The DADO Parallel Computer

Salvatore J. Stolfo
Department of Computer Science
Columbia University
New York City, N. Y. 10027

CUCS-63-83

Abstract

DADO is a parallel, tree-structured machine designed to provide significant performance improvements in the execution of large production systems. A full-scale production version of the DADO machine would comprise a large (on the order of a hundred thousand) set of processing elements (PE's), each containing its own processor, a small amount (8K bytes, in the current prototype design) of local random access memory, and a specialized I/O switch. The PE's are interconnected to form a complete binary tree.

This paper describes the organization of, and programming language for two prototypes of the DADO system. We also detail a general procedure for the parallel execution of production systems on the DADO machine and outline how this procedure can be extended to include commutative and multiple, independent production systems. We then compare this with the RETE matching algorithm, and indicate how PROLOG programs may be implemented directly on DADO.

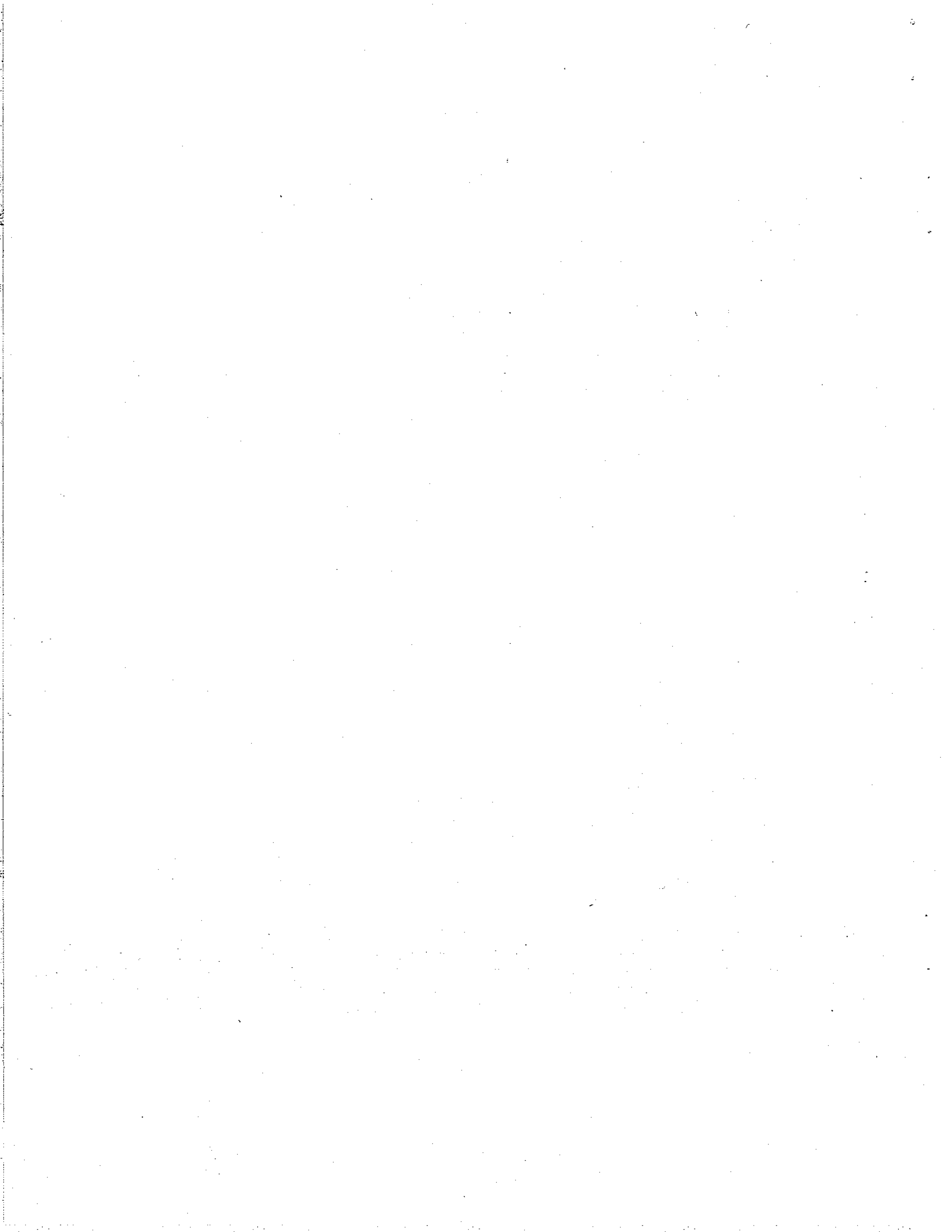


Table of Contents

1 Introduction 1
1.1 Production Systems 1
1.2 Goal of the Research 1
2 The DADO Machine Architecture 3
2.1 The Binary Tree Topology 3
3 The DADO Prototypes 7
3.1 The Prototype Processing Element 8
3.2 The PE kernel 9
3.2.1 SIMD Mode of Operation 9
3.2.2 MIMD Mode of Operation 10
4 Programming DADO 10
4.1 Conventional PL/M 11
4.2 Parallel Processing Primitives: PPL/M 11
4.3 MIMD Mode Primitives 12
4.4 Examples 14
5 The Production System Algorithm 16
5.1 Allocation of Productions and Working Memory 17
5.2 The Matching Phase 18
5.3 The Selection Phase 19
5.4 The Action Phase 19
5.5 Specialized Production Systems 20
5.6 Discussion 20
5.6.1 Compiling patterns 21
5.6.2 Data Elements may contain variables 21
5.6.3 Temporal Redundancy 21
5.6.4 WM-subtree overflow 22
5.6.5 Duplicate WM elements 22
6 Future Research 22
7 Conclusion 23

List of Figures

Figure 1: An Example Production. 2
Figure 2: Interconnection of two Leiserson Chips. 4
Figure 3: The Leiserson Printed Circuit Board. 5
Figure 4: Hyper-H embedding of a binary tree. 6
Figure 5: The DADO Prototype Processing Element. 9
Figure 6: Loading DADO sequentially. 15
Figure 7: Associative Probing: using DADO as a content-addressable memory. 16
Figure 8: Functional Division of the DADO tree. 17

1 Introduction

As knowledge-based systems grow in size and scope, they will begin to push conventional computing systems to their limits of operation. Even for experimental systems, many researchers reportedly experience frustration based on the length of time required for their operation. For applications requiring real-time response from an expert system (for example, electronic warfare or autonomous robot control systems) conventional implementations may not be practical.

DADO [Stolfo et. al. 1982, Stolfo and Shaw 1982] is a parallel, tree-structured machine designed to provide highly significant performance improvements in the execution of very large *production systems*. Production systems form the basis for a wide range of approaches to the implementation of knowledge-based software. A number of working systems implemented by researchers in the field of Artificial Intelligence (AI) have demonstrated the considerable utility of rule-based representation schemes applied to a number of significant tasks requiring extensive domain expertise. Medical diagnosis [Davis 1976], the identification of unknown chemical compounds [Buchanan and Feigenbaum 1978], mineral exploration [Duda et. al. 1979] and telephone cable maintenance [Vesonder et. al. 1983] are just a few examples. As has been reported by several researchers, rule-based systems appear well-suited to the aquisition of knowledge from human experts, and are easily implemented and readily modified and extended.

1.1 Production Systems

A *production system* [Newell, 1973; Davis and King 1975; Rychener, 1976] is defined by a set of rules, or *productions*, which form the *production memory* (PM), together with a database of assertions, called the *working memory* (WM). Each production consists of a conjunction of *pattern elements*, called the *left-hand side* (LHS) of the rule, along with a set of actions called the *right-hand side* (RHS). The RHS specifies information which is to be added to (asserted) or removed from WM when the LHS successfully matches against the contents of WM.

In operation, the PS repeatedly executes the following cycle of operations:

1. *Match:* For each rule, determine whether the LHS matches the current environment of WM.

2. *Select:* Choose exactly one of the matching rules according to some predefined criterion.

3. *Act:* Add to or delete from WM all assertions specified in the RHS of the selected rule.

For pedagogical reasons, we will initially restrict our attention to the case in which both the LHS and RHS are conjunctions of predicates in which all first order terms are composed of constants and existentially quantified variables. Data elements in WM will have the form of arbitrary ground literals in the first order predicate calculus. (When PROLOG is considered in a later section, we briefly describe how WM elements may contain general first-order terms.) A negated pattern in the LHS causes the matching procedure to fail whenever WM contains a matching ground literal, while a negated pattern in the RHS causes all matching data elements in the WM to be deleted.

An example production is presented in figure 1. (Variables are prefixed with an equal sign.)

1.2 Goal of the Research

In practical applications of the sort anticipated by most researchers in the field of AI, the set of productions (and hence the set of LHS patterns against which WM must be matched on each cycle) are expected to typically be quite large. In the case of the R1/XCON program [McDermott 1981], for example, roughly 2400 specialized productions presently exist to configure a Digital Equipment Corporation VAX computing system. To fulfill their promise for the very-large-scale embodiment of

Figure 1: An Example Production.

```
(part-category =part electronic-component)
(used-in =part =product)
(Supplied-to =product =customer)
(NOT Manufactured-by =part =customer)
 --> (Dependent-on =customer =part)
  (NOT Independent =customer)
```

domain-specific expertise, production systems are likely to require at least an order of magnitude more rules, making the question of efficiency a potentially critical concern.

Because the matching of each rule against WM is essentially independent of the others (at least in the absence of contention for data in WM), it is natural to attempt a decomposition of the matching portion of each cycle into a large number of tasks suitable for physically concurrent execution on parallel hardware. While this task is in fact considerably more complicated than it might first appear, we believe the immense potential value of a powerful and highly general *production system machine* warrants serious attention by parallel machine architects and VLSI designers.

Thus, simply stated, the goal of the DADO machine project is the design and implementation of a (cost effective) high performance *rule processor* capable of rapidly executing a production system cycle for very large rule bases (ideally in an amount of time independent of the number of rules). Our goals do not include the design of a high-speed parallel processor capable of (a fruitless) parallel search through a combinatorial solution space.

Much of the experimental research conducted to date on specialized hardware for AI applications has focussed on the realization of high-performance, cleverly designed, but for the most part, architecturally conventional machines. (MIT's LISP Machine exemplifies this approach.) Such machines, while quite possibly of great practical interest to the research community, make no attempt to employ hardware parallelism on the massive scale characteristic of our own work.

Recently, several AI researchers (see [Nilsson 1980], for example) have suggested that significant increases in the performance of contemporary AI systems might be realized through distributed processing or the use of specialized parallel hardware. Some attention has been given to issues of parallelism in system organizations for cooperating distributed AI subsystems [Lesser and Erman 1979, Lesser and Corkill 1979]; special hardware for high speed property inheritance and related operations in systems based on semantic network-like formalisms [Fahlman 1979, Hillis 1982]; and the design of machines supporting the parallel execution of certain relational algebraic operations having practical importance in large-scale knowledge-based systems [Shaw et. al. 1981; Bonuccelli et. al. 1983]. The potential applications of very large scale hardware parallelism to the execution of rule-based systems, however, has remained largely unexplored.

In this paper, we describe DADO, a tree-structured, multi-processor based architecture that utilizes the emerging technology of VLSI systems in support of the highly efficient parallel execution of large-scale production systems. Our research has convinced us that DADO may support many other AI applications including the very rapid execution of PROLOG programs and a large share of the symbolic processing typical of knowledge-based systems.

A small (15 processor) prototype of the machine, constructed at Columbia University from components supplied by Intel Corporation, is operational. Based on our experiences with constructing this small prototype, we believe a larger DADO prototype, comprising 1023 processors, to be technically and economically feasible for implementation using current technology. We believe that this larger experimental device will provide us with the vehicle for evaluating the performance, as well as the hardware design, of a full-scale version of DADO implemented entirely with custom VLSI circuits.

2 The DADO Machine Architecture

DADO is a fine-grain, parallel machine where processing and memory are extensively intermingled. A full-scale production version of the DADO machine would comprise a very large (on the order of a hundred thousand) set of *processing elements* (PE's), each containing its own processor, a small amount (8K bytes, in the current design of the prototype version) of local random access memory (RAM), and a specialized I/O switch. 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 1972]), 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 single conventional coprocessor, 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 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 (on different data) at a given point in time. This flexible architectural design supports *multiple-SIMD* execution (MSIMD). Thus, the machine may be logically divided into distinct partitions, each executing a distinct task, and is the primary source of DADO's speed in executing a large number of primitive pattern matching operations concurrently, to be detailed shortly.

The DADO I/O switch, which will be implemented in custom VLSI and incorporated within the 1023 processing element version of the machine, has been designed to support communication between physically adjacent tree neighbors. In addition, a specialized combinational circuit incorporated within the I/O switch will allow for the very rapid selection of a single distinguished PE from a set of candidate PE's in the tree. Currently, the 15 processing element version of DADO performs these operations in firmware embodied in its off-the-shelf components.

In the following sections we outline the reasons for implementing a binary tree organization. We then discuss the precise semantics of both execution modes of a DADO PE, and the methods employed to simulate each in the current DADO prototype design. Subsequently we define PPL/M, a variant of the PL/M language, providing several primitives for specifying parallel computation on DADO. The basic DADO algorithms for production system execution are then described and evaluated.

2.1 The Binary Tree Topology

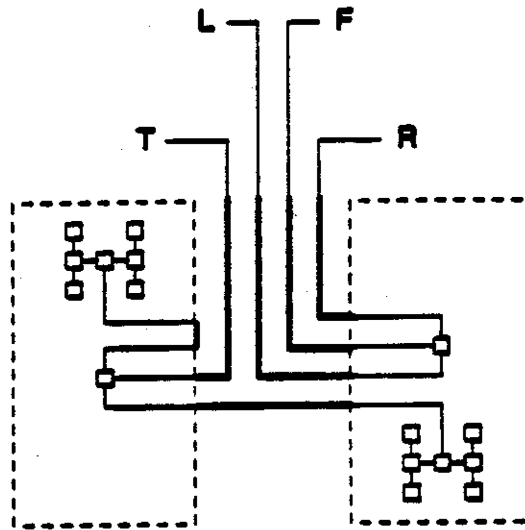
As VLSI technology continues its downward trend in scaling, many PE's may be implemented on a single silicon chip. If the minimum feature size is halved, for example, four times as many components can be placed on a single chip. Thus, future microcomputer technology may provide additional speed, function and storage capacity of a single PE on a chip. Alternatively, as is the case with many of the approaches to fine-grain parallelism, many simpler processors may be integrated on the same chip. It is crucial, therefore, to interconnect a large number of processors in the most area-efficient topology possible. Further consideration must also be given to methods which efficiently drive the large number of device components to be placed on the chip, and which are not restricted by the severe pin-out limitations of packaging technology.

In our initial work, several alternative parallel machine architectures were studied to determine a suitable organization of a special-purpose production system machine. High-speed algorithms for the parallel execution of production system programs were developed for the perfect shuffle [Schwartz 1980] and binary tree machine architectures [Browning 1978]. Forgy [1980] proposed an interesting use of the mesh-connected ILLIAC IV machine [Lowrie et. al. 1975] for the parallel execution of production systems, but recognized that his approach failed to find all matching rules in certain circumstances. Of these architectures, the binary tree organization was chosen for reasons of efficient implementation in VLSI technology.

First we note that the entire binary tree of PE's can be implemented using a number of identical chips. This design, first reported by Leiserson [1981], embeds both a complete subtree of PE's and a single interior node on each chip (see figure 2). Four data ports enter the chip. One, called the T port, connects to the root of the chip's subtree, while the other three ports, called F, L and R, connects the single interior PE node to its father, left child and right child, respectively.

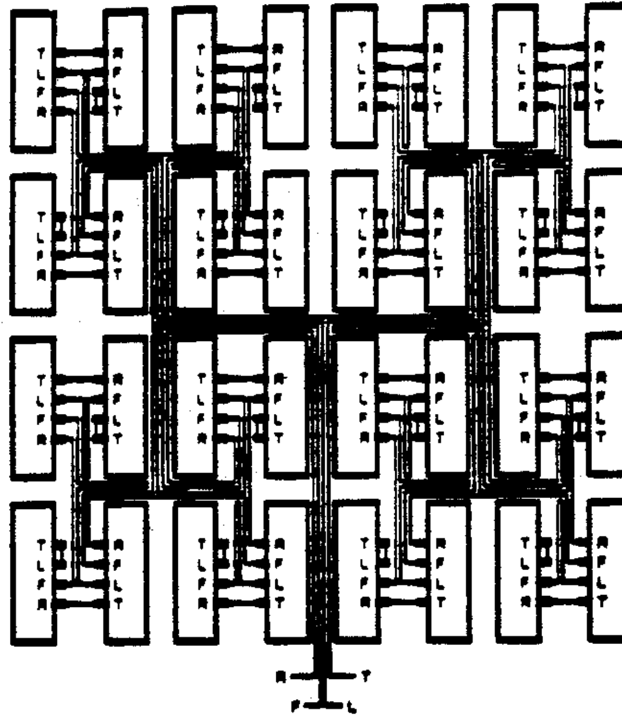
A simple recursive procedure allows the construction of an arbitrarily large binary tree using only chips of this type. Figure 2 illustrates this construction for two chips. Note that the resulting circuit consists of a larger binary tree, together with a single unconnected interior node. This scheme may be extended to allow the construction of a planar printed-circuit board layout (also due to Leiserson), which is illustrated in figure 3. Note that the area required for routing wires within the PC board is strictly proportional to the number of chips, allowing the efficient implementation of boards of arbitrary size.

Figure 2: Interconnection of two Leiserson Chips.



The subtree of PE's incorporated within each chip is configured according to the "hyper-H" embedding as first reported by Browning [1980] (see figure 4). This construction is highly regular, is area-optimal (in that the amount of silicon area is proportional to the number of PE's) and is easily extended to incorporate larger numbers of PE's as device dimensions scale downward. (Note that the number of ports

Figure 3: The Leiserson Printed Circuit Board.

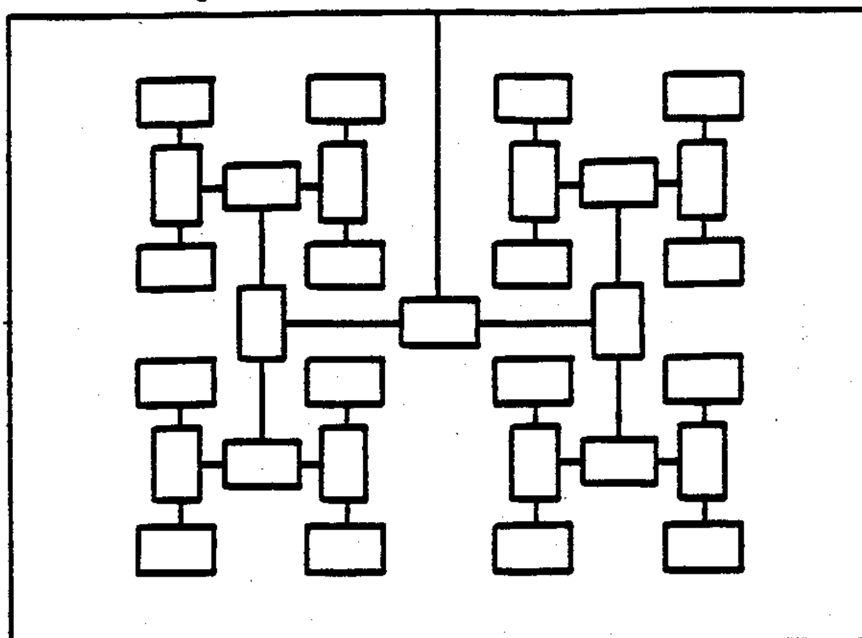


entering the Leiserson chip remains constant as larger numbers of PE's are incorporated.) It should be stressed that this architectural design is not only highly efficient from a theoretical perspective, but inexpensive to implement (and replicate) in a working device.

Although binary trees implemented in this fashion may seem to scale indefinitely, an interesting theoretical result reported in [Patterson et. al. 1981] suggests that synchronous binary tree systems with massive numbers of PE's may not be practical due to the asymptotically growing lengths of wires in the tree. This "wire length problem" may introduce severe clock skew as well as other anomalous electrical problems. Our statistics suggest that for the DADO system this is not a severe limitation since the number of PE's in a full-scale version would be limited to a manageable number. (See [Fisher and Kung 1982] for a discussion of this problem in terms of clock skew for synchronous systems and possible solutions.)

Other factors contributed to our choice of designing DADO as a binary tree machine. The most important of these factors is the requirement of broadcasting data to a very large number of processors. The contemporary models of VLSI computation dictate that global communication requires an amount of time no less than the logarithm of the number of recipients. It should be noted that no architecture based on components having a *bounded valence* (that is, a fixed maximum number of external connections) can perform this function in less than logarithmic time. While this lower bound is achieved by tree-structured machines and by certain other "area-expensive" organizations, like the perfect-shuffle, other topologies do not share this property. For example, linear arrays require an amount of time for broadcast that is proportional to the number of recipients, while mesh-connected devices require time that is proportional to the square root of the number of recipients. In such systems consisting of many thousands of PE's, a significant delay is unfortunately introduced, thus requiring complex pipelined communication schemes suggesting an asynchronous design.

Figure 4: Hyper-H embedding of a binary tree.



Finally, we note that binary trees do have certain limitations of practical importance. Although broadcasting a small amount of information to a large number of recipients is efficiently handled by binary trees, the converse is, in general, unfortunately not true. That is, for certain computational tasks (permutation of data, for example) the effective bandwidth of communication is restricted by the top of the tree. It is easy to prove that if n items stored at the leaves of a binary tree are to be randomly permuted, the number of such items which must pass through the root of the tree is proportional to n. Fortunately, as we shall see shortly, this "binary tree bottleneck" does not arise in the execution of production systems or PROLOG.

Many of the decisions made in designing DADO were strongly influenced by the organization of the NON-VON supercomputer [Shaw 1982] and the Caltech tree machine [Browning 1978]. Perhaps the best way to distinguish DADO from these two tree machine architectures is by considering the modes of execution of each of the constituent PE's, and the implications for the hardware design.

The proposed Caltech tree machine is a full MIMD device incorporating thousands of PE's in a full-scale version. Each PE executes its own independent program and thus requires a substantial amount of local memory as is the case in the DADO machine. Communication is supported by a buffered message passing protocol, where the recepient of each message is identified by relatively complex I/O circuitry at each node, whereas other forms of communication (for example, global broadcast) are implemented by sequential logic.

NON-VON, by comparison, is a full SIMD, massively-parallel synchronous device incorporating millions of simple, *highly-area efficient* PE's, each associated with only 64 bytes of local RAM. In general, each NON-VON PE executes an instruction broadcast from a single control processor, located at the root of the tree, and thus requires a highly-efficient method of global broadcast. The I/O switch incorporated within each node of the NON-VON tree contains a few inverters driving the signals along the broadcast bus, and therefore communication is implemented by high-speed combinational logic.

DADO, on the other hand, is capable of executing in both SIMD and MIMD modes, and thus contains elements of both machine designs. DADO incorporates a combinational I/O switch similar to that

employed in NON-VON. However, each DADO PE may drive the I/O switch, in addition to the single coprocessor of DADO. Thus, DADO also supports very high speed global broadcast. However, because of the replication of substantial programs within various PE's in the tree, a DADO PE has been designed with a more general (8 bit) processor as well as an 8K byte RAM. Thus, DADO cannot achieve the same processor density as is possible in NON-VON.

The DADO design attempts to synergistically merge the advantages of both the NON-VON and the Caltech tree machine. It is not clear whether or not the NON-VON approach to single-instruction stream, massive parallelism will be substantially limited by its inability to execute independent programs concurrently. Nor is it clear whether or not the Caltech approach of large-scale parallelism, albeit substantially lower than that of NON-VON for certain computational problems, can achieve the same throughput capable of NON-VON. It is our hope that experimentation with the DADO prototype may provide some of these answers, and begin to elucidate the precise nature of the tradeoffs involved with both approaches.

3 The DADO Prototypes

A 15-element *DADO1* prototype, constructed from (partially) donated parts supplied by Intel Corporation, has been operational since April 25, 1983. The two wire-wrap board system, housed in a chassis measuring 3.5 by 18.5 by 17.5 inches volume (roughly the size of an IBM PC), is clocked at 3.5 magahertz producing 4 million instructions per second (MIPS). (The effective useable MIPS is considerably less due to the significant overhead incurred in interprocessor communication. For each byte quantity communicated through the system, 12 machine instructions are consumed at each level in the tree while executing an asynchronous, 4-cycle handshake protocol.) DADO1 contains 124K bytes of user random access storage and 60K bytes of read only memory. A much larger version, *DADO2*, is currently under construction which will incorporate 1023 PE's constructed from two commercially available Intel chips. DADO1 does not provide enormous computational resources. Rather, it is viewed as the development system for the software base of DADO2, and is not expected to demonstrate a significant improvement in the speed of execution of a production system application.

DADO2 will be implemented on 16 printed circuit boards, manufactured through the DARPA supported MOSIS silicon foundry system, and housed in an IBM Series I cabinet (donated by IBM Corporation). The system, which will be integrated within a standard 19 inch rack, provides 8 megabytes of user storage. A DEC VAX 11/750 (partially donated by DEC Corporation) serves as DADO2's coprocessor (although an Apollo or SUN workstation may be used as well) and is the only device a user of DADO2 will see. Thus, DADO2 is considered a transparent back-end processor to the VAX 11/750. The DADO2 system will have roughly the same hardware complexity as a DEC VAX 11/750 system, and if amortized over 12 units will cost in the range of 70 to 90 thousand dollars to construct considering 1983 market retail costs. The DADO2 custom I/O chip is planned for implementation in gate array technology and will allow DADO2 to be clocked at 12 megahertz, the full speed of the Intel chips. The effective machine instruction cycle time achievable is 1.8 microseconds, producing a system with a raw computational throughput of 570 million instructions per second. Note that little of this computational resource is wasted in communication overhead, as in the DADO1 machine, since asynchronous communication is replaced with a synchronous combinational logic circuit.

In the following sections we detail the prototype processing element design as well as the software systems implemented for the prototypes.

3.1 The Prototype Processing Element

Each PE in the DADO1 prototype system incorporates an Intel 8751 microcomputer chip, serving as the processor, and an 8K X 8 Intel 2186 RAM chip, serving as the local memory. (A simple logic gate packaged in a Texas Instrument TI-7408 chip is used to properly integrate the RAM and processor.) DADO2 will incorporate a slightly modified PE. The Intel 2187, which is fully compatible with but faster than an Intel 2186, replaces the DADO1 RAM chip allowing the processor to be clocked at its fastest speed. Further, the custom I/O chip will contain extra circuitry to quickly refresh the Intel 2187, and thus replaces the TI chip employed in DADO1. The resulting system consists of a 3 chip PE, 64 of which may be integrated on a single printed circuit board.

Although the original version of DADO had been designed to incorporate a 2K byte RAM within each PE, an 8K byte RAM was chosen for the prototype PE to allow a modest degree of flexibility in designing and implementing the software base for the full version of the machine. In addition, this extra "breathing room" within each PE allows for experimentation with various special operations that may be incorporated in the full version of the machine in combinational circuitry, as well as affording the opportunity to critically evaluate other proposed (tree-structured) parallel architectures through software simulation.

(It is worth noting though that the proper choice of "grain size" is an interesting open question. That is, through experimental evaluation we hope to determine the size of RAM for each PE, chosen against the number of such elements for a fixed hardware complexity, appropriate for the widest range of production system applications. Thus, future versions of DADO may consist of a number of PE's each containing an amount of RAM significantly larger or smaller than implemented in the current prototype systems.)

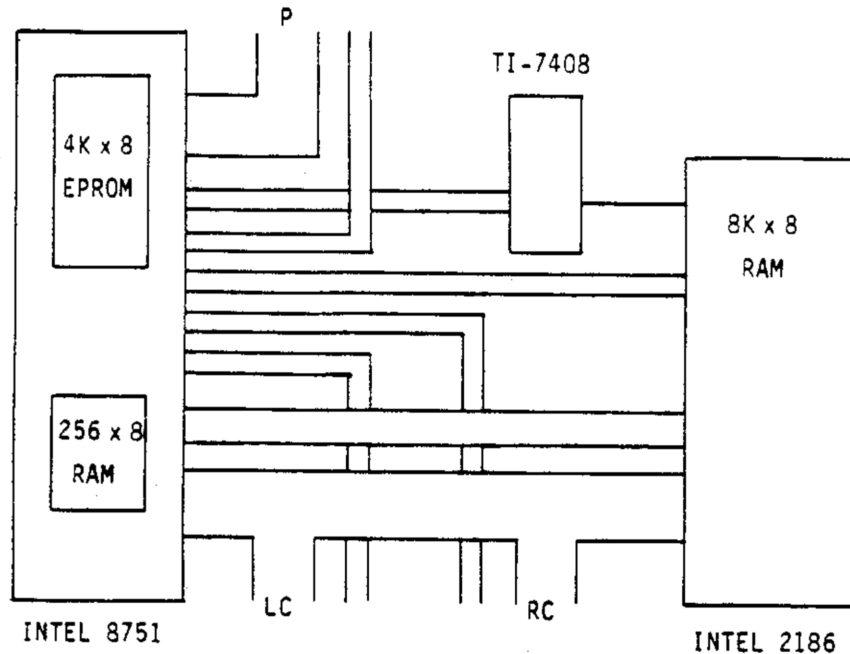
The Intel 8751 is a moderately powerful 8-bit microcomputer incorporating a 4K eraseable programmable read only memory (EPROM), and a 256-byte RAM on a single silicon chip. One of the key characteristics of the 8751 processor is its I/O capability. The 4 parallel, 8-bit ports provided in a 40 pin package has contributed substantially to the ease of implementing a binary tree interconnection between processors. Indeed, DADO1 was implemented within 6 months of delivery of the hardware components. Figure 5 illustrates the DADO1 prototype PE at about twice actual dimensions.

In DADO1 the communication primitives and execution modes of a DADO PE are implemented by a small *kernel system* resident within each processor EPROM. The specialized I/O switch envisaged for the larger version of the machine is simulated in the smaller version by a short sequential computation. As noted, the 1023 element prototype would be capable of executing in excess of 570 MIPS (on 8 bit data), assuming inter-processor communication to be implemented with a combinational logic I/O switch. Although pipelined communication is employed in the kernel design, it is expected that only 150 million instructions per second would be achieved using the current design. Thus, the design and implementation of a custom I/O chip forms a major part of our current hardware research activities.

It should be noted that, in keeping with our principles of "low-cost performance," we have selected a processor technology one generation behind existing available microcomputer technology. For example, DADO2 could have been designed with 1023 Motorola 68000 processors or Intel 80286 chips. Instead, we have chosen a relatively slow technology to limit the number of chips for each PE, as well as to demonstrate our most important architectural principals in a cost effective manner.

Furthermore, since the Intel 8751 does not press current VLSI technology to its limits, it is surely within the realm of feasibility to implement a DADO2 PE on a single silicon chip. Thus, although DADO2 may appear impressive (an inexpensive, compact system with a thousand computers executing roughly 600 million insructions per second) its design is very conservative and probably at least an order of magnitude less powerful than a similar device using faster technology. It is our conjecture though that the machine will be practical and useful and many of its limitations will be ameliorated as VLSI continues its downward trend in scaling. (DADO3 may serve to prove this conjecture.)

Figure 5: The DADO Prototype Processing Element.



3.2 The PE kernel

As noted, the 4K EPROM of the Intel 8751 stores the system kernel of a PE, which includes code performing the most basic communication and synchronization functions as well as the simulation of SIMD and MIMD modes of execution. Presently, the kernel software occupies less than 1K bytes of EPROM. Thus, many frequently used procedures (pattern matching, for example) are planned for implementation in DADO2's EPROM.

The kernel system is designed in such a way as to logically divide the 8K RAM space of the Intel 2186 chip into two portions for each of the execution modes. The size of these two portions is specified by the software declarations. By convention the initial portion of RAM, referred to as *SIMD RAM*, is a reserved data space for variables and constants operated upon by a PE while in SIMD mode. The remaining portion of RAM is used for storage of code, as well as the local variables used during the MIMD mode of operation. (The 8 megabyte memory provided in DADO2 is fully distributed in 8K quantities. A us program written for DADO is, thus, limited to an 8K address space.) A set of reserved memory locations within the SIMD RAM have special significance to the kernel system. In the following sections we define each of these reserved locations when appropriate and briefly describe their use.

3.2.1 SIMD Mode of Operation

A processor in SIMD mode (henceforth, a ***SIMD PE***) can be instructed to enter one of two states, as determined by the contents of a special single bit variable, called EN1, resident within the SIMD RAM of the PE. 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. The kernel simulates the SIMD mode of operation in the following way.

SIMD ENABLED state

A DADO PE in SIMD enabled state will repeat the following steps:

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).

3. Execute the instruction.

SIMD DISABLED state

A DADO PE in SIMD disabled state will repeat the following steps:

1. Accept an instruction from the broadcast bus.

2. As in the enabled case, it will pass the instruction on to its descendants if they are logically connected, however ...

3. The instruction is *ignored* unless it is one of the following special functions, to be detailed shortly:

   - RESOLVE

   - ENABLE

   - a communications instruction (SEND, RECV, BROADCAST or REPORT)

3.2.2 MIMD Mode of Operation

Likewise, the kernel system simulates the MIMD mode of operation (henceforth, a *MIMD PE*) in the following way:

1. A MIMD PE is ***logically disconnected*** from its parent. (Thus, instructions from the broadcast bus will not be accepted.)

2. The PE executes code from its local memory (Unless otherwise instructed, the tree below the processor remains logically connected and thus, can be utilized as a SIMD processor.)

3. Upon terminating its MIMD operation, it enters SIMD disabled state, after broadcasting an instruction to disable its descendants.

4 Programming DADO

PL/M [Intel 1982] 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. A superset of PL/M, which we call PPL/M, has been implemented as the *system-level language* for the DADO prototypes. PPL/M provides a set of facilities to specify operations to be performed by independent PE's in parallel. In this section we discuss the additions to PL/M of new data types, built-in functions and syntactic conventions for the parallel execution of DADO programs.

4.1 Conventional PL/M

Before defining the primitives for parallel computation on DADO, we begin with a brief overview of PL/M. Intel's PL/M language is based on:

- a statement-oriented syntactic structure based largely on *PL/I*,

- a full range of supported statements typical of a high-level language including assignment, nested if, case, and several forms of iteration (while, and auto-increment),

- block structure, employing several forms of the *PL/I* DO statement,

- a full range of data type facilities including arrays, structures and *pointer-based dynamic variables*, as well as subroutine and function definition statements,

- and lastly, 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. Explicit data definition and typing is specified primarily with the DECLARE statement.

4.2 Parallel Processing Primitives: PPL/M

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 variables and procedures that are resident within each PE. The second addition is a syntactic construct, the *DO SIMD block*, which delimits PPL/M instructions broadcast to descendent SIMD PE's. (In the following definitions, optional syntactic constructs are represented within meta brackets.)

The SLICE attribute:

DECLARE variable[(single-array-dimension)] type SLI(;

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

Each declaration of a SLICEd variable will cause an allocation of space for the variable to occur within the SIMD RAM of each PE. SLICEd procedures are automatically loaded within the MIMD portion of RAM (by an operating system executive resident in DADO's coprocessor).

Within a PPL/M program, 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 enabled SIMD PE. (Thus, at times it is convenient to think of SLICEd variables as vectors which may be operated upon, in whole or in part, in parallel.) 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-statement0;
  .
  .
  .
 r-statementn;
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 (user defined SLICEd procedures may not execute any MIMD mode primitives), or

- a call to a local user defined procedure, by way of the *MIMD* function (to be detailed shortly).

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 of type SLICE by default. Examples of the use of these features are provided in a later section.

4.3 MIMD Mode Primitives

In addition to the full range of instructions available in PPL/M, a DADO PE in MIMD mode will have available to it the following list of built-in functions. (It should be noted that DADO's coprocessor may execute the full range of PPL/M instructions as well.) These functions have been modelled after the machine instructions employed in the NON-VON supercomputer as reported in [Shaw 1982]. 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 SLICEd variable A1, resident in all PE's,
is set to zero except in the "first" PE.
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 A8 variable 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

Call RECV(<neighbor-PE>);
-- the contents of register A8 is set to the value stored in IO8 of <neighbor-PE>. <neighbor-PE> may be one of: LC, RC, 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 variable of all descendent PE's are set high, thus enabling the entire subtree.

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

The BROADCAST function is used to communicate a specified BYTE value from a MIMD PE or DADO's coprocessor to all (enabled) PE's in the subtree it roots. The REPORT instruction, on the other hand, provides the means for the contents of a variable of a single enabled PE to be communicated to the MIMD PE. As a side effect, the DADO2 I/O chip provides the means for the byte quantity to be simultaneously BROADCAST to all (connected) processors within the tree. The REPORT instruction is intended for use only when it is known that at most one PE is currently enabled, for example, after use of a RESOLVE instruction detailed below.

The SEND and RECV instructions are used for communication among physically adjacent PE's. Two special SLICEd variables of type BYTE, called A8 and IO8, take part in the data transfer operation. Unlike the RECV instructions, a PE can not SEND data to its parent, since this operation would be undefined if both children of that parent were enabled. A parent is capable of receiving data from its children through the use of RECV instructions. It should be noted that 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. In the case of a disconnected or nonexistent PE all I/O operations return a value of 0.

A PE may be disabled by transferring a 0 into its EN1 variable using an ordinary assignment statement in PPL/M, or by use of the DISABLE function. 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.

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, also a SLICEd variable, 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 found in the tree, and low if the candidate set was found to be empty. This condition code is stored in the SLICEd variable CPRR, which exists within the MIMD PE.) 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 MIMD PE through a sequence of REPORT commands.

If the candidate set is first saved (using another flag 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 employed as a sort of "outer loop" in a number of parallel DADO algorithms.

In DADO1, the RESOLVE function is implemented using special sequential code, embedded within the EPROM, that propagates a series of "kill" signals in parallel from all candidate PE's to all (higher-numbered) PE's in the tree. In DADO2, the RESOLVE operation has been generalized to operate on 8-bit data, producing the *maximum* value stored in some candidate PE. Repeated use of this max-RESOLVE function allows for the very rapid selection of multiple byte data. This circuit has proven very useful for a number of DADO algorithms which made use of the SEND and RECV instructions primarily for ordering data within the tree. The use of the high-speed max-RESOLVE often obviates the need for such communication instructions. Consequently, the view of DADO as a binary tree architecture has become, fortuitously, nearly transparent. The 1 bit RESOLVE implemented in DADO1 is exhibited in the examples which follow.

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 local user defined procedure to be executed. (PPL/M provides a very simple and direct means for specifying the address of an object within a program, including the base address of a subroutine.) 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 to the tree above in SIMD mode by way of the EXIT function. Several algorithms for the synchronization of MIMD PE's within DADO have been reported elsewhere [Stolfo 1981].)

4.4 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 6: Loading DADO sequentially.

```
/* 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; /* An instance */
  DECLARE Not-done BIT SLICE;  /* of each of these SLICE    */
  DECLARE Index BYTE SLICE;    /* variables apprears in each PE */
  DECLARE Buffer(64) BYTE;
  DECLARE i BYTE;

  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;   /* All PE's not yet loaded with data */

    A1 = Not-done; /* have A1 set high.                 */

    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));  /* The single enabled PE */

     Intelligent-record(Index) = A8; /* will execute this*/

     Iddex = Index + 1;              /* code alone.      */

    END;

   END;

  Call READ(Buffer);

  END;  /* Repeat for other PE's in the DADO tree. */

END;
```

The second example implements the most basic operation for associative matching on DADO. This procedure was the first PPL/M program executed on the DADO1 system.

Figure 7: Associative Probing: using DADO as a content-addressable memory.

```
ASSOCIATIVE-PROBE: PROCEDURE (Search);

 DECLARE Intelligent-Record(64) BYTE SLICE; /* An instance */
 DECLARE Index BYTE SLICE;  /* of each appears in every PE.*/

/*We assume each of the instances of Intelligent-Record
  have been previously loaded within the DADO tree.*/

 DECLARE i BYTE;    /* i is local to this routine.*/
 DECLARE Search(64) BYTE; /*The search string is provided by
                           some external source. */

 DO SIMD;

  Call ENABLE; /*All descendent PE's enter SIMD enabled state.*/

  A1 = 0;  /* All A1 flags within the tree below are cleared. */

 END;

 DO i = 1 to length(Search) ; /*Repeat the following for
                          each character of the search string.*/

  DO SIMD;
   Call BROADCAST(i); /*The value of the index variable i is
                       broadcast and loaded in each SLICEd
                       A8 variable within the tree.        */
   Index = A8;        /*and transfered to a local SLICEd
                       variable.*/

   Call BROADCAST(Search(i)); /*The ith character of the
                       search string is then broadcast and
                       loaded in each A8 register.*/
   EN1 = A8 = Intelligent-Record(Index);
                      /*After comparing the search
                        character, currently stored in A8,
                        with the locally stored data,
                        disable those PE's which do not
                        match (by assignment of logical 0
                        to the enable variable EN1).     */
  END;

 END;

 DO SIMD;
  A1 = 1; /* Only those PE's that remain enabled, that is
           only those which match the search string, will
           set their A1 variables high. */

  Call RESOLVE;  /*Lastly, we test for whether or not
                  any A1 flags in the tree are high.
                  CPRR is set accordingly. */
 END;

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

END ASSOCIATIVE-PROBE;
```

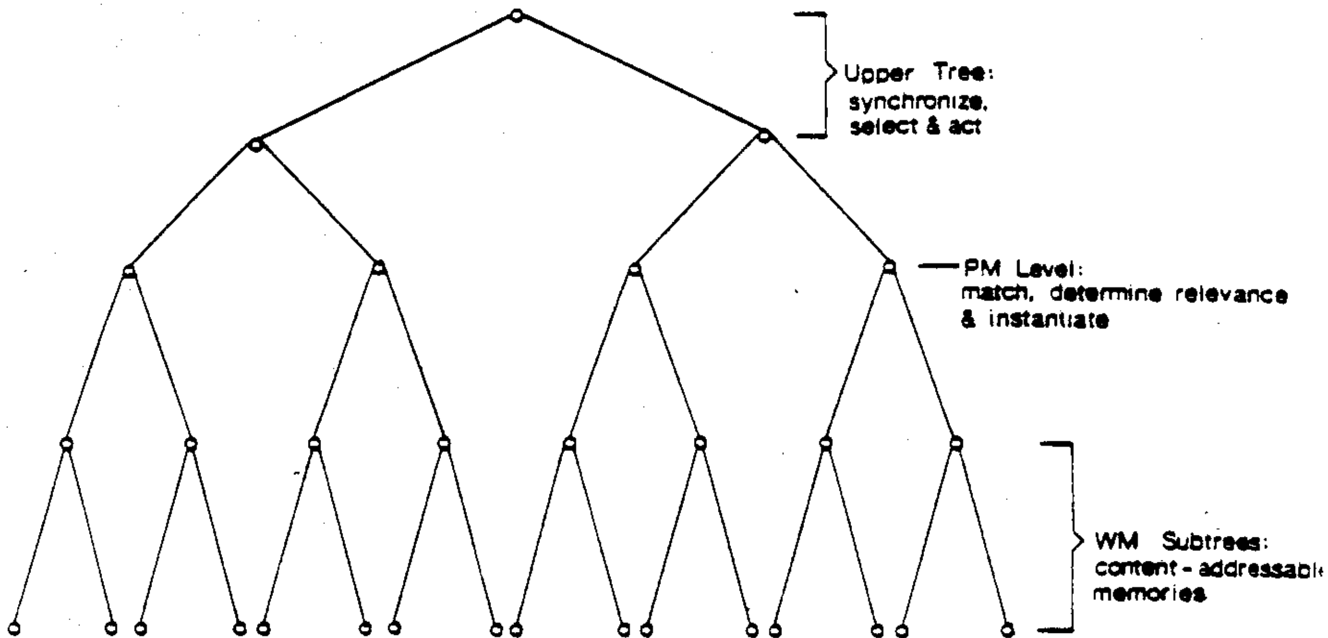
5 The Production System Algorithm

The general production system algorithm implemented on DADO is presented in this section. Following this we compare the algorithm with RETE-based matching systems and outline several ways the algorithm may be modified to adequately treat various anomalous situations.

5.1 Allocation of Productions and Working Memory

In order to execute the production system cycle on the DADO machine, the I/O switches are configured in such a way as to divide the DADO machine into three conceptually distinct components. One of these components consists of all PE's at a particular level within the tree, called the *PM-level*, which is chosen in a manner to be detailed shortly. The other two components are the *upper portion* of the tree, which comprises all PE's located above the PM-level, and the *lower portion* of the tree, which consists of all PE's found below the PM-level. This functional division is illustrated in figure 8.

Figure 8: Functional Division of the DADO tree.



Each PE at the PM-level is used to store a single production (although this restriction is easily relaxed at the expense a modest cost in time for matching). The PM-level must thus be chosen such that the number of nodes at that level is at least as large as the number of productions in PM. The subtree rooted by a given PE at the PM-level will store that portion of WM that is *relevant* to the production stored in that PE. A ground literal in WM is defined to be relevant to a given production if its predicate symbol agrees with the predicate symbol in one of the pattern literals in the LHS of the production, and all constants in the pattern literal are equal to the corresponding constants in the ground literal. Intuitively, the set of ground literals relevant to a given production consists of exactly those literals that might match that production, given appropriate variable bindings.

The constituent subtrees that make up the lower portion of the tree will be referred to as the WM-subtrees. For simplicity, we will assume in this paper that each PE in a WM-subtree rooted by some production contains exactly one ground literal relevant to that production. (In a fashion similar to that for production allocation, "packing" techniques may be employed at the expense of a modest increase in time for matching). It should be noted that, since a single ground literal may be relevant to more than one production, portions of WM may in general be replicated in different WM-subtrees.

During the match phase, the WM-subtrees are used as *content-addressable memories*, allowing parallel matching of a single pattern element in time independent of the size of WM. The upper portion of the

tree is used to *select* one of the matching productions to be executed in O(log P) time, where P is the number of productions, and to broadcast the *action* resulting from this execution. Details of these functions follow.

5.2 The Matching Phase

At the beginning of the matching phase, all PE's at the PM-level are instructed to enter MIMD mode, and to simultaneously (and independently) match their LHS against the contents of their respective WM-subtrees. The ability to concurrently match the LHS of all productions accounts for some, but not all, of the parallelism achieved in DADO's matching phase. In addition, the matching of a *single* LHS is performed in a parallel manner, using the corresponding WM-subtree as an *associative processing* device. The simplest case involves the matching of a single LHS pattern predicate containing at most one instance of any variable. (The reader may wish to peruse the Associative-Probe procedure detailed in figure 7 before proceeding.) In order to match the predicate

(Part-category =part electronic-component),

for example, the PM-level PE corresponding to the production in question would first broadcast a sequence of instructions to all PE's in the WM-subtree that would cause each one to simultaneously compare the field beginning in, say, its fifth RAM cell (the location of some SLICEd variable, for instance) with the string (or some syntactic token representing) "Part-category". All non-matching PE's would then be *disabled*, causing all subsequent instructions to be ignored for the duration of the match. Next, the string "electronic-component" would be broadcast, along with the instructions necessary to match this string against, say, the field beginning in the thirty-fifth RAM location of all currently enabled PE's. After again disabling all non-matching PE's, the only PE's still enabled would be those containing a ground literal that matches the predicate in question. If this were the only predicate in the LHS, matching would terminate at this point. As noted, the time required for this matching operation depends only on the complexity of the pattern predicate, and not on the number of ground literals stored in the WM-subtree.

We should mention though that the depth of the DADO tree defines DADO's machine cycle time. Thus although we state that access to WM is achievable in time independent of the number of WM elements, it is actually dependent on a logarithmic function of the number of PE's in the tree. Since WM is fully distributed among the majority of available PE's, it is also logarithmic in the number of WM elements. However, this delay is bounded by log(n) combinational gate delays, which is proportional to the latency period for access to a conventional RAM of comparable size and hardware complexity, and thus may be ignored in analysis of the time complexity.

The general matching algorithm, which accommodates a LHS consisting of a number of conjoined predicates, possibly including common pattern variables, is considerably more complex. In this case, after associatively probing for the first pattern predicate, each value contained in a matching WM element, stored in the same relative location as a pattern variable, is sequentially enumerated and used for further associative probing for subsequent patterns. In the worst case, this operation may require enumeration of each of the elements in WM. However, the high-speed content addressable memory operations reduce the "look-up" time to a constant factor, and obviates the need for expensive overhead incurred by indexing schemes required of sequential implementations. The result of this general matching operation is a set of variable bindings corresponding to all possible instantiations of the production in question that are consistent with the contents of WM.

5.3 The Selection Phase

Since each production is asynchronously matched against the data stored in its WM-subtree, the production matching phase will in general terminate at different times within each PM-level PE. At the end of the matching phase, the PM-level PE's must thus be *synchronized* before initiation of the selection phase. In support of this synchronization operation, each PM-level PE sets a local flag upon completion of its own matching task. The DADO RESOLVE circuit permits the DADO tree to compute a logical conjunction of these flags in time equal to O(log n) gate delays. (In this case, the max-RESOLVE function of DADO operates on flags set to 0 upon termination, and 1 otherwise. A low bit result signals synchronization.) DADO's tree-structured topology, along with the combinational, as opposed to sequential, computation of this n-ary "logical AND", lead to a synchronization time which is dominated by that required for matching, and which may, as in the case of WM access, be ignored in analysis of the time complexity of the production system cycle.

The selection of a single production to "fire" from among the set of all matching productions also requires time proportional to depth of the tree. Unlike the synchronization operation, however, the primitive operations required for selection are computed using sequential logic in DADO1. We assume that each PM-level PE performs some local computation prior to the synchronization operation that yields a single, numerical *priority rating*. PE's containing matching productions are assigned positive values, while other PM-level PE's are assigned a priority of zero. We also assume that each PM-level PE has a distinct *PE tag*, stored in SLICEd variable within its local memory, which may be used to uniquely identify that PE.

After synchronization, all PM-level PE's are instructed to enter SIMD mode. Each such PE is then instructed to send its priority rating to its parent. Each parent compares the priority ratings of its two children, retaining the larger of the two, along with the unique tag of the "winner". The process is repeated at successively higher levels within the tree until a single tag arrives at the root. This tag is then broadcast to all PM-level PE's for matching, disabling all except the one having the highest priority rating, which remains enabled for the action phase. Note that in DADO2, this operation is replaced by a few sequential steps employing several applications of the max-RESOLVE circuit.

5.4 The Action Phase

At this point, the "winning" PE is instructed to instantiate its RHS, which is then reported to the root. Next, all PM-level PE's are enabled, and the RHS of the winning instance is broadcast to all. The details of the action phase are made more complex by the importance of avoiding unnecessary replication of WM literals within the lower portion of the tree, and of reclaiming local memory space freed by the deletion of such literals. These functions are based on associative operations similar to those employed in the matching operation.

The PE's at the PM-level are instructed to enter MIMD mode and to concurrently update their WM-subtrees as specified by the RHS of the winning instance.

First, the PM-level PE's perform an associative probe for each literal to be deleted from WM, enabling only those PE's in the WM-subtrees whose local memories are to be reclaimed. The enabled PE's are then instructed by the PM-level PE to overwrite their stored ground literal with a special *free-tag* identifying empty PE's. This tag is the target of the subsequent associative probe executed for each of the ground literals to be added to WM.

When processing an asserted literal, the PM-level PE first determines whether or not the literal is relevant to its stored production. Next, the associative operation identifies those relevant literals which are not present in the WM-subtree, and thus are to be stored in some empty PE.

After probing for the free-tag, all PE's are disabled except the empty PE's. To avoid duplication of asserted literals, all but one of these PE's is disabled by the RESOLVE circuit. The asserted literal is then broadcast to the one enabled PE.

As in the matching phase, the action phase in general will terminate at different times in each PM-level PE. After synchronization, another cycle of production system execution begins with the production matching phase.

5.5 Specialized Production Systems

The general scheme for production system execution on DADO can be extended to support *commutative* production systems, as well as "cooperating expert systems" based on *multiple, independently executing* production systems.

A commutative production system allows each of the matching rules on every cycle of operation to be selected for execution. The same combinatorial hardware used in the action phase to select a single arbitrary "free" PE supports this operation by enumerating each of the matching productions in an arbitrary sequential order. Each of the RHS's so reported to the root are then processed by the action phase.

In our exposition of the general production system algorithm, it was assumed that the upper tree was rooted at the (physical) root of DADO (see figure 8). Since each PE in the DADO tree can execute its own independent program, the upper tree can be rooted at an arbitrary internal node of DADO. Thus, multiple, independent production systems are executed on the DADO machine by rooting a forest of upper trees at various levels of the DADO tree. (The "buddy system" of memory allocation provides a simple means to allocate multiple production systems on DADO.) Communication among these independent production systems is implemented in the same fashion as communication among the PM-level PE's during the action phase of the (commutative) production system cycle.

5.6 Discussion

By way of summary, the basic DADO algorithm for PS execution operates in the following way:

1. By assigning a single rule to a unique PE at a fixed level within the tree (referred to as the PM-level), executing in MIMD mode, each rule in the system is matched concurrently. Thus, the time to calculate the set of matching rules on each cycle is independent of the number of productions in the system.

2. By assigning a data item in Working Memory (WM) to a single PE below the PM-level executing in SIMD mode, WM is implemented as a true hardware *content-addressable memory*. Thus, the time required to match a single pattern element in the LHS of a rule is independent of the number of facts in WM.

3. Lastly, the selection of a single rule for execution from the conflict set is also performed in parallel. Thus, the logarithmic time lower bound of comparing and selecting a single item from a collection of items is achievable on DADO as well.

This algorithm offers a number of advantages over the RETE algorithm reported by Forgy, while maintaining much of RETE's efficient characteristics. We quote from [Forgy 1982]:

> ...Certainly the [RETE] algorithm should not be used for all match problems; its use is indicated only if the following three conditions are satisfied.
>
> - The patterns must be compilable [to more primitive match tests]...
>
> - the objects must be constant. They cannot contain variables or other non-constants as patterns can.

- The set of objects must change relatively slowly. Since the algorithm maintains state between cycles, it is inefficient in situations where most of the data changes on each cycle.

5.6.1 Compiling patterns

In its current form, the DADO algorithm does not provide a means to compile patterns into primitive match tests, although it does not directly exclude this possibility. However, the ability of a DADO PE to execute code independently of other PE's permits pattern matching tests common to several rules to be performed in parallel, as well as a more powerful pattern match operation, *unification*, discussed below.

5.6.2 Data Elements may contain variables

Data items within DADO's WM may contain variables or other non-constants. In this case, the Associative-Probe procedure is replaced by a SLICEd unification procedure local to each PE in a WM-subtree. Thus, an entire partially instantiated pattern element is first broadcast to all PE's, locally unified, and variable bindings are subsequently reported from those which successfully matched the pattern element in question.

This capability forms the basis of the implementation of PROLOG on DADO. [Taylor et. al. 1983] describes this procedure modified to permit the entire set of PROLOG clauses to be fully distributed throughout the DADO tree. The sequential semantics of PROLOG is maintained in the reported design through the use of the max-RESOLVE circuit applied to integers associated with each clause. Each of these integers represent the "position" of the clause in the PROLOG data base, and thus determines the order in which clauses are reported to the coprocessor and subsequently applied. (This parallel associative PROLOG implementation is the focus of a doctoral investigation undertaken by Stephen Taylor working in collaboration with Gerald Maguire.)

It should be noted that the introduction of general first order terms within elements stored in WM has substantially complicated the design of the general matching procedure. This difficulty is a direct consequence of the unification process which generates objects that may grow exponentially. For example, in order to unify the literals:

$$p(f(x_1, x_1), f(x_2, x_2), \ldots, f(x_{n-1}, x_{n-1}))$$
$$p(x_2, x_3, \ldots, x_n)$$

x_n is forced to be bound to a term consisting of 2^n symbols. Thus, the implementation requires a representation scheme based on pointer structures (which can represent the unified literal in question in linear space), and dooms any attempt to represent such objects in character form to failure. (See the linear time unification algorithm reported in [Paterson and Wegman 1978].) The 8K RAM space of a DADO PE is more than sufficient to adequately handle such objects. (A report presently in preparation by Taylor describes the use of the Paterson and Wegman algorithm in a distributed environment.)

5.6.3 Temporal Redundancy

The DADO algorithm does not restrict the amount or scope of WM modifications, but rather permits large global changes to be made to WM very efficiently (by broadcasting such changes from the root PE). However, the DADO algorithm as outlined above does not save state between cycles. Rather, in situations in which few WM changes are made on each cycle, the DADO algorithm recomputes much of its match results calculated on the previous cycle. However, the basic DADO algorithm can be easily extended to directly implement this temporal redundancy by executing the match for only those literals recently asserted in WM while saving previous rule instantiations directly within the WM-subtrees. An implementation including this feature is presently under development.

Lastly, we note that the basic DADO algorithm can be modified to accomodate certain anomalous situations which may arise in practice. We briefly describe these in turn.

5.6.4 WM-subtree overflow

In the event that the number of literals to be stored within a WM-subtree is too large, two productions may be stored within a single PE one level higher than the PM-level. The resulting configuration produces a WM-subtree twice as large, at the expense of slowing the match phase to accomodate two sequential production matchings. Other allocation schemes are possible. For example, the entire upper portion of the DADO tree may be used to store productions for matching, at the expense of a log(P) time matching operation, where P is the depth of the upper tree. This last configuration is particularly useful when considering the following problem.

5.6.5 Duplicate WM elements

As noted, an instance of a WM element relevant to several rules will exist within several distinct WM-subtrees. In order to achieve the maximum for parallel matching of rules, in the worst case an exact copy of WM may exist for each rule. (Contrast this with "shared memory" models of computation in which a single global memory is accessible to some large number of asynchrous processors.) This duplication problem is imposed on us by the fully distributed model of storage and computation in the DADO machine. In order to reduce the number of duplicate elements, a *subsumption* principle may be used to effectively partition the productions distributed within the upper tree while maintaining a log(P) time for rule matching.

If the LHS of rule P_1 is a generalization of the LHS of rule P_2 (that is, rule P_1 matches a superset of the literals matched by P_2) then P_2 is placed in the subtree rooted by P_1. Rule P_2 thus shares a subset of WM-subtrees accessible to P_1 . If the LHS of P_1 is disjoint from that of P_2 then both rules may be placed in sibling subtrees without sharing a common WM-subtree. Finally, in the case where the LHS of P_1 overlaps with that of P_2, but it is not a generalization of P_2, then they may either be located within the same PE, or their relative positions may be determined by their respective relationships with other rules in the production system.

There are many degrees of freedom and tradeoffs involved with this allocation scheme (which forms a major part of a doctoral investigation being conducted by Daniel Miranker).

6 Future Research

Thus far, 12 people have written PPL/M programs for DADO. The applications that have been written, at various stages of completion, include system-level diagnostics and AI applications.

The diagnostic programs, which are currently being integrated within the kernel system, exercise the processor and RAM chips whenever a PE is in a non-busy state. The coprocessor has been designed to periodically assess the status of the entire system by performing a high-speed logical disjunction of the error flags of all PE's to identify any that may have failed. Furthermore, we have included in the design of DADO2 a simple sequential circuit passing through each connector in the system which is used to detect any faulty connections. Other than this simple "hardware hack", we have paid little attention to the issue of fault tolerancy thus far. Nonetheless, the statistics on the error rate of the Intel chips we have employed indicates that soft errors will appear in DADO2 every 1800 hours of operation.

The bulk of our effort has concentrated on the development of the interpreter for the parallel execution of production system programs. A restricted model of production systems, Winston's animal program [Winston 1977], has been implemented in PPL/M and is currently being tested. Our plans include the completion of an interpreter for a more general version of production systems in the coming months including a direct implementation of the RETE matching algorithm. A modified algorithm for the rapid evaluation of *hierarchical production systems*, typified by MYCIN-like systems, is being investigated for implementation on DADO as well. Indeed, the envisaged PROLOG implementation may subsume this effort.

Fahlman [1979] has proposed a special-purpose parallel architecture for high-speed property inheritance in systems based on semantic network-like formalisms. Although it is too early in our investigations to make any precise claims, we believe that DADO may in fact provide significant improvement in the execution of semantic network based systems over von Neumann machines. Currently, we have implemented the essential elements of a *frame-matching* operation, but have not yet explored the possibilities of applying DADO's hardware parallelism to property inheritance operations.

Lastly, we note the relationship of LISP to DADO. Part of our work has concentrated on providing LISP with additional parallel processing primitives akin to those employed in PPL/M. Thus, we have been actively pursuing the opportunity of providing SLICEd list structures within a conventional LISP environment.

More importantly, though, we have begun to formulate the essential aspects of LISP execution which may be regarded as purely associatively-based, and thus suitable for direct execution on DADO. Examples of such operations include:

- finding variable bindings on an association list,

- property list operations, including the access and instantiation of function definitions,

- finding and allocating, as well as freeing, a *cons cell* from a large space of free memory cells.

The time required for each of these operations on a sequential machine is, in general, linear in the size of the list structures in question. For certain of these operations space-expensive hashing may reduce the time to a constant. Within DADO, on the other hand, these operations may be executed in constant time without a significant overhead in storage management (see [Bonar and Levitan 1981]).

By way of summary, it is our belief that DADO can in fact support the high-speed execution of a very large class of AI applications. Coupled with an efficient implementation in VLSI technology, the large-scale parallelism achievable on DADO will indeed provide significant performance improvements over von Neumann machines. We are presently preparing detailed experiments to empirically evaluate the performance of DADO2. If pressed to give some indication of its capabilities, we have estimated that DADO2 may execute R1/XCON, for example, at an average rate in excess of 150 production system cycles per second. Presently, R1/XCON runs on a VAX 11/780 at a rate from 2 to 600 cycles per minute. The envisaged implementation of R1/XCON, which provides this rough estimate, consists of a PM-level of 32 PE's, performing the match, 31 PE's within the upper tree, performing selection, and 30 PE's in each of the 32 WM-subtrees.

7 Conclusion

A large part of our work continues to involve the analytical investigation of new parallel algorithms and languages for AI applications. Several researchers are actively investigating methods for the rapid execution of *frame*-based systems, as well as methods for improving the performance of "conventional" AI languages including a parallel implementation of PROLOG. New methods for the parallel execution of ***hierarchical production systems*** are being investigated as well.

A large share of our efforts, though, are devoted to the hardware design and implementation of a larger experimental device. Although adequate for further development of the software base for DADO, the 15 element DADO1 system is too limited in storage capacity and processing power to demonstrate a significant performance improvement in the execution of AI systems. (A large share of the machine's processing power is utilized in system control and interprocessor communication.) However, as the number of processors in the system increases, the proportion of this "wasted" processing power will decrease dramatically. Concomitant with increasing the number of processing elements in the system, a

set of new technical problems dealing with interprocessor communication and fault-tolerance must be solved to achieve the predicted speed-up.

However, these problems can only be investigated if a large-scale, experimental device is implemented. Thus, using a slightly modified hardware design of the DADO1 system, we are currently implementing a much larger version of DADO, comprising 1023 processing elements. This version, DADO2, will incorporate a custom VLSI chip, currently being designed at Columbia University, to perform the most basic communication functions in combinational logic. This custom IC is expected to produce a significant improvement in operating speed, and would be a required component of a full version of DADO implemented entirely in VLSI. The existing DADO software will require only minor modification to run on the newer design.

Our future plans include the demonstration of the DADO2 prototype using several existing large-scale expert systems which use the production system paradigm. Digital Equipment Corporation has expressed an interest in supplying a copy of R1/XCON for implementation on DADO. Bell Laboratories has also expressed a willingness to supply a copy of ACE, an expert system that has been developed to perform telephone cable maintenance. Other systems are being actively sought from other sources in the Artificial Intelligence community.

Acknowledgements

It is a great pleasure to thank the many people and organizations who have contributed to the DADO project. Daniel Miranker, a Ph.D. student at Columbia, is responsible for many of the detailed hardware and software designs of the machine and is the driving force in the effort to implement the OPS family of languages on the system. Stephen Taylor, also a Ph.D. student, working closely with Chris Maio, Andy Lowry and faculty co-investigator Professor Gerald Maguire have made tremendous progress in specifying the execution of PROLOG on DADO. Their design of the PROLOG system is as elegant as the hardware solutions provided by our project engineer, Shunsaku Ueda, who deserves special mention. Professor David Shaw has had a tremendous impact on our work, and we are very grateful for his involvement. Many researchers have contributed in substantial ways, too numerous to specify in great detail. We thus would like to acknowledge the efforts of Janvid Cheng, Eugene Dong, Wai Man Wong, Jody Weiss, Mike Weisberg, Jim Gilpatrick, Monique Fei, Daphne Tzoar, Doug Degroot, Mark Lerner, Alex Pasik and Ted Sabety. We would also like to thank Lanny Forgy, Allen Newell and Mike Rychener for very interesting and thought-provoking conversations about DADO.

The Defense Advanced Research Projects Agency is our primary source of support through contract N00039-82-C-0427. Intel Corporation has contributed most of the components used in the construction of DADO1, and continues to support our development effort of DADO2. Digital Equipment Corporation has provided computational resources for our software development. Valid Logic Systems has donated the prototype boards used in the construction of DADO1 and has continued to aid our research. Finally, we acknowledge the assistance of IBM Corporation for providing components for DADO2 as well as taking a more active role in supporting our research. Presently, IBM researchers are helping to prepare experiments for the statistical analysis of our PROLOG work.