LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-92

REPORT ON THE
WORKSHOP ON DATA FLOW COMPUTER AND PROGRAM ORGANIZATION

DAVID P. MISUNAS

NOVEMBER 1977

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

MIT/LCS/TM-92

REPORT ON THE WORKSHOP ON DATA FLOW COMPUTER AND PROGRAM ORGANIZATION

David P. Misunas

November 1977

MIT/LCS/TM-92

REPORT ON THE

WORKSHOP ON DATA FLOW COMPUTER AND PROGRAM ORGANIZATION

David P. Misunas

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)

CAMBRIDGE MASSACHUSETTS 02139

Report on the

Workshop on Data Flow Computer and Program Organization

David P. Misunas
MIT Laboratory for Computer Science

The following report comprises an edited transcription of presentations made at the Workshop on Data Flow Computer and Program Organization, held at M.I.T. on July 10-14, 1977 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 summarized. For further details, the interested reader should consult the bibliography at the end of the report.

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 Bill Ackerman, Dean Brock, Randy Bryant, 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.

TABLE OF CONTENTS

Session 0. Welcome
Jack B. Dennis, MIT

In the last decade, there has been much interesting work having to do with the idea of data-driven program structure and computer architecture, but there has never been any professional technical conference at which people engaged in such research would naturally tend to gather. The original goal of this workshop was to gather this group of people together, to let each other know the nature of their work, what their progress had been, what they consider to be the current issues, and what problems need to be solved to bring concepts of data-driven computation into practical application. Recently, the M.I.T. Laboratory for Computer Science has entered into negotiations with Lawrence Livermore Laboratory concerning possible support from LLL and ERDA for exploiting the ideas of data-driven computer architecture with relation to LLL's computational problems. Thus, there is a second objective for the workshop: to acquaint the LLL people with the status of research in this area, and to see if these ideas are applicable to problems of interest to their projects and if the performance potential is sufficiently attractive to make a gain over the computational facilities which they presently have. So the original idea of the workshop has changed slightly, with increased emphasis on sessions devoted to applications and studies of the potential performance of data-driven computer architectures.

Session 1. Research Status and Objectives
Chairperson, Jack B. Dennis

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

The data flow research group at MIT has been devoting most of its research on computer architecture to one type of data flow computer. The architectural principles we have been studying are manifested in four different levels of data flow computer. The level 1 machine implements the programming constructs of scalar variables, conditional constructs and iteration. It has a very primitive capability in terms of what we know of modern programming languages. This machine is appropriate for computations which have a small amount of program information and a small amount of data. A particular area of application which is attractive for a level 1 machine is signal processing, where the aim is often to get a very high throughput, but the amount of data that has to be handled by the machine at any instant is relatively small, and the program is relatively small. The size of the program in such a machine is limited by the fact that the entire program must be stored in instruction cells comprising the active memory of the machine.

A level 2 machine handles data structures through incorporation of a memory section which will hold a large amount of data in structure form. Again in this machine, the size of the program is limited by the number of instruction cells in the machine's active memory, but the amount of data on which one can operate is dependent only on the size of the memory section, which may be arbitrarily large. A problem that we are studying deeply now is the structure of this memory section and the way in which data structure operations should be implemented in terms of the section.

A third level of machine in which we are interested is one designed such that the instructions of the program do not have to reside permanently in the instruction cells of the machine. Rather, there can be an instruction memory which holds complete programs, and those instructions which are most active are represented in the instruction cells of the machine as they are required by the activity of a program. Again, this is a machine which would support scalar variables, conditionals, iteration, and operations on data structures. Any higher level operations expressed in the applications program would have to be translated out by the compiler and programming system.

Level 4 is the ultimate machine in which we would like to exploit the full expressive power of data flow languages, including ideas about procedures, recursive procedures, and the processing of streams as important kinds of data

values. Also, the level 4 machine should handle applications that require nondeterminate programs. Our ideas are fuzzy concerning the exact implementation of this level machine. Several studies of procedure implementation mechanisms for such a processor have been made, but much work remains to be done.

During the last year, we have come to realize that the architectural concepts with which we have been working provide an interesting basis for realizing a type of distributed function computer system. It is possible to divide the instruction memory and associated interconnection networks of the machine into subsections which don't communicate with each other. Each section of the instruction memory can then be associated with several processing units, and in this manner the whole machine can be partitioned into sections which can operate independently, with the exception that an interconnection network must be utilized to convey results generated by one section of the machine which become operands of another section of the machine. Similarly, the structure capability of such a system can be realized by a group of structure processors, one associated with each section of the machine.

The issues currently concerning the MIT research group cover a broad spectrum ranging from programming language semantics to logic design and architecture description languages. In language issues, we have been very interested in streams as an extension of our ideas of data flow programming languages. We've studied how to deal with nondeterminacy in programs and are examining mathematical foundations appropriate to the semantics of a programming language which allows the expression of nondeterminate computation.

We are currently investigating all four levels of data flow processor. However, our emphasis at this time is on the level 2 processor since this level is most appropriate for application to LLL's problems, and this is the most likely machine to be the basis of a large scale development project. We are also further investigating the level 1 processor since there are some friends of the group who are very interested in signal processing applications, and a level 1 machine would be an appropriate small scale prototype. Such a prototype would be both useful and would give us a testbed for checking out our ideas about programming, testing, and verifying hardware.

We have studied some applications -- the fast Fourier transform as an example of a signal processing computation, the global circulation model for weather prediction, the aircraft collision avoidance problem, and a simplified version of the hydrodynamics problem of interest to LLL. Along with these application studies, we have done some work on performance analysis, examining

the structure of a data flow program and predicting what performance will be achieved when that program is run on a certain type of data flow machine. Much remains to be done in the area of studying the structure of the machine and the structure of the program to arrive at conclusions regarding potential performance.

We have put much effort into developing an architecture description language specialized to packet communication architectures such as the data flow processor. Systems with such architecture are structured of modules which can communicate through the transmission of discrete information packets. We are interested in the architectural description language for a number of reasons. We would like to use the description of a data flow machine as input to a simulation facility, so that once the machine is described, it could be simulated to verify the appropriateness of the machine design and to measure its potential performance. For verification, we'd like to have a formal language for describing units of the machine, so one could prove correctness of the machine in the same manner that one attempts to prove correctness of a program. Another reason we're interested in the architecture description language is because we'd like to study means of building fault-tolerance into packet communication architecture, and the language provides a good idea of the nature of the units that one is trying to make fault tolerant. Finally, the language provides the necessary description for construction of a packet communication system.

In the area of implementation, we've developed logic designs for the active memory of a level 2 machine, investigated interconnection network structure, and studied the specification for a microcomputer that could be used as a unit in building some of our data flow machines.

II. Arvind, University of California - Irvine

The goal of the data flow project at the University of California at Irvine is to develop an architecture capable of utilizing large numbers of processors. By large numbers we mean hundreds or thousands, the exact number depends upon performance and hardware. We believe that the problems involved in utilizing LSI technology are NOT related to simply providing an interconnection mechanism or to designing specialized machines. Rather, the problems really arise with respect to the programmability of such machines -- how would you ever program if you had so many processors which have to work in semi-autonomous fashion?

We view data flow as a solution to the problems of von Neumann type computers, specifically the problems of low level machine languages which are far removed from user languages, the restrictions on the exploitation of parallelism

due to the required sequential and centralized control of computational processes, and the linear organization of storage cells which is almost contrary to the storage organization needed by most of the advanced programming languages.

Our work was inspired by the data flow base language developed by Dennis. By extending his language through the addition of tags to each token, we developed a new way of interpreting the language. This new interpreter, the Unravelling Interpreter (U-interpreter), then spurred the development of ideas on how a machine should be built. The U-interpreter is very well suited for a machine which is composed of large numbers of identical processors, and the power of these component processors is relatively independent of the interpreter.

The interpreter can be imagined as a large system composed of a number of processors which are always looking for tokens, where a token in this system contains a data item and a tag specifying some initiation of some statement in a certain procedure. One input token of each operation is arbitrarily designated an allocation token, and upon production, searches for a free processor. Upon finding one, the token occupies the processor, causing it to assume the personality of the specified instruction and look for any other required operands. When all operands have been found, the instruction is executed, a result is produced, and the processor is freed to wait for the arrival of another allocation token.

Each processor in our system has a link to a memory system and to a communication system. Our system differs from the M.I.T. machine in that instructions of that machine wait in a special memory until they are enabled, at which time they flow out to the processors. In our system, instructions wait at the processors, and execute upon becoming enabled. Also, in our system, memory references are made from the processor, rather than by the transmission of distinct instructions.

Our group has concentrated on the development of a high level language, the Irvine Data Flow Language (ID). The language has asynchronous control structure, clean semantics, and is block structured. It has facilities for programming with streams, and models nondeterminate computation with the help of monitors and the nondeterminate merge, where monitors are used in the sense of resource managers. The language incorporates abstract data types and functional programming facilities for manipulating procedure definitions. Currently, we are incorporating a basic security and protection model.

We have designed a new base language that is used for expressing the semantics of the high level language. The base language also serves as a basic model for the proposed computer architecture. The language has easily

identifiable procedure domains, loop domains, and there exist well-defined translation rules from ID to the base language.

We are also performing work on the semantics of ID and of the base language. This work has been greatly influenced by the work of Backus, especially by the notion that there are no types, that functions and values look exactly alike. Also, along with this work, we are examining models of nondeterminacy and their influence on the languages.

We currently have two operating compilers on a PDP-10. One translates programs from ID to the base language; the other, translates from ID to LISP. These compilers do not incorporate the stream features of ID, however, work is underway to develop a compiler with that capability.

The architecture itself has been extensively studied through simulation, primarily to determine how programs behave, what kinds of demands they make on bus and processing resources. Our primary concern has been to find efficient implementation schemes.

III. Steve Landry, University of Southwestern Lousiana

Research in data flow languages and architecture at the University of Southwestern Lousiana has been led primarily by Dr. Bruce Shriver. The nucleus of people working on the data flow project arose from a project which had as its goals the study of virtual machines and the specification and realization of secure multi-level virtual machines.

Currently, data flow concepts are being considered in five major research efforts:

1. The development of a general data flow simulator;

2. The specification and modeling of a large scale system using a data flow language;

3. The generation of highly improved microcode for parallel/horizontal systems;

4. The use of data flow to describe highly parallel user specifiable arithmetic units; and

5. The use of data flow language and architecture as an implementation alternative in the realization of a nonprocedural language which is based on the semantic principles of relations.

Very early in our studies, it became clear that a tool which could be used to realize or simulate general data flow operations was very desirable. After studying the various alternatives, we made the decision to develop such a simulation facility on our Multics system. The design group felt strongly that the package should not be bound to a single data flow language or architecture since the simulator would be used in the evaluation of language feature alternatives. A user of the system is given building blocks with which to construct his own simulator, allowing evaluation of various sets of primitive nodes.

The second research activity involves the specification of a large scale system using a data flow language. The original stimulus for this work was Paul Kosinski's paper "A Data Flow Language for Operating Systems Programming." This paper seemed an open challenge to operating system designers and implementers, showing how use of a data flow programming language attacks a large number of the problems faced in the development of an operating system. It is the objective of this research to use the design, specification, and implementation of an operating system as an example for evaluating the suitability of a data flow programming language for accomplishing such a task.

The application of data flow principles to the generation of highly improved microcode for systems which utilize horizontal control stores (i.e. systems in which multiple micro-operations are specified in a single micro-instruction) has resulted in the development of a high level microprogramming language for several microcoded processors. An algorithm specified in our internal language is rendered loop free, then a data dependency graph is developed from the loop-free representation. Next, the graph is constrained to reflect actual system resources. This "constrained" graph is then utilized to generate the microcode.

Data flow techniques are also being considered as a tool for describing the operation of user specified arithmetic units. The primary emphasis of this project is the realization of an arithmetic unit for which the user can completely specify the characteristics of operation. This would allow user control over such characteristics as rounding strategies or unit responses to various anomalies.

We are also utilizing data flow primitives in a research project directed toward the design and implementation of a non-procedural language. The language and implementation will embody the semantic principles of relations realized by using dynamic data flow path directives. Programming in such a language describes

in terms of relations or sets of relations of how things interrelate without regard to explicit sequencing.

IV. Al Davis, Burroughs Corp.

The group at Burroughs has had the goal of developing a set of workable systems ideas for distributed control parallel processing. The ideas which we are attempting to exploit can be characterized under three subtopics: properties of the system, properties of the representation, and properties of the hardware.

The basic system goal is that it should be cost effective. Data flow and data-driven approaches to computation can be cost effective, especially in the light of the sort of hardware which one tends to use to build things these days. Another goal of major importance to us is that the actual performance shouldn't be a surprise. Performance should be a function of the process representation, which may or may not allow certain amounts of parallelism, and of the amount of physical resources which one has available, which may vary with time. Any other factor should have a negligible effect on performance, and one of our goals has been to minimize the effect of other factors.

Furthermore, the system should admit to very high performance. Thus, the system supports not only concurrency of a horizontal nature which exploits independent operations or functions, but also, executes in a pipelined sense. Streams of activity pass over resources of the machine, giving rise to concurrency of temporal independence. Resource allocation is performed as a dynamic function during execution of a process, in contrast to the architectures proposed by Irvine and M.I.T.

The program representation used is a form of data-driven nets similar to those developed at M.I.T. This representation is designed to cleanly express a problem, utilizing a programming methodology which is quite different from conventional methodologies and yields a program structure which is a function of the problem, rather than the language. The data-driven form of the representation facilitates formal analysis and also appears to be useful as a modeling and descriptive tool.

The primary hardware goal is to have a system with distributed control which is arbitrarily extendible. In addition, the search for a decent storage model has occupied a significant portion of the work over the past four years. The architecture which has evolved on the basis of some of these goals is an asynchronous recursive architecture which allows a distributed operating system

and has some very nice economic properties in view the sorts of LSI technologies which are available today.

The development plan is based on the waffle principle of doing anything; that is, the first waffle is always bad, so you throw it away. The first waffle was a waffle in which we intended to do everything in exactly the way we wanted to do it, not being influenced by the sorts of components and the sorts of programs other people are writing. The machine was built for its own right in an attempt to achieve the goals we have.

The first waffle has been completed. In that machine we took a low level approach and investigated some of the basic primitives of such systems and the basic ways that these systems behave. We intend to use this knowledge in the construction of a second waffle, that we intend other people to use and find fault with.

V. Jean-Claude Syre, C.E.R.T. - Toulouse, France

In Toulouse, we are currently constructing a single assignment computer called the LAU system. LAU stands for Langage a assignation unique (in French), in English, SAL or single assignment language. This project developed in mid 1973, and its inception was heavily inspired by the paper by Tesler.

Between 1973 and 1974, we carried out formal studies of single assignment languages and examined the implications of such languages on the principles of machine design. The single assignment rule states that a variable will be assigned a value at most once during program execution. This implies a natural expression of the inherent parallelism in a program and a data-driven program sequencing. The language implies a standard sequencing control, i.e. that a statement is ready as soon as its operands are evaluated and can be executed at any later time.

During the next two years, we studied the feasibility of a single assignment system. The main goal was to develop a complete hardware/software system based on single assignment, with single assignment as the base for both language and architecture design. The language and architecture were designed to be general-purpose and to utilize existing hardware and software tools.

We specified a first version of the high level language at that time. The goals of the language design effort were that it be readable and debuggable. The language has the drawbacks of no dynamic features, no recursive features, and no

high level operators on vectors or matrices. We currently have a compiler which accepts the language as specified at that time, while we are completing the definition of the remainder of its features, such as synchronization expressions, type definitions, and so forth.

Our next step was to define a machine language and a general architecture. We then developed in parallel a compiler and simulator for the envisioned machine. The simulator is fully parameterized, and we have used it to measure such characteristics as the parallelism realizable within an execution of a program, both in the arithmetic units and globally.

We are currently constructing a single assignment elementary processor consisting of a memory subsystem, 32 elementary execution units, and a control unit. This construction project should be completed in late 1978.

VI. Roy Zingg, Iowa State University

At Iowa State University, we are have recently initiated a project on data-directed computation. We have developed a simulation facility which simulates a feedback interpreter, working at a logical simulation level and not tied to any specific architecture. We are interested in translation support requirements and have under development a translator that accepts a high level language which we have specified. We are also interested in the effect of certain optimizing transformations when applied to a serial language.

Our future plans extend to the study of memory systems, the data structures such memory systems should support, and the necessary operations upon those data structures. An architecture we might propose would include the ability to unravel loops and would support streams and reentrancy of code. We're currently looking at a further version of the simulator that would include some of these architectural features, and are interested in eventually seeing some of our ideas in hardware.

Session 2. Applications
Chairperson, Lance Sloan

I. George Michael, Lawrence Livermore Laboratory

Lawrence Livermore Laboratory is interested in data flow for reasons of speed. An example of one of the large problems that we routinely face is that of calculating partial differential equations in difference form in meshes that are generally logically square, using principally Lagrangian prescriptions to describe the flow of gasses and fluids in a large network. A typical large problem may be 20,000 zones in size. That's naturally not the limit of what people want to do, its the limit of what they can reasonably think of accomplishing in finite time at present.

According to actual measurements on the CDC 7600, there are 6900 instructions in the inner loop of an example computation, of which approximately 2,000 are floating point instructions. The 7600 will calculate this problem at about 2.5×10^6 floating point operations per second (FLOPS). With the problem tailored to the CDC STAR, it is possible to achieve about 5×10^6 FLOPS. However, the execution on the STAR is a highly tailored thing. It is difficult to find lots of interesting and challenging physics problems that can make use of a STAR. That's one of the reasons we're looking at data flow. A large problem may require 10^{11} floating point operations. At current execution rates, this would take about 100 hours. Nobody's doing 100 hour problems, 30 would be more representative. That's part of the reason we need speed.

The other reason we're here is that from what we know of the future, or at least the next 15 to 20 years, there are only a few promising architectures on the horizon which might be applicable to LLL problems. If you misuse a STAR, it can go as slow as a 6600. If you use it very well, you can get up to 2.5 times the 7600. A number of other machines promise to extend this capability by a factor of three to five. However, what we really need is a machine which will yield a performance of a factor of 25 or so over the 7600 around 1980.

I view data flow as a way to map the computer onto a problem. Commercially, we always end up with a computer, and then we somehow have to warp our physics to fit on whatever this guy decided was the proper way to build a computer. This was acceptable when we could achieve large factors of speedup. Nobody was complaining that he had to rewrite his programs or that it took very special intimate knowledge of the tricks in the arithmetic unit or something like that. Now, the payoffs are not nearly as great from generation to generation, and

so data flow has a few very interesting properties which are valuable to LLL. One of them is that the architecture now has the characteristic of fitting the problem. Second is that you can find out, because of the theoretical underpinning, that you've done the right thing before you have a 2-1/2 or 4 or 6 or 8 million dollar bunch of solder and wires sitting in front of you, mocking you, saying, "You've made another mistake." You don't have to build it if its not right. So, in that sense it has the same virtue as a program. The third thing is that its theoretical underpinning allows optimization of the hardware/software mix. In some sense, that's what data flow has to be to make it pay off the right way.

II. Chris Hendrickson, Lawrence Livermore Laboratory

As a physicist working on a code at LLL, I put in the physics for the designers to use. What I'm going to do is give you an idea of what I'm up against and, hopefully, set up some kind of feedback so that perhaps you can see in the design of your machines what you might do to make my job easier.

Lawrence Livermore Laboratory has always been in the forefront in the purchase of computers, we often buy serial number one. The code I'm responsible for, Coronet, has been on the 7094, STRETCH, the 6600, the 7600, and finally the STAR. It was running at about one MegaFLOP on the 6600, 5 on the 7600, and with tremendous work, we got approximately a factor of three speedup by moving to the STAR. To make the work necessary to move from machine to machine worthwhile, the speed increase has to be on the order of three to five.

Ten to fifteen of our codes are in use one-third of the time. The users of these codes may be willing to reprogram/restructure for maximum efficiency and a large increase in speed. In the case of transforming Coronet to run on the STAR, the effort was twelve man-years. Other users are not interested in putting in such an effort. However, any new machine must be able to run their smaller problems with modest speed gains.

The code Coronet is the simplest of the two-dimensional codes we have at the laboratory. It currently runs on the 7600 and the STAR. On the 7600, it can run 15,000 zones at a speed of 4.5 MFLOPS, or 15.7 MIPS, taking approximately 300 microseconds per zone. Each zone has 2000 floating point operations and 1700 storage references, requiring the memory to run at about 340 MHz.

Due to the STAR 100 experience, we are very wary of new computers, especially those that are conceptually new. The STAR 100 originally looked really great. Looking at the literature back in 1968, everybody was really excited about

it. However, it has a couple of problems, and, unfortunately, one of the problems was in an area where we spend 20% of our time. It took us many years to break this to some extent. I'm wary about the existence of such a problem on a data flow machine. Is there some kind of a flaw in the design of the machine where it will do fast Fourier transforms very nicely, where it will do Lagrange interpolation very nicely, but it won't run Coronet, for example?

What I would like to see as a goal is the production of a machine which is 25 times more powerful than a 7600 as far as speed goes. Looking at the kinds of storage references we're talking about, this implies a memory bandwidth on the order of 9 GHz, which is about five times faster than the 7600 and similar to the STAR. The speed of the machine must be about 100 MFLOPS, not counting any of the controls or other similar functions. Such a machine must have a readable user language. Also, there must be a Fortran to data flow translation process for those who aren't willing to recode.

III. Bob Meyer, Clarkson College of Technology

The three general application areas of interest to me are image processing, seismic processing, and multi-dimensional systems of partial differential equations. A simple example of such is a first order two dimensional recursive digital filter. In such a computation, an input array of points is processed through simple linear combination of a neighborhood of points on the input array with a neighborhood of points on the output array to produce the next point on the output array.

Characteristics of such a computation which make it of interest to data flow are:

1. Reasonably large amounts of data being processed, in which there is a limited degree of coupling on the data points. This allows a high degree of parallelism with only limited interaction.

2. Very few functional types are necessary, requiring only a few types of functional units.

3. The problem has "pure" data-driven control flow with no branching, looping, or decision making. This makes it simple to execute.

These problems operate in a pipeline fashion. Once the computation is started, the functional units should all be kept busy. Hence, data flow appears to be quite useful for such applications.

IV. Jack Dennis, MIT

On one hand, we consider our data flow machines to be general purpose machines because they are language-based and therefore can execute any program in the language for which the machine is set up. On the other hand, they are special purpose because the various levels of machine are restricted in the languages they will support. Also, to run a large application that's going to tax the machine to its limit, the machine has to be tailored to match the problem. However, this is true for any machine.

To evaluate a data flow machine versus a machine of conventional architecture, we have to keep in mind that data flow machines are so radically different that there's no way of comparing them on the basis of such measures as throughput rate at a memory interface, because a data flow machine may not have a memory interface in the conventional sense. So, the only way we have of evaluating such a processor is to run an entire application and compare its performance on the two machines.

The largest problem we have studied is the execution of the global circulation model on a Level 2 machine. This problem is data intensive, but the size of the program is reasonable. The problem exploits two kinds of concurrency: spatial, or between instructions of a program, and pipelined, which is brought about by the simultaneous activity within the machine. One takes advantage of both kinds of concurrency in designing a data flow program.

The global circulation model is represented on a large three dimensional rectangular grid. The computation we want to perform over this grid involves a number of state variables which are used to represent the state of the atmosphere. The computation rule is to find, say, the temperature at a point in the grid by taking the temperature at the previous time step for the grid and adding some incremental values which are obtained by difference approximations based on the temperature and other values at the current time step. This computation is independent for each grid point, allowing enormous potential for concurrency.

The number of instructions in a data flow program to execute this computation would be correspondingly large. So we're faced with the problem of arranging the structure of the program and the corresponding structure of the machine so we can gain enough parallelism to make the computation go on at the rate we would like to see and so the parts of the machine are effectively utilized.

This computation is currently implemented on an IBM 360/95 equipped with four megabytes of addressable core memory. It requires 12.5 seconds per time step for its computation on this machine, or 4 milliseconds per cell of the grid. We have designed a data flow machine which can execute the computation 100 times faster or at the rate of 40 microseconds per cell.

Analysis of the complete data flow program reveals that the machine level program will consist of about 13,000 instructions. If the data flow version is to have a hundred-fold performance increase over the 360/95 implementation, the processor/memory interconnection networks must be able to perform packet switching at 175 MHz and the instruction execution delay should be no more than 20 microseconds. These speeds are quite readily achievable for the processor structures under discussion, implemented using conventional medium speed logic technology.

Several problems related to the structure of this machine and the language it supports have been revealed by this study. The pipeline organization of the program necessitates special instructions in the data flow program for efficient execution of data structure operations. In addition, a higher level language which can be efficiently translated into the machine level representation is necessary for the expression and understanding of the computation. Such a language may be based on an extension of the language studied by Ken Weng.

V. Tim Rudy, Lawrence Livermore Laboratory

I'm going to talk about a LLL application that presents some unique problems for a data flow machine, although I think they're solvable. I'm somewhat optimistic about the data flow machine because I'm concerned that there isn't much left in the way we're going. After completion of the Cray II and the STAR 100C, that's the end of the rope, and that's only another factor of three. We still like to talk factors of five.

I'm concerned with a code, Grok, that is more complicated than Coronet, and has some strange mesh optimization procedures that allow it to run problems that Coronet can't. Each point in the grid has a unique identifier and the neighborhood of each point changes from time step to time step. Computing the zone mass and point energy necessitates mapping from point to zone and zone to point, which presents some special problems for the structure handler of a data flow computer. The temperature updates for this code do not utilize data from a previous time step as with the weather problem; but rather, simultaneous components. This should place unusual requirements on a data flow machine since the machine may

need to do many memory operations for only a few floating point operations.

One of the tragedies which has occurred in benchmarking machines in the past has been to run the simplest model as a benchmark and apply it without some of the bad conditions that arise in a code such as Grok, for example, table lookup. On the STAR, this code would do quite well on most of the computation part. However, for operations such as these which are very scalar and very localized, I'm very concerned that they won't work. That's the same concern I share for the data flow machine, that it has to not only do those operations that are well suited for the architecture very well, but it can't be degraded too much for inherently scalar operations. I understand you can't run Fortran through a data flow machine and make it work, but there's a tradeoff.

I view the data flow machine as a way of achieving speed through parallelism with slower components. The Cray machine is built with 50 nanosecond ECL memory, but we don't necessarily want to build a three million word machine with this technology. We want to get parallelism with slower, less power-hungry components, and that's, hopefully, what the data flow machine will offer us.

Session 3. Language Issues
Chairperson, Arvind

I. Randy Bryant, MIT

This summer, we have been examining some of the computational problems of LLL and how they might be represented in data flow form and executed on a data flow machine. As a first step, we have looked at some of LLL's programs and attempted to translate them into data flow like languages, reaching a number of realizations in the process.

Efficient translation from Fortran to data flow would be very difficult. For example, a Fortran array could translate into one of three different things in data flow. The three possibilities are:

1. A single entity, such as when passing a data flow structure as an argument;

2. A set of identical elements upon which identical operations are performed in data flow, as a set of parallel operations; and

3. A sequence of values, corresponding to a stream in data flow.

Depending on the algorithm and intention, one might want any one of these three different things for a given Fortran array represented in a data flow graph. Utilization of a high level data flow language would allow the user to more clearly specify his intentions, but a translator probably could not do an efficient job.

In examining LLL's programs, I found it necessary to translate them first from Fortran to Algol and then directly from Algol to a data flow representation. I found this translation to data flow very difficult, primarily due to the problems of setting up initial conditions, setting up the loops in the system, and ensuring that a program would return to its initial state upon completion. In a medium-sized program, the data flow version had approximately twice as many control operations as arithmetic operations. In execution of such a program on a level 2 machine, most of the instruction cells of the active memory would be occupied with rather uninteresting things.

A high level language based on streams seems the most efficieint manner to implement the type of numerical functions of interest here. A preliminary version of such a language has turned out to be very useful for this and has a

straightforward translation to a data flow schema.

II. Arvind, University of California - Irvine

The language ID, under development at Irvine, is an expression-oriented, block-structured language. A variable in ID always represents a line in a data flow graph, never a memory cell. Every token in the language can represent either a single value or a structure. A stream is formed of a sequence of tokens, followed by an end-of-stream token.

Assignment in ID comprises naming the output link of an expression, not assignment of a value to a cell. A block in the language consists of a number of statements followed by a return instruction. No names within the block are accessible outside the block. Ordering of the statements in a block is immaterial, due to single assignment, ordering of operation is by names. The language also incorporates facilities for loops, procedure definition, abstract data types, and monitors.

The power of stream operations is exploited in the language through use of three constructs: each, all, and next. The each construct allows the application of a procedure to each element of a stream. The all operator allows capabilities such as returning all the values from a loop to form a stream. The next operator permits the selection in sequence of the elements of a stream.

One of the goals of the language design effort was to make its structure as conventional as possible. Anyone able to write a program in Algol should have no problem using ID.

III. David Wise, Indiana University

Our project started out with a concept of stylized recursion, believing in a style of programming best characterized as pure Lisp. We attempted to develop a useful subset of pure Lisp into something that would work on an iterative machine. A key discovery of this effort was the fact that one can construct a data structure without completely specifying its contents. Exploitation of this discovery tremendously alters the order of evaluation of programs, without affecting meaning. Also, this opens up numerous opportunities for parallelism and translation to a data-driven base language.

We characterize application programs as having environments that are safe

and side effect free. Single assignment is an example of a language with this characteristic. We get it through function invocation and parameter binding, so the whole issue becomes one of binding.

A suspension is a promise to deliver something at a later date. A sequential file or a stream can be represented as a two-component record consisting of a suspension of the current element and the remainder of the stream. For example, in our system a program to generate a list of all the integers will run and return a box with two suspensions in it. Until that structure is transmitted to a device which attempts to print, that is a sufficient answer to the program. It is the job of a device driver to pull output from the program and put it on the device. Traversing the data structure to print it causes the suspension to be delivered upon, and the values appear. If the structure is not used, it will remain suspended. At a system level, if nothing is printed, nothing runs.

We are currently examining target machines for our ideas, and data flow may very well be the appropriate machine. The development of a compiler is waiting on choice of a target machine. What we have developed is a language which has expressiveness that allows us to represent algorithms in a recursive, referentially transparent manner, free of any hardware design.

IV. Jack Dennis, MIT

We have been examining the incorporation of nondeterminacy through use of the nondeterminate merge operator. This is required in such places as the input module of an airline reservation system, merging the inputs into a single stream and allowing the main program to be determinate. Use of the merge operator has thereby allowed us to express an airline reservation system for an arbitrary number of flights and an arbitrary number of agents.

The system interface consists of an input stream of requests from all the agents, merged into a single stream, and an output stream consisting of a responses which are distributed to the agents. The airline reservation system has as data a specification of the set of agents with which it communicates and also the set of flights for which it is handling reservations. There may be an arbitrary number of agents and an arbitrary number of flights. This airline reservation system is nondeterminate because transactions requesting data in auxiliary storage may require an arbitrary time to return, and in order to make the system operate effectively, these transactions must be processed in a nondeterminate manner.

Input requests are distributed to agent modules. These agent modules

distribute the requests to flight modules according to the flight specification in each request. The merge operator is used for two purposes: to merge the requests incident on a flight module into a single stream of requests for that flight module and to merge the outputs of all the flight modules into a single stream which serves as the output of the system. The data base for the system amounts to the collection of flight modules to which the requests are distributed.

V. Paul Kosinski, IBM

Programming in data flow really doesn't need to be all that different from the programming style one is used to. I have developed a programming language which is readily translated to data flow and the structure of which greatly resembles the structure of conventional programming languages. The syntax and semantic base of my language is different from that of Arvind, but, other than that, the two are very similar.

Expressions in the language are very similar to ordinary mathematical expressions. The difference is that variables are more like variables in mathematics than in conventional programming -- they don't correspond to cells. An example of this difference is the idea of updating an array. To maintain freedom from side effects, an update on an array generates a copy of the array with the designated element changed. However, this doesn't necessarily mean that an implementation need generate a copy.

Definitions in the language are very much as they are in mathematics. The order of definitions only affects understanding of the program, not its execution. Conditionals are similar to ordinary conditionals with the exception that a variable appearing on one branch must appear on all.

The language departs slightly from conventional languages in the design of its loop construct. A loop may have multiple exits, containing a number of return statements which break it into sections. The statements in each section are evaluated in their implied order, at which time a decision whether or not to return is made. This replaces the while construct and has been shown to be slightly more general.

Streams in the language are only allowed in loops. Streams are referenced via get and put operations on input and output streams, respectively. Get *stream* yields a value from the input, and put *stream* places a value on the output. Only one transaction on each stream is allowed in each iteration, with the exception of conditional expressions. One branch of a conditional may or may not execute a get

or put operation on the stream referenced in the other branch.

Session 4. Architecture
Chairperson, Roy Zingg

I. Klaus Berkling, GMD - St. Augustin, Germany

One of the main subjects of computer architecture is the problem of getting operators and operands together. There are four essentially different methods of doing this:

1. The conventional method in which an operator has associated with it an address which designates the operand, and from which the operand is fetched for execution;

2. The data flow method in which an operand has the address of the operator associated with it;

3. The ring distribution method, as utilized by lambda calculus and Lisp; and

4. The tree distribution method, such as curry combinators and reduction languages.

The best method remains to be determined. At GMD, we are building a reduction language machine which utilizes a combination of the third and fourth methods.

II. Bill Ackerman, MIT

The structure handling facility under development is designed to support arrays and records of the type that occur in conventional programming languages. Structures are implemented as binary trees; that is, as acyclic directed graphs in which each node is either a leaf (elementary value) or has two immediate subordinates. In the latter case, the arcs to the subordinates are labelled in a manner which allows the directed path from any node to any descendant to be specified by a compound selector.

The structure handling facility implements two data structure operations: select and append. The select operation takes as argument a structure (binary tree) and a bit string and returns the substructure reached by following a path through the tree indicated by the bit string. The returned value may be an elementary value or a structure. The append operation takes a structure, a bit

string selector, and a value. It returns a structure equivalent to the input structure except that the input value is located at the position designated by the selector, replacing whatever was previously located there. The appended value may be an elementary value or a structure.

A structure controller has been designed which efficiently implements creation, transformation, retrieval, and deletion operations on data structures, using a packet memory system for the storage of the structures. The packet memory system which contains the data structures has the property that it can be expanded both laterally and vertically. That is, it can be realized as separate smaller units, each handling a subset of the total address space. Furthermore, these units can be realized as a hierarchy of units, with the higher level units containing only the most active data. These lateral and vertical expansions are the data flow equivalent to the common techniques of interleaving and use of a cache, respectively.

III. Al Davis, Burroughs

I'm interested in a class of machines which I call recursively structured machines. Some of the concepts of such machines are exposed very well in Gluskov's 1974 IFIP paper. That is, there should be a particular level of machine language, there may be limitless levels of machine language, depending on how much one wishes to pay. The storage system of such a machine is also recursively structured and organized. This recursive structure permits an arbitrary number of machine elements.

If the machine is opened at any level, the structure of the machine at that level is exactly the same as the structure at any other level. If we consider the basic module of computing as a processor and some store, in a recursive machine, the processor is composed of another processor/store combination and so forth, until there's no more substructure. In the multi-processor case, a processor is defined as a group of processors and some store.

Viewed in a non-recursive fashion, the system structure resembles a tree. The paths between the nodes of the tree are queues, allowing significant pipelining in the system. Each module at a node of the tree can do one of two things with a data flow program that reaches it. First, it can execute the program in place. This option is always used if the program is sequential in nature. If the program has some parallelism and the module has some physical substructure upon which to exploit that parallelism, the process is decomposed and sent to those physical resources.

The language of the machine is a data-driven language. One advantage of this is that, given a data flow program, one can arbitrarily cut out pieces of the net, execute them elsewhere, and insert the results of the executed subnet on the lines where they would have been had the subnet not been removed. To exploit this data-driven language on the recursive architecture, it is currently necessary to preprocess a data flow program, achieving a highly modular form which can be readily decomposed by a processor for exploitation of its concurrency on the available resources.

IV. Andy Boughton, MIT

We have been conducting extensive studies on the complexity and performance of interconnection network designs for use in packet communication systems. This work has centered around two basic types of networks and has examined the structure of such networks along with their associated complexity and delay.

The research on interconnection network design has been concerned with the complexity required to construct a network with a particular number of inputs and outputs and with a particular level of performance. Complexity is measured in terms of the number of modules required by the network. Performance is composed of two components, network throughput and network depth. Throughput is the rate at which the network will accept packets, whereas depth corresponds to the average time a packet spends in the network.

Two basic networks, from which most interconnection networks can be constructed, have been studied in detail. These are the concentration network, which has a greater number of inputs than outputs, and connection network, which has the same number of inputs as outputs. We have also studied two specific varieties of networks with application to the design of data flow processors: arbitration networks and distribution networks. An arbitration network is an interconnection network with a larger number of inputs than outputs, whereas a distribution network is an interconnection network with fewer inputs than outputs. We have developed constructions for arbitration and distribution networks which are simple compositions of concentration and connection networks. The results of these studies have yielded very promising throughput and performance characteristics for these communication structures.

V. Milos Ercegovac, UCLA

I wish to discuss some relationships which may affect efficiency of implementation of data flow architectures. Within a machine, there are three main levels of representation:

1. Data level or elements only;

2. Algorithm level or operators; and

3. Program level or composition of operators.

We need to achieve a better understanding of the relationships between these representation levels; that is, how the representation of the lowest level of data affects the representation of programs and finally affects the architecture of systems. Clearly, issues at the level of algorithms and treatment of numbers significantly affect such complexity issues as speed and cost of implementation.

I have been studying an evaluation method that transfers a given problem into a system of linear equations that is iteratively solved to any precision using a digit-by-digit left-to-right algorithm on functional units which are no more complex than adders. This method is simple and fast, utilizing a length-independent operation (add) with a single digit bandwidth to achieve a variable precision result with simple control. Computation in the fixed-point domain can presently be done for such problems as polynomials, rational functions, and certain arithmetic expressions.

An on-line algorithm allows the computation of the j-th digit of the result of the basis of $(j + \delta)$ digits of the operand. The on-line delay, δ, is small, one for addition and four for division. This method also has a simple single-digit bandwidth, utilizing primitive operators. This configuration is more complex than the first evaluation method, but requires less time.

VI. Glen Miranker, MIT

The various schemes for procedure activation in a data flow processor exploit techniques for dynamic renaming of operators of a program to give distinct identity to operations occurring in distinct procedure activations. I have been investigating the implementation of these techniques through the addition of memory relocation mechanisms to perform the memory mapping function.

There are several ways of invoking a procedure in a data flow language that are consistent with the data flow model. The simplest method is a single argument apply operator. The effect of apply P is intuitive. When a data value arrives on the input arc of the operator, a copy of the data flow graph for P is made and the data value is placed on the input link of the graph of procedure P. As each of the outputs for this activation of P is produced, it is passed from its output link to the corresponding output link of the apply operator and hence to its successor nodes. To be semantically correct, P must be properly terminating. Briefly, this means that P produces (after some finite time) one output value on each of its output links and then undergoes a finite number of additional actor firings.

This procedure mechanism is implemented in a data flow processor through addition of a relocation box. A request to the memory for some node a of activation s of procedure P causes retrieval of node a with all the names in its destination fields changed to have suffix s. The relocation box then passes the node back to the active memory. It is assumed that with the sole exception of the relocation box and one special functional unit, no other component of the data flow processor can distinguish if a node name has a suffix appended or not. The essential idea is that a complete node name (i.e. a node name plus an appended suffix) is treated everywhere but the relocation box and the distinguished functional unit as a single entity -- a node designation.

VII. Jed Donnelley, Lawrence Livermore Laboratory

The machine I envision for the execution of data flow programs is essentially a huge programmable logic array. Each operator of a data flow program is stored in one cell of the array and connected to its input and output operators through links from the cell to its neighbors. Each cell contains a personality register which designates the function to be performed by the cell. In this fashion, a data flow program to be executed is simply mapped onto the machine.

This system has the properties that it can be built of entirely identical chips and of a logic family with the lowest cost * power/gate. With 100-200 gates per cell and, if one could obtain in a 1980 time frame, 200-400K gates per chip, a machine would have 1K - 4K cells per chip, or about 32 - 64 on an edge. A machine built of these chips should have a large configuration, i.e. 1K - 100K chips or about 10^6 - 10^8 cells. Such a system also has low engineering costs and significant potential for fault tolerant implementation.

Session 5. Performance and Simulation
Chairperson, David Misunas

I. Susan Conry (Susan Conry Meyer), Clarkson College of Technology

Through examination of the characteristics of a data flow program and its mapping onto a machine, which defines the implementation of that particular program, we can gain significant information concerning the performance of the program on that machine. The behavior of a data flow program can be characterized by a data dependency graph which explicitly describes the pattern of data flow in the program and yields information concerning the program behavior.

Examining performance of a program on the MIT machine, we note that there is a fixed constellation of functional units, each performing a specialized function. Each operation can be done only on a functional unit of a specific type. In addition to the data dependencies of the program, one is also physically constrained by the allocation of operations to resources in execution. By examining the maximum length path through a period of execution, summing the number of operations performed and the number of operand paths, and forming a ratio of the number of operations in a period with that sum, one can bound the execution time. This form of analysis also clearly demonstrates that when two operations are in conflict, performance can be significantly affected by the choices made.

In the Irvine machine, one has a number of general-purpose units, and one does not assign functions to any one unit. The performance results achieved from this sort of analysis of the Irvine machine are not fundamentally different from those developed for the MIT machine, performance of the Irvine machine appears to be basically equal to the best that can be achieved if one has functional units of the right number and type on the MIT machine.

II. Bob Thomas, University of California - Irvine

The simulation facility at Irvine has been running for about a year, primarily since the development of the U-interpreter. The simulator accepts as input an encoded form of a data flow graph and yields the result of the specified computation and an analysis of the resource utilization in terms of processors, bus utilization, and time.

We have examined a number of algorithms, primarily matrix multiplication,

quicksort, and the Gaus-Seidel simultaneous linear equation approximation. From simulation of these computations we have measured the number of processors allocated and the number of processors executing at any time, the number of tokens present on the bus structure, and the experimental time distribution for processor and token waiting time.

The results of the simulation studies indicate that time is linear with the mean communication delay and mean execution time, independent of the time distribution. With low variance in the mean communication delay and mean execution time, the critical path length varies little with relative changes in these values. Also, the contribution of the mean delay and mean execution time to the total resources consumed are almost additive. Finally, variance in the mean delay and execution time has an adverse effect on time and resources. However, it may be possible to minimize this effect by appropriate scheduling.

III. Steve Landry, University of Southwestern Louisiana

Our simulation work is intended to aid the study of algorithms at the data flow graph level. Such a facility could be utilized to validate algorithms that are specified in data flow as well as to study their behavior as far as parallelism is concerned.

The basic structure of the data flow simulator consists of three major parts. The translation phase performs syntax and semantic checks and builds an internal representation of the data flow program. The interpreter phase performs the stepwise realization of the data flow machine execution as well as provides the basic probing and debugging interface. The termination phase optionally provides measurement results and dumping capabilities based on user specified requests.

Two user interfaces are supported by the simulator. A graphics interface on a GT40 allows the user to construct and alter program definitions and to trace the flow of tokens during simulation. The other interface utilizes a nonprocedural, high level, language to describe data flow programs and also provides interactive monitoring and debugging capabilities.

All of the simulator (with the exception of the GT40 support code) is written in PL/1. User supplied node realization procedures may be written in a language of the user's choice since they are viewed by the simulator simply as callable routines. The design and coding for a first version of the simulator has been completed, and we are currently in the debugging and testing phase.

IV. Arthur Oldehoeft and Roy Zingg, Iowa State University

The simulator at Iowa State University accepts as input a data flow graph in an encoded form. The simulator interprets the program graph, collecting statistics at each step, which are then summarized and printed. We have made no attempt to design hardware support or to define a data flow user language.

Measurements from the simulator were obtained while executing various program graphs and depend highly on system parameters which define the number and execution time of functional units. The primary performance measures utilized in the execution of actual data flow programs were the speedup, defined as the time to execute a program graph sequentially divided by the time to execute the graph in parallel, and the maximum and average resource utilization. The simulation results for programs which had modest resource requirements indicate that significant speedups were achieved even with unoptimized data flow programs.

V. Randy Bryant, MIT

I have investigated the possibility of simulating a data flow computer on a distributed computer system, for example a network of microprocessors. By exploiting the concurrency and modularity of a data flow computer, such a simulation could also be highly concurrent and modular.

Besides modeling the functional behavior of the system, a proper simulation must also model its time behavior. To avoid placing real-time constraints on the simulation processes, a time-independent algorithm for simulating the time behavior is required. Furthermore, to avoid the need for a high-speed central controller for the simulation, all these time simulation algorithms must be decentralized, requiring special control operations to prevent the simulation from deadlocking and to ensure its proper termination.

I have developed algorithms for controlling such a simulation and have established their correctness. These algorithms allow a number of computations to proceed at different locations concurrently, where each computation has only a limited amount of information about the state of the rest of the system. As is typical of many parallel computations, it is difficult to prove the correctness of these algorithms, yet proofs of correctness are almost imperative, considering the many potential forms of incorrect behavior. Fortunately, since a simulation need only model the behavior of some other system, one need only prove that at no point will the values produced by the simulation diverge from those produced by the simulated system, nor will the simulation deadlock or fail to terminate. Thus,

rather than facing the more general issue of proving the correctness of parallel computations, I was able to develop specialized techniques for this particular problem.

Session 6. Implementation
Chairperson, Bob Jump

I. Clement Leung, MIT

I'm currently examining the problem of fault tolerance in the context of packet communication systems. The general feature of a packet communication system is that it is composed of a number of modules that communicate by sending packets to each other. Each module of such a system can be decomposed into a subsystem which again is structured as a packet communication system. Specific packet communication systems which have been proposed, such as the data flow computer and the packet memory system, exhibit a huge degree of modularity, lending themselves to the introduction of reconfiguration capabilities and graceful degradation.

Two approaches to fault tolerance fit well in the structure of a packet communication system. The first is that of static redundancy, or fault masking. Such techniques include coding redundancy, the use of error correcting codes, modular redundancy, and timed redundancy. The other approach to fault tolerance is that of fault detection and diagnosis, followed by repair and/or reconfiguration. Due to the high degree of parallelism in the processor, it seems desirable to isolate a fault when it occurs and before its effects propagate through the system. However, the application of these techniques to packet communication systems involves the development of asynchronous voting schemes, which are currently unknown, and the study of such techniques is one of the primary targets of this research.

II. Bob Meyer, Clarkson College of Technology

We did not have as an original objective the design of a machine that was yet another alternative data flow architecture. Rather, we were motivated by the increasing availability of processors for use in systems today and set out to study the interconnection of these processors into a system. We view data flow as a natural way of solving the control problems that exist in such a multi-processor distributed system.

The system we developed is composed of a collection of modules. An interface module translates a data flow program into a net description which is executed on the remainder of the machine. A scheduler module holds the complete program, allowing the processors to act as a cache. When an instruction receives

all its operands, it is assigned to a computation activation processor (CAP) module. Each CAP module executes the instruction contained in an arriving packet and produces a result which is returned to the scheduler module, where it is distributed to the necessary destination instructions.

The interconnection paths of the machine and the allocation and collection means are sequential in nature. Hence, the system is intended to exploit parallelism at a high level, that is, in terms of functional execution, rather than at the lower levels of gate-level design.

III. Jean-Claude Syre, CERT - Toulouse, France

Our high level language is very similar to conventional programming languages. An object in the language has attributes consisting of a name, defined operations, and a set of environmental rules which define how and when one can operate on the object. The sequencing rules of the language ensure that a statement is "ready" as soon as its operands have been produced and can be executed at any later time. An expand statement represents parallelism in the same fashion as a "parallel for" statement and is an extension of array operators. The command allows programmer control over the amount of parallelism exploited in the enclosed statements.

We have developed a compiler and have and compiled approximately fifty programs written in the language. These programs have been chosen from various application areas such as numerical evaluation, signal processing, business, and radar processing. In developing these programs, we have found the language easy to program in and easy to teach to students.

Our machine exploits parallelism between jobs, tasks in a job, instructions in a task, and within an instruction (pipelining). The system is a multiprocessor in which each component processor is capable of exploiting the data-driven parallelism of a given task. A task in the system is assigned by a task supervisor to an idle processor. When task outputs are computed, the task supervisor stores the data produced by the task and checks for any new tasks to be executed.

Within each processor, ready instructions run independently and free from hardware constraints. A processor consists of a control subsystem which maintains the status of each instruction, a memory subsystem which stores the actual instructions and data, and an execution subsystem which consists of a number of elementary processors for the execution of instructions.

Active instructions are found through examination of status tags contained in the control subsystem. Upon being activated, an instruction is fetched from the memory subsystem and assigned to an elementary processor. The elementary processor requests the necessary operands from the memory subsystem, and, upon their receipt, executes the instruction, returning any result to the memory subsystem. The processor then requests the control subsystem to appropriately set the tags of the destination instructions and becomes free to accept a new instruction.

We have developed a fully parameterized simulator and have used it to simulate the execution of numerous programs and applications on a processor with our structure. Our initial simulation work evaluated the expand statement. We discovered that there is an optimal degree of expansion in the machine, equal to approximately the square root of the number of data processors. This is caused by a trade-off between the overhead involved in processing the statement versus the parallelism achieved. We have obtained very promising results for execution time and parallelism achievable in numerous applications.

IV. Katsu Amikura, MIT

I have investigated the implementation of portions of the M.I.T. data flow computer. To examine the methods and technologies of such an implementation, the study has concentrated on the most complex part of the processor, the instruction cell block. The cell block under study is the basic building block of the memory of the computer and is composed of 16 distinct instruction cells, each of which holds one instruction of a data flow program in execution on the processor.

An instruction cell performs a number of complex operations, including the reception of packets, the loading of operands, various managerial operations to update the status of the cell, the examination of enabling conditions, and the transmission of its contents to a processing unit. In addition, each cell must contain a mechanism for initial loading of the program, a facility to dump its the contents, and an error mechanism for handling received packets that do not have the required format.

The behavior of a cell block was first formally described in an architecture description language. This description was then utilized to generate data flow interconnection graphs and a Petri net control graph of the system. From these graphs, the design was generated from a top-down decomposition of the specifications, utilizing conventional components and asynchronous communication disciplines for both external and internal communication.

V. Suhas Patil, University of Utah

I have also been studying the realization of the instruction cell block of the M.I.T. machine, but by an entirely different method. The object of my examination is to make use of a kind of programmable logic array to do the implementation.

The logic array implements control structures which allow sequencing of operations, condititional branching, subroutine calls, splitting of a process into parallel processes, merging of parallel processes, and synchronization of processes. The program for an array is specified in terms of a Petri net. Programming the array involves selecting appropriate configurations for the cells in the body of the array and at the edge of the array to reflect the Petri net specification for the desired function.

From the studies, I am quite hopeful that the entire instruction cell block can be realized on two chips with this technique. The programmable logic array chip will perform the necessary processing functions, and a memory chip will maintain the necessary data and state information. The exact implementation of the cell block will, of course, depend on the resolution of the space-speed tradeoffs.

Session 7. Specification and Verification
Chairperson, Susan Conry (Susan Conry Meyer)

I. Clement Leung, MIT

The architecture description language (ADL) under development is intended to serve as a formal language for the specification of packet communication systems. This language should provide a medium for system documentation and human communication, a formalism for design verification, and a language interface to a design automation and simulation facility. ADL complements existing computer hardware description languages in that is is designed for architecture description at the algorithmic behavior/system structure level, not for straightforward translation into existing component technology. The novel features of ADL include a type facility for defining system structure, the adoption of data flow as a basis for its operational semantics, state variables for implementing functions on data streams, and monitors for sharing data objects.

The basic textual element of the ADL is a module, which is the description of an architectural unit. An ADL module has one of two forms: it is either a structural description or a behavioral description of an architectural unit. A structural description of a unit is appropriate if the unit is conceived as an interconnection of simpler units as in the case of a data flow processor conceived as a whole. Behavioral descriptions are required for modules which by themselves constitute complete descriptions of the corresponding architectural units.

The behavior of a module is synthesized by composing expressions. The semantics of expression evaluation is based on the principle of data flow. Each evaluation of an expression is initiated as soon as a new set of operands is available and the results of the previous evaluation are no longer needed. Many expressions can thus be viewed as functional modules with well-defined input and output interfaces. The functional capability of these elementary expressions is expanded in two steps. The concept of a module state which can be updated is incorporated to allow definition of functions on data streams. Next a simplified version of Hoare's monitors is added, introducing non-determinism via shared state variables. The expansion is carefully structured so that expression evaluation can still be governed by the flow of data, although explicit signalling is required to evaluate expressions merely for their side-effects.

II. David Ellis, MIT

In a packet communication system, there exists no centralized facility for coordinating the action of different modules; as a result, data processing and communication within the elements comprising such a system are asynchronous and concurrent. All the modules in a packet system share the same basic principle of operation: a module receives packets on its input channels, processes them internally, and generates packets to be placed on its output channels. There may be an arbitrary finite delay between the time a module receives a packet and the time the module generates and sends out its response to that packet. The fact that packet modules and systems must be able to tolerate such delays is an essential consequence of their asynchronous operation. The above principles of operation apply to an entire packet system, just as they apply to the individual modules that form that system. Packet systems are data-driven in the sense that the progress of a computation in a packet system is determined by the passage of packets through the system.

A crucial property of these systems is that they act nondeterminately; that is, a module in such a system is free to choose among any of a set of equally valid alternative responses to its given input. The admissibility of nondeterminate behavior supports the design of packet systems which take advantage of their asynchronous operation in achieving more efficient use of their computational resources than conventional systems.

The notion of correctness for packet systems bears a close relationship to the manner in which the issues of system structuring and composition are treated within the framework of packet communication architecture. At a very intuitive level, a system is correct if it satisfies certain conditions laid out for it in advance. For packet systems, these conditions take the form of behavioral specifications. More precisely, the behavior is a relationship between inputs received and outputs generated in response to those inputs. A packet system, therefore, is correct if this relation satisfies a given set of specifications.

An approach to proving correctness of these systems has been developed and complete proofs have been worked out for several example systems. This methodology for describing the behavior of packet systems not only makes formal verification possible, but also has proven a significant aid to understanding the operation of such asynchronous, nondeterminate systems.

III. Dean Brock, MIT

I'm currently studying the equivalence of two semantic models of data flow computation. The first semantic model provides the operational semantics of a data flow graph. This model corresponds intuitively to execution of the graph on a data flow machine. The other model provides the denotational semantics of a data flow program. These semantics are very similar to those given for conventional programming languages.

The two objectives of this research are to:

1. Provide the data flow programmer with a conventional semantic basis with which to reason about data flow programs; and

2. To prove that this semantic basis faithfully reflects data flow program execution.

While this research has to date been concerned with a determinate language, further research will investigate the implications of non-determinate language features.

IV. Paul Kosinski, IBM

Data flow programming languages are especially amenable to mathematization of their semantics in the style of Scott and Strachey. That is, a data flow operator can readily be viewed as a function from input data sequences to output data sequences. However, coping with nondeterminate programs is a more challenging problem, as the functions must be from sets of sequences to sets of sequences, and finding a partial order in which the functions are continuous is difficult.

It is possible to obtain a straightforward partial order by considering sets of tagged sequences of data. Each data sequence in the set has associated with it zero or more tags, each of which identifies the sequence of arbitrary decisions made by a nondeterminate operator which contributed to the existence of that data sequence. Two sets are compared by matching up the tags on each element of the first set with the corresponding tags on the elements of the second set. Only then are the data sequences compared by the prefix ordering. This relation may be shown to be a true partial ordering of sets of tagged sequences.

Data flow programming languages have cleaner mathematical semantics than ordinary programming languages. Because they are basically applicative in nature

and local in effect, the functions act solely on the data without states, continuations, or other complications. The tags associated with the data sequences do complicate matters of course, but this complexity is for the purpose of dealing with nondeterminacy, which is not addressed by states, continuations, etc. Furthermore, the tags serve double duty. First, they allow the construction of a straightforward partial order. Second, they are necessary to the specification of how operators functionally transform input sets of sequences to output sets of sequences. Hence, they are less onerous than they might seem at first.

BIBLIOGRAPHY

Ackerman, W. B., *Interconnections of Determinate Systems*, Computation Structure Group (Note 31), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, July 1977.

Ackerman, W. B., *A Structure Memory for Data Flow Computers*, S. M. Thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Massachusetts, August 1977.

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.

Amikura, K., *A Logic Design for the Cell Block of a Data Flow Processor*, S. M. Thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Massachusetts, August 1977.

Arvind, and K. P. Gostelow, "A Computer Capable of Exchanging Processors for Time," *Proceedings of IFIP Congress 1977*, August 1977.

Arvind, and K. P. Gostelow, *Microelectronics and Computer Science*, Department of Information and Computer Science, (TR 106), University of California - Irvine, Irvine, California, 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, *Programming in a Viable Data Flow Language*, Department of Information and Computer Science (TR 77), University of California - Irvine, Irvine, California, March 1976.

Arvind, and K. P. Gostelow, *Semantics of Loop Expressions in ID*, Department of Information and Computer Science (Data Flow Note 11), University of California - Irvine, Irvine, California, March 1976.

Arvind, and K. P. Gostelow, "Some Relationships Between Asynchronous Interpreters of a Data Flow Language," *Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts*, August 1977. Also, Department of Information and Computer Science (TR 88A), University of California - Irvine, Irvine, California.

Arvind, K. P. Gostelow, and W. Plouffe, *Compilation of ID Loops Involving Streams in a New Base Language*, Department of Information and Computer Science (Data Flow Note 15), University of California - Irvine, Irvine, California, July 1977.

Arvind, K. P. Gostelow, and W. Plouffe, "Indeterminacy, Monitors and Dataflow," To appear in *The Sixth ACM Symposium on Operating Systems Principles*, November 1977, to appear in *Communications of the ACM*. Also, Department of Information and Computer Science (TR107), University of California - Irvine, Irvine, California.

Arvind, and W. Plouffe, *Definition of Data Flow Operators*, Department of Information and Computer Science (Data Flow Note 14), University of California - Irvine, Irvine, California, June 1977.

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* (G. Goos, Karlsruhe, and J. Hartmanis, Eds.), 1972, 217-246.

Berkling, K. J., "A Computing Machine Based on Tree Structures," *IEEE Transactions on Computers C-20*, 4(January 1971), 404-418.

Berkling, K. J., "Reduction Languages for Reduction Machines," *Proceedings of the Second Annual Symposium on Computer Architecture*, January 1975, 133-140.

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

Brock, J. D., *Formal Semantics of Data Flow Language*, S. M. Thesis in preparation, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Massachusetts, expected January 1978.

Bryant, R. E., *Simulation of Packet Communication Architecture Systems*, S. M. Thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Massachusetts, June 1977.

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

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

Conry, S. E., "Transformations on a Model for Parallel Computation," *Proceedings of the Conference on Petri Nets and Related Methods*, MIT, July 1975, to appear.

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

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., *Architecture of DDM1: A Recursively Structured Data Driven Machine*, Technical Report, University of Utah, Salt Lake City, Utah, 1977.

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

Davis, A. L., *Data Driven Nets -- A Class of Maximally Parallel Output-Functional Program Schemata*, Burroughs IRC Report, San Diego, California, 1973.

Davis, A. L., *An Overview of Data Driven Machine 1*, Technical Report, Burroughs ASDO, San Diego, California, 1976.

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

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.

Dennis, J. B., *A Language Design for Structured Concurrency*, Computation Structures Group (Note 28-1), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, February 1977.

Dennis, J. B., "First Version of a Data Flow Procedure Language," *Programming Symposium: Proceedings, Colleque sur la Programmation*, (B. Robinet, Ed.), *Lecture Notes in Computer Science 19* (G. Goos and J. Hartmanis, Eds.), 362-376. Also, Laboratory for Computer Science (TM-61), MIT, Cambridge, Massachusetts, May 1975.

Dennis, J. B., "Packet Communication Architecture," *Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing*, August 1975, 224-229. Also, Computation Structures Group (Memo 130), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, August 1975.

Dennis, J. B., "Programming Generality, Parallelism and Computer Architecture," *Information Processing 68*. North-Holland Publishing Co., Amsterdam, 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* (G. Goos, Karlsruhe, and J. Hartmanis, Eds.), 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. Also, Computation Structures Group (Memo 108), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, August 1974.

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," *Proceedings of the Second Annual Symposium on Computer Architecture*, January 1975, 126-132. Also, Computation Structures Group (Memo 102), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, August 1974.

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., D. P. Misunas, and C. K. C. Leung, *A Highly Parallel Processor Based on the Data Flow Concept*, Computation Structures Group (Memo 134), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, January 1977.

Dennis, J. B., and K.-S. Weng, "Application of Data Flow Computation to the Weather Problem," *Proceedings of the Symposium on High Speed Computer and Algorithm Organization*. Also, Computation Structures Group (Memo 147), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, May 1977.

Ellis, D. J, *Formal Specifications for Packet Communication Systems*, Ph. D. Thesis in preparation, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Massachusetts, expected January 1978.

Ellis, D. J, *Internal Specifications for Packet Communication Systems*, Computation Structures Group (Note 29), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, July 1977.

Ellis, D. J, *Proving Correctness of a Packet Communication System*, Computation Structures Group (Note 30), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, July 1977.

Friedman, D. P., and D. S. Wise, "CONS Should Not Evaluate Its Arguments," *Automata, Languages and Programming* (S. Michaelson and R. Milner, Eds.), Edinburgh University Press, Scotland, 1976, 257-284. Also, To appear in *IEEE Transactions on Computers*.

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.

Friedman, D. P., and D. S. Wise, "Output Driven Interpretation of Recursive Programs, or Writing Creates and Destroys Data Structures," *Information Processing Letters 5*, 6(December 1976), 155-160.

Friedman, D. P., and D. S. Wise, "Recursive Programming Through Table Look-up," *Proceedings of the ACM Symposium on Symbolic and Algebraic Computation*, 1976, 85-89.

Friedman, D. P., and D. S. Wise, *Functional Combination*, Computer Science Department, (Technical Report 27), Indiana University, 1976. Also, To appear in *Programming Languages*.

Friedman, D. P., and D. S. Wise, "Aspects of Applicative Programming for File Systems," *Proceedings of an ACM Conference on Language Design for Reliable Software, SIGPLAN Notices 12*, 3(March 1977), 41-55.

Friedman, D. P., and D. S. Wise, "An Environment for Multiple-Valued Recursive Procedures," *Programmation* (B. Robinet, Ed.), 1977, 182-200.

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*, August 1976, 255.

Gurd, J. R., and I. Watson, "A Multilayered Data Flow Computer Architecture," *Proceedings of the 1977 International Conference on Parallel Processing*, August 1977.

Hewitt, C. E., and H. Baker, "Actors and Continuous Functionals," *Proceedings of the IFIP Working Conference on the Formal Descriptions of Programming Concepts*, August 1977, 16.1-16.21.

Isaman, D. L., *Systems of Data-Structuring Operations for Parallel Processors*, Ph. D. Thesis in preparation, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Massachusetts, expected January 1978.

Jacobsen, R. G., and D. P. Misunas, "Analysis of Structures for Packet Communication," *Proceedings of the 1977 International Conference on Parallel Processing*, August 1977. Also, Computation Structures Group (Memo 151), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, September 1977.

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

Kahn, G., and D. MacQueen, *Coroutines and Networks of Parallel Processes*, Insitut de Recherche d'Informatique et d'Automatique (Rapport de Recherche No. 202), November 1976.

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.

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., "Mathematical Semantics and Data Flow Programming," *Conference Record of the Third ACM Symposium on Principles of Programming Languages*, January 1976, 95-103. Also, Computation Structures Group (Memo 135), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, December 1975.

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

Leung, C. K. C., *Formal Properties of Well-Formed Data Flow Schemas*, Laboratory for Computer Science (TM-66), MIT, Cambridge, Massachusetts, June 1972.

Leung, C. K. C., D. P. Misunas, A. Neczwid, and J. B. Dennis, "A Computer Simulation Facility for Packet Communication Architecture," *Proceedings of the Third Annual Symposium on Computer Architecture*, January 1976, 58-63. Also, Computation Structures Group (Memo 127-1), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, November 1975.

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.

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*, 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* (G. Goos, Karlsruhe, and J. Hartmanis, Eds.), 1972, 285-298.

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 Processors," *Proceedings of the 1977 International Conference on Parallel Processing*, August 1977.

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*, S. M. Thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Massachusetts, June 1975.

Misunas, D. P., "Deadlock Avoidance in Data-Flow Architecture," *Proceedings of the Third Milwaukee Symposium on Automatic Computation and Control*, April 1975. Also, Computation Structure Group (Memo 116), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, February 1975.

Misunas, D. P., "Error Detection and Recovery in a Data-Flow Computer," *Proceedings of the 1976 International Conference of Parallel Processing* August 1976, 117-122. Also, Computation Structures Group (Memo 142), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, September 1976.

Misunas, D. P., "Performance Analysis of a Data-Flow Processor," *Proceedings of the 1976 International Conference of Parallel Processing* August 1976, 100-105. Also, Computation Structures Group (Memo 128-1), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, September 1976.

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., "Structure Processing in a Data-Flow Computer," *Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing*, August 1975, 230-235. Also, Computation Structures Group (Memo 129), Laboratory for Computer Science, MIT, Cambridge, Massachusetts, August 1975,

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 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 (TR 77-2), Iowa State University, Ames, Iowa, March 1977.

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

Plas, A., et al., "LAU System Architecture: A Parallel Data Driven Processor Based on Single Assignment," *Proceedings of the 1976 International Conference on Parallel Processing*, 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 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*, August 1977.

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.

Sohnenburg, C. R., *A Configurable Parallel Computing System* Department of Electrical Engineering (SFL TR 82), University of 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.

Syre, J. C., D. Comte, and N. Hifdi, "Pipelining, Parallelism and Asynchronism in the LAU System," *Proceedings of the 1977 International Conference on Parallel Processing*, August 1977.

Tesler, L. G., and H. J. Enea, "A Language Design for Concurrent Processes," *Proceedings of the AFIPS Conference 32*, 1968, 291-293.

Treleaven, P. C., *Principal Components of Data Flow Computers*, Computing Laboratory (TR 108), University of Newcastle upon Tyne, Newcastle upon Tyne, England, July 1977.

Weng, K.-S., *Stream-Oriented Computation in Recursive Data Flow Schemas*, Laboratory for Computer Science (TM-68), MIT, Cambridge, Massachusetts, October 1975.