PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

DESIGN OF AN EFFICIENT DATAFLOW ARCHITECTURE WITHOUT DATA FLOW

Guang R. Gao
School of Computer Science
McGill University
3480 University
Montréal, Quebec H3A 2A7

René Tio
School of Computer Science
McGill University
3480 University
Montréal, Quebec H3A 2A7

Herbert H.J. Hum
Centre de recherche
informatique de Montréal
1550 de Maisonneuve O.
Montréal, Quebec H3G 1N2

ABSTRACT

An efficient static dataflow architecture based on an *argument-fetching* data-driven principle has recently been proposed by Dennis and Gao [5]. This architecture opens possibilities in combining the technologies of existing high performance conventional pipelined architectures with the strengths of the dataflow model of parallel computation. The key feature is that data never "flows" in the new architecture even though instruction scheduling remains data-driven. This paper outlines the instruction set architecture design of the argument-fetching dataflow architecture. Some important aspects, the addition of a structure memory to for arrays, split-transaction array accesses and interprocessor communication support in the instruction set design, are described. We also discuss extensions for a dynamic argument-fetching dataflow architecture where multiple function invocations can be effectively supported.

1 INTRODUCTION

In recent years, we have witnessed remarkable advancements in such AI applications as knowledge information processing and robotics. These advancements have spawned an increasing demand for higher performance in computing power. More and more researchers agree that, to meet these increasing challenges, parallel processing is becoming a necessity. The next generation of computer systems may employ parallel computers with tens, hundreds and even thousands of processors together to accomplish a task. It becomes increasingly clear that traditional von Neumann style programming and architectures are inadequate to meet such technological challenges [4].

The dataflow model of computation offers an alternative with a sound, simple, yet powerful model of parallel computation. However, there have been serious doubts that dataflow processor architectures can compete with the efficiency of their conventional counterparts. One major concern is the amount of data token flow in the processor. In order to exploit fine-grain parallelisms, the dataflow model appears to require a higher volume of data traffic — a criticism common to all proposed dataflow architectures. The overhead becomes even higher in most dynamic dataflow architectures (such as the tagged-token dataflow architectures) due to the matching of tokens needed in the critical datapath [3,12].

A new static dataflow multiprocessor architecture based on an *argument-fetching* data-driven principle has recently been proposed by Dennis and Gao [5]. This new architecture opens possibilities in combining the technologies of existing high performance conventional pipelined architectures with the strengths of the dataflow model of parallel computation. The key feature is that data never "flows" in the new architecture while instruction scheduling remains data-driven.

In traditional proposals for a dataflow processor, the data value computed by one instruction is transmitted to its destination instruction to signal that the input operand is available. In such *argument-flow* dataflow architectures, there is more data movement involved than necessary. The problem lies in the decision to keep data information (values) and control information (addresses) bound together in packets "flowing" through the processor. In the argument-fetching dataflow architecture, the data and signaling roles of the information packets are separated and an instruction fetches its own arguments from a data memory just like in conventional processor architectures.

The new architecture has two parts: a *dataflow instruction scheduling unit* (DISU) and a *pipelined instruction processing unit* (PIPU). The PIPU is an instruction processor that uses conventional tech-

niques to achieve fast pipelined operation. The DISU holds the data-dependency signal graph of the collection of nodes allocated to the processing element, and maintains a record of which nodes are enabled for execution by the firing rules of the signal flow graph (see sec. 4.2.) Each of the enabled nodes in this pool is available for execution by the PIPU. The new architecture has a significant advantage over vector pipelined processors as well as RISC processors [5]. The highly pipelined processing power of the new architecture can be exploited by the dataflow software pipeline for array operations in scientific numerical computation [6,7,8].

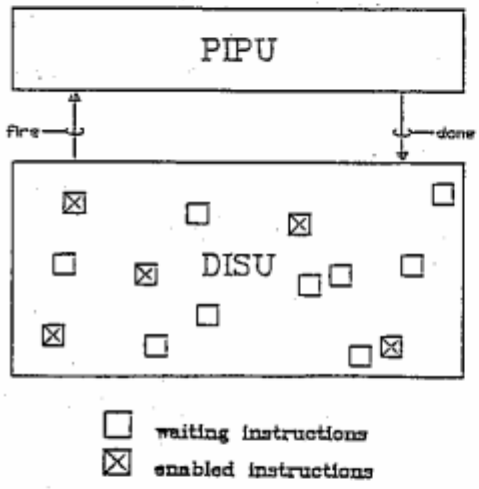

Figure 1: An Argument-Fetching Dataflow Processor

2 THE ARGUMENT FETCHING ARCHITECTURE

The argument-fetch dataflow processor consists of two major sections (fig. 1) designed specifically to perform instruction execution and scheduling functions. The Dataflow Instruction Scheduling Unit (DISU) stores the *signal graph* of the dataflow program and is responsible for identifying and "firing" instructions that are available for execution. The Pipelined Instruction Processing Unit (PIPU) executes these instructions and informs the DISU when each instruction finishes execution.

The *fire* link in figure 1 is for transmitting the addresses of enabled instructions from the DISU to the PIPU. The *done* link is for transmitting back to the DISU the addresses of instructions which have completed their processing in the PIPU, together with a *condition code* used in sending conditional signals.

The PIPU can be considered a conventional pipelined processor without a PC — the von Neumann style program counter — while the DISU performs the role of a PC by providing addresses of candidate executable instructions. The difference is that this "PC" is data-driven and maintains not one, but a *pool* of concurrent candidates.

We have extend the architecture proposed in [5] to include a structure memory unit for handling arrays and an I/O unit for interprocessor communications to the PIPU.

2.1 The Pipelined Instruction Processing Unit

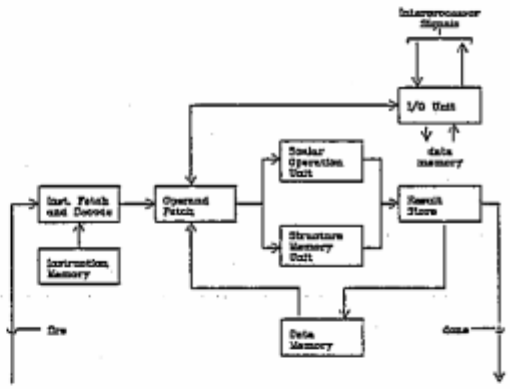

Figure 2: Structure of the PIPU

The organization of the PIPU is shown in figure 2. It consists of four major pipeline stages to handle instruction fetch and decode, operand effective address calculation and fetch, instruction execution (consisting of a scalar operation unit and a structure operation unit), and result store.

The scalar operation unit performs arithmetic and logic functions (such as basic fixed and floating point arithmetic) as well as scalar memory operations, while the structure memory unit performs data structure oriented memory operations, such as arrays accesses. The I/O Unit is used for interprocessor communications, specifically, fetching data from remote PEs (sec. 5.3). Our architecture also provides built-in

primitives for handling FIFO buffers.

The unique character of the PIPU stems from the absence of a program counter and its related control logic.

2.2 The Dataflow Instruction Scheduling Unit

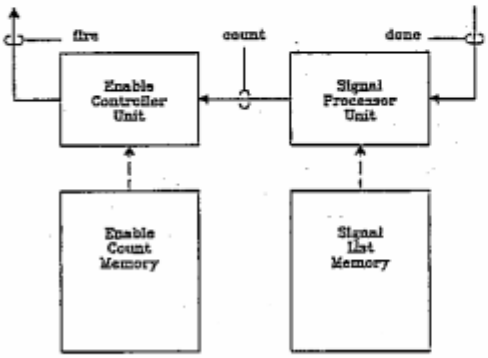

Figure 3: Structure of the DISU

The DISU (fig. 3) consists of a *signal processing* unit (SP) and an *enable controller* (EC) unit. The signal graph of a program is represented in the DISU by the signal lists stored in the *signal list memory* of the SP unit. Each signal list represents a set of signal arcs leaving the associated node of the signal graph. The *enable count memory* of the enable controller unit holds *count* and *reset* status values for each node in the signal graph.

In response to a **done** signal from the PIPU for instruction n_i, the SP unit retrieves the signal lists for n_i and sends a **count** signal for each entry in the active lists (the set of active lists is determined by the condition code returned with the **done** signal.) The EC unit receives the **count** signal and decrements the *count* value of the indicated node. When this count value reaches zero, an "enable" flag for this instruction is set and the *reset* value is copied back into *count* to prepare for the next firing cycle of the instruction (e.g. in the next iteration of a loop.) The EC unit continuously monitors all the enable flags and issues fire signals for enabled nodes.

3 THE PROGRAM AND INSTRUCTION FORMAT

A dataflow program graph G for the argument fetching architecture is represented by a *program tuple* $\{P, S\}$, where P is a set of PIPU instructions and S a *signal flow graph*, represented by a set of signaling instructions. Formally,

G ::= <P,S>

Each actor in the dataflow program graph has an entry in both P and S sections of the program tuple. The instructions in P (p-instructions) contain no information about the sequence of execution. Instead, the sequencing information is given separately by the signal flow graph S.

3.1 Instruction Format for the PIPU

The instruction graph P is a list of instructions where:

P ::= <p-inst-list>

<p-inst-list> ::=
    <p-instruction>
    | <p-instruction> <p-inst-list>

<p-instruction> ::=
    <opcode> <op-address> <op-address>
                <result-address>

<op-address> , <result-address> ::=
    <mode> <address>

Each p-instruction is a three address instruction common to conventional architectures. These instructions are stored in PIPU instruction memory and are executed by the PIPU (see section 3.2). The effective address can be calculated from the address field and the addressing modes.

3.2 Signal Graph Format for the DISU

The signal graph S of the program tuple determines the sequencing of the instructions. Formally, the graph consists of a list of signal nodes:

S ::= <s-node-list>

<s-node-list> ::=
    <s-node>
    | <s-node> <s-node-list>

<s-node> ::=
<signal-count> <signal-list>

Each signal node, or s-instruction, contains three address lists designated the unconditional, true and false signaling lists. These signal lists are used in the implementation of conditional expressions [10]. The signal count consists of both enable-count and reset-count fields to enforce the dataflow firing rules.

4 INSTRUCTION EXECUTION AND SCHEDULING PROCESS

The operational semantics of a dataflow program graph is described by the firing rules of the actors in the graph. In the argument-fetching architecture, the firing rules are implemented jointly by the PIPU and DISU, where the PIPU performs the actual execution of an operation and the DISU performs the scheduling of the operation. These two phases are called the *execution phase* and the *scheduling phase*.

4.1 Execution phase

The execution phase of an instruction in the PIPU begins when a firing signal for that instruction is sent to the PIPU. The firing signal contains the address of a p-instruction, which the PIPU will retrieve and execute in a conventional pipelined manner. When the operation completes, the execution phase ends and a done signal is generated and sent to the DISU.

4.2 Scheduling phase

The DISU performs the scheduling function by processing a done signal from the PIPU. A done signal has the following format:

<done-signal> ::= <address> <condition-code>

where <address> is a pointer to the signal node counterpart of the "done" p-instruction, and <condition-code> is either T, F or U. When a done signal is received, the corresponding s-instruction is retrieved and a count signal is sent to the enable controller unit for each address in the active signal lists of the signal node (fig. 4). The rules of signaling are:

- the addresses in the unconditional signal list are always signaled;

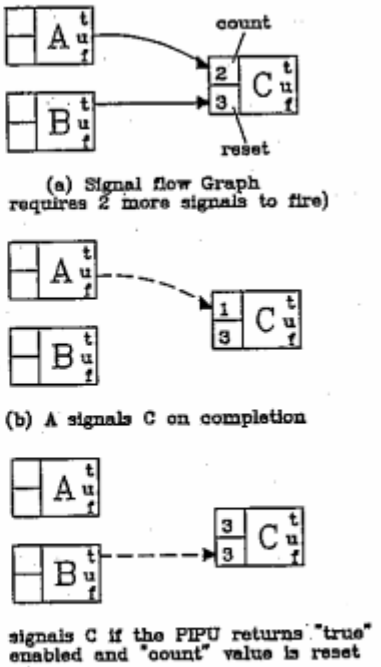

Figure 4: DISU Scheduling Phase

- if the condition code is T (F), the addresses in the true (false) list are also signaled.

The count signal contains the address of the status information of an s-instruction. The signal causes the EC unit to retrieve the status of the specified instruction and decrement its *enable count* field. The instruction is identified as *enabled* when this count reaches zero. At that time, the *count* field of the enabled s-instruction is returned to the value of *reset*. Finally, the EC unit chooses an enabled instruction and sends a fire signal to the PIPU. Since there may be more than one enabled instruction, the EC unit uses a scheduling mechanism to determine the order in which the instructions are fired. Such a scheduling mechanism should be "fair" [6] to ensure that the machine does not repeatedly fire a group of instructions without giving attention to other enabled instructions.

Currently, the enable flag bits in the EC unit are organized as a two dimensional array. The selection of instructions for execution is done by checking each row in turn and sending the contents of any non-zero row to a column encoder. No row is considered again until all other rows have been examined. The column encoder scans each flag bit in the selected row in turn, issuing a fire command for each bit that is on.

This selection scheme is fair in the following sense: If we define a *sweep* as the time required by the row logic to loop through all rows and return to some starting row r, then the mechanism guarantees that all cells that become enabled during one sweep will be fired by the next.

5 STRUCTURE MEMORY OPERATIONS

These instructions are extensions of conventional dataflow operations and are used to support scalar and array accesses in structure memory.

5.1 Regular Load/Store Operations

Load and Store operations are used to transfer scalar datum between data memory and structure memory. A Load operation from a certain address must be preceded by a Store at the same address. Hence, there must be some signal path from the Store actor to its corresponding Load in order for the Load instruction to fire safely (fig. 5).

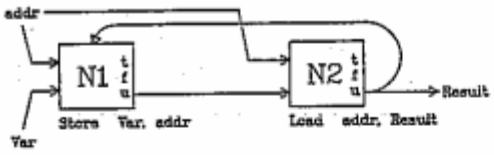

Figure 5: Load and Store Node Signaling

5.2 Array Append and Select Operations

Append takes as operands the base address of the array A, the index into the array i, and a value, and writes this value into the array memory location corresponding to $A[i]$. Select loads a value from array memory at base address A, index j. Since there may be no direct correlation in the order which array elements are being produced and consumed (i.e., i and j may be generated by different functions in different code blocks,) a synchronization bit similar to the I-structure valid/invalid bit [2] must be included with each location in array memory. An Append operation sets the valid bit of the memory location that it is currently writing to. A Select operation must check and wait until the valid bit of a memory location is set before attempting to read it. Therefore, the done signal of the Select operation may have to be delayed while waiting for the required datum.

5.3 Implementation of array operations using Load/Store

Array Append and Select operations can be implemented by split-transactions based on the the scalar Load/Store operations. We introduce two new instructions, *S-Index* and *L-Index*, for computing memory addresses of Append/Select operations from the array base address and the index.

When the base A and index value i for L-Index is available, the corresponding L-Index instruction is executed in the structure memory unit of the PIPU. First, the address of the data in structure memory is computed from A and i. Then it checks the valid bit of the corresponding memory location and, if reset, the issuing of the **done** signal will be delayed. When the index and data values for the Append operation becomes available, the S-Index instruction fires and computes the memory address of the array element. The operation will then (1) activate the Store instruction and (2) release all corresponding pending done signals "parked" in the structure memory unit. To guarantee that the Store operation is fired before the Load operations activated by the released done signals, the extended argument-fetching dataflow architecture provides a "short-cut". The signal link between S-Index and Store does not go through the usual PIPU-DISU-PIPU cycle. Instead, the done signal of the S-Index will act as the fire signal for the Store instruction and is routed directly back to the fire input of PIPU, bypassing the DISU altogether. Because of this, the signal arc from the source node of the datum to be stored enters the S-Index instruction instead of the Store, as S-index is responsible for firing its corresponding Store (fig. 6). Since there are no signal arcs at all to the Store instruction, the structured Store will not have an s-instruction counterpart in the DISU.

This split-transaction realization of Append/Select are an attempt to achieve I-Structure style memory latency tolerance while avoiding the overhead of creating extra memory traffic, as only Load/Store operations are sent to memory. The S-Index/L-Index

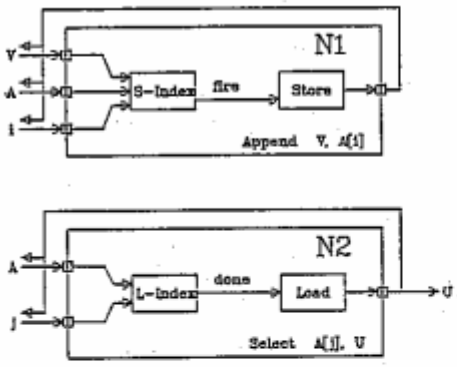

Figure 6: Implementing Append/Select using Load and Store

instructions can be handled elsewhere where a "valid status" bit map of the structure memory unit is maintained.

6 INTERPROCESSOR COMMUNICATIONS IN THE ARGUMENT FETCHING ARCHITECTURE

Dataflow architectures show great potential in multiprocessor applications. There is no software cost needed for synchronization since the data driven model of computation eliminates the need for interrupts, context switches, or busy waits. Additionally, the multi-threaded model ensures that no enabled instruction will be blocked from execution by a waiting instruction.

Previous dataflow architectures based their communications on the argument flow model where the source instruction originates the routing transaction to a remote target. This lacks flexibility and incurs the overhead of having to add data duplicating instructions for routing between multiple PEs (since each instruction may route data to at most one remote PE.) In this section, we propose an interprocessor communication model based entirely on the argument-fetching scheme.

The new method behaves as if there were no boundaries between PEs at all; instructions on different processors are able to interact with each other directly in an interprocessor argument-fetching fashion. Remote data is specified by an addressing mode in one (or both) of the input operands of a p-instruction. When a p-instruction with a remote argument fires, the instruction is removed from the PIPU pipeline and an I/O request for each remote operand is queued in a *parking store* in the local I/O unit. The I/O unit services each request on a first-come-first-served basis by fetching the data from the memory of the remote PE (via the remote I/O unit) and matching the received data with the corresponding request in the parking store. After a blocked instruction has received all of its operands, it is released from the I/O unit and rejoins the PIPU pipe. When the local p-instruction is done, its s-instruction image will send an *interprocessor count* signal to the remote DISU, which will then proceed as if this were any normal count signal.

Since data is fetched rather than sent from the result register (i.e., the originator is the target instruction,) data from one PE may be accessed by several other PEs without adding extra instructions for duplicating data. Since the blocked instruction does not impede other instructions from being executed, as long as there are enough enabled instructions, the processor will be kept usefully busy and can tolerate delay due to interprocessor communications.

Currently, we are investigating interconnection techniques for a multi-PE argument fetching model. In particular, we will be constructing a multiprocessor simulation based on the hypercube network.

7 FUNCTION INVOCATIONS — TOWARD A DYNAMIC ARGUMENT FETCHING ARCHITECTURE

The application spectrum of dataflow architectures is not only limited to high speed numerical computing. In particular, we have been study the applications of the architecture to AI, where the support of multiple function invocations and recursive functions becomes crucial. This represents a departure from the static model towards a dynamic one.

A function invocation instance contains *activity templates* of each actor in the invoked function. The activity template of an actor N, in a function invocation instance F_i, must be "distinct enough" from all other activity templates of N in other invocation instances F_j, where $i \neq j$.

We propose to associate a frame of consecutive memory space for each function invocation called a *function overlay*. These are used to store operand values of PIPU instructions in the function invocation

(the data memory overlay), as well as other dynamic information such as the count values in the DISU (the enable memory overlay). The operand values in the data memory overlay can be accessed with an offset plus a base address. The same base addressing scheme can be applied to the count values in the enable memory overlay. The static portion of an actor now consists of (1) a PIPU part, consisting of an opcode, operand offsets, and a result offset, and (2) a DISU part, consisting of signal list offsets, initial enable count, and reset count.

The key attribute for a function invocation is the base address of its overlays. In our implementation, the **fire**, **done**, and **count** signals must be augmented with base addresses. The instruction and signal formats must also be modified. Although the base address can be considered as an association of color (or tags) to the signals, the new architecture is entirely based on the argument-fetching principle, thus no data token flow (and hence no color matching) exists in the processor. This makes it unique from most proposed dynamic dataflow architectures.

Currently, we are considering the introduction of a special module, the *Memory Overlay Manager* (MOM), for providing dynamic memory management support of multiple function invocations. As for the DISU, we hope to remove the function invocation scheduling (performed by the MOM) from the *critical path* of the processor architecture. A more detailed presentation can be found in [9].

8 CONCLUSIONS

In this paper, we report the research work and preliminary results on the argument-fetching dataflow architecture as an emerging research project in our group — the Advanced Computer Architecture and Program Structures Group at the School of Computer Science of McGill University.

We have described the instruction set architecture design of the argument-fetching dataflow architecture, in particular the program and instruction format, as well as the instruction scheduling and execution mechanism of the new machine. We have also described other important extensions to the basic architecture, namely the structure memory, array operations, and interprocessor communication support in the instruction set design. We briefly discuss the extensions toward a dynamic argument-fetching dataflow architecture where multiple function invocations can be effectively supported.

It is our plan to conduct a more precise analysis and comparison of the advantages and disadvantages between the new architecture and other dataflow or conventional processor architectures. Members in our group are currently studying the performance aspects of the new architecture through simulation. In particular, for the static argument-fetching architecture, we are developing code mapping strategies for a multiprocessor architecture based on compilation techniques for functional languages such as Val and SISAL [1,13]. We also plan to continue our research in the dynamic architecture for AI applications. Portions of the research are being or are expected to be performed in collaboration with other researchers both at McGill and in other institutions.

9 ACKNOWLEDGEMENTS

We thank Dr. J.B. Dennis for many long hours of discussions over a wide range of topics; several ideas inspired by these discussions are included in this paper. We also thank the participants of recent workshops sponsored by our group which provided many constructive suggestions, in particular, the following members of our group: Wong-Kook Hong, Zaharias Paraskevas and Luc Boulianne.

REFERENCES

[1] Ackerman, W.B. and Dennis, J.B., *Val — A Value-Oriented Algorithmic Language: Preliminary Manual*, Technical Report 218, Laboratory for Computer Science, MIT, Cambridge, MA, June 1979.

[2] Arvind, Culler, D.E., Iannucci, *Two Fundamental Issues in Multiprocessing*, first appearance as MIT Computational Structures Group Memo 226-2, Jul. 1983. Subsequent versions of the paper were published in several computer science conferences.

[3] Arvind, Culler, D.E., *Dataflow Architectures*, Annual Reviews in Computer Science, Vol.1, pp.225–253, 1986.

[4] Backus, J., *Can Programming Be Liberated from the Von Neumann Style? A Function Style and Its Algebra of Programs*, CACM,

Vol.21, No.8, Aug. 1978.

[5] Dennis, J.B., Gao, G.R., *An Efficient Pipelined Dataflow Processor Architecture*, to appear in the Proceedings of the IEEE and ACM SIGARCH Conf. on Supercomputing, Florida, Nov. 1988.

[6] Gao, G.R., *A Pipelined Code Mapping Scheme for Static Data Flow Computers*, Ph.D dissertation, Laboratory for Computer Science, MIT, Cambridge, MA, 1983.

[7] Gao, G. R., *A Pipelined Solution Method for Tridiagonal Linear Systems*, Proceedings of 1986 International Conference on Parallel Processing, IEEE Computer Society, pp 84-91, Aug. 1986.

[8] Gao, G. R., *A Pipelined Code Mapping Strategy for Data Flow Supercomputers*, to appear on the Proceedings of the Third International Conference on Supercomputing, Boston, MA, May 1988.

[9] Gao G.R., Hum H., *Function Application Support in the Argument-Fetching Dataflow Architecture*, Advanced Computer Architecture and Program Structures Group Memo 03, School of Computer Science, McGill University, Montreal, May 1988.

[10] Gao, G.R., Paraskevas, Z., *Efficient Software Pipelining in an Argument-Fetching Dataflow Architecture* (Preliminary Version), Advanced Computer Architecture and Program Structures Group Memo 02, School of Computer Science, McGill University, Montreal, Feb. 1988.

[11] Gao, G.R., Tio, R., *Instruction Set Definition for the Argument Fetching Dataflow Architecture, Version 1.0*, (Preliminary Version), Advanced Computer Architecture and Program Structures Group Memo 01, School of Computer Science, McGill University, Montreal, Feb. 1988.

[12] Gurd, J. R., Kirkham, C. C. and Watson, I., *The Manchester Prototype Dataflow Computer*, CACM, Vol. 28, No. 1, Jan. 1985.

[13] McGraw, J., *SISAL: Streams and Iterations in a Single Assignment Language, Language Reference Manual*, Lawrence Livermore National Laboratory, CA., Mar. 1985.