Computation Structures Group Progress Report 1984-85

Computation Structures Group Memo 252
June 1985
revised March 1986

COMPUTATION STRUCTURES

Academic Staff

J. B. Dennis

Visitor

J. E. Stoy

Research Staff

W. B. Ackerman
G. A. Boughton
W. Y-P. Lim

Graduate Students

T-A. Chu
G-R. Gao
B. Guharoy
S. Jagannathan
B. Kuszmaul
S. Markowitz
K. Theobald
E. Waldin III
T. Wanuga

Undergraduate Students

C. Goldman
E. Gornish
W. Hamdy
D. Kravitz
B-H.Lim
E. Lyons
D. Marcovitz

Support Staff

N.Tarbet

Introduction

The Computation Structures Group is involved in two projects concerned with the design of computer systems using concepts of data flow program execution and functional programming languages. In our work on static data flow architecture, the goal is machines capable of high performance, low cost, and improved programmability in the number crunching domain [8, 9]. In the VIM project [10] , the objective is to build an experimental computer system that demonstrates the merit of these ideas for general purpose information processing by a user community.

Static Data Flow Architecture

The Group's effort on static data flow architecture [8, 9] has been devoted to developing the design of a practical machine for high performance scientific applications. With the availability of commercial, pipelined floating point chips for 64-bit addition and multiplication, we envision a data flow processing element capable of performing 10 million floating point operations per second. A data flow multiprocessor utilizing these elements could comprise 64 processing elements yielding a total peak performance of 640 megaflops. Each processing element would include a large memory for holding array values generated during computation, and would be interconnected with other processing elements by a packet routing network.

Processing Element Design

The major work done in the past year has been the evolution of our architecture for a data flow processing element into an efficient and practical scheme. The participants include graduate students T.-A. Chu, G.-R. Gao, T. Hegg, W. Lim, K. Theobald, T. Wanuga, staff members W. A. Ackerman, G. A. Boughton and W. Lim, and several undergraduates.

An important change has been a revision of the program execution model for a static data flow processor. The new model, called "argument fetching", is logically equivalent to the old "token flow" model, but achieves better utilization of memory cycles and space, especially when used with presently available high performance floating point processing chips. In the token flow model, an instruction sent each of its successor instructions a message containing the result. In the argument fetch model, the messages only give an indication to the successor instructions that they should fire. The messages do not contain the result. The successor instructions fetch the result from predetermined locations in a random access memory. This model permits greater separation of the data paths from the instruction control mechanism, and allows tighter coupling of arithmetic processors to the memory.

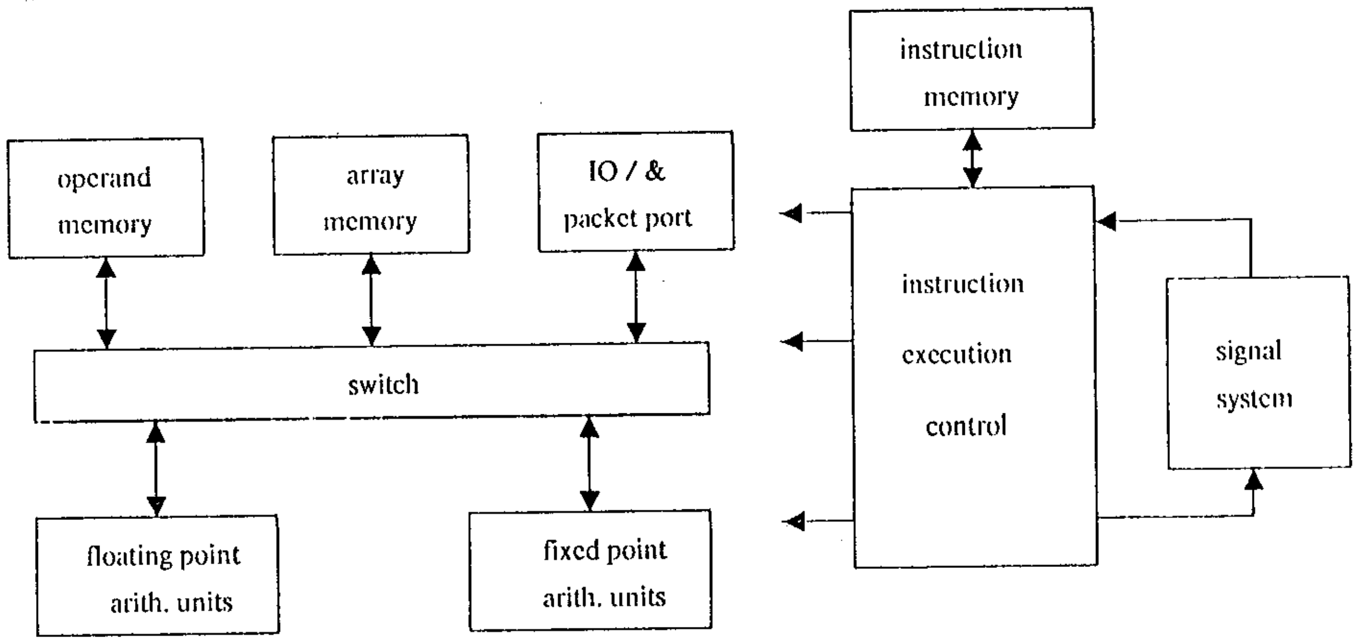

Figure 1-1: Structure of a data flow processing element.

Our current vision of the processing element is shown in Figure 1. It is composed of memories, arithmetic units, a signal system, and an instruction execution control box. While many components of the processing element correspond to components of a conventional processor, the Signal System is new. This system handles the bookkeeping required to determine when all operands for a data flow instruction are available and the instruction is ready for execution.

We have studied possible designs for each of the major components, keeping in mind the approaches that might be used for their fabrication—gate arrays, standard cell, full custom VLSI, and commercially available chips.

The set of arithmetic processors will include commercially available chips. This is especially attractive due to the availability of pipelined adders and multipliers using the IEEE floating point standard that offer as much as ten megaflops of performance. For the fixed point units, commercial units may be used, but it is is not certain that these fit well into our design and specialized implementations may be required.

The heart of the Signal System is the Enable Memory that records the number of times each instruction has been signalled by other instructions, and determines which instructions have received needed signals and are therefore "enabled" for execution. The Enable Memory will be implemented using custom VLSI; an experimental prototype has been fabricated. The signal system also contains a signal list memory which holds, for each instruction, the list of instructions to be signalled once the instruction has been executed. One proposal for the format of the signal lists and the functionality of the controller has been developed by D. Marcovitz.

A crucial element is the collection of data paths that form the interconnection of the arithmetic units and the primary memories. The need to support a very high data transfer rate implies that many connections will be needed, and a bit-sliced structure of the data path appears most appropriate.

Several approaches to implementing the instruction execution control are being considered. To minimize the number of overhead instructions that perform no arithmetic, a powerful set of modes for fetching operands and results has been proposed, including modes for entering and removing elements of FIFO queues and for exchanging data packets with other processing elements. In the implementation the instruction execution control must have access to a pointer memory in which pointers and other types of control information are stored. Since each instruction must go through several stages of processing—access of control information, address computation, operand fetch, operation, result store—pipelined operation is essential to achieve a high throughput. Alternative structures for the instruction execution control have been proposed but many details need further work. A semicustom VLSI implementation is anticipated.

The Enable Memory

The enable memory (EM) is a key component of the static data flow machine responsible for the sequencing of instructions. It keeps a record of how many signals each instruction cell has received and marks cells that have received the required number of signals. These marked cells are said to be "enabled". One task of the Signal System is to select enabled cells one at a time and forward their numbers to the Instruction Execution Control. It is important for smooth operation of the data flow machine that each enabled instruction be executed promptly—that it not happen that other enabled instructions keep jumping ahead, causing a long delay for one instruction. G.-R. Gao and K. Theobald have designed a prototype Enable Memory chip using MOSIS CMOS 3-micron technology. It has about 4000 transistors and a capacity of 128 instruction cells. This prototype is expected to run at about 10 MHz, handling signals from completed instructions at the rate of 2.5 MHz. The chip has been fabricated and is awaiting testing and evaluation.

A Self-timed Design Methodology

T.-A. Chu is conducting doctoral research on a self-timed design methodology for VLSI systems. This methodology is applicable to VLSI and capable of exploiting concurrency. The methodology consists of a high-level organization principle, which was first proposed by Dennis, and implementation techniques for self-timed circuits. A graph model called the Signal Transition Graph (STG) has been developed by Chu for the specification and direct synthesis of asynchronous self-timed control circuits. A STG is a directed graph with vertices representing the signal transitions at nodes in the corresponding circuit module, and arcs representing the precedence constraints between signal transitions. The signal nodes are the input and output terminals of logic components of the circuit module. The STG can be thought of as a specialization of the Petri Net model in the following way. A Petri Net is a bipartite directed graph consisting of two types of vertices, places and transitions. It is capable of modeling concurrent systems in general. Finite State Machines represent one extreme of the model by using only places and no transitions; hence, it has very limited capability for describing asynchronous concurrent circuits. Our STG model differs from Petri Nets in two important ways. First, it uses only transitions and no places; this type of graph has been known as marked graphs. Secondly, a pair of transitions {+,-} is associated with every node of a circuit, instead of a single transition as used in Petri Nets. These specializations allow one to use STG to specify many types of control circuits easily and also allow direct and efficient synthesis of the hardware modules. STG can model all self-timed control circuits, those data-dependent and data-independent, those with conflicts and with concurrency.

Two self-timed VLSI chips have been designed using the methodology. One is a CMOS self-timed Two-by-Two Packet Router that works at around 10 MHz, the second is a NMOS FIFO chip that works at around 4 MHz [6]. The router is the constituent component of a communication network for the proposed data flow computer that our group is developing. A large routing network can be put together using the self-timed two-by-two routers. The advantages of using self-timed components are the modularity of construction and the absence of a global clock. The FIFO chip has a novel organization called a Ring Buffer using distributed control circuits. This organization permits the trade-off between area, latency and throughput rate. It is believed that these are two of the first truly asynchronous, large-scale systems successfully implemented as single chips.

Fault Detection and Recovery Strategies for a Static Data Flow Machine

K. Theobald has begun an analysis of fault-detection and fault-tolerance strategies for the static data flow machine. The goal of the research is the development of general strategies for adding redundancy to detect, and in some cases correct, errors resulting from hardware failures. Since the data flow machine itself is still in the design phase, the strategies are to be sufficiently flexible to adapt to design changes. Therefore, the research does not extend to the level of circuit or logic design; circuit and logic diagrams are only used when illustrating a general strategy by example. However, the research is comprehensive, ranging from hardware analysis to compilation techniques.

The preliminary phase of the project consisted of determining how the choice of strategies is constrained by the nature of the machine and its intended use. For example, since the goal of the machine is high-speed computing, temporal redundancy (running the same program several times on the same machine, and voting on the results) is unacceptable, because the throughput of the machine is reduced two-fold or worse. On the other hand, the machine is not intended for real-time applications, where a momentary delay in a computation might be disastrous (e.g., aircraft control), so it may use dynamic redundancy (selective re-execution of only those program steps that were executed erroneously).

The next phase involved reviewing current VLSI technology to determine the major sources of hardware failures in such systems. From this analysis, low-level fault models have been devised and used as primitive failure models to see how hardware failures are manifested at the logical and program levels.

The bulk of the research has been devoted to deciding how and where to add redundancy selectively to the hardware to detect (and possibly correct) errors caused by hardware failures. The system was divided into two parts, the processing elements and the routing network. The processing element was further divided into the functional execution unit and the sequencing control system, whose failure modes are different. Fault-detection and correction strategies have been devised for these subsystems. The only major portion remaining is the perfection of a time-out system to detect a certain class of errors in the Enable Memory (part of the sequencing control subsystem). This effort continues and builds on the work of Leung [13, 14].

The Interaction of Routing Network Traffic and Data Flow Instruction Scheduling

A simulation model of a packet switched routing network for the static data flow machine, in the form of an indirect binary n-cube composed of two-by-two routers, has been expanded and transposed from a previous model [4, 5]developed in the simulation language Simula. The new simulation model operates as a pseudo event-driven simulator written in Pascal, using a more detailed model of the two-by-two router. The expanded model of the two-by-two router models the timing of the actions of its individual components (buffers, multiplexors, demultiplexors, input and output pins, etc.).

The expected performance of the routing network, in terms of throughput, latency, blocking of network inputs, and buffer utilization, is being examined with this simulator. Studies have been made concerning the placement of buffers at various places within the router (e.g., on the inputs, internal to, or on the outputs of the router). Also examined were the effects of different patterns of communication across the network (e.g., does each processing module talk to only a few other modules or many other modules). Currently, the effects of different chip sizes and packet sizes (i.e., how many bytes per packet) are being investigated.

Studies so far have shown that the placement of buffers internal to the routers (as opposed to on the router inputs or outputs) can greatly improve several aspects of the network performance. Different patterns of communication across the network have been shown to produce widely varying levels of network performance, depending on which processing modules communicate with each other. A simple method for assigning code blocks to processing modules such that the network always performs quite well has been developed. The effect of increasing packet size seems to be only a slight degradation in network performance (less than 15% in throughput and latency for a doubling of the packet size).

The Program Transforming Compiler

A compiler that produces efficient data flow machine code from programs written in the applicative language VAL [2] is crucial to the success of the static architecture in large scale scientific applications. To keep the architecture simple and efficient for computations involving large arrays of numerical data, most of the decisions about resource assignment (allocation of data structures to space in array memory and of data flow instructions to processing elements) have been entrusted to the compiler.

To accomplish an effective resource allocation, the compiler must transform the program so that its structure is a good match to the processing power and memory space of the target machine. Hence, the number of processing elements and the sizes of the instruction and array memories will be parameters of the compiler. Global program transformations are needed because the several sections of a large program must be capable of operating together in a way that allows full utilization of performance without requiring inordinately large amounts of memory for intermediate results. It is reasonable to attempt such global program transformations because VAL is a functional or applicative language, and therefore interactions among program parts occur only at points that are evident from the syntactic structure of the program—the impossibility of "side effects" removes the major difficulty that inhibits use of global optimizations in compilers for conventional languages.

The conceptual basis for such a compiler has been laid down in the doctoral thesis of Ackerman [1], which develops loop unfolding and array interleaving optimizing transformations for data flow programs, and in the graduate research of Gao [11, 7]which explores the concept of *pipe-structured* programs that support the pipelined execution of data flow machine instructions. Accordingly, compilation of a large program will proceed by the following steps:

1) Syntax analysis; type inference and checking; static semantic checks: Converts program into abstract syntax tree.

2) Graph generator: Converts program from abstract tree into "data flow graph" representation.

3) Simple machine independent optimizations: removal of records; constant folding; common subexpressions.

4) Linker: Combines modules to form complete program for a task to be performed by the machine.

5) Analysis I: develop space/time estimate for presentation to user—for the complete program if possible, otherwise, by modules with indication of where the analysis is incomplete and why.

6) User interaction: The user augments the data generated by the compiler's analysis and confirms the transformations to be applied.

7) Transform I: Array interleaving and loop unfolding: This implements the basic space/time tradeoff involved in matching a problem to the machine

8) Transform II: Pipelining: This implements sections of the program as "maximally pipelined" code blocks wherever profitable.

9) Analysis II: Prediction of code performance and memory requirements on the basis of the selected program structure (from analysis and advise).

10) Code generation: generate code for the target machine.

The program transforming compiler will accept valid VAL programs and produce target code for static data flow computers. It will attempt to generate *pipe-structured* object code whenever the source code allows. For some programs optimal code of this form can be produced algorithmically, given parameters characterizing the machine. In these cases, nearly full performance of the machine will be realized for any program that allows sufficient concurrency without exhausting the machine's memory. For other programs, program transformations will be used to convert the code into a form that can make better use of the target machine. The choice of transformations to be applied will be guided by the compiler's structural analysis of the source program, and the programmer's advice given in response to the compiler's analysis. The advice will concern the degree of interleaving to be used for arrays and the degree of unfolding of loops to be used by the compiler. This advice is needed for programs in which the compiler's analysis is incomplete due to the absence of such information as: the number of cycles performed by loops (termination test depends on a computed value); size of arrays (index range is not a compile-time constant); the frequency of selecting arms of a conditional (when the computations are of very different form for the different arms). We envision that the advice file will have the form of a tree representation of the program structure with annotations that constitute the advice.

Ackerman has begun implementation of optimizing transformations for data flow programs. At this writing the global unfolding of record structures has been accomplished. This sort of global optimization would be considerably more difficult to implement for a language that was not applicative.

The VIM Project

The VIM project aims to develop an experimental computer system based on principles of data-driven instruction execution and functional programming languages [10]. The project is unique in striving for a system that will serve multiple users with a degree of semantic coherence well beyond what contemporary computer systems are able to offer. It also differs from other efforts to build systems to support functional programming in that the issues of efficient execution of functional programs over a hierarchical memory are addressed and solutions developed. The system uses a base language that is a form of acyclic data flow graph, and a user language VIMVAL that is an extension and revision of the functional language VAL [2].

VIMVAL

Effort on refining the design of VIMVAL during the past year has focused on the role of modules, and the means of binding free names in modules to form executable program units. In the VIM system, program modules are written in VIMVAL and compiled separately. The programmer creates a runable program by using the *bind* command of the VIM system shell to associate values with the free names of some module. The resulting *closed* module defines a value which may be bound to a free name of yet another module.

In this way a runable program unit that is a tree (actually a directed acylic graph) of linked modules can be constructed. The absence of cycles in the linked program is guaranteed by requiring that only closed modules be bound to free names.

Usually the value defined by a (closed) module will be a function, but we extend the notion of module to include any value representable as VIMVAL text such as an integer, a string, or any data structure built of arrays and records. Thus a module can define any constructable data object which has a VIMVAL type specification. In this way, traditional data objects and functions are treated as equals. This feature avoids a problem present in many programming environments: how to handle the global constants of a program. Instead of having to set up some kind of global environment for such values, they can be collected in a VIMVAL record that is the value of a closed module. Adjusting any of the "global" parameters is simply a matter of recompiling the parameter module and rebinding the modules that use them.

The restriction to acyclic structures of closed modules does not rule out recursion and mutual recursion. Such functions may be expressed within a single VIMVAL module as a set of mutually recursive function definitions. Such a set should be conceived of as a unified entity. Viewed from outside, the recursive set is an just an ordinary function definition. This idea is illustrated by the example in [10], and is one embodiment of the principle that modules are independently written program units that should be comprehensible without reference to the internal workings of other modules.

In the VIM system, we require that all code be type checked before execution, providing a guarantee that no type conflicts will occur during program execution. Within a VIMVAL program module most type specifications are optional, since the compiler can infer the type of an object from its context. However, in certain situations the compiler may be unable to infer the type of an expression. In such cases the programmer must insert a type specification. At other points a programmer may wish to insert a type specification just to improve program readability.

However, omitting type specifications from the module header would be inconsistent with the principle that a user need not inspect the module body to determine its legitimate uses. Therefore, each module has a header that gives type information for each argument, each result, and each free name of the module. This information must be sufficiently complete that the user of the module may construct a module that uses the given module and know that no type conflicts will occur when any legal binding is performed to create a runable program unit.

Since complete type information is not always needed in module headers to support this need, the Vim system provides for *partial type definitions.* For example, a programmer may wish to specify that an argument or result is an array, but not the type of the array elements, because their type is immaterial. Then the type specification *array[t]* may be used where no type specification is given for *t* indicating that it is a *type variable.* One use of partial type specifications is to constrain the range of types so the compiler may produce more efficient code. Another use is to specify polymorphism, as in the header of a sort program

```
    function Sort ( Data: array[value], Keys: array[tag], Test:
        function (x,y: tag) returns(boolean))
        returns(array[value], array[tag])
```

where *value* and *tag* are type variables [8].

Various language constructs for expressing the computation of array objects have been investigated. These computations can normally be expressed using simple array operations, such as *append,* in a tail recursive function definition. However, in many cases where the computation is expressed as a tail recursive function definition, it is difficult for the compiler to generate code that uses storage efficiently. In addition, the programmer may find it awkward or inconvenient to write the computation in this form. We have found it desirable to include a special language construct for these cases that permits easy generation of good code and that provides the programmer with a simple, semantically clean language construct.

A related area of investigation involves general recursion in VIMVAL. General recursion allows recursive equations to define data structures. Some array computations can be considered as special cases of general recursion. For example, if there is a recurrence relationship among the elements of an array, then the entire array may be easily defined by a set of equations that specify this recurrence relationship. However, several difficulties are raised by the inclusion of general recursion in the language, among them the capability for the programmer to express circular data structures. A problem of current interest is finding an efficient implementation of general recursion within the framework of the VIM system.

Operational Models for VIM

Two projects concerning the implementation of large data structures and the backup of information in VIM are being completed as master's theses. So these studies may be done in terms of a precise understanding of the program execution model of VIM, formal operational models for VIM have been formulated by B. Guharoy and S. Jagannathan. The first model **L1** specifies the semantics of the data flow graphs that represent programs executed by VIM. In L1 there is no explicit modeling of hardware elements of the implementation—the model is abstract and suitable for understanding the correctness of a compiler from the VIMVAL language into data flow graphs. This model and the other models to be mentioned consist of a set of system states and an interpret function that defines a set of non-deterministic transitions between states. In L1 a system state has three components: a set of function activations; a heap which contains, among other things, structure values and function templates; and the set of enabled instructions (enabled nodes of the data flow graph). This latter set contains those instructions in the activations that have received all operands and signals necessary to enable execution. There is no concept of memory in this model; it treats structure types as sets and structure values are simply elements in these sets. The interpreter is a state transition function mapping from a state and an enabled instruction to a set of possible next states. The rationale for developing this model is two-fold. First, it provides a simple but precise mathematical description of system operation as a formal interpreter of dataflow program graphs. Second, this model serves as a suitable basis, through augmentation and refinement, for rigorously defining aspects of system implementation, as described below.

For the study of the implementation of data structures in the VIM memory hierarchy, Guharoy has developed a second operational model **L2**. This model includes a storage model that reflects a physical storage hierarchy consisting of a main store and disk. Here the program representation is the same as in **L1** but the heap is modeled by a collection of fixed-size memory *chunks* which are either *accessible* (meaning they reside in fast (main) memory) or *inaccessible* (meaning that the chunk is held only on the disk memory). The model includes the reference count scheme by which chunks are identified as unreachable and therefore the space occupied is available for reallocation.

In his master's thesis Guharoy establishes correctness of a set of algorithms for an implementation of general array operation using chunk allocation in a hierarchical memory. This is achieved by showing that **L2** and **L1** have equivalent behavior using the proof method developed by D. Berry [3] and C. McGowan [15].

Backup and Recovery in VIM

In his master's degree research, S. Jagannathan has developed the design of a backup and recovery system for the VIM computer system. The goal is to design procedures to guarantee the security of *all* accessible information in the system against loss or corruption as a result of hardware failure. The mechanisms to achieve data security on conventional systems are based on the existence of a file system that uses memory space independent of that used by the basic program execution facility and exploit the fact that updates of files take place relatively infrequently and usually replace the file in its entirety. This implementation has the unfortunate consequence that all activity performed since the last update action is lost when the system fails. The backup and recovery system proposed for VIM takes advantage of the novel aspects of VIM—its powerful applicative base language and the data-driven architecture of the system.

In VIM, long-lived objects are bound to names in an *environment* associated with each user. The backup system preserves the environment by copying values onto the backup store as they are defined in response to user commands. The information placed in backup store by this mechanism is called *quiescent data.* The backup system also maintains an *activation log* of all active computations in the system. When the system fails, recovery is accomplished as follows: First, the recovery system uses the quiescent backup data to restore a past system state; then interpreting the activation log will bring the system forward to the its state immediately before the failure. The activation log is a directed tree in which nodes correspond to function activations and arcs represent caller/callee relationships. The thesis shows how the execution rules for certain VIM instructions may be augmented to incorporate the necessary building of the activation log. It is also shown how the recomputation of values can be avoided by storing pointers to computed results in the activation log, thus preventing long recovery times for long running commands.

The backup store consists of a *stable storage* device used to hold the activation log, and a large mass storage device, such as tape. Information on stable storage is immune to the effects of hardware failure and is easily accessed and updated. Stable storage is presumed to be too expensive to hold the entire backup state, so a cheaper storage device, such as redundant tapes, is needed to hold quiescent data.

Jagannathan has designed the backup system to handle the creation of stream elements produced in a demand-driven fashion and has included optimizations to handle tail recursive functions.

Functional Language Implementation on the Connection Machine

Programs written in an applicative language exhibit a large degree of parallelism, and the connection machine [12] is a highly parallel machine. In his master's thesis, B. Kuszmaul has hypothesized that the connection machine will perform well running such programs, and that there is much to be learned about applicative programming from a high performance test bed. He is studying several implementation schemes for applicative languages through simulation on the connection machine. These include static data flow [8], VIM-style dynamic data flow [10], and combinator reduction [16]. For this study Kuszmaul has developed a new programming model for the connection machine well suited to writing highly parallel interpreters for applicative languages. The objective is to evaluate the suitability of the connection machine for each of the implementation schemes mentioned above. VIM is an experimental computer system based on principles of data-driven instruction execution and functional programming languages. The project is unique in striving for a system that will serve multiple users with a degree of conceptual coherence well beyond what conventional computer systems are able to offer. It is also distinguished from other efforts to build systems to support functional programming in that the issues of efficient execution of functional programs over a hierarchical memory are addressed and solutions developed. The system uses a base language that is a form of acyclic data flow graph, and a user language VimVal that is an extension and revision of the functional language Val.

Publications

1. Ackerman, W.B. "How to Design Simulatable CMOS Integrated Circuits," VLSI Memo No. 85-237, MIT, Cambridge, MA, March 1985.

2. Chu, T-A. "Design of a CMos Self-timed Two-by-Two Packet Router," CSG Memo 242, MIT Laboratory for Computer Science, Cambridge, MA, December 1984.

3. Dennis, J.B. "Modeling the Weather with a Data Flow Computer," IEEE Transactions on Computer," July 1984.

4. -. "Data Flow Computation," *Control and Data Flow:Concepts of Distributed Programming.* Manfred Broy, ed. Springer-Verlag, Berlin, Heidelberg, New York, 1985.

5. Gao, G-R. "Pipelined Mapping of Homogeneous Data Flow Programs," Proc. IEEE Intern. Conf. on Parallel Processing," Cambridge, MA, August 1984.

6. Lim, W. Y-P. "A Design Methodology for Stoppable Clock Systems," CSG Memo 240, MIT Laboratory for Computer Science, Cambridge, MA, August 1984.

7. Markowitz, S. "VLOE: A Val Language-oriented Editor," S.B. thesis, Department of Electrical Engineering and Computer Science, Cambridge, MA, September 1984.

Theses Completed

1. Boughton, G.A. "Routing Networks for Packet Communication Systems," Ph.D. dissertation, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, September 1985.

2. Guharoy, B. "Data Structure Management in a Data Flow Computer System," S.M. thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1985.

Theses in Progress

1. Gao, G-R. "A Pipeline Code Generation Scheme for Static Data Flow Computers," Ph.D. thesis, Department of Electrical Engineering and Computer Science, Cambridge, MA, expected January 1986.

2. Jagannathan, S. "Guaranteeing Data Security in a Dynamic Data Flow Machine," S.M. thesis, Department of Electrical Engineering and Computer Science, Cambridge, MA, expected September 1985.

3. Theobald, K. "Adding Fault-Tolerance to a Static Data Flow Supercomputer," S.M. thesis, Department of Electrical Engineering and Computer Science, Cambridge, MA, expected September 1985.

4. Wanuga, T. "Routing Network Performance in a Static Data Flow Machine," Department of Electrical Engineering and Computer Science, Cambridge, MA, expected September 1985.

Talks

1. Dennis, J.B. Six lectures on Data Flow Computation, Intern. Summer Course on Control Flow and Data Flow: Concepts of Distributed Programming, Marktoberdorf, Germany, August 1984.

2. -. Lectures on Data Flow Computing given at RIACS Workshop on Evaluation Study of Data Flow Computation, Moffett Field, CA, September 1984.

3. –. "On the Form and Use of Distributed Computer Systems," Keynote address given at IEEE Conf. on Distributed Systems, Denver, CO, May 1985.

4. Gao, G-R. "Pipeline Mapping Scheme for Array Operations on Static Data Flow Computers," IBM, Yorktown Heights, NY, August 1984.

5. –. "Principals of Pipeline Code Generation Scheme for Static Data Flow

Computers," RIACS Workshop on Evaluation Study of Data Flow Computation, Moffett Field, CA, September 1984.

References

1. Ackerman, W.B. Efficient Implementation of Applicative Languages. MIT/LCS/TR-323, Laboratory for Computer Science, MIT, Cambridge, Mass., March, 1984.

2. Ackerman, W.B. and Dennis, J.B. VAL --A Value -oriented Algorithmic Language: Preliminary Reference Manual. Report MIT/LCS/TR-218, Laboratory for Computer Science, MIT, Cambridge, Mass., June, 1978.

3. Berry, Daniel M. Block Structure : Retention or Deletion. Proc. of the Third Annual Symposium on the Theory of Computing, May, 1971.

4. Boughton, G.A. Routing Networks for Packet Communications. MIT/LCS/TR-341, Laboratory for Computer Science, MIT, Cambridge, Mass., June, 1985.

5. Boughton, G.A. *Routing Networks for Packet Communications.* Ph.D. Th., Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass., September 1985.

6. Chu, T-A. The Design, Implementation and Testing of a Self-timed Two-by-Two Packet Router. Computation Structures Group Memo 225, Laboratory for Computer Science, MIT, Cambridge, Mass., February, 1983.

7. Dennis, J.B. and Gao, G-R. Maximum Pipelining of Array Operations on Static Data Flow Machine. Proc. of Intern. Conf. on Parallel Processing, August, 1983, pp. 331-335.

8. Dennis, J.B., Gao, G-R., and Todd, K.R. "Modeling the Weather with a Data Flow Supercomputer". *Transactions on Computers C-33,7* (July 1984), 592-603.

9. Dennis, J.B., Lim, W. L-P., and Ackerman, W.B. The MIT Data Flow Engineering Model. Information Processing, IFIP, 1983, pp. 553-560.

10. Dennis, J.B., Stoy, J.E., and Guharoy, B. VIM: An Experimental Multi-user System Supporting Functional Programming. Proc. of the Workshop on High-level Computer Architecture, May, 1984.

11. Gao, G-R. An Implementation Scheme for Array Operations in Static Data Flow Computers. MIT/LCS/TR-280, Laboratory for Computer Science, MIT, Cambridge, Mass., August, 1982.

12. Kuszmaul, B.C. Simulating Applicative Architectures on the Connection Machine. M.S. thesis, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, Mass.. Unpublished.

13. Leung, C. K-C. N. Fault Tolerance in Packet Communication Computer Architectures. MIT/LCS/TR-250, Laboratory for Computer Science, MIT, Cambridge, Mass., September, 1980.

14. Leung, C-K. C-N. and Dennis, J.B. Design of a Fault-tolerant Packer Communication Computer Architecture. 10th Ann. Sym. on Fault Tolerant Computer Systems, October, 1980, pp. 328-335.

15. McGowan, C. An Inductive Proof Technique for Interpreter Correctness. Courant Institute Symposium on Formal Semantics of Programming Languages, 1970.

16. Turner, D. A. "A New Implementation Technique for Applicative Languages". *Software - Practice and Experience 9* (1979), 31-49.

TABLE OF CONTENTS

COMPUTATION STRUCTURES 1

LIST OF FIGURES

Figure 1-1: Structure of a data flow processing element. 3

especially when used with presently available high performance floating point processing chips. In the token flow model, an instruction sent each of its successor instructions a message containing the result. In the argument fetch model, the messages only give an indication to the successor instructions that they should fire. The messages do not contain the result. The successor instructions fetch the result from predetermined locations in a random access memory. This model permits greater separation of the data paths from the instruction control mechanism, and allows tighter coupling of arithmetic processors to the memory.

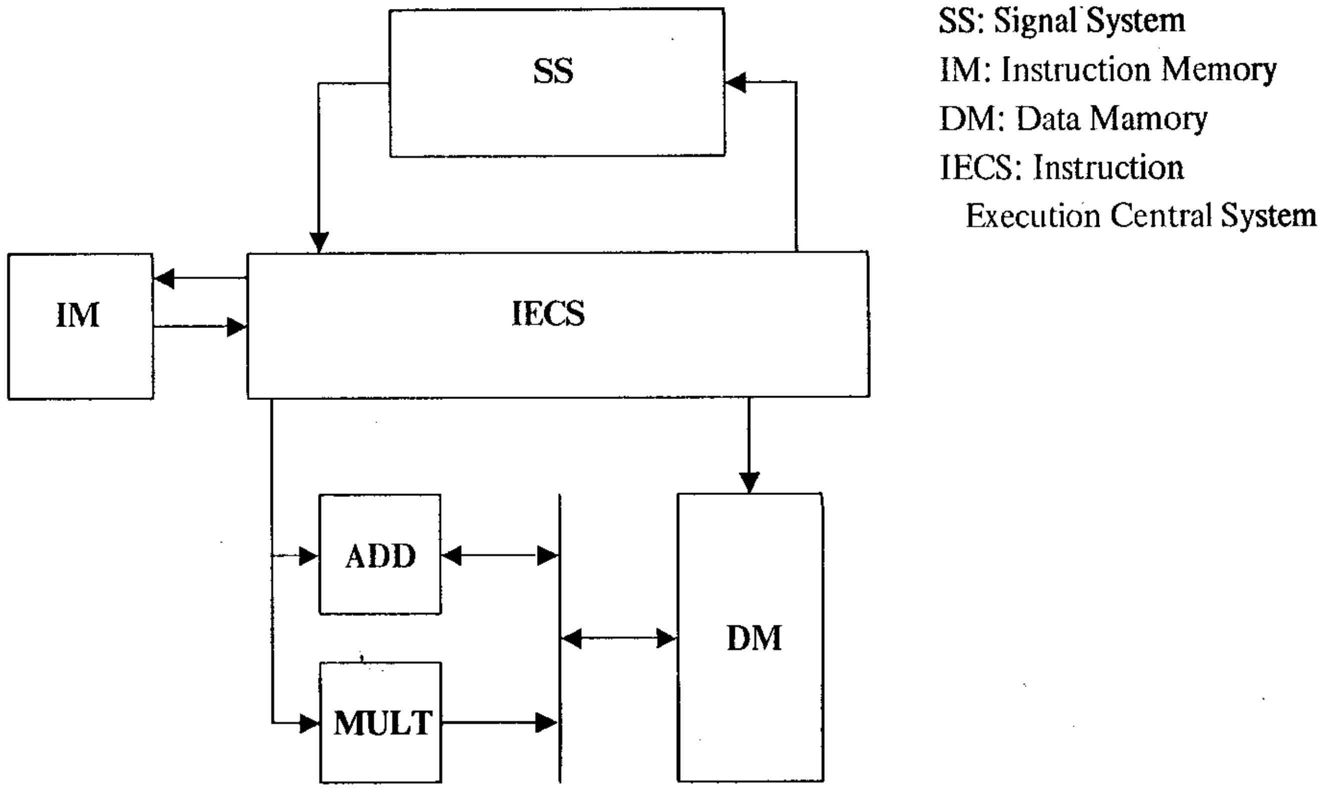

Figure 1-1: Structure of a data flow processing element.

Our current vision of the processing element is shown in Figure 1. It is composed of memories, arithmetic units, a signal system, and an instruction execution control box. While many components of the processing element correspond to components of a conventional processor, the Signal System is new. This system handles the bookkeeping required to determine when all operands for a data flow instruction are available and the instruction is ready for execution.