is linear in the number of edges in the graph. Now that the problem has been decomposed, we can perform a process similar to that described previously. The algorithm is still quadratic, but for much smaller graphs.

Interval analysis is linear as long as the graphs are reducible. However, this is not always the case. Hecht and Ullman have come up with a scheme for doing flow analysis which is not quite as nice as intervals in that loops are not such an intuitive idea. However, their algorithms apply in roughly the same time frame to irreducible graphs as well as reducible ones. And some loop information is obtainable from this approach, although techniques for processor assignment have not been developed.

III. Lynn Montz, M.I.T.

My research has been concerned with safety and optimization transformations for data flow programs. This work has the main goal of determining methods for translating high level data flow programs into machine language

representations.

At the moment, we plan to accomplish this translation through a two step process. First, the high level data flow program is translated into a data flow graph. The second step involves translating the data flow graph into a machine representation.

The firing rules of a data flow program state that there can never be two tokens simultaneously present on one arc of a graph. To ensure this within the processor, we must add acknowledge arcs to the program. Petri net theory assures us that the proper assignment of acknowledge arcs to form one token directed cycles will ensure safety of the graph, the quality we are seeking.

The use of acknowledge arcs provides deterministic and deadlock free data flow productions for each data flow language construct, allowing us to show that the resulting interconnections are determinate and deadlock free. The algorithms under development will allow the assignment of acknowledge arcs to a program, creating the one token directed cycles, and will then remove unnecessary acknowledge arcs, optimizing the solution.

IV. Karl Ottenstein, Michigan Technological University

My thesis research is concerned with program translation and optimization. While offering the potential for translation of conventional languages into data flow graph languages, my work is presently concerned with von Neumann machines and languages.

The research recognizes that certain transformations on standard intermediate program forms, such as directed acyclic graphs (DAGs), abstract syntax trees, three address code, or prefix notation, are reasonably complex and looks for a better form that utilizes the information contained in a DAG-like form in order to perform these transformations more directly. Essentially, what we're talking about is the extension of the data-dependency concept inherent to a DAG to a complete program: a cyclic data dependency graph.

The first step in the translation involves the extraction of a simple data flow graph for each basic block of the program. This data flow graph consists of a data dependency graph for the assignment statements, an ordered list of input and output variables, a set of nodes which have no predecessors in the graph (inputs to the block), a set of nodes which represents the definitions in the block which are available to other portions of the program, and finally, one distinguished node which as the root node of the terminating conditional determines the execution-time successor of the block.

We then construct a cyclic graph from all these simple graphs. After determining the set of definitions which can reach a particular block, the uses of variables are linked to definitions found elsewhere in the program. The resulting data flow graph can be compressed since many nodes represent temporary variables only, and can be eliminated from the graph. The program is then compressed further through examination backwards from outputs to inputs, eliminating any nodes or edges which are not reachable.

I have developed recursive graph-marking algorithms for performing scalar propagation, subsumption, dead code removal and constant propagation in near linear time. An algorithm has been developed for performing redundant common subexpression elimination on programs with reducible control-flow graphs. This algorithm is more complete than any other known practical algorithm and, the various algorithms are simpler than those for other intermediate forms partly because all use-definition relationships are explicit. In my thesis, I present methods for translating these graphs back into von Neumann languages and sketch a method for translating into a data flow language. The applicability of these methods to data flow machine programming seems clear.

Session 5. Architecture
Chairperson: Elliott Organick, University of Utah

I. David Klappholz, Columbia University

Our work is not within the data flow framework; however, most of the underlying architectural insights are in conceptual agreement with data flow. Most of the reasons why we feel our architecture will work are exactly the same as the reasons the data flow people feel their architectures will work.

Our machine is called CHoPP, the Columbia Homogeneous Parallel Processor. The only goal of the design is to achieve orders of magnitude speedup of computation. The machine is to be general purpose, but general purpose for those problems that require very much more computation than we can do in reasonable time today. This design is not directed at applications such as business data processing, where there is no parallelism, and, in fact, there's no need for speedup.

The user of our machine is required to have a parallel algorithm which he codes as a parallel algorithm. But unlike machines such as Illiac IV, CM*, or C.MMP, the user codes a parallel algorithm assuming an effectively unbounded number of autonomous processors. The actual number of processors and their configuration is totally irrelevant to him. He can also assume a conflict-free shared memory. In particular, the user should write his parallel algorithm totally disregarding any notion of memory to task assignment, processor to task assignment, overlaying, or reference resolution, these are all done for him by a combination of hardware and software.

The user language looks like a conventional language from which side effects have been removed by prohibiting shared variables from being accessed except under synchronization control. The types of constructs that we add to the language are things like Spawn Task which simply creates an instance of a process. We also add synchronize message constructs such as Read Buffer and Write Buffer, where a buffer is a mailbox or Dijkstra message buffer.

The design criteria for the processor require an architecture to be conceived without predicting in advance the rate of demand and traffic pattern of demands for memory accesses, allocation of resources, or communication. We thus have to support the maximum that can occur. To accomplish these criteria, our design has no central control and no central intelligence.

The physical interconnection structure of the machine is that of a k-dimensional binary cube. The links between the vertices are bidirectional packet transmission links. We have chosen a particular deterministic routing algorithm that has the nice property that a memory packet satisfying a memory request travels over the

same path as the packet requesting that particular fetch from memory, which will have nice side effects on the rest of the design. This routing is under local control and is deadlock free.

Each node in the machine consists of a processor and a memory bank. The memory bank is not a local memory bank, it's just one of the banks forming the large shared memory. There are also a number of input ports, a number of output ports and a routing controller, with queues located on most links.

Our machine has a latency problem in that the processor cannot execute the next instruction in a program until a packet returns from the memory with the instruction. This latency is larger than in a conventional architecture due to the communication delays through the k-cube. We eliminate this latency by executing several tasks at a time on the processor. A processor contains a bank of multiple register sets, each containing the environment for a different task, and multiplexes between tasks in order to keep busy.

I feel that the M.I.T. data flow design and our design are equivalent in that the two designs will either succeed or fail together with each other. Fundamentally, underneath the obvious differences, the intuitions that lead to both designs had to do with the feeling that central control is absolutely deadly, central intelligence is absolutely deadly, that packet architecture with various kinds of sorting networks can be used to avoid all that. The one point on which I would criticize the M.I.T. design, the one point where I think it differs significantly from ours, is that the design has communication networks, but there's no notion of forcing the traffic in the networks to be uniform.

II. Suhas Patil, University of Utah

Our ideas on architecture crystallized last summer when we tried to combine the good features of a number of the various data flow machines such as the M.I.T. machine and the Irvine machine. It's hard to beat the von Neumann architecture for doing localized computing. Hence, our processing elements are similar to microprocessors, as in the Irvine design. However, the communication bus of that design is rather cumbersome, so we utilize routing network communication structures similar to those in the M.I.T. design.

Our architecture has arbitration and distribution networks, as the M.I.T. machine, but the networks have been superimposed. This provides bidirectional links which can carry information between processors and memory and has the advantage that it is not necessary for information to travel all the way through the networks to reach a destination.

The processing units of our machine are on the order of complexity of a microcomputer, together with the memory present at the physically same location. Although the memory is associated with a processor, it is in reality one bank of a large memory system and each memory bank is accessible from any processor.

The architecture is able to exploit locality of information. In effect, the architecture makes sense if one is able to argue that locality is an important aspect of data flow machines. The structure of our communication network allows the average amount of time any information requires to travel within the machine to be small, even when the machine is operating in a mode of mimicking the M.I.T. machine.

III. Phillip Treleaven, University of Newcastle upon Tyne

The highly concurrent computing systems project at the University of Newcastle involves the investigation of a new form of general-purpose MIMD computer architecture and programming language, in which programs are regarded as implicitly parallel and serialism has to be indicated explicitly. This investigation centers on two forms of program representation, namely a machine level representation embodied in an MIMD computer architecture, which we refer to as the "generalized control flow" organization, and a new style of high level concurrent programming language, based on a "structured directed graph" representation.

The generalized control flow (GCF) organization is, at least in the abstract, somewhat conventional in appearance, consisting of (i) memory locations that are used to store data, (ii) data instructions that perform computations on this data, and (iii) control instructions which organize the execution of a computation. The major difference between the conventional control flow organization and the GCF organization is that control instructions in the latter may be viewed as forming a directed control graph through which partial controls (control signals or tokens) flow. Using control instructions like FORK and JOIN, these partial controls can fan-out to activate multiple streams of code, or can fan-in to synchronize the streams. A partial control may also activate a process which can run concurrently with the calling process.

It is the responsibility of the programming language compilers for a GCF computer to optimize the style of control and data instructions that are generated, depending both on the resources of the target computer and on the style of language being translated. For example, for a computer of a limited store size, a compiler for a conventional programming language might generate highly sequential code that reused storage, and a single partial control which simulated the operation of a program counter in a von Neumann computer. Whereas, for appropriate problems where high performance is essential and a "pure" data flow calculation is adequate, such as Mesh calculations or fast Fourier transforms, programs could be encoded in some form of concurrent programming language, probably based an a single assignment rule, and

would be translated into a data flow style of representation to take advantage of any inherent concurrency. In such cases, the control graph will correspond to the flow of data. However, for other more general types of calculations, where additional sequencing constraints seem to be needed, extra control instructions can be provided to supply just the necessary synchronization, without incurring the over-specification of sequence characteristic of von Neumann computers.

Partial controls in the GCF computer define the addresses of independent tasks (each of which can be a single instruction or a process) that are ready to be processed by the computer. These names are grouped together in what can be viewed as an unordered set of tasks that provide a.pool of work for the computer.

The basic structure of the GCF computer consists of three units: a Task Unit, storing the instruction addresses of tasks requiring processing, a Processing Unit, comprising a possibly large number of processing elements, and a Memory Unit, providing storage for the programs and data.

When a processing element becomes idle through the completion of a task, it selects a next instruction address from the Task Unit. Using this address, the processing element fetches the instruction from the Memory Unit and decodes it to obtain a specification of the required operation, the memory addresses of the input and output operands, and the address(es) of the following instruction(s). The processing element follows a conventional instruction execution cycle and then places the next instruction address(es) in the Task Unit before returning to the idle state.

The addressing mechanism of the GCF computer is central to its concurrent operation, it uniquely identifies each object (data element or instruction) stored in the Memory Unit, as well as distinguishing between concurrent activations (invocations) of a specific procedure either within a single program or within separate programs. An address within the GCF computer consists of two fields: a user name allocated at compiler time to a particular object (the relative address of an instruction) and an environment name which is allocated at runtime to identify a particular environment (a given invocation of a process). The environment name is passed around the system via the next instruction address and is appended where appropriate, by a processing element, to the addresses used in accessing the Memory Unit.

One of the key difficulties in implementing a concurrent computer, such as the GCF, is how to design the architecture so that the units which make up the computer can interact asynchronously, allowing the processing elements to access information in the Task Unit and Memory Unit in parallel without getting extensive memory clashes. The solution currently under investigation is a ring-based architecture in which all asynchronous units communicate via packets of information placed on one or more slotted rings, where a slotted ring can be viewed as a circular conveyer belt subdivided into a number of individual segments, each capable of holding a packet.

Thus, the Task Unit could be represented by a ring in which each slot contains a next instruction address. This ring-based approach also allows concurrency to be exploited in the execution of individual instructions by decomposing (pipelining) the operations performed in the normal instruction fetch-decode-execute cycle and allocating them to separate units around the ring.

In terms of performance, there is clearly a limit to the number of units which can be connected to a single ring or even a set of rings. For this reason, the ring-based computer is considered to form one module (board) in some larger GCF computer in which a number of such modules would be interconnected into a tightly coupled network. The problem of interconnecting these modules, whether to form a high-speed machine or a desk-top computer, is interlinked with the problems of reallocating the units comprising the various rings and scheduling work among the modules. Our current ideas for overcoming problems of routing information around a GCF network are based on the addressing mechanism described previously, and equating addresses in a particular range, say, to a Memory Unit in a particular module. In this case the address of an object is used for routing data in a similar fashion to a telephone number.

IV. Ian Watson, University of Manchester

The research program at the University of Manchester has as its major motivation the exploitation of parallelism in the design of very high speed machines. We view data flow as one approach to designing a flexible parallel architecture for such high speed processors. The project has as secondary motivations the realization of a cost effective and reliable design.

Our design is being carried out under two constraints: the machine must be relatively cheap and it must be fast. To hold costs down, the machine should not require a very large store. This implies that there can be no code duplication for the execution of procedures, iterative computations, or concurrent execution of a computation on parallel data. To build a fast machine, we cannot waste processing power. Nodes of a program should not be allocated to a processor until all inputs are available. Also, there can be no unnecessary holdups such as would be arise if all input tokens were available but the node description was being used or waiting for output. As a final goal, the design should be modular, allowing extensibility to realize a more powerful machine and reducibility in the event of partial failure.

To realize these goals, all tokens in the language carry a label which contains a unique procedure invocation identifier, a specification of the iteration level, and a data structure identifier specifying the array index. The node firing rule requires a set of inputs with matching labels to be available, removing the need for code duplication or unnecessary holdups.

The matching operation performed by a node is simplified through limiting the number of input tokens to a node to two and may be performed in a content addressable memory which acts as a pseudo-associative memory.

The architecture is organized as a single ring. A Switch accepts initial input tokens to a program and places them in a Result Queue on the ring. The Result Queue is simply a buffer, holding tokens which are waiting for something to happen. Each token in the Result Queue consists of a value, a label, and a node address which designates the node description to which this token is destined.

A token is removed from the head of the Result Queue and, if destined for a two-input operation, goes to a pseudo-associative Matching Store where the name field, consisting of the label and node address, is used to fetch the other operand. If the other operand is not available, the token is stored in the Matching Store. If the other token is found in the Matching Store, or if the operation requires only one operand, the token(s) travel to an Instruction Store which contains a description of nodes, each node description consisting of an operation, two addresses for results, and a literal (if required).

The operation is retrieved from the Instruction Store, and, in conjunction with the tokens, is sent off to a Processing Unit. The Processing Unit, which consists of one or many processors, accepts the operation and data items and eventually produces a result token which is presented to the Switch either for entry into the Result Queue or for transmission to the system output.

The ring operation is pipelined, with a typical length on the order of fifteen, so there is a necessity to not only have enough tokens available to keep the processing units busy, but also to fill the pipelines.

This single ring structure has some limitations. The main limitation is that there is no parallelism in the storage. The sort of technology from which we're intending to build the prototype has a store operation on the order of 200 nanoseconds and an instruction execution time of 1 to 3 microseconds. We could keep approximately 10 processors busy in such a system, but after that, adding more processors would accomplish nothing, because the store would be a bottleneck. If we look at high speed technology with 20 nanosecond stores and instruction execution speeds on the order of 100 nanoseconds, we still reach an ultimate limitation based on the store speed.

What we really want is a very high speed machine capable of providing parallelism in both processing and storage. This leads us to a multiple ring architecture, consisting of the same ring structure duplicated a number of times. The rings share a common switch to allow communication among all the rings and with the external environment. This switch is not a bottleneck due to its pipeline structure as a

binary switching tree.

We have order code and hardware simulators currently available. Preliminary simulation results suggest that we can exploit around 90 per cent of the available processor power on a single ring architecture. In a multiple ring architecture (up to 10) the figure is more on the order of 70 per cent of available power. We are currently planning and performing further simulation to get a better handle on these figures.

V. Kim Gostelow, UC - Irvine

Our work has followed two basic principles: the notion of an activity as a unit of computation and the concept of locality, or that those activities closely related in a logical way should be executed closely together in a physical way. Procedures and loops are natural boundaries of locality, but the principle is valid at many levels. In any case, we especially recognize procedures and loops and call an instance of execution of either a "logical domain."

We have been using a simulator to test our architectural ideas, and almost all of our architecture plans are now present in the simulator. Basically, our processing elements (PEs) are complex pipelined machines connected together by a token communication system, currently envisioned as a ring. Also connected to the PEs is a memory system for holding structure values. It seems that three PEs connected to a single memory controller is about right, and all memory controllers are connected together on another bus. Thus, a PE sees a single memory system which is actually a distributed system.

The memory controllers connected together on a ring form a single physical domain. We expect a physical domain to be of fixed size with, say, sixteen, thirty-two, or whatever, PEs. The complete data flow machine is then an n-dimensional collection of interconnected physical domains.

This system represents a number of changes from our original ideas. We learned early that purely logical addressing of tokens is not workable. Each PE now, just before it outputs a token, uses an assignment function to map the activity name of the output token's destination to a physical processor address. All activities in the same logical domain use the same assignment function; thus, two sources with the same destination activity will both send their tokens to the same PE.

Memory bandwidth and distance of a data structure from the requesting PE is important. In fact, we have a simple structure-copy mechanism whereby the first request to a structure in a non-local controller's memory causes that structure to be automatically copied to the requester, since much of the time further requests will be

made of that same structure by that same PE. We represent structures in memory either as balanced trees or as linear arrays (for static data such a program code).

PEs are pipelined. Since many activities may be assigned to the same PE, a stream of input tokens is constantly arriving at the input of a PE. The PE's first job is to sort tokens into groups according to activity name. When a token which completes an activity arrives, that activity's input tokens are queued to the execution section which accepts the activity and carries it out. Output tokens are produced and handed to the output section which computes the assignment function and places the tokens on the token ring. Or, if the resulting tokens are assigned to the producing PE, they are moved to the input section for sorting.

We also found that by judicious selection of operator names in the compiler and the assignment function f, the token ring can be made into a pipe. Thus, if operator i sends a token to operator j in the program, then f(i) < f(j) greatly increases the performance, where the direction of the ring is from low numbered to higher numbered PEs.

Finally, to promote even greater pipelining along the ring, we found that a second counter-rotating ring, with appropriate assignment of loops to PEs, was very effective in improving performance -- even when the two rings ran at speeds less than 1/2 the speed of the original single ring.

VI. Ken Weng, M.I.T.

My interests lie in seeing an architecture similar to the M.I.T. Form 3 or 4 machine which does not introduce additional architectural complexity to support the concepts of procedure invocation and streams.

I firmly believe that a data flow processor should support recursion instead of iteration, avoiding cyclic data flow schemas and alleviating the problems associated with variations in the execution time of instructions. The use of recursion allows a dramatic reduction in the number of acknowledge signals required in the computation. Recursion allows the specification of a semantics for streams in a very simple manner and allows efficient implementation of streams on the processor. Also, the implementation of a Forall construct can be readily based upon recursion.

If we have a procedure P within which there are two activations of another procedure Q, at runtime we must have some means of uniquely identifying the different instances of the procedure Q. One way is to have a unique identifier for each instantiation of the procedure, based on a counter or a unique identifier pool. Within an activation of a procedure, each token carries a data value and a destination address, consisting of the procedure name, the instruction number, and the unique identifier.

This forms a logical address space which is mapped (through the distribution network) into the physical memory resources.

Instead of copying a procedure upon invocation, it is possible to perform dynamic fetching of instructions upon demand. One way to assign responsibility for fetching instructions is to distinguish at compile time the arcs of these data flow graphs responsible for initiating requests for instruction fetches, allowing prefetching of instructions and avoiding the delays associated with waiting for all operands prior to fetching an instruction.

There are not really that many choices as to how procedures can be implemented. The difficulty is in choosing the best alternative within the constraints of the particular architecture.

The execution time of a data flow program can be decreased either by making the operations faster or the communication delays shorter. Since the communication delays in a data flow processor are significant, I would like to see the processor folded up in the manner suggested by Patil, with data structures and procedure activations distributed through the network. If dynamic allocation mechanisms are also used, then the communication delays in the machine should be significantly reduced.

Session 6. Implementation
Chairperson: Phil Treleaven, University of Newcastle

I. Dominique Comte, C.E.R.T.-D.E.R.I., Toulouse

The LAU multiprocessor system consists of a number of processing elements sharing a common secondary storage, a job management system, a task management system, and an I/O supervisor. Each processing element in the system executes a single task in concurrent fashion, exploiting the parallelism between the instructions of the task.

We are currently constructing a processing element which interfaces to a minicomputer host which compiles the high level programs and downloads them for execution. The processing element consists of a data control unit to hold the status of data items, an instruction control unit to hold the status of instructions under execution, a main memory to hold the data items and instructions, execution processors to perform the execution of the instructions, and bus managers to coordinate the transfer of data within the processor.

The main memory of the processing element consists of a number of smaller, interleaved banks of memory, in conjunction with an input multiplexer and an output demultiplexer. Each bank of memory has 4K by 64 bits of 480 nanosecond static RAM, with a cycle time of 480 nanoseconds for the memory. The input of a bank consists of a FIFO queue of length 40 to hold requests for that particular bank. Using a board size on the order of twelve inches by twelve inches, the eight banks of memory in the prototype system occupy eight boards, with the input multiplexer and output demultiplexer each occupying a single board.

An execution processor in the system is built as a horizontally microprogrammable sequential processor, using 2900 series ICs. Each processor has a 16 bit wide data path and a cycle time of 200 nanoseconds, using approximately 280 ICs and fitting on one standard LAU board. There will be a total of 32 processors in the system, approximately 8,950 ICs.

The control unit, consisting of the instruction control memory and the data control memory, is built of Schottky TTL technology with cycle times of 120 and 200 nanoseconds, respectively. The instruction control unit fits on two boards and utilizes 250 ICs. The data control unit occupies one board and 200 ICs.

The bus managers grant the bus to any requesting processor within one bus cycle, 60 nanoseconds. Hence, a round robin scheme is not sufficient, we must utilize a simple mechanism with a priority encoder and decoder to arbitrate among bus requests and grant the bus to one of the processors. There are two bus managers per board,

incorporating approximately 150 ICs in Schottky TTL technology.

The complete LAU machine configuration, containing 32 execution processors, consists of a total of 49 boards, representing approximately 11,900 ICs. We are well into the construction of this machine and expect the prototype to be operational within six months.

II. William Cote, Wayne State University

Rick Riccelli and I have been working on the design of a data driven processing element. A primary objective in the design was to produce a machine that executes basic data flow programs and could be expanded to handle procedures. Another objective was that the machine should have consistent control structure for all instruction types. Third, the machine should be designed around standard devices.

We felt that the machine should maintain information on the actor type: the class of instruction and the type within that class, the arc to which the output arc of the actor is directed, the presence or absence of tokens on an arc, the token values, and the specific rule for firing.

In the case of binary operations, this information is kept in a four word instruction cell, called an instruction word. For an instruction, the instruction word specifies the destination address, two input values, and the type of operation. A link has a fanout of three, so it has a single input value and three destination specifications. The switch actor has two destination addresses, a data input, and a Boolean input.

To keep track of tokens, we added in parallel with each instruction word four bits of content addressable memory (CAM). For binary operations, one bit indicates whether a token is present on the output arc, and two bits indicate if the input tokens are available. For the link, the status bits correspond to the three outputs and the input. For the switch, there are two output and two input status bits.

To know the specific firing rule for an actor, we added an extra bit to each of the four words of an instruction. This indicates whether each instruction word should be rewritten when the actor fires.

The system structure consists of the instruction memory in conjunction with the CAM flag memory, a rebroadcast queue to hold data values whose destinations are currently occupied, a set of functional units, instruction, data, and control buses to distribute packets among the functional units, and a control section to handle traffic on the buses.

A sequence control unit maintains a list of the functional units which are available. This unit interrogates the CAM for an instruction in the enabled state which can be executed on that type of functional unit. At the same time, it tells an instruction control unit which functional unit is to be utilized. The instruction packet is then read out of the instruction memory and sent off to the functional unit. When the functional unit completes its computation, it signals a data-in control unit that it wishes the data bus. When the data bus is available, the data-in control informs the functional unit that it has the bus, and the result value is transmitted to the data-in control. The data-in control then interrogates the CAM to see if the destination arc is available. If so, the data item is written into the instruction memory; otherwise, it is placed in the rebroadcast queue.

III. Milos Ercegovac, UCLA

I've been studying algorithms for reducing the bandwidth of interconnection networks and achieving a low-cost speed-up in overlapping computations. By allowing for certain unusual properties at the data representation level, one is able to obtain algorithms with unusual properties.

An on-line algorithm is characterized by the property that the result is computed in a digit-by-digit fashion, with the most significant digit first. The j-th digit of the result is computed as soon as $j+\delta$ digits of the operands are available. The on-line delay δ is just one for addition, subtraction, multiplication, and square root. For division, δ is three or four. This gives us a significant speedup in computation.

Since we communicate at the level of digits, these algorithms allow reduced bandwidth of interconnections and I/O. They allow the use of low-cost error-detection arithmetic codes. Their use allows a modular implementation which is extendable through the addition of identical modules. In addition, significance monitoring is easy to incorporate in such a system. Such a system can be formed of reconfigurable arrays of on-line units as special operators to match the problem structure.

These techniques allow an increase in computation speed through the addition of a simple on-line arithmetic unit to the instruction cells of the machine. This would permit local processing of arithmetic operations, significantly reducing the packet traffic in the communication networks.

IV. Robert Keller, University of Utah

I'm going to speak about the evaluation model for our loosely-coupled parallel processor. The use of the language Lisp is motivated primarily from consideration of symbol manipulation applications rather than numeric applications.

However, there are characteristics of the language that should be of use to the LLL people, at least by transitivity.

Flow graph Lisp has the domain of binary trees over some set, say the set of integers united with the set of characters and some special terminator symbol nil. The operators in the language will be the cons operator, the converse operators car and cdr, a test to see if a tree is an atom, and an "if then else" conditional, in addition to the usual simple arithmetic operators. We represent procedure invocation in the language by productions in a graph grammar. The occurrence of a symbol representing a procedure should be thought of as being replaced by the body of the procedure represented by the symbol.

An array is represented as a list, which is a special case of a tree in which all the right-hand subtrees are simply atoms. A two dimensional array is just a list of lists, and so forth. This representation has the disadvantage that it doesn't take advantage of the contiguity of storage allocation, but that's an optimization problem that we should be able to cope with.

Strict cons serves two roles -- it's a paring function and a synchronizing function, evaluating both arguments. Lenient cons, on the other hand, serves merely as a pairing function, no evaluation is performed. This has important properties for increasing the asynchronism and hence the parallelism in a computation. Also, there are certain good semantic properties that hold for lenient cons that don't hold in general. For example, taking the car of a cons of two things allows us to "short-circuit" one of the inputs to the outputs. This allows us to do computations on trees while they're still being created, permitting the use of streams and streams of streams. Thus, proper implementation requires a demand-driven evaluation strategy.

The demand-driven strategy of our evaluator simplifies the program graphs (especially conditionals), provides a great deal of flexibility, yielding a nice way of deciding when to expand procedures and preventing run-away recursions, and doesn't require arrival of all the data prior to initiation of the evaluation. The evaluator copies the code, which is an advantage in our case by preventing contention for memory locations. However, this is not yet a firm decision, our scheme may easily be converted to pure code. We can do recursion without the procedure-instantiation labels of the M.I.T. and Irvine machines. There's no associative memory involved in the machine, except possibly to find the meaning of a procedure name. This memory can be eliminated if there is no dynamic creation of processes.

The remaining work contains a large number of optimizations which must be performed, both in terms of conventional code optimization and the representation of the data and the evaluator itself. We have to consider space allocation and reclamation. There is the possibility of combining data and demand driven computations which has interesting implications.

V. Clement Leung, M.I.T.

Our group has conducted quite a bit of work on implementation. Katsu Amikura studied the logic design of a cell block in our processor. An S.B. thesis has recently been completed on the design of functional units based on on-line algorithms, in the fashion studied by Milos Ercegovac. Work is continuing on an architecture description language which we expect to use as a hardware specification tool and on how to incorporate fault tolerance into the hardware design.

My work involves the study of fault tolerance. The model I am working with is an interconnection of modules, called a packet communication system. Each module of the system has a well defined interface in terms of a set of input ports and a set of output ports. Each port communicates over wires which carry a data encoding using an asynchronous packet communication protocol. The research has the goal of specifying certain properties for how to design these modules. If we can build modules that satisfy these goals, then we can build fault tolerant systems.

The failure modes in the model are "stuck-at-0", "stuck-at-1", and "transmit a random pulse train." Given these failure modes and modules which are designed to satisfy the property that the electrical signals presented at an input port are always interpreted by the hardware as a stream of packets, then the behavior of a faulty module falls in one of a few classes.

The fault-tolerance study will develop methods of providing complete single-fault coverage; that is, a system will be able to continue correct operation following a single-module failure. We would like a failure to be readily diagnosable and easily serviceable, and we would like the fault tolerant property to be realized in a way that avoids increasing software complexity.

We are studying the use of modular redundancy techniques in the implementation of fault tolerance. However, use of these techniques in asynchronous systems raises a number of problems such as how to allow the system to treat anomalies in module behavior that appear as long delays before any output is produced.

The overall strategy for incorporating fault tolerance in a data flow processor involves the utilization of redundant structure and information in the various subsystems of the processor. Since a different redundant representation may be more viable in each subsystem, we must provide conversion between the redundant representations at the subsystem interfaces. The different kinds of subsystems appear to require different techniques to provide graceful degradation in the event of component failure, and the studies of how to incorporate such facilities in each case

are in a sense independent.

VI. Daniel Schwabe, UCLA

The Systems Architects Apprentice (SARA) being developed at UCLA by Gerald Estrin and Wilson Ruggiero provides an environment suitable for the design of systems utilizing data flow architecture. SARA provides simulation and analysis tools and supports a design methodology which has multilevel modeling power, supporting abstraction and refinement of designs and composition of pretested or preverified building block models. The system supports a requirement-driven design discipline, providing a well-rounded system for computer system development.

SARA could be used to design data flow architectures through refinement of the design down to the composition of data flow primitives. We could extend our PL/1 preprocessor PLIP to include the data flow primitives. Alternatively, we can represent data flow primitives in the graph model of behavior (GMB) primitives. Furthermore, data flow primitives could be transformed into existing control and data flow models. This would then allow us to create a library of preestablished SARA building blocks which have been implemented with data flow primitives and compose systems with these building block models.

SARA is a requirement-driven system. Every SARA design begins with a set of requirements to be satisfied by the system and a set of requirements (or assumptions) to be met by the environment of the system. To manage complexity, SARA supports multilevel models with either top down refinement or bottom up abstraction. Every design must end by composing models of well-understood building blocks.

SARA is currently a collection of programs residing on a 360/91 at UCLA and on the Multics system at M.I.T. and can be accessed via the ARPA network. The system is still under development and usable at your own risk.

VII. Ian Watson, University of Manchester

During the next three years we will be building a single ring prototype machine based on the principles described in our previous talk in the architecture session. The machine will be constructed from medium speed technology to investigate design principles and will be extensible to a multiple ring architecture at a later date.

The processing unit will consist of ten Schottky bit slice microprocessors, working on a 32 bit word with a floating point add time on the order of three microseconds. The result queue will be of size 16K by 96 bits with a 200 nanosecond

read/write cycle. A word in the queue will consists of 32 bits of data, a 36 bit label, and 24 bits for the node address, with 4 extra bits. The matching store will be the same size. The instruction store will be 16K by 32 bits with the same cycle time. The overall instruction execution speed of the processor will be approximately three MIPS.

The processing unit will be micro-programmed with a RAM microprogram store to ensure flexibility. The unit will consist of the ten processors connected by an input bus and an output bus, with an input control unit to allocate an executable node to the first free processor in sequence and an output control unit to accept results from the processors.

The result queue is to be implemented as a RAM with recirculating pointers to simulate a first-in, first-out queue. Similar schemes using two independent blocks of store are under consideration to permit concurrent read-write. The instruction store utilizes a conventional RAM employing a simple segmentation system to designate a location in the store. The matching store is built from RAM and utilizes hardware parallel hash techniques to act as a pseudo content addressable memory with an access time similar to that of RAM.

The prototype machine is to contain approximately 2500 ICs, of which 1000 are used in the store, and to cost approximately 50,000 pounds ($95,500).

Session 7. Performance and Simulation
Chairperson: Susan Conry, Clarkson College

I. Randy Bryant, M.I.T.

I'm interested in developing analytical models for interconnection networks, in particular, networks for interconnecting a large number of independently operating parallel units. I'm focusing initially on the arbitration and distribution networks of the M.I.T. data flow processors, but I feel that the general ideas which I'm studying apply to many of the other networks which have been discussed here.

My proposed approach is to condense the elements in the network into a series of groups of elements, called chunks. Packets are then modeled as a fluid or continuous medium flowing between the chunks. This approach is inspired by the manner in which things are done in fluid mechanics and physics.

A system is characterized at any point in time by a number of state variables. A set of equations can then be developed to describe how the state variables change with time. These characteristic equations should form a set of differential equations, highly nonlinear and not corresponding to any known physical processes. A characteristic equation for the network is then built up from conservation of packets, the system structure, the system parameters, and the assumed distribution of packets.

II. Andy Boughton, M.I.T.

My work is concerned with the development of structures that one might use to design the routing networks in the M.I.T. data flow machine. An arbitration network of the machine has a large number of inputs and a smaller number of outputs, and each input has a tag specifying on which output it is to appear. I decompose the arbitration network into a concentration network that doesn't look at the tags, funneling packets from a large number of inputs to a smaller number of outputs, and another network that has the same number of inputs as outputs and does all the routing of packets. It can be shown that nothing is lost by ignoring the tags in the first stage of this decomposed system. Similarly, a distribution network (with fewer inputs than outputs) can be decomposed into a square concentration network and a tree of switches.

Last year I described the design of concentration networks. This year my work has been concerned with the design of connection networks, those networks with an equal number of inputs and outputs which sort the packets on the inputs and distribute each to the output specified by the associated tag. The easiest way to do

this is through use of a crossbar switch consisting of a switch tree, a level of FIFO buffers, and a level of arbitration. However, this approach requires N^2 FIFOs and is too expensive for general use.

An alternate approach involves limiting the size of the structure by breaking the network into a number of stages, each stage consisting of a column of switches followed by a column of arbitration units. I'm currently looking at other methods of splitting up concentration networks to keep both the width and the depth at minimum values. I'm also looking at ways of avoiding such networks, together with their artificial constraints on the flow of data, through the use of certain probabilistic network structures.

III. Arch Oldehoeft, Iowa State University

Our work has the objective of measuring the parallelism in computer programs that can be exploited by data flow machines. To this end, we've looked at a number of real programs, not necessarily programs with a lot of parallelism. Some of our future efforts will involve looking for applications where parallelism can be exploited.

We assume an underlying architecture which is a feedback interpreter with operands received from predecessor instructions and acknowledgments from successor instructions. The data types include real, integer, Boolean, structures, and sequential files. The operations we allow are the standard arithmetic, logical, and trigonometric operations; select and append; read, write, readedit and writedit for files; and apply for procedures.

Our first approach involved building a simulator for a data flow machine and capturing measurements during simulated execution of programs. This approach had the drawback of being a very expensive process, every data flow instruction cost between .7 and 1 cent to simulate. In addition, the memory constraints of the computer we were using would often not allow us to execute the program at all. Furthermore, the statistics we realized were often difficult to relate to the program being simulated.

Our second approach was to construct the program graph at some level of detail and derive formulas which characterize a program in terms of execution time. We first assigned numeric values as execution times for base machine instructions. This information was used during compilation to construct characteristic timing equations for the sequential execution time and parallel execution time as functions of the branching probabilities and the number of loop iterations. We then formed a ratio of the two times and used that ratio as a criteria for parallelism.

The numbers realized from this graphical approach differed by less than ten per cent for most programs from the numbers realized from simulation of the programs. The major difference between the two approaches appears to be in the hiding of overlap in computation caused by the approximations in the graphical approach. The major culprit in this is the repeat-until clause. Other causes are possibly inaccurate estimates of branching probabilities and the possible inaccuracies introduced by the assumption that innermost loops dominate the computation.

We have done some preliminary work on the incorporation of streams and vector operations into the simulator. While much work remains to be done in this area, the preliminary results indicate that, for appropriate computations, significantly larger speedups are achievable through use of data flow techniques.

IV. Roy Zingg, Iowa State University

Our simulation work has tried to avoid tying the simulation to any specific implementation. We are more interested in just what parallelism there is in a program and the consequent resource demands, than how close some implementation can come to efficiently executing the program.

The current version of the simulator measures only scalar parallelism, there is no unraveling, no streams, no vector functional units. The simulation incorporates acknowledge signal concepts and has essentially a complete backward flow of tokens for that purpose. Also, the simulator explicitly implements merge instructions.

The simulator consists of an instruction memory containing currently active procedures. When a result is sent to an instruction, the instruction identifier is put on a check list as potentially ready for execution. An enabler examines items on the check list and places enabled instructions on the ready list. The enabled instructions are then transferred to an execution list which is maintained by scheduled completion times for the instruction packets on the list. When an instruction is put on the execution list, acknowledge signals are sent to the appropriate predecessor instructions.

A decoder distributes instruction packets from the execution list to the functional units as they are scheduled for execution. At the conclusion of their execution, results are sent back to the instruction memory. A file memory is used for I/O, and a program memory contains copies of procedures not currently active.

We simulated student programs, System 360 subroutine packages, and algorithms from CACM. The programs performed such operations as root finding, numerical integration, and solution of simultaneous linear equations. Program sizes ranged from 69 data flow instructions (35 high level language statements) to 353 data

flow instructions (137 high level language statements).

For each program a single set of input data was selected to exercise those branches most likely to be exercised in practice and to ensure at least five iterations of each loop encountered. An adequate number of functional units were allocated to insure minimum parallel execution time. We assumed that most functions took one unit of simulated execution time, with the exceptions that trigonometric and hyperbolic functions took two units, exponential and inverse trigonometric functions required three, and identity and merge in general required zero units of time.

The simulation achieved speedups on the order of 1.5 to 3.5. If sequential I/O is ignored, the speedups range up to 4. We find that in some cases, the parallelism is due to the I/O, and in other cases, the I/O is holding it up. The results of the simulation indicate that in order to achieve significant degrees of parallelism, we have to take advantage of things that are vectorizable.

Session 8. Specification and Verification
Chairperson: Arch Oldehoeft, Iowa State University

I. Earl Schweppe, University of Kansas

I'm reporting on some work done by Elizabeth Unger and myself on a concurrent model or a natural model of concurrent computation. We hope to achieve concurrency intrinsically and naturally with a language that is locally unordered, but well structured. The basic control philosophy is data driven with the addition of constructs which we call stimulation and termination.

A simple statement in the language, called a request, specifies that an action is to be applied to certain material objects which will produce other resultant objects. The basic principle governing the ordering of execution among requests within the model is data driven. This ordering is based upon the availability of the action, the data object, and an authorization to use each object involved.

The model incorporates two conditions, which are simple Boolean expressions. These stimulation conditions and termination conditions govern the eligibility for execution and termination of execution of an action, respectively.

We've done some work on aggregation of data objects. We provide for both ordered and unordered collections of objects. We have also done work on repetition (similar to the Forall constructs), which we treat as being distinct from iteration or recursion. Recursion occurs without anything special, we allow activation without all data necessary for completion. We have done some formal work, we can easily prove that the language is universal. With some restriction, we can prove determinacy. We have put a fair amount of effort in comparing the model with other concurrent models such as concurrent Pascal. Our model has the advantages over the other models that it unifies a number of things in a manner that is simple and natural, it is oriented toward a data processing production environment, it is oriented toward a diverse network environment, and it provides very flexible control over activities.

II. Bill Ackerman, M.I.T.

Dean Brock and I have shown that history functions cannot possibly describe non-determinate packet communication systems. A packet communication system is a conceptual model that arises in both hardware and software and turns out to be of great use in data flow systems. The formal semantics of packet systems are important to both software and hardware. For software, they allow definition of programming languages for verification purposes. For hardware, the semantics allow verification of the hardware structure.

The semantic model of a packet system is that input data values enter a module or subsystem asynchronously and results of the computation are eventually emitted, with no timing constraints. This is the common model of the semantics of software systems, but is unusual for hardware. The semantics of packet systems is generally represented by history functions which relate the total history of all the tokens that have entered the system to the history of the tokens it will eventually emit.

There exists a theorem that, for determinate systems, the interconnection of any collection of determinate systems is determinate and its history function can be computed from nothing but the history functions of the components and the method of their connection. This capability demonstrates that history functions are a consistent semantic model, abstracting a consistent amount of information.

Similarly, we can characterize a nondeterminate system by a nondeterminate history function, that is a history function which maps each input history into the set of possible output histories that could have occurred. This can be specified with a history relation or a multiple-valued function. It can be shown that history functions form a consistent semantic model for acyclic systems.

However, the model breaks down when the systems are both nondeterminate and cyclic. We can demonstrate this by exhibiting two systems which are indistinguishable in terms of their history functions but which, in fact, behave differently when interconnected. In particular, there is a third system in which these two systems cannot be substituted for each other without affecting the history function of the larger system.

III. Suhas Patil, University of Utah

At last year's workshop, I indicated how one might implement the units of the M.I.T. data flow computer using programmable logic arrays to provide the necessary flexibility at a reasonable cost. Last year it was not clear what facilities would be available for creating these devices. I would like to take this time to give a brief update on what might be possible.

The National Science Foundation has given the University of Utah a grant to set up facilities for creating the first set of logic arrays as a feasibility demonstration. We hope to have the first of these devices ready by the end of 1979. The devices we are thinking about will have 48 columns and 256 rows and are to be implemented in either I^2L, short-channel MOS, or CMOS. The initial devices will be mask programmed, but we envision the appearance of field programmable and erasable devices soon after. A number of semiconductor manufacturers have indicated an interest in production of these chips after our development, the time scale on that is

perhaps two years.

BIBLIOGRAPHY

Ackerman, W. B., *A Structure Controller for Data Flow Computers*, Computation Structures Group (Memo 156), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, January 1978.

Ackerman, W. B., *A Structure Memory for Data Flow Computers*, Laboratory for Computer Science (TR-186), MIT, Cambridge, Massachusetts, August 1977.

Ackerman, W. B., "A Structure Processing Facility for Data Flow Computers," *Proceedings of the 1978 International Conference on Parallel Processing*, August 1978.

Ackerman, W. B., and J. B. Dennis, *VAL -- A Value-Oriented Algorithmic Language: Preliminary Reference Manual*, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, In Preparation.

Adams, D. A., *A Computation Model With Data Flow Sequencing*, School of Humanities and Sciences (Technical Report CS 117), Stanford University, Stanford, California, December 1968.

Adams, D. A., "A Model for Parallel Computations," *Parallel Processor Systems, Technologies, and Applications* (L. C. Hobbs, D. J. Theis, J. Trimble, H. Titus, and I. Highberg, Eds.), Spartan Books, New York, New York, 1970, 311-333.

Amikura, K., *A Logic Design for the Cell Block of a Data Flow Processor*, Laboratory for Computer Science (TM-93), MIT, Cambridge, Massachusetts, December 1977.

Arvind, and K. P. Gostelow, "A Computer Capable of Exchanging Processors for Time," *Information Processing 77: Proceedings of IFIP Congress 77* (B. Gilchrist, Ed.), August 1977, 849-853.

Arvind, and K. P. Gostelow, *Dataflow Computer Architecture: Research and Goals*, Department of Information and Computer Science (TR 113), University of California - Irvine, Irvine, California, February 1978.

Arvind, and K. P. Gostelow, "Microelectronics and Computer Science," *Proceedings of the 2nd IEEE(G PHP)/ISHM University/Industry/Government Microelectronics Symposium*, University of New Mexico, Albuquerque, New Mexico, January 1977.

Arvind, and K. P. Gostelow, *A New Interpreter for Data Flow and Its Implications for Computer Architecture*, Department of Information and Computer Science (TR 72), University of California - Irvine, Irvine, California, October 1975.

Arvind, and K. P. Gostelow, *Semantics of Loop Expressions in ID*, UCI Dataflow Architecture Project (Note 11), University of California - Irvine, Irvine, California, March 1977.

Arvind, and K. P. Gostelow, "Some Relationships Between Asynchronous Interpreters of a Dataflow Language," *Formal Description of Programming Concepts* (E. J. Neuhold, Ed.), August 1977, North-Holland Publishing Company, New York, New York, 95-119.

Arvind, K. P. Gostelow, and W. Plouffe, "Indeterminacy, Monitors and Dataflow," *Proceedings of the Sixth ACM Symposium on Operating Systems Principles, Operating Systems Review 11*, 5(November 1977), 159-169.

Arvind, K. P. Gostelow, and W. Plouffe, *The (Preliminary) Id Report*, Department of Information and Computer Science (TR 114), University of California - Irvine, Irvine, California, May 1978.

Arvind, K. P. Gostelow, and W. Plouffe, *Programming in a Viable Data Flow Language*, Department of Information and Computer Science (TR 77), University of California - Irvine, Irvine, California, March 1976.

Ashcroft, E. A., and W. W. Wadge, "Lucid, a Nonprocedural Language with Iteration," *Communications of the ACM 20*, 7(July 1977), 519-526.

Baer, J. L., D. P. Bovet, and G. Estrin, "Legality and Other Properties of Graph Models of Computations," *Journal of the ACM 17*, 3(July 1970), 543-552.

Bährs, A., "Operation Patterns: An Extensible Model of an Extensible Language," *International Symposium on Theoretical Programming* (A. Ershov, and V. A. Nepomniashy, Eds.), *Lecture Notes in Computer Science 5*, 1972, 217-246.

Bic, L., *A Basic Model for Protection in Dataflow*, UCI Dataflow Architecture Project (Note 22), Department of Information and Computer Science, University of California - Irvine, Irvine, California, September 1977.

Bic, L., *Confined Interprocess Communication*, UCI Dataflow Architecture Project (Note 26), Department of Information and Computer Science, University of California - Irvine, Irvine, California, March 1978.

Bic, L., *An Extended Model for Protection in Dataflow*, UCI Dataflow Architecture Project (Note 23), Department of Information and Computer Science, University of California - Irvine, Irvine, California, November 1977.

Bic, L., *Propriety Services for Dataflow*, UCI Dataflow Architecture Project (Note 27), Department of Information and Computer Science, University of California - Irvine, Irvine, California, February 1978.

Bic, L., *Protection in Dataflow*, UCI Dataflow Architecture Project (Note 25), Department of Information and Computer Science, University of California - Irvine, Irvine, California, November 1977.

Bic, L., *Security and Protection in a Dataflow Computer System*, Department of Information and Computer Science, University of California - Irvine, Irvine, California, February 1978.

Boughton, G. A., *Routing Networks and Data Flow Architectures*, S. M. Thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Massachusetts, May 1978.

Brock, J. D., *Operational Semantics of a Data Flow Language*, S. M. Thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Massachusetts, September 1978.

Brock, J. D., and W. B. Ackerman, *An Anomaly in the Specifications of Nondeterminate Packet Systems*, Computation Structures Group (Note 33-1), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, January 1978.

Bryant, R. E., *Simulation of Packet Communication Architecture Computer Systems*, Laboratory for Computer Science (TR-188), MIT, Cambridge, Massachusetts, November 1977.

Bryant, R. E., and J. B. Dennis, *Concurrent Programming*, Computation Structures Group (Memo 148-2), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, July 1978. To appear in *Research Directions in Software Technology* (P. Wegner, Ed.), 1978.

Campos, I. M., and G. Estrin, "Concurrent Software System Design Supported by SARA at the Age of One," *Third International Conference of Software Engineering*, May 1978, 230-242.

Campos, I. M., and G. Estrin, "SARA Aided Design of Software for Concurrent Systems," *AFIPS Conference Proceedings 47, 1978 National Computer Conference*, 325-336.

Chamberlin, D. D., "The 'Single-Assignment' Approach to Parallel Processing," *AFIPS Conference Proceedings 39, 1971 Fall Joint Computer Conference*, November 1971, 263-269.

Ciccarelli, E., *Strict Semantic Equations for Data Flow Programs*, Computation Structures Group (Note 26), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, August 1976.

Comte, D., G. Durrieu, O. Gelly, A. Plas, and J. C. Syre, "Parallelism, Control and Synchronization Expressions in a Single Assignment Language," *SIGPLAN Notices 13*, 1(January 1978), 25-33.

Conry, S. E. (see also S. C. Meyer)

Conry, S. E., and J. R. Jump, "Functional Equivalences in a Model for Parallel Computation," to appear in *Information and Control.*

Cote, W. F., and R. F. Riccelli, "The Design of a Data Driven Processing Element," *Proceedings of the 1978 International Conference on Parallel Processing*, August 1978.

Crooks, L., *Analysis of Airplane Collision Avoidance Algorithm Written in Data Flow*, Computation Structures Group (Note 25), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, May 1976.

Davis, A. L., "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine," *Proceedings of the Fifth Annual Symposium on Computer Architecture, Computer Architecture News 6*, 7(April 1978), 210-215

Davis, A. L., *The Architecture of DDM1: A Recursively Structured Data Driven Machine*, Technical Report UUCS-77-113, University of Utah, Salt Lake City, Utah, October, 1977.

Davis, A. L., *Data Driven Net Queueing Phenomenon*, Burroughs IRC Report, San Diego, California, 1975.

Davis, A. L., *An Overview of Data Driven Machine I*, Technical Report, Burrought ASDO, San Diego, California, 1976.

Davis, A. L., *Structured Data*, Burroughs IRC Report, San Diego, California, 1975.

Davis, A. L., *SPL - A Structured Programming Language*, Ph. D. Thesis, University of Utah, Salt Lake City, Utah, 1972.

Davis, A. L., *System and Method for Concurrent and Pipeline Processing Employing a Data Driven Network*, U. S. Patent 3,978,452, issued August 31, 1976.

Davis, A. L., *Systems Aspects of Data Driven Nets*, Burroughs IRC Report, San Diego, California, 1975.

Denning, P. J, "Operating Systems Principles for Data Flow Networks," *Computer 11*, 7(July 1978), 86-96.

Dennis, J. B., "First Version of a Data Flow Procedure Language," *Programming Symposium: Proceedings, Colloque sur la Programmation* (B. Robinet, Ed.), *Lecture Notes in Computer Science 19*, 1974, 362-376.

Dennis, J. B., "A Language Design for Structured Concurrency," *Design and Implementation of Programming Languages: Proceedings of a DoD Sponsored Workshop* (J. H. Williams and D. A. Fisher, Eds.), *Lecture Notes in Computer Science 54*, October 1976.

Dennis, J. B., "Packet Communication Architecture," *Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing*, August 1975, 224-229.

Dennis, J. B., "Programming Generality, Parallelism and Computer Architecture," *Information Processing 68*. North-Holland Publishing Co., Amsterdam, Netherlands, 1969, 484-492.

Dennis, J. B., and J. B. Fosseen, *Introduction to Data Flow Schemas*, Computation Structures Group (Memo 81), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, September 1973.

Dennis, J. B., J. B. Fosseen, and J. P. Linderman, "Data Flow Schemas," *International Symposium on Theoretical Programming* (A. Ershov, and V. A. Nepomniashy, Eds.), *Lecture Notes in Computer Science 5*, 1972, 187-216.

Dennis, J. B., C. K. C. Leung, and D. P. Misunas, *Specification of the Instruction Cell Block for a Data Flow Processor*, Computation Structures Group (Data Flow Design Note 1), Laboratory for Computer Science, MIT, December 1975.

Dennis, J. B., and D. P. Misunas, "A Computer Architecture for Highly Parallel Signal Processing," *Proceedings of the ACM 1974 National Conference*, November 1974, 402-409.

Dennis, J. B., and D. P. Misunas, *Data Processing Apparatus for Highly Parallel Execution of Stored Programs*, U. S. Patent 3,962,706, issued June 8, 1976.

Dennis, J. B., and D. P. Misunas, *The Design of a Highly Parallel Computer for Signal Processing Applications*, Computation Structures Group (Memo 101), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, August 1974.

Dennis, J. B., and D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," *The Second Annual Symposium on Computer Architecture: Conference Proceedings*, January 1975, 126-132.

Dennis, J. B., D. P. Misunas, and C. K. C. Leung, *A Highly Parallel Processor Using a Data Flow Machine Language*, Computation Structures Group (Memo 134), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, January 1977. To appear in *IEEE Transactions on Computers*.

Dennis, J. B., and K.-S. Weng, "Application of Data Flow Computation to the Weather Problem," *High Speed Computer and Algorithm Organization* (D. J. Kuck, D. H. Lawrie, and A. H. Sameh, Eds.), 1977, 143-157.

Ellis, D. J, *Formal Specifications for Packet Communication Systems*, Laboratory for Computer Science (TR-189), MIT, Cambridge, Massachusetts, November 1977.

Estrin, G., "A Methodology for Design of Digital Systems - Supported by SARA at the Age of One," *AFIPS Conference Proceedings 47, 1978 National Computer Conference*, 313-324.

Estrin, G., and R. Turn, "Automatic Assignment of Computations in a Variable Structure Computer System," *IEEE Transactions on Electronic Computers EC-12*, 6(December 1963), 755-773.

Feridun, A. M., *Design of an On-Line, Byte-Level Pipelined Arithmetic Processor*, Computation Structure Group (Memo 162), Laboratory for Computer Science, Cambridge, Massachusetts, July 1978.

Fitzwater, D. R., and E. J. Schweppe, "Consequent Procedures in Conventional Computers," *AFIPS Conference Proceedings 26, Fall Joint Computer Conference*, 465-476, 1964.

Friedman, D. P., and D. S. Wise, "The Impact of Applicative Programming on Multiprocessing," *Proceedings of the 1976 International Conference on Parallel Processing*, (P. H. Enslow Ed.), August 1976, 263-272.

Gelly, O., *et al.*, "LAU System Software: A High Level Data Driven Language for Parallel Programming," *Proceedings of the 1976 International Conference on Parallel Processing* (P. H. Enslow, Ed.), August 1976, 255.

Glauert, J., *A Single Assignment Language for Data Flow Computing*, M. Sc. Dissertation, Department of Computer Science, University of Manchester, Manchester, England, January 1978.

Gurd, J., and I. Watson, "A Multilayered Data Flow Computer Architecture," *Proceedings of the 1977 International Conference on Parallel Processing* (J. L. Baer, Ed.), August 1977, 94.

Gurd, J., I. Watson, and J. Glauert, *A Multilayered Data Flow Computer Architecture*, Department of Computer Science, University of Manchester, Manchester, England, July 1978.

Glushkov, V. M., M. B. Ignatyev, V. A. Myasnikov, and V. A. Torgashev, "Recursive Machines and Computing Technology," *Information Processing 74: Proceeding of the IFIP Congress 74* (J. L. Rosenfeld, Ed.), 1974, 65-70.

Hankin, C. L., P. E. Osmon, and J. A. Sharp, *A Data Flow Model of Computation*, Department of Computer Science, Westfield College, Hampstead, London, 1978.

Jacobsen, R. G., *Analysis of Structures for Packet Sorting Networks*, Computation Structure Group (Memo 163), Laboratory for Computer Science, Cambridge, Massachusetts, July 1978.

Jacobsen, R. G., and D. P. Misunas, "Analysis of Structures for Packet Communication," *Proceedings of the 1977 International Conference on Parallel Processing* (J. L. Baer, Ed.), August 1977, 38-43.

Jaffe, J. M., "The Use of Queues in the Parallel Data Flow Evaluation of 'If-Then-Else', 'While' Programs," *Proceedings of the 1978 Conference on Information Sciences and Systems*, March 1978, 451-456.

Jaffe, J. M., and L. Montz, *Two Data Flow Solutions of Laplace's Equation*, Computation Structures Group (Note 37), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, July 1978.

Kahn, G., "The Semantics of a Simple Language for Parallel Programming," *Information Processing 74: Proceeding of the IFIP Congress 74* (J. L. Rosenfeld, Ed.), 1974, 471-475.

Kahn, G., and D. MacQueen, "Coroutines and Networks of Parallel Processes," *Information Processing 77: Proceedings of IFIP Congress 77* (B. Gilchrist, Ed.), August 1977, 993-998.

Karp, R. M., and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM Journal of Applied Mathematics 14*, (November 1966), 1390-1411.

Kessels, J. L. W., "A Conceptual Framework for a Nonprocedural Programming Language," *Communications of the ACM 20*, 12(December 1977), 906-913.

Kosinski, P. R., "A Data Flow Language for Operating Systems Programming," *Proceedings of ACM SIGPLAN-SIGOPS Interface Meetings, SIGPLAN Notices 8*, 9(September 1973), 89-94.

Kosinski, P. R., *A Data Flow Programming Language*, IBM T. J. Watson Research Center (RC 4264), Yorktown Heights, New York, March 1973.

Kosinski, P. R., *Denotational Semantics of Determinate and Non-Determinate Data Flow Programs*, Ph. D. Thesis in preparation, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Massachusetts, expected January 1979.

Kosinski, P. R., "Mathematical Semantics and Data Flow Programming," *Conference Record of the Third ACM Symposium on Principles of Programming Languages*, January 1976, 95-103.

Kosinski, P. R., "A Straightforward Denotational Semantics for Non-Determinate Data Flow Programs," *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, January 1978, 214-221.

Leung, C. K. C., *ADL: An Architecture Description Language for Packet Communication Systems*, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, In preparation.

Leung, C. K. C., *Formal Properties of Well-Formed Data Flow Schemas*, Laboratory for Computer Science (TM-66), MIT, Cambridge, Massachusetts, June 1975.

Leung, C. K. C., D. P. Misunas, A. Neczwid, and J. B. Dennis, "A Computer Simulation Facility for Packet Communication Architecture," *Third Annual Symposium on Computer Architecture: Conference Proceedings*, January 1976, 58-63.

Lloyd, S. C., *Parallel Computation Schemata (PCS): A Constructively Determinate Model with Dynamic Operand Resolution and Distributed Control*, Ph. D. Thesis, Department of Computer Science, Duke University, Durham, North Carolina, December 1974.

McNally, M. E., *The Design of an Arbitration Network for a Data-Flow Processor*, Computation Structure Group (Memo 164), Laboratory for Computer Science, Cambridge, Massachusetts, July 1978.

Meyer, S. C. (see also S. E. Conry)

Meyer, S. C., *An Analysis of Two Models for Parallel Computation*, Ph. D. Thesis, Department of Electrical Engineering, Rice University, Houston, Texas, December 1974.

Meyer, S. C., "An Analytic Approach to Performance Analysis for a Class of Data Flow Processors," *Proceedings of the 1976 International Conference on Parallel Processing* (P. H. Enslow, Ed.), August 1976, 106-115.

Miller, R. E., and J. Cocke, "Configurable Computers: A New Class of General Purpose Machines," *International Symposium on Theoretical Programming* (A. Ershov, and V. A. Nepomniashy, Eds.), *Lecture Notes in Computer Science 5*, 1972, 285-298.

Minne, J., *Stream Programming in RED Lanugages*, UCI Dataflow Architecture Project (Note 24), Department of Information and Computer Science, University of California - Irvine, Irvine, California, December 1977.

Miranker, G. S., *An Approach For Proving Packet Communications Architectures Correct*, Computation Structures Group (Note 27), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, October 1976.

Miranker, G. S., *Design and Correctness of a Data Flow Procedure Mechanism*, S. M. Thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Massachusetts, January 1977.

Miranker, G. S., "Implementation of Procedures on a Class of Data Flow Procedures," *Proceedings of the 1977 International Conference on Parallel Processing* (J. L. Baer, Ed.), August 1977, 77-86.

Miranker, G. S., *Implementation Schemes for Data Flow Procedures*, Computation Structures Group (Memo 138), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, May 1976.

Miranker, G. S., *Proving Packet Communications Architectures Correct*, Computation Structures Group (Memo 143), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, September 1976.

Misunas, D. P., *A Computer Architecture for Data-Flow Computation*, Laboratory for Computer Science (TM-100), MIT, Cambridge, Massachusetts, March 1978.

Misunas, D. P., "Deadlock Avoidance in Data-Flow Architecture," *Proceedings of the Third Milwaukee Symposium on Automatic Computation and Control*, April 1975.

Misunas, D. P., "Error Detection and Recovery in a Data-Flow Computer," *Proceedings of the 1976 International Conference of Parallel Processing* (P. H. Enslow, Ed.), August 1976, 117-122.

Misunas, D. P., "Performance Analysis of a Data-Flow Processor," *Proceedings of the 1976 International Conference of Parallel Processing* (P. H. Enslow, Ed.), August 1976, 100-105.

Misunas, D. P., *Performance of an Elementary Data-Flow Processor*, Computation Structures Group (Memo 115), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, February 1975.

Misunas, D. P., "Petri Nets and Speed Independent Design," *Communications of the ACM 16*, 8(August 1973), 474-481.

Misunas, D. P., "Report on the Workshop on Data Flow Computer and Program Organization," *Computer Architecture News 6*, 4(October 1977), 6-22.

Misunas, D. P., "Structure Processing in a Data-Flow Computer," *Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing*, August 1975, 230-235.

Montz, L., *Safety and Optimization Transformations for Data Flow Programs*, S. M. Thesis in preparation, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Massachusetts, expected January 1979.

Nadler, D. R., *Data Flow Computer Performance for the GISS Weather Model*, Computation Structures Group (Memo 159), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, March 1978.

Oldehoeft, A. E., S. A. Thoreson, C. Retnadhas, and R. J. Zingg, *The Design of a Software Simulator for a Data Flow Computer*, Department of Computer Science (Technical Report 77-2), Iowa State University, Ames, Iowa, April 1977.

Oldehoeft, A. E., S. Allan, S. A. Thoreson, C. Retnadhas, and R. J. Zingg, *Translation of High Level Programs to Data Flow and Their Simulated Execution on a Feedback Interpreter*, Department of Computer Science (Technical Report 78-2), Iowa State University, Ames, Iowa, 1977.

Patil, S. S., *An Asynchronous Logic Array*, Laboratory for Computer Science (TM-62), MIT, Cambridge, Massachusetts, May 1975.

Patil, S. S., "Cellular Arrays for Asynchronous Control," *Proceedings of the ACM 7th Annual Workshop on Microprogramming*, September 1974, 178-185.

Patil, S. S., "Micro-Control for Parallel Asynchronous Computers," *Proceedings of the Euromicro Workshop*, June 1975.

Patil, S. S., "On Structured Digital Systems," *1975 International Symposium on Computer Hardware Description Languages and Their Applications Proceedings*, September 1975, 1-6.

Patil, S. S., R. M. Keller, and G. Lindstrom, *An Architecture for a Loosely-Coupled Parallel Processor (Draft)*, Department of Computer Science (UUCS-78-105), University of Utah, Salt Lake City, Utah, July 1978.

Peterson, J. L., *Modelling of Parallel Systems*, Ph. D. Thesis, Department of Electrical Engineering, Stanford University, Stanford, California, December 1973.

Plas, A., D. Comte, O. Gelly, and J. C. Syre, "LAU System Architecture: A Parallel Data Driven Processor Based on Single Assignment," *Proceedings of the 1976 International Conference on Parallel Processing* (P. H. Enslow, Ed.), August 1976, 293-302.

Rodriguez, J. E., *A Graph Model for Parallel Computation*, Laboratory for Computer Science (TR-64), MIT, Cambridge, Massachusetts, September 1969.

Rumbaugh, J. E., "Data Flow Languages," *Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing*, August 1975, 217-219.

Rumbaugh, J. E., "A Data Flow Multiprocessor," *Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing*, August 1975, 220-223.

Rumbaugh, J. E., "A Data Flow Multiprocessor," *IEEE Transactions on Computers C-26*, 2(February 1977), 138-146.

Rumbaugh, J. E., *A Parallel, Asynchronous Computer Architecture for Data Flow Programs*, Laboratory for Computer Science (TR-150), MIT, Cambridge, Massachusetts, May 1975.

Schroeder, M. A., and R. A. Meyer, "A Distributed Computer System using a Data Flow Approach," *Proceedings of the 1977 International Conference on Parallel Processing* (J. L. Baer, Ed.), August 1977, 93.

Science Research Council, *The Coordinated Programme of Research in Distributed Computing Systems*, Science Research Council, England, 1978.

Seeber, R. R., and A. B. Lindquist, "Associative Logic for Highly Parallel Systems," *Proceedings of the AFIPS Conference 24*, 1963, 489-493.

Shapiro, R. M., H. Saint, and D. L. Presberg, *Representation of Algorithms as Cyclic Partial Orderings*, Applied Data Research (CA-7112-2711), Wakefield, Massachusetts, 1971.

Sonnenburg, C. R., *A Configurable Parallel Computing System*, Department of Electrical Engineering (SFL TR 82), University of Michigan, Ann Arbor, Michigan, October 1974.

Stoy, J. E., *Proof of Correctness of Dataflow Programs*, Computation Structures Group (Memo 110), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, September 1974.

Sullivan, H., and T. R. Bashkow, "A Large Scale, Homogeneous, Fully Distributed Parallel Machine, I," *Proceedings of the Fourth Annual Symposium on Computer Architecture*, March 1977.

Sullivan, H., T. R. Bashkow, and D. Klappholz, "A Large Scale, Homogeneous, Fully Distributed Parallel Machine, II," *Proceedings of the Fourth Annual Symposium on Computer Architecture*, March 1977.

Phillip Treleaven	University of Newcastle-Upon-Tyne
Ian Watson	University of Manchester, England
Ken Weng	M.I.T. Laboratory for Computer Science
Charles Wetherell	Lawrence Livermore Laboratory
Rob Witty	Rutherford Laboratory, Oxon, England
Roy Zingg	Iowa State University