PPL/M: The System Level Language for Programming the *DADO* Machine*

Salvatore J. Stolfo
Daniel Miranker
Mark D. Lerner

CUCS-104-84

Columbia University
February 29, 1984

*This research was supported in part by the Defense Advanced Research Projects Agency under contract N00039-82-C-0427 and the New York State Science and Technology Foundation, as well as grants from Intel Corporation, International Business Machines, Hewlett Packard, Digital Equipment Corporation, Valid Logic Systems, and Bell Laboratories.

Table of Contents

List of Figures

1. Introduction

DADO [Stolfo and Shaw 82; Stolfo 82] is a fine-grain parallel computer designed specifically for the rapid execution of Artificial Intelligence (AI) software. Two software systems are presently under development for *DADO* which implement two language facilities supporting the high-speed execution of Production Systems (PS) and Logic Programs:

- Herbal (named in honor of Herbert Simon and Allen Newell, inventors of the AI PS paradigm) is similar in style to OPS [Forgy 82].
- LPS (a Logic Programming System) is similar in syntactic style to Prolog.

For a number of years we have studied a variety of machine models to determine a suitable (and relatively inexpensive to realize in hardware) parallel machine organization for these two application domains. A number of algorithms have been studied which were designed to capture the inherent parallelism in a wide range of PS and Logic Programming applications. These algorithms led to the current model of the machine, to be described shortly.

A prototype machine, which we call *DADO1*, has been operational at Columbia University since April 1983. *DADO1* consists of 15 commercially available microprocessors each associated with a single random access memory chip. These 15 processing elements (PEs), interconnected as a complete binary tree, serve as a testbed for the development of software for a larger prototype. The larger *DADO2* machine will comprise 1023 PEs and is expected to be completed within two years. The reader is encouraged to see [Stolfo 83] for details concerning the hardware implementation of *DADO*, and the rationale for its design.

In the present paper we describe PPL/M, which is the first system level programming language implemented for *DADO*. PPL/M was rapidly developed using a number of commercially available compiler systems. The PPL/M language, as defined in subsequent sections, served to test our ideas and clarify our thinking about programming the machine. Our experiment has been successful. A simple PS interpreter has been written in PPL/M and demonstrated on *DADO*.

The PPL/M experiment has also helped considerably towards identifying a suitable LISP-based programming language for *DADO* which provides a more convenient environment for AI programming. In [Weisberg 84] we report on the current status of the ||PSL (Parallel Portable Standard Lisp) implementation and identify those aspects of the language derived from PPL/M. For the present paper we describe only PPL/M. We begin with a brief overview of the *DADO* machine architecture.

2. The *DADO* Machine Architecture

The power and number of processing elements expected to appear in a full scale version of the machine are topics of ongoing experimental research; however, it is expected that many thousands of processors each capable of efficiently executing algorithms of significant complexity (e.g., unification, pattern matching, etc) will be used. The PE's are interconnected to form a ***complete binary tree***.

Within the *DADO* machine, each PE is capable of executing in either of two modes. In the first, which we will call SIMD mode (for single instruction stream, multiple data stream [Flynn 72]), the PE executes instructions broadcast by some ancestor PE within the tree.

In the second, which will be referred to as MIMD mode (for multiple instruction stream, multiple data stream), each PE executes instructions stored in its own local RAM, independently of the other PE's. A conventional ***host processor***, adjacent to the root of the *DADO* tree, controls the operation of the entire ensemble of PE's.

When a *DADO* PE enters MIMD mode, its logical state is changed in such a way as to effectively "disconnect" it and its descendants from all higher-level PE's in the tree. In particular, a PE in MIMD mode does not receive any instructions that might be placed on the tree-structured communication bus by one of its ancestors. Such a PE may, however, broadcast instructions to be executed by its own descendants, providing these descendants have themselves been switched to SIMD mode.

The *DADO* machine can thus be configured in such a way that an arbitrary internal node in the tree acts as the root of a tree-structured SIMD device in which all PE's execute a single instruction (on different data) at a given point in time. This flexible architectural design supports the logical division of the machine into distinct partitions each executing a distinct task. This is called multiple-SIMD (MSIMD) operation.

Details concerning the specific hardware design of the machine are beyond the scope of this paper. We focus here upon the execution semantics of a *DADO* PE, as defined above, and detail the language constructs implementing the various modes of operation. Specifically, PPL/M provides:

- Constructs specifying SIMD mode of computation. While in SIMD mode, a PE may be:
    - ***enabled***, in which case it will receive an instruction from its parent, send the instruction to its two children, execute the instruction, and continuously repeat these steps.
    - ***disabled***, in which case it repeatedly receives an instruction from its parent, and sends it to the children without executing it.

- Constructs specifying MIMD mode of computation, in which the processor executes instructions from its local RAM.

- Global communication instructions:
    - *Broadcast*, to send data from a MIMD mode processor down the tree to its SIMD mode descendants
    - *Report*, to send data from one designated SIMD mode processor up the tree to the MIMD mode root
    - *Send* and *Recv*, to provide tree neighbor communication between physically adjacent processors
    - *Resolve*, to select one processor from a collection of distinguished processors.

It is convenient to think of the host as the root of the *DADO* tree, which always executes as a MIMD mode PE. Thus, in the following, when we refer to a MIMD mode PE our comments also apply to the host processor. All of the PPL/M system is resident in the host. PPL/M programs, though, can be executed by the host and by any PE of *DADO*. Facilities resident in the host manage the loading of the tree.

The PPL/M language provides a precompiler and a software kernel to support these operations.

3. PPL/M: Parallel PL/M

Herbal **and LPS are** designed as very high level user application languages for *DADO*. Consequently, the view of *DADO* as a massively parallel binary tree-structured machine is nearly transparent within these programming formalisms. Both are to be implemented in PPL/M or the closely related ||PSL language. Thus, in order to maximize performance, PPL/M provides constructs that directly access the *DADO* machine structure. Therefore, PPL/M may be viewed as a rather low-level parallel programming language, suitable for tree-structured machines.

PPL/M is a superset of the PL/M language [Intel 82]. Before detailing the syntax and semantics of PPL/M, we begin with a brief introduction to PL/M.

PL/M is a high-level language designed by Intel Corporation as the host programming environment for applications using the full range of Intel microcomputer and microcontroller chips. Some of PL/M's salient characteristics include:

- block structure, employing several forms of the PL/1 DO statement,
- a full range of data structure [illegible]lities including arrays, structures and pointer-based dynamic variables
- "strong typing" facilities (thus, data and subroutine definition statements are provided)
- a statement-oriented syntactic structure
- all data is either of type BIT, BYTE or WORD (2 bytes).

A PL/M program is constructed from blocks of associated statements, delimited by either a DO or PROCEDURE statement, and a terminating END statement. As is typical of a block oriented language, nesting is permitted following the usual conventions for variable scoping.

We will describe each of the executable statements briefly in turn. (In the following definitions, symbols appearing within the bounds of square brackets[ ] are optional, whereas symbols appearing within set brackets {} are alternates.)

Assignment statement

```
     identifier [,identifier]* = expression;
```

The expression follows the usual conventions with the added provision of implicit type conversion between BYTE and WORD data. Implicit conversion of BIT data is prohibited. (Refer to the section on data structures in the PL/M manu~~al.)~~ Multiple assignment is unpredictable if a variable appears on both sides of the assignment operator.

IF statement

```
IF relational-expression THEN statement;
   [ELSE statement;]
```

The relational expression provides the full range of logical and relational operators, resulting in a value of type BIT.

Simple DO statement

```
[label:]DO;
        statement-0;
             .
             .
             .
        statement-n;
       END [label];
```

The statement may be a data definition whose scope is defined by the bounds of the block.

Iterative DO statement

```
DO counter = start-expression TO limit-expression
                             [BY step-expression];
 statement-0;
   .
   .
   .
 statement-n;
END;
```

Each expression is evaluated once prior to the loop, while the termination test is performed on each entry into the loop.

DO WHILE statement

```
DO WHILE relational-expression;
 statement-0;
   .
   .
   .
 statement-n;
END;
```

The relational-expression must result in a value of type BIT.

DO CASE statement

```
DO CASE select-expression;
  statement-0;
    .
    .
    .
  statement-n;
END;
```

The select-expression must yield a BYTE or WORD value, which is used to select a single statement for execution. Eighty-four cases is the maximum allowable number. If the select-expression is out of range, results are unpredictable.

CALL statement

```
CALL name[(parameter list)];
```

The name must be the name of an untyped procedure. Indirect calling is possible by specifying the address operator defined below.

Definition statements

```
    label-name: statement;
```

Labels are defined by use and are subject to the same scoping rules as variables.

Explicit declaration and typing is done primarily with the declare statement.

```
    DECLARE variable [(single array dimension)] type;
    DECLARE (variable list) type;
```

The type of variable may be:

- BIT
- BYTE
- WORD
- STRUCTURE (variable type [,variable type*])
- {BIT BYTE WORD} BASED variable

Strings and constants can be manipulated by operating on memory referenced indirectly through based variables and pointers. For example,

```
    DECLARE ptr WORD;
    DECLARE string(64) BYTE BASED ptr;
```

Any reference to **string** will use the current WORD value stored in the variable ptr as the base address. Based variables used in conjunction with the dot operator perform all of the indirect addressing capabilities of a high level language.

The dot (.) operator

```
    . variable
```

This operator returns the address location (a value of type WORD) of variable. It can also be used with constant lists as for example:

```
    .('ABC')
```

The dot operator serves the dual purpose of structure variable qualification. If **x** is of type structure with subcomponents **y** and **z**, each component is referenced by **x.y** and **x.z**.

Procedure definitions

```
    name: PROCEDURE [(parameter list)] [type];
        statement-0;
          .
          .
          .
        statement-n;
      END name;
```

Typical conventions are used with type conversion of arguments. Untyped procedures are CALLed, while typed procedures are referred to within expressions as a function call.

3.1 Parallel Processing Primitives of the PPL/M Language

Many of the design choices made in the definition of PPL/M were influenced by the methods employed in specifying parallel computation in the GLYPNIR [Lowrie 75] language implementation for the ILLIAC IV processor. PPL/M provides two syntactic constructs and ten primitive functions which significantly enhance the PL/M language. These allow specification of parallelism, communication between processors, and selection of particular PEs.

The first construct for programming the SIMD mode of operation of *DADO* is the SLICE attribute. This defines a variable or procedure that will be stored at the same location in each PE. A SLICEd variable may be viewed as a vector which can be operated upon in parallel. SLICEd procedures are automatically loaded and stored at the same location within each PE.

The second addition is a syntactic construct, the *DO SIMD* block. This delimits the parallel instruction sequences which are executed by SIMD-enabled PE's. These blocks are translated into PL/M by the PPL/M precompiler.

The PPL/M language software allows users to employ all parallel processing primitives. The compiler generates the synchronization and communication primitives as calls to kernel software.

3.1.1 The Slice Attribute

The SLICE declaration is a mechanism to guarantee that identically named objects (variables and procedures) reside at the same location in all processors. Before a variable is used in a *DO SIMD* block, the programmer must declare it to have the SLICE attribute. The precompiler restricts SLICE procedures to address only SLICEd variables. This allows the rapid execution of the SIMD instruction stream, since instructions do not require address modification before they are executed. For example.

```
    DECLARE variable[(single array dimension)] type SLICE;
    name: PROCEDURE[(parameter list)] [type] SLICE;
```

3.1.2 The DO SIMD Block

An assignment of a value to a SLICE variable causes the data transfer to occur concurrently within each enabled SIMD PE, and must be written within the scope of a DO SIMD block. The expression on the right hand side (rhs) of an assignment statement is evaluated concurrently within all enabled descendants. Each evaluation utilizes the local store of the PE in which it is performed. Specifically, a constant appearing on the right hand side is assigned to the left hand side variable, and all rhs expressions are evaluated independently of other PEs.

Invocation of a SLICE procedure occurs when the PL/M CALL statement is written within the scope of a DO SIMD block. It results in the concurrent transfer of control within each SIMD enabled PE. This serves two purposes. First, all PL/M statements may occur within a SLICE procedure, whereas only a subset of PL/M is allowable within a DO SIMD block (as described below). Second, repeated code sequences can be written once as a subroutine and invoked from several blocks.

In general, invoked SLICEd procedures may require different amounts of time in distinct PEs. Consequently synchronization is implicitly enforced during execution of communication primitives, and is directly supported by the hardware. Thus, execution of the instructions which follow a primitive will not proceed until all SIMD PEs terminate the execution of the operation.

The complete syntax is:

```
DO SIMD;
 r-statement-0;
  .
  .
  .
 r-statement-n;
END;
```

The r-statement is restricted to be either
- an assignment statement *incorporating only SLICE variables and constants* or
- a call to a subroutine that has been declared to be of type SLICE.
- a call to a system level subroutine.

For example, the statements
```
DO SIMD;
  X=5;
END;
```
where X is of type **BYTE SLICE**, will assign the value 5 to each occurrence of X in every SIMD enable PE. The statements
```
DO SIMD;
  X=2*X+1
END;
```
will update the value of X in the SIMD enabled processors by operating upon the value which resides in each processor.

A non-SLICE variable may appear within an r-statement only as an argument to the BROADCAST function (to be defined shortly). The BROADCAST function provides the means to communicate data used by a PE in MIMD mode to descendant PEs executing in SIMD mode. In the following example, **MVAL** is any variable, and **SIMDVAR** is any variable with the SLICE attribute.
```
DO SIMD;
  CALL BROADCAST(MVAL);
  SIMDVAR=A8;
  X=2*X+SIMDVAR;
END;
```

This will update the value of x in each PE by adding the root value of MVAR (broadcast to the local SLICEd variable A8, discussed below) to twice the local x value.

3.1.3 Added Built-in Communication Primitives

The following is a detailed description of the communication primitives, which are invoked with the PL/M CALL statement. The communication primitives in *DADO1* are implemented in software. They are being implemented in hardware for the larger *DADO2* machine.

The primitive communication operations use the following user accessible variables to specify the status of the parallel operations. These variables are resident in all PEs as SLICEd variables, although use of CPRBYTE and CPRR is restricted to MIMD mode processors.

- EN1. (boolean). When set to *true* the SIMD processor is enabled, and set to *false* to disable the processor.
- A1. (boolean). Prior to the resolve operation, the SIMD processors which are to participate in the operation set this bit. The resolve operation will leave this bit set in only one processor.
- CPRBYTE. (byte). Receives data which is transmitted up the tree from a single enabled SIMD PE to the MIMD mode PE issuing the REPORT primitive.
- CPRR. (boolean). Receives the status of the resolve operation.
- IO8. (byte). Provides the data to be transmitted to other processors.
- A8. (byte). Receives data which is transmitted from other processors, by the BROADCAST, SEND and RECV primitives.

A PE may disable itself by transferring a 0 into its EN1 register using an ordinary assignment statement. In a typical application, the contents of EN1 will be set to the result of some boolean test prior to the execution of such a store instruction, resulting in the selective disabling of all PEs for which the test fails. This technique supports the "conditional" execution of a particular code sequence. Following the execution of such a sequence, an ENABLE instruction is issued to "awaken" all disabled PEs. In combination with appropriate register, transfer and logical operations, this approach may be used to implement more complex conditionals, including nested "IF-THEN-ELSE" constructs embedded within a DO SIMD block.

The primitive communication operations will now be described. (The PPL/M syntax is indicated in boldface.) All operations may be viewed as being issued by a MIMD mode PE, with the entire tree of SIMD mode descendants participating in the operation.

Call RESOLVE

RESOLVE is the basic operation to control information flow to the top of the tree. It selects at most one PE from a candidate set, and indicates to the MIMD mode PE whether a PE was chosen.

The candidate set consists of PEs with A1=1. PEs with A1=0 are ignored. After execution of this instruction the first PE encountered in an inorder traversal, whose A1 flag is set, is considered the winning PE. The A1 flag in this PE remains set to 1, while all other PEs have their A1 bit set to zero. The control processor's CPRR bit indicates the status of the operation: 1 indicates that a PE was selected. CPRR will be set to 0 only if all PEs had A1=0 before the RESOLVE. No PEs are enabled or disabled by this routine.

In applications where several PEs must be identified (for example, if all ties are to be examined) the A1 bits are stored in a local save-area before the resolve operation. After the resolve operation, by execution of an instruction to move the contents of A1 to EN1, only one processor will remain enabled.

The single enabled PE may now store a 0 into the save-area of the A1 bit. The tree is re-enabled and the A1 is restored from the save-area. This technique is used iteratively to enumerate each member of the candidate set until CPRR is zero. For example:

```
DO SIMD;
  CALL ENABLE;              Initially enable processors
  A1=BOOLEAN(SomeTest);     Set up initial condition
END;
DO SIMD;
  CALL ENABLE;       Re-enable everyone within the loop
  SaveValue=A1;      Save the A1 bit
  CALL RESOLVE;      Find one winner
  EN1=A1;            Turn off everything else
  SaveValue=0;       Enabled processor leaves candidate set
  CALL Process;      Enabled processor executes arbitrary code
END;
```

The RESOLVE instruction may also be used to provide the control processor with a binary completion code. As shown below, this programming technique allows PEs to receive and operate upon data as long as one PE remains enabled. The following code sequence illustrates a method by which a candidate set of values may be communicated to all processors (by the GetNextBinding routine), and then used in a match process (by the DoMatch routine). This process continues until the DoMatch routine stores 0 into EN1.

```
more=1;
DO WHILE more;
  DO SIMD;
    Call GetNextBinding; Obtain more information for matching.
    Call DoMatch;        This will disable the processor when a match found.
    Call Resolve;        Tell Control Processor when all processors are done.
  END;
  more=CPRR;             CPRR will be true if any PEs want to keep processing.
END;
```

Call REPORT

The ***report*** routine transfers data from descendant PEs up the tree to the MIMD root PE. The value in the A8 register of an enabled descendant PE is written into the CPRBYTE of the root processor. If more than one descendant is enabled, then the lowest numbered PE (according to tree-inorder traversal) is used to resolve the conflict. This conflict resolution does not require any additional time, but is actually a byproduct of the way the *DADO* circuitry is designed.

The following is an example of transferring a single byte to the root:

```
DO SIMD;
  A8=Simd Var;
  CALL Report;
END;
Mimd_Var=CprByte;
```

Call BROADCAST(<byte>)

This is the primary mechanism for downward communication in the tree. The argument of BROADCAST is placed on the broadcast bus and is stored into the A8 register of SIMD enabled processors in the subtree. If some descendant PE is in MIMD mode, that PE as well as its entire set of descendants will not receive the byte, since they execute instructions independently of their ancestors.

Call SEND(<neighbor-PE>)

The SEND and RECV instructions implement tree neighbor communication. They have been found of infrequent, though important, usage in the algorithms written to date, and consequently are implemented in firmware in both the *DADO1* and *DADO2* machines.

The instruction sends the contents of the A8 byte into the IO8 byte of the designated neighbor. The neighbor may be any of the following:

LC	left tree child
RC	right tree child

A PE is not permitted to SEND to its parent since the semantics would be undefined if two descendants of a PE attempted to SEND simultaneously.

Call RECV(<neighbor-PE>)

This routine receives the value of a neighbor PE. The A8 byte of the originator PE receives the value of the IO8 byte of the neighbor. The neighbors may be designated as:

LC	left tree child
RC	right tree child
P	parent node

We illustrate this instruction with the following code sequence to mark and label each level of the machine.

```
Mark: PROCEDURE(Level);
  DECLARE (Level,I) BYTE;
  DECLARE Markbit BYTE SLICE;
  DO SIMD;
    CALL ENABLE;          Enable all processors.
    IO8=1;                Set output byte to 1.
    CALL RECV(P);         Receive byte from parent. Root received 0 since it has no parent.
    Markbit=NOT(BOOLEAN(A8)); Only parent is "1" now.
  END;
  I=0;
  DO WHILE I<Level;   Send the "1" down to proper level.
    DO SIMD;
      IO8=EXPAND(Markbit);     Store output byte.
      CALL RECV(P);            Send "markbit" down a level.
      Markbit=BOOLEAN(A8);     Receive new "markbit."
    END;
    I=I+1;                Keep track of the level.
  END;
END Mark;
```

Call MIMD(<address>)

This instruction is used to partition the tree into independently executing subtrees. The address specified in the instruction is broadcast down the tree. The SIMD enabled PEs then logically disconnect themselves and enter MIMD mode.

After disconnection from their parents, the MIMD PEs begin execution of code stored in their local RAMs. The address given as the actual parameter to the CALL MIMD instruction is the address of the procedure to be activated in each PE. Any SIMD-disabled processors become descendants of their nearest MIMD mode ancestor.

The MIMD mode of operation is terminated in two phases, which may be performed in any order. To describe this process we introduce the following vocabulary. Prior to the operation there was one root to the tree; this is called the ***originator*** node. Subsequent to the operation there may be many disconnected MIMD processors; these are called *roots of mimd subtrees*.

The processors return to SIMD-disable state as soon as the two conditions are satisfied:

1. Each root of a mimd subtree calls the EXIT routine to indicate it is prepared to return to SIMD disabled state.
2. The originator node calls the SYNC routine to restore the logical state and reconnect the children.

Call EXIT

This routine is executed by the roots of mimd subtrees when they need to reconnect to the tree. The subtree is placed into SIMD disabled state, and the reconnection will complete when the parent of the new tree completes its call to SYNC. After completion of the sequence, the PE will resume participation in all communication primitives.

Call SYNC

This routine is executed by the originator of a CALL MIMD when it is prepared for its children to reconnect to the tree. The logical state of the tree is restored to recognize the children.

This routine operates by polling its children until they have executed the EXIT routine. The polling presently waits until the children respond. It is also possible for a user program to periodically check the communication path, and execute user-supplied instructions until the children are ready.

Call ENABLE

This sets the EN1 register of all descendant PEs to 1, thus enabling all processors.

Call DISABLE

This sets the EN1 register of all descendant PEs to 0, thereby disabling all processors.

Figure 3-1 summarizes the communication primitives.

	<- STATE OF THE				
	<- SIMD PART OF THE PEs ->		<- MIMD PART OF PEs ->		
Routine Name	<- before ->	<- after ->	<- before ->	<- after ->	Routine Purpose
	<- USE OF ROUTINE			->	
RESOLVE	A1 bits on if processor is to be used in the resolve.	A1 bits set to zero, except for first one.	Not applic-able.	CPRR set if any A1 bit is high.	Selects one PE from the candidate set.
REPORT	Data to send up to the root is stored in A8 byte of SIMD enabled child.	Not applic-able.	Not applic-able.	CPRBYTE at root receives data sent up the tree.	Transmit data up the tree from one child.
BROADCAST	Root of the SIMD subtree broadcasts a single byte argument.	A8 byte in the enabled descendants receive the data.	None.	None.	Transmits data down the tree, from root to children.
SEND	Source stores byte in its IO8, and calls routine with name of the destination.	Single byte is placed into the A8 byte of the destination PE.	None.	None.	Tree neighbor communic-ation: transmit.
RECV	The source stores a byte in its IO8, and the dest-ination selects the source.	Single byte is placed in the A8 of the PE which initiates the RECV.	None.	None.	Tree neighbor communic-ation: read.
ENABLE	Executes call.	Changes EN1 value.	Initiates call.	None.	Enables a SIMD processor.
DISABLE	Executes call.	Changes EN1 value.	Initiates call.	None.	DISABLES a SIMD processor.
MIMD	None.	SIMD enabled processor enters MIMD mode.	Initiates call with name of a mimd routine.	None.	Partition tree into multiple mimds.
EXIT	Descendant executes to terminate its MIMD mode and return to SIMD.				
SYNC	Parent executes to regain control of its children and synchronize.				

Figure 3-1: Summary of *PPL/M* Primitives

3.2 Examples

Code for two fundamental operations is presented in this subsection: the first loads the *DADO* tree sequentially with data from some external source; the second is used to associatively mark all PEs which store data that match a given search string. These two code sequences were the first to successfully execute on the *DADO1* machine.

Figure 3-2: Sequentially Loading *DADO*

```
   We will assume that this program is executed within
  DADO's CP. The system function READSTR loads string
  data into a buffer from some external source.

SEQLOAD: PROCEDURE:
  DECLARE Intelligent-record(64) BYTE SLICE EXTERNAL;
  DECLARE Not done BYTE SLICE;
  DECLARE (Index,Length) BYTE SLICE;
  DECLARE I BYTE;
  DECLARE Buffer(64) BYTE;

  DO SIMD;
    CALL SENABLE;    ALL PE'S ARE ENABLED
    NOT DONE = 1;    ALL SLICES INITIALIZED
    INDEX = 0;
  END;

LOADLOOP:
   pick a pe to load the next record into
  DO SIMD;
    CALL Enable;
    A1 = BOOLEAN(Not Done);
    CALL Resolve;    Only one A1 is now set
    EN1 = A1;        Selectively disable all but one pe
    Not_Done = 0;
  END;

  IF Cprr=0 THEN     If tree is full
  DO;
    Call Writestr(.Mfull);
    RETURN;
  END;
  CALL Readstr(.Buffer,.Length);    Data provided by external source
  IF Buffer(0) =(') THEN RETURN;

  DO I= 0 TO LENGTH-1;
    CALL Broadcast(Buffer(I));
    DO SIMD;
      Intelligent Record(Index) = A8;
      Index = IndEx + 1;
    END;
  END;
  DO SIMD;
    Intelligent_Record(Index)=0;
  END;
  GOTO LOADLOOP;
END SEQLOAD;
```

The second example implements the most basic operation for associative matching on *DADO*.

Figure 3-3: Associative Probing

```
ASSPRO: PROCEDURE (BUFPTR,LENGTH);
        declare BUFPTR WORD;
        declare LENGTH BYTE;
        DECLARE INDEX BYTE SLICE;
        DECLARE I BYTE AUXILIARY;
        DECLARE ATBUFPTR BASED BUFPTR BYTE;

        DO SIMD;
           CALL ENABLE;                   Initially enable all PEs
        END;

        DO I = 0 TO LENGTH-1;             Broadcast bytes to look for
         CALL Broadcast(I);               First send the index
         DO SIMD;
           Index = A8;
         END;
         CALL Broadcast(Atbufptr);        Then send the data
         DO SIMD;
           Disable PEs that don't match
           En1, A1 = A8 = Intelligent_Record(Index);
         END;
         Bufptr = Bufptr+1;
        END;
        DO SIMD;
         CALL Resolve;
        END;
END asspro;
```

3.3 Implementation of PPL/M

The PPL/M language was implemented by a precompiler which analyses source code and replaces it with sequences of calls to system level subroutines*. In addition to supporting the above parallel primitives, the precompiler performs syntax checking and error message generation. The result of preprocessing is compiled by the Intel compiler on an Intel Microcomputer Development System. After compilation, code is downloaded into the *DADO1* tree and executed. For example, the statement:

```
DO SIMD;
  X=5;
END;
```

compiles to a set of statements that invoke the system routine SIMD to perform a parallel assignment of the constant 5 to the sliced variable X occurring in each simd enabled processor.

As noted, parallel code sequences are programmed as DO SIMD blocks. These are translated by the precompiler into parameterless subroutines. The precompiler

*The precompiler was implemented on a DEC VAX/11-750 using Lex [Lesk 75] and YACC [Johnson 75] and consists of 20 lexical rules and 78 parsing rules.

generates ***test-and-jump*** sequences to test whether the PE is enabled. Disabled processors only execute system functions, as these require the full cooperation of all PEs in the tree. Tests are inserted:

- after assignment to EN1, since this could disable a processor
- after all CALL instructions, since these may also affect the setting of EN1.

The inserted jump instructions transfer execution to the next CALL statement within the DO SIMD block. This guarantees the execution of kernel routines having global effects in the machine.

The precompiler does not generate unnecessary tests or jumps. For example, no test is required after a call statement which is the last in a DO SIMD block, or after a CALL ENABLE instruction.

For example, the following code from a production system interpreter is shown in both PPL/M and the resulting PL/M. This code transmits a string from a SIMD enabled PE to the root processor. Commentary has been italicized.

PPL/M Code

```
StringLen=-1; Done=0;           Initialise loop
DO WHILE (DONE=0);
  stringlen=stringlen+1;
  DO SIMD;
     A8=Lstring(Index);         Move in byte
     Index=Index + 1;           Increment pointer
     CALL REPORT;               Fetch the byte
  END;
  String(Stringlen)=CprByte;    Store it
  IF String(Stringlen)=0 THEN Done=1;    Check if done
END;
```

PL/M Result

```
StringLen=-1; Done=0;   Initialise loop
DO WHILE (done=0);
  Stringlen=Stringlen+1;
  DO;
       CALL SIMD;       Invoke the kernel routine, which will goto s2
       GOTO E2;         When SIMD returns, execution is complete
   S2: IF NOT EN1 THEN GOTO L3;  Skip user code if SIMD disabled
       A8=Lstring(Index);               Move in byte
       Index=Index + 1;                 (increment pointer
   L3: CALL REPORT;     Note label generated for communication primitive
       RETURN;          Terminate the SIMD routine
   E2: END;             End of DO SIMD block
  String(Stringlen)=CprByte;            Store it
  IF String(Stringlen)=0 THEN Done=1;   Check if done
END;
```

The kernel support of SIMD mode execution has been implemented on *DADO1* in two ways. First, it can broadcast machine instructions which the descendant PEs immediately execute. Alternatively, the instructions can be pre-stored in blocks within all descendant PEs, with execution invoked by transmission of the routine's unique identification. There are performance tradeoffs between these methods.

A SIMD machine conserves space when it broadcasts its instructions for execution, yet the time needed to broadcast and store the instructions may exceed the time to execute them. A faster technique is available when needed, which stores compiled procedures in the descendant processors, and subsequently broadcasts the function address to invoke the procedure. This technique can also be used to broadcast code blocks; it requires a SLICE procedure which receives instructions from the broadcast bus, stores them at a prespecified address, and subsequently transfers control to this address.

4. Conclusion

The PPL/M language is important for several reasons. First, it provides the necessary environment for programming a parallel machine. This has allowed us to learn important techniques for implementing and debugging algorithms. Moreover, PPL/M has helped to resolve several hardware design issues that would otherwise be difficult to understand without code sequences to study.

Nevertheless, PPL/M is not completely sufficient for our needs. It suffers from many drawbacks. It does not provide a sufficient level of abstraction to develop high level algorithms; for example it is unable to pass arbitrary data structures in the tree. Moreover the PL/M-51 compiler generates very inefficient code, and supports neither abstact datatypes or recusion.

We consequently embarked on ||PSL LISP, which provides additional facilities to ameliorate many of the shortcomings of PPL/M. The PPL/M language exists, however, and has been demonstrated on an operational prototype parallel computer. It is hoped that our experience in developing a systems-level programming language for a massively parallel computer may help to guide others who are investigating similar machine architectures.

References

Browning S., "Hierarchically Organized Machines", In Mead and Conway (Eds.), *Introduction to VLSI Systems*, 1978.

Browning S., *The Tree Machine: A Highly Concurrent Computing Environment*, Ph. D. Thesis, California Institute of Technology, 1980.

Flynn M. J., "Some Computer Organizations and Their Effectiveness", *IEEE Transactions on Computers*, 1972.

Forgy C. L., *A Note on Production Systems and ILLIAC IV*, Technical Report 130, Department of Computer Science, Carnegie-Mellon University, 1980.

Griss M. L., and A. C. Hearne, "A Portable LISP Compiler." *Software - Practice and Experience*, 11:541-605, June, 1981.

Intel Corporation, *PL/M-51 Users's Guide for the 8051 Based Development System*, Order Number 121966, 1982.

Johnson S. C., *Yacc: Yet Another compiler Compiler*, Computing Science Technical Report No. 32, 1975, Bell Laboratories, Murray Hill, NJ 07974.

Leiserson C. E., *Area-Efficient VLSI Computation*, Ph. D. Thesis, Department of Computer Science, Carnegie-Mellon University, 1981.

Lesk M. E., "Lex – A lexical Analyzer Generator," *Comp. Sci. Tech. Rep. No. 39*, Bell Laboratories, Murray Hill, New Jersey (October 1975).

Lowrie Duncan D., T. Layman, D. Daer and J. M. Randal, "*GLYPNIR*- A Programming Language for *ILLIAC IV*", *Comm. ACM*, 18 3, March, 1975.

Miranker D., "Herbal, A Production System for the *DADO* Machine," Technical Report (in preparation) Department of Computer Science, Columbia University, 1984.

Stolfo S. J., and D. E. Shaw, "Specialized Hardware for Production Systems," Department of Computer Science, Columbia University, 1981.

Stolfo S. J., and D. E. Shaw, "DADO: A Tree-structured Machine Architecture for Production Systems," in the *Proceedings of National Conference on Artificial Intelligence*, 1982.

Stolfo S. J., "The *DADO* Parallel Computer," Department of Computer Science, Columbia University Technical Report," submitted to *AI Journal*, 1983.

Stolfo S. J., "Knowledge Engineering, Theory and Practice," Proceedings of the IEEE *Trends and Applications*, 1983.

Taylor S., "LPS, A Logic Programming System: Motivations and Goals" Technical Report (in preparation) Department of Computer Science, Columbia University, 1984.

Taylor S., A. Lowry, G. Q. Maguire, and S. J. Stolfo, "Logic Programming using Parallel Associative Operations," in *1984 International Conference on Logic Programming*, February 6-9, 1984.

Weisberg M., M. Lerner, G. Q. Maguire, and S. J. Stolfo, "||PSL: A Parallel Lisp for the *DADO* Machine," submitted to *ACM Conference on Lisp and Functional Programming*.