Programming the DADO Machine: An Introduction to PPL/M*

Salvatore J. Stolfo
Daniel Miranker
David Elliot Shaw

CUCS-34-82

Columbia University
November 15, 1982

*This research was supported in part by the Defense Advanced Research Projects Agency under contract N00039-82-C-0427.

Errata: Programming The DADO Machine

Page 8:

"...Untyped procedures are CALLed, while *typed* procedures are referred to within expressions as a function call."

Page 9:

CPRR should be declared as type BIT.

Page 10:

```
Call RECV(<neighbor-PE>);
                -- the contents of register A8 is set to the value
                   stored in I08 of <neighbor-PE>. <neighbor-PE>
                   may be one of: LC, RC, LN, RN and P (parent)
```

Table of Contents

List of Figures

1 Introduction

DADO [Stolfo and Shaw, 1982] is a highly parallel, tree-structured machine designed to provide significant performance improvements in the execution of Artificial Intelligence software. The *DADO* prototype, currently being constructed at Columbia University, comprises 1023 processing elements (PE's) each consisting of an Intel 8751 microcomputer chip and an Intel 2186 8K by 8 RAM chip. The PE's are interconnected in a complete binary tree. A full version of *DADO* would comprise on the order of a hundred thousand PE's each consisting of a much smaller amount of local memory, roughly 2K bytes of RAM. (The 8K RAM employed in the *DADO* prototype was chosen to allow a modest amount of flexibility in designing and implementing the software base for the full version of *DADO*.) In addition, a specialized combinatorial I/O switch is incorporated in the full *DADO* design to perform the most basic communication primitives at much higher speeds than is possible with sequential logic, as it is implemented on the prototype machine.

The Intel 8751 is a powerful 8-bit microcomputer incorporating a 4K Eraseable, Programmable ROM (EPROM), and a 256 byte RAM on a single silicon chip. One of the key characteristics of the 8751 processor is its I/O capability. The four parallel, bi-directional, 8-bit ports provided in a 40-pin package, has substantially contributed to the ease of implementing a binary tree interconnection between processors.

Certain aspects of the *DADO* machine are modelled after *NON-VON* [Shaw, 1982; Shaw, et al., 1981], a tree-structured, highly parallel machine containing a larger number of much simpler processing elements.

In *NON-VON*, most of the PE's are severely restricted in both processing power and storage capacity, and are thus not typically used to execute independent programs. Instead, a single *control processor* (CP), located at the root of the *NON-VON* tree, typically broadcasts a single stream of instructions to all PE's in the tree. Each such instruction is then simultaneously executed (on different data) by every PE in the tree. This mode of operation has been referred to in the literature of parallel computation as single instruction stream, multiple data stream (SIMD) execution [Flynn, 1972].

Within the *DADO* machine, on the other hand, each PE is capable of executing in either of two modes. In the first, which we will call *SIMD mode*, the PE executes instructions broadcast by some ancestor PE within the tree, as in the *NON-VON* machine. 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.

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 all of 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 at a given point in time.

DADO supports communication between physically adjacent tree neighbors, as well as communication between PE's that are adjacent in a logical linear ordering embedded within the tree. (The *NON-VON* I/O switch supports the *in-order* tree enumeration, whereas *DADO* supports the *bounded-neighbor* ordering through sequential logic. Thus, two adjacent PE's are never more than 3 tree edges apart. [Shaw, 1982] provides a complete specification of various methods of embedding a linear ordering on a tree.)

In the following sections we detail the precise semantics of both execution modes, and have outlined the methods employed to simulate each in the current *DADO* prototype design. In subsequent sections we define PPL/M, a variant of the PL/M language providing several additional primitives for parallel computation.

2 SIMD Mode of Operation

A processor in SIMD mode (henceforth, a SIMD PE) can be instructed to enter one of two states which is determined by the contents of a special single bit register called EN1. If EN1 is set high (logical 1) within a PE, the processor will be in the SIMD *enabled* state, otherwise it is in the SIMD *disabled* state.

2.1 SIMD *ENABLED* state

A *DADO* PE in SIMD enabled state will:

1. accept an instruction from the broadcast bus (received from its parent),
2. pass the instruction on to its descendants, provided the PE is *not a leaf* processor AND its immediate tree neighbors (children) are logically connected (see below), and
3. the instruction is executed by the PE.

2.2 SIMD *DISABLED* state

A *DADO* PE in SIMD disabled state will

1. accept an instruction from the broadcast bus, and
2. as in the enabled case, it will pass the instruction on to its descendants if they exist, however
3. the instruction is *ignored*, unless it is one of the special functions to be detailed shortly:
   - RESOLVE
   - ENABLE
   - Communications instruction (SEND, RECV, BROADCAST, REPORT)

3 MIMD Mode of Operation

A PE in MIMD mode of operation (henceforth, a MIMD PE) will:

1. be *logically disconnected* from its parent (Thus, instructions from the broadcast bus will not be accepted.)
2. executes code from its local memory (The tree below the processor remains logically connected and thus, can be utilized as a SIMD tree.)
3. execute the entire Intel 8751 instruction set,
4. executes the SIMD instructions it broadcasts to its descendents,
5. enters SIMD disabled state, after broadcasting an instruction to disable its descendents, when it terminates its MIMD operation.

4 SIMD Instruction Set

We have defined a superset of PL/M [Intel, 1982], which we have come to call PPL/M, which provides a set of facilities to specify operations to be performed by independent PE's in parallel. In this section we first discuss the assignment of special registers to memory, and then detail the additions to PL/M of new data types, new built-in functions and DO blocks for the SIMD operation of the machine.

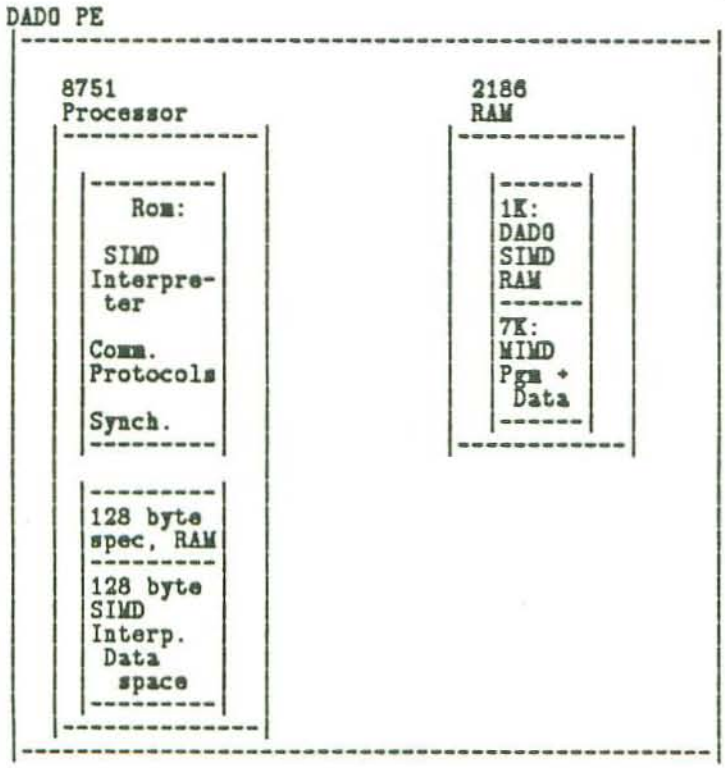
The external 8K RAM (referred to as AUXILIARY space within a PL/M program), is logically divided into a 1K portion, which stores the data space for the SIMD operation of the PE, and a 7K data/program space for storage of MIMD code. The 4K EPROM (referred to as CONSTANT space within a PL/M program) is used for system level code performing the most basic communication and synchronization instructions.

4.1 SIMD RAM

To facilitate the conversion and simulation of *NON-VON* code on *DADO*, the *NON-VON* registers are assumed to be resident within the system. Thus, any PL/M program can reference the *NON-VON* registers without prior need to define them explicitly. The allocation of memory, as depicted in figure 1, is defined as follows:

NON-VON Logical Register	DADO RAM location (relative)
A8	0
B8	1
C8	2
X8	3
Y8	4
Z8	5
IO8	6
MAR	7
A1-Z1,IO1,EN1	8 (bits 0-7)
NV-RAM 64-byte	9-73
DADO Logical Registers	
CPIO Register	74
CPRR Resolve Register	75
DADO-RAM 947-byte	76-1023

Figure 1: The *DADO* memory map



4.2 PPL/M: Parallel PL/M

Before defining the primitives for parallel computation on *DADO*, we begin with a brief introduction to PL/M.

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

- block structure, employing several forms of the PL/I DO statement,
- a full range of data structure facilities 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 manual.) 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. 84 cases are the maximum allowable number. If the select-expression is out of range, disaster will strike (refer to the manual).

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 {MAIN
AUXILIARY
CONSTANT}
[{EXTERNAL
PUBLIC}]

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 AUXILLIARY;
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 untyped procedures are referred to within expressions as a function call.

4.3 Parallel Processing Primitives

The following two syntactic conventions have been added to PL/M for programming the SIMD mode of operation of *DADO*. The design of these constructs was influenced by the methods employed in specifying parallel computation in the GLYPNIR language [Lowrie, et al., 1975] designed for the ILLIAC IV parallel processor. The SLICE attribute defines a variable that is defined to be resident within each PE for which the declaration applies. The second addition is a syntactic construct, the DO SIMD block, which delimits instructions broadcast to SIMD PE's.

The SLICE attribute

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

name: PROCEDURE[(parameter list)] [type] SLICE;
```

Each declaration of a SLICEd variable will cause an allocation of space for the variable within the *DADO* SIMD RAM. SLICEd procedures will be automatically loaded within the MIMD portion of RAM (by an operating system executive resident in DADO's CP). As an example, the following declaration defines the *NON-VON* and *DADO* SIMD RAMs.

```
DECLARE DADO-MEMORY
            STRUCTURE
                  (A8 BYTE,
                   B8 BYTE,
                   C8 BYTE,
                   X8 BYTE,
                   Y8 BYTE,
                   Z8 BYTE,
                   IO8 BYTE,
                   MAR BYTE,
                   A1 BIT,
                   B1 BIT,
                   C1 BIT,
                   X1 BIT,
                   Y1 BIT,
                   Z1 BIT,
                   IO1 BIT,
                   EN1 BIT,
                   NV-RAM(64) BYTE,
                   CPIO BYTE,
                   CPRR BYTE,
                   DADO-RAM(947) BYTE) AUXILIARY
                                       PUBLIC SLICE;
```

An assignment of a value to a SLICEd variable will cause the transfer to occur within each enabled SIMD PE concurrently. A constant appearing in the right hand side will be automatically BROADCAST to all enabled PE's. Thus, the statement

```
X=5;
```

where X is of type BYTE SLICE, will assign the value 5 to each occurrence of X in each SIMD PE. However, statements which operate upon SLICEd variables can only be specified within the bounds of a DO SIMD block.

DO SIMD block

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

The r-statement is restricted to be either

- an assignment statement *incorporating only SLICEd variables and constants* or
- a call to a subroutine that has been declared to be of type SLICE.

A non-SLICEd variable may appear within an r-statement only as an argument to the BROADCAST function to be defined shortly. The parameters of a SLICEd subroutine are assumed to be also of type SLICE by default. Examples of the use of these facilities is provided in the concluding section.

4.4 Added Built-in Communications Primitives

Besides the full range of instructions available in PL/M, a *DADO* PE in MIMD state will have available to it the following list of built-in functions (each defined to be of type SLICE). These have been modelled after the instructions employed in the *NON-VON* supercomputer. For consistency, the *NON-VON* registers are used in precisely the same manner as that defined in the *NON-VON* instruction set.

```
Call RESOLVE;  -- the A1 registers in all PE's except the 'first' PE
                  are set to zero.  The register CPRR in the
                  MIMD PE is set high. If no descendent PE has A1=1,
                  CPRR is set low.

Call REPORT;   -- the contents of A8 in the one enabled descendent PE
                  is written to the register CPIO in the MIMD PE.  If
                  more than one descendent PE is enabled, the result
                  is undefined.

Call BROADCAST(<byte>);
               -- the value of the single byte argument is stored
                  in the IO8 register of every descendent SIMD PE.

Call SEND(<neighbor-PE>);
               -- the contents of register IO8 of <neighbor-PE> is
                  set to the value stored in A8. <neighbor-PE> may be
                  one of:  LC left tree child
                           RC right tree child
                           LN left linear order neighbor
                           RN right linear order neighbor

Call RECV(<neighbor-PE>);
               -- the contents of register A8 of <neighbor-PE> are
                  changed to the value stored in IO8. <neighbor-PE>
                  may be one of: LC, RC, LN, RN and P (parent)

Call MIMD(<address>);
               -- any ENABLED SIMD PE will enter MIMD mode of
                  operation and execute code stored locally in RAM
                  starting at address <address>

Call EXIT;     -- the MIMD PE will terminate its MIMD operation.
                  The PE will issue an instruction to SIMD
                  descendants to disable themselves (set EN1 low)
                  and will reconnect itself to its parent in SIMD
                  disabled state.

Call ENABLE;   -- the EN1 register of all descendent PE's are set
                  high, thus enabling the entire tree.

Call DISABLE;  -- the EN1 register of all descendent PE's are set
                  low, thus disabling the entire tree.
```

The BROADCAST instruction is used to communicate a specified BYTE constant from the MIMD PE or CP to all (enabled) PE's in the tree below. (Note that this constant may in fact be the value of a variable in the CP or MIMD PE.) The REPORT instructions, on the other hand, provide the means for the contents of the

A8 register of a single enabled PE to be communicated to the CP. The REPORT instructions are intended for use only when it is known that at most one PE is currently enabled – for example, immediately following execution of a RESOLVE instruction (discussed below). The effect of a REPORT instruction is not defined in the case where more than one PE is enabled.

The SEND and RECV instructions are used for communication among physically and linearly adjacent PE's – that is, between PE's that are either physically adjacent within the tree, or logically adjacent with respect to the total ordering imposed on the nodes in the linearly adjacent neighbor communication mode. When a PE executes a RECV instruction having either P, LC, RC, LN or RN as its argument, its A8 register takes on the value stored in the IO8 register of the specified (physically or logically) adjacent PE. When a particular PE executes a SEND instruction, on the other hand, the contents of its A8 register is transferred to the IO8 register of the adjacent PE specified as its operand.

Unlike the RECV instructions, however, a PE can not SEND data to its parent, since the semantics of this operation would be undefined if both children of that parent were enabled. Thus, only LC, RC, LN and RN are legal operands for the SEND instruction. It should be noted, however, that the parent is capable of receiving data from its children through the use of RECV LC and RECV RC instructions. The semantics of the SEND and RECV instructions are not immediately apparent in the case where the operand PE is currently disabled. In such cases, it is the recipient's status, and not that of the originator, which determines whether data is in fact transferred. Specifically, it is always possible to RECV data from a PE, regardless of whether it is enabled, but an attempt to SEND data to a disabled PE will not result in a transfer of data.

A PE may be disabled by transferring a 0 into its EN1 register using an ordinary assignment statement in PPL/M. 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 PE's 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 PE's. 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 RESOLVE instruction is used in practice to disable all but a single PE, chosen arbitrarily from among a specified set of PE's. First, the A1 flag is set to one in all PE's to be included in the candidate set. The RESOLVE instruction is then executed, causing all but one of these flags to be changed to zero. (Upon executing a RESOLVE instruction, one of the inputs to the MIMD PE will become

high if at least one candidate was in fact found in the tree, and low if the candidate set was found to be empty. This condition code is stored in the "logical register" CPRR, which exists within the MIMD PE or CP.) By issuing an assignment to EN1, all but the single, chosen PE may be disabled, and a sequence of instructions may be executed on the chosen PE alone. In particular, data from the chosen PE may be communicated to the CP or MIMD PE through a sequence of REPORT commands.

If the candidate set is first saved (using another flag register in each PE), each of the candidates can be chosen in turn, subjected to individual processing, and removed from the candidate set, allowing the sequential processing of all candidates. Typically, the individual processing performed for each chosen candidate involves the broadcasting of information contained in, or derived from, that candidate to other PE's within the *DADO* tree. This paradigm for sequential enumeration is thus employed as a sort of "outer loop" in a number of highly parallel *NON-VON* and *DADO* algorithms.

In the *DADO* prototype, the A1 flag is preserved in that PE which would be assigned the lowest number in an bounded-neighbor enumeration of all nodes in the tree. The RESOLVE function is implemented using special sequential code, embedded within the ROM, that propagates a series of "kill" signals in parallel from all candidate PE's to all higher-numbered PE's in the tree. (As is the case for all of the global communication functions, the RESOLVE operation would be very fast if implemented in combinational logic; thousands of candidates might be "killed" in less than a microsecond in *DADO*, for example. All communication primitives in our current prototype will be implemented with sequential logic. Future research will be devoted to the implementation of a custom VLSI chip for all of the most essential communications on the machine.)

Finally, the MIMD function causes an enabled SIMD PE to begin executing in MIMD mode. The argument address is first broadcast as the base address of the portion of MIMD RAM within which the local subroutine to be executed resides. Return to SIMD mode is performed by the EXIT function when the MIMD PE terminates its computation. Synchronization can be performed with sequential logic to explicitly test whether or not data may be transfered to the MIMD PE. Thus, when such a test indicates that data may be transfered, the MIMD PE has terminated its operation and reconnected itself in SIMD mode. Algorithms for the synchronization of MIMD PE's have been presented elsewhere [Stolfo, 1981].

5 Examples

Code for two fundamental operations are presented in this section: the first loads the *DADO* tree sequentially; the second is used to associatively mark all PE's that match a given search string.

Figure 2: Sequentially Loading *DADO*

```
/* We will assume that this program is executed within
   DADO's CP. The system function READ is used to load
   string data into a buffer from some external source. */

 DO;
  DECLARE Intelligent-record(64) BYTE SLICE;
  DECLARE Not-done BIT SLICE;
  DECLARE Index BYTE SLICE;
  DECLARE Buffer(64) BYTE;
  DECLARE i BYTE;
  DECLARE DADO_MEMORY EXTERNAL;

  DO SIMD;
   Call ENABLE;  /* All PE's are enabled. */
   Not-done = 1; /* All slices initialized. */
   Index = 0;
  END;

  Call READ(Buffer); /* Data provided by some
                        external source. */

  DO WHILE length(Buffer) > 0; /* AND CPRR */

   DO SIMD;
    Call ENABLE;
    A1 = Not-done;
    Call RESOLVE;  /* Only one A1 is now set. */
    EN1 = A1;  /* Selectively disable all but one PE. */
    Not-done = 0;
   END;

   IF NOT CPRR THEN quit; /* No PE's enabled, thus overflow.*/

   DO i = 0 to length(Buffer) - 1 ;

    DO SIMD;
     Call BROADCAST(Buffer(i));
     Intelligent-record(Index) = I08;
     Index = Index + 1;
    END;

   END;

  Call READ(Buffer);

  END;

END;
```

The second example implements the most basic operation for associative matching on *NON-VON* and *DADO*.

Figure 3: Associative Probing

```
DO;

 DECLARE Intelligent-Record(64) BYTE EXTERNAL;
 DECLARE Index BYTE SLICE;
 DECLARE i BYTE;
 DECLARE Search(64) BYTE;

 Call READ(Search);

 Call ENABLE;

 DO i = 0 to length(Search) - 1 ;

  DO SIMD;
   Call BROADCAST(i);
   Index = IO8;  /* This is an alternative method of addressing
                    SLICEd arrays. */
   Call BROADCAST(Search(i));
   EN1 = IO8 = Intelligent-Record(Index); /*Disable those
                                           that do not match.*/
  END;

 END;

 Call RESOLVE;

 IF CPRR THEN /* we have responders! */ ;

END;
```

REFERENCES

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

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

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

Shaw, David Elliot, Salvatore J. Stolfo, Hussein Ibrahim, Bruce Hillyer, Gio Wiederhold and J. A. Andrews, "The NON-VON Database Machine: A Brief Overview", Database Engineering 4(2), 1981.

Shaw, David Elliot, The NON-VON Supercomputer,Technical Report, Department of Computer Science, Columbia University, 1982.

Stolfo, Salvatore J., The DADO USERSGUIDE, DADO Project Report, Department of Computer Science, Columbia University, 1981.

Stolfo, Salvatore J. and David Elliot Shaw, "DADO: A Tree-structured Machine Architecture for Production Systems", Proc. National Conference on Artificial Intelligence, Carnegie-Mellon University and University of Pittsburgh, August, 1982.