AN OVERVIEW OF THE DADO PARALLEL COMPUTER

Mark D. Lerner, Gerald Q. Maguire Jr., &
Salvatore J. Stolfo

CUCS-157-84

An overview of the *DADO* parallel computer

by MARK D. LERNER, GERALD Q. MAGUIRE JR., and SALVATORE J. STOLFO
Columbia University
New York, New York

ABSTRACT

DADO is a special purpose parallel computer designed for the rapid execution of artificial intelligence expert systems. This article discusses the *DADO* hardware and software systems, with emphasis on the question of *granularity*. *DADO* is designed as a fine-grain machine constructed from many thousands of processing elements (PEs) interconnected in a complete binary tree. Two prototype systems, *DADO1* and *DADO2*, are detailed. Each PE of these prototypes consists of a commercially available microprocessor chip, memory chips, and an additional semicustom I/O processor designed at Columbia University. The software includes a kernel and parallel languages. Under development are several artificial intelligence systems, including a production system interpreter, a logic programming language, and an expert system building tool.

INTRODUCTION

A considerable amount of interest has recently been generated in specialized machine architectures designed for the very rapid execution of Artificial Intelligence (AI) software.[1] The Japanese Fifth Generation Machine Project, for example, promises to deliver a functioning device capable of computing solutions of large PROLOG programs at execution rates on the order of many thousands to perhaps millions of *logical inferences per second* (LIPS). Such a device will require high-speed hardware executing a large number of primitive symbol manipulation tasks many times faster than today's fastest computers. This rather ambitious goal has led some researchers to suspect that a fundamentally different computer organization is necessary to achieve this performance. Thus, *parallel processing* has assumed an important position in current AI research.

Since 1980, we have been exploring various parallel processing architectures suitable for the rapid execution of AI software implemented in production system (PS) and logic programming (PROLOG, for example) form.[1,2,3,4] A number of parallel algorithms have been developed and studied, each designed to capture the inherent parallelism in a wide range of symbol manipulation tasks. This study has led to the development of a specific machine architecture which has come to be called *DADO*.[5,6,7]

DADO is a binary tree-structured multiprocessor architecture incorporating thousands of moderately powerful processing elements (PEs). Each PE consists of a fully programmable microcomputer with a modest amount of local memory (in the range of 16 to 20 Kbytes, in the second prototype) and a specialized I/O chip designed to accelerate inter-PE communication. A full-scale production version of the machine may comprise many thousands of processors implemented in VLSI technology.

Other parallel tree-structured machines have been proposed in the literature for accelerating a wide range of applications from database operations to more general applications involving numerical operations.[8,9] *DADO* distinguishes itself from these other architectures in several ways.

First, *DADO* is not fully detailed. That is, although we have settled upon the notion of large-scale parallelism by incorporating thousands of PEs interconnected in a complete binary tree, the granularity (storage capacity and functionality) of each PE remains an open theoretical and important practical issue. We are thus hedging our bets by remaining uncommitted to any specific PE design. By studying "real-world" applications executed on a *DADO* prototype, our proposed architectural studies will shed more light on the granularity suitable for a production version of the machine.

Second, *DADO* is designed for a specialized set of applications implemented with the AI production system and logic programming formalisms.[3] Thus, we can expect *DADO* to provide important information for other researchers interested in accelerating AI computation.

Third, the execution modes of a *DADO* PE are rather unique. Each PE may operate in Single Instruction Multiple Data (SIMD) mode[10] whereby instructions are executed as broadcast by some ancestor PE in the tree. Alternatively, a PE may operate in Multiple Instruction Multiple Data (MIMD) mode by executing instructions from its local RAM. Such a PE may, however, broadcast instructions for execution by descendant PEs in SIMD mode. This rather simple architectural principle allows *DADO* to be fully partitioned into a number of distinct "sub-*DADO*s," each executing a distinct task. Thus, *DADO* may potentially implement a variety of parallel algorithms spanning a vertical spectrum from the small grain operations typical of the parallel manipulation of databases up to more coarse-grain activities associated with multitasking. Indeed, it is our experience that this flexible architectural design provides a rather powerful "semantic-base" allowing relatively easy programming of these various parallel activities.

Fourth, our current design of the second prototype machine, *DADO2*, employs two physical binary tree interconnections. This provides fault tolerance as described in a later section.

Fifth, we have designed *DADO* around commercially available technology rather than designing everything from scratch. In addition, the software has been designed with portability in mind (such as changing the type of microprocessor). Thus, it is quite possible that future *DADO* machines could rapidly be implemented and directly employed in real-world settings much more quickly than other proposed machines waiting for technology to catch up with their architectural needs.

In a number of previous papers,[3,11,12,13,14,15,16,17,18,19] we have detailed the hardware design of the *DADO* machine, as well as the software systems that have been implemented to date and currently under development. The algorithms that were invented for the parallel execution of production systems and logic programs were not sufficient alone for deciding several specific hardware design issues. The decision was made, therefore, to build functioning prototype systems that provide the means to carry out empirical studies. Towards that end, it was necessary to develop concurrently both hardware and software systems that implement and demonstrate the most important architectural principles of the machine while providing the necessary laboratory environment for experimentation and study.

In the following sections, we detail the status of the *DADO1* and *DADO2* prototype hardware and software systems. Several kinds of software systems are under development. Low-level system software includes the kernel and application language support resident in each *DADO* PE. The parallel languages are PPLM,[13] and a newly developed high-level parallel LISP (PPSL).[17,19] Artificial Intelligence (AI) software consists of several very-high-level systems: OPS5,[20] HerbAI,[21] and LPS[10,18] languages. Knowledge acquisition software (such as DTEX[22]) uses these layers to develop a working expert system which is then available to users.

HARDWARE

The hardware configuration of *DADO* consists of a large number of processing elements (PEs) connected by a tree-structured bus. Each PE in the prototype systems consists of an Intel 8751 microprocessor with 4 Kbytes of onchip EPROM, 128 bytes of onchip RAM, 16 Kbytes of RAM with parity, and an I/O processor. The initial *DADO1* prototype PE was built without the I/O processor, which is simulated by a software kernel for parallel communication resident in each EPROM.

The I/O processor provides very fast bidirectional communication and selection of data. It also supports both a *global interrupt* and *context switch*. This allows any processor to interrupt the entire machine and place it into debug mode. When an interrupt occurs, the host processor is given control. It can then read and write the individual processor memories as part of the debugging process. Subsequently, the machine can be restarted without loss of any information.

As noted, the PEs are interconnected as a binary tree. The root PE is connected to two descendant PEs, which in turn are connected to two descendants each. This interconnection topology was selected for two reasons. (1) It can be very efficiently implemented in VLSI technology, and (2) this architectural configuration is well suited for a large class of problems.

A prototype of the hardware, *DADO1*, has been operational since April 1983. It has been used to develop further hardware and software. Using gate array technology, we have designed a specialized I/O processor for the *DADO2* PE to accelerate inter-PE communication. These I/O chips are interconnected in a binary tree. Each microcomputer chip in a *DADO2* PE is also interconnected in a second complete binary tree. These two tree interconnections provide a measure of fault tolerance. Thus, if a *DADO2* I/O chip fails, the processor connections will allow the machine to continue operating, as evidenced by the small 15 element *DADO1* prototype which employs processor interconnections exclusively. Alternatively, if a *DADO2* processor fails, the I/O chip interconnection will allow easy fault isolation and diagnosis. Software for reconfiguration and fault tolerance, however, is not yet developed and not part of the current effort.

Work is largely complete on the larger *DADO2* machine. This computer will be configured with 1023 PEs and 16 megabytes of RAM. The construction of this machine involved the design and integration of the I/O processor; the connection circuitry in the form of circuit boards and a backplane; the power supply and cabinet; and a high speed I/O interface to a conventional host.

Communication

DADO hardware prototyping began three years ago in 1981. At that time, the Intel 8751 was the only commercially available single chip microcomputer in existence that provided 4 parallel 8 bit ports. These ports allowed us to conveniently interconnect a number of PEs directly without additional logic or chips. The 4K onchip EPROM also allowed us to conveniently implement software with the opportunity to reprogram the ROM as the software evolved.

If we were to initiate prototyping today, we would use a 32 bit two address microcomputer chip. Such a design is expected to immediately deliver a 16 fold speed advantage over the 8751.

Synchronization between processors was originally implemented with a four-cycle handshake protocol, implemented in software. This is being replaced with a semicustom I/O processor, which is expected to provide substantial performance improvements for two reasons; (1) the I/O chip can transfer a byte of data throughout the entire machine in the time needed for a typical microprocessor instruction, (2) the microprocessor is no longer burdened with the communication task.

The I/O chip performs in hardware three main parallel operations: (1) flow of information from one processor to its descendants (broadcast), (2) the selection of the largest (or smallest) value of a set distributed throughout the machine (max/min resolve), and (3) the flow of information from descendants towards their ancestors (report). In addition, the I/O processor provides the hardware to help monitor and debug large software systems (global interrupt and context switch), as well as memory support.

The I/O processor was completely designed at Columbia University. We used sophisticated CAD/CAM tools in this phase. A VALID ScaldSystem served in all phases of development: from schematic entry using a hierarchical methodology, through rough simulation, and then board layout. The design was verified at LSI Logic with their LDS simulator, and subsequently fabricated using approximately 1850 gates of HCMOS gate array.

Circuit Boards

The extraordinarily regular layout of the binary tree made it possible to design relatively simple circuit boards. Most of the interconnections are within the PEs, between a RAM chip and its processor. These wires are shorter than 1 inch in length. The remaining lines are for communication and clock signals. These interconnections are very regular. Indeed, all 32 boards are completely interchangeable.

The *DADO* circuit board was designed using two CAD tools; the VALID ScaldSystem and an Advanced Electronic Design (AED) workstation running Caesar CAD software. The VALID was used to create a drawing of the board. We developed software on the VALID to produce a complete holelist from this drawing. The holelist indicates the positions

of all drill holes, as well as the endpoints of the connecting wires. Hierarchical design features of the VALID allowed us to specify the interconnections once and then replicate them as needed.

The AED workstation was used to the design artwork which describes the power, ground, and silkscreen layers of the circuit board. We subsequently used software to verify that the holelist agreed with the artwork. The DARPA MOSIS silicon foundry then produced full-sized transparencies of the three layers. The actual board fabrication was done by Multiwire Corporation, using the artwork and holelist. The motherboard was also designed with the AED station, and has been fabricated by MOSIS.

Packaging

To minimize the volume of the machine we chose to place 32 processors on each board and to maintain a very low profile on each board. To achieve these goals, we used two techniques: (1) The I/O processor is packaged in a 64 pin PGA package which occupies 1 square inch of surface area. We exploited this square package to develop an efficient component layout. (2) Components are placed into eyelets. These occupy less height above the board than the traditional sockets, while providing the excellent mechanical and electrical qualities that are required for a new machine.

Each of the 32 boards houses 32 processors, 32 I/O processors, and 512 Kbytes of RAM with parity. The resulting hardware is simpler than a DEC VAX-11/750, and has a cumulative processing power in excess of 570 MIPS.

Interface to Control Processor

The *DADO* machine functions as an attached processor controlled by a conventional computer such as a DEC VAX or AT&T 3B machine. The conventional computer serves as a control processor (CP). *DADO* communicates with the CP in two ways: either an RS232 connection at 9600 baud, or with a parallel interface. The latter allows us to use the direct memory access (DMA) mode of a DEC DR11W. To achieve large data transfer rates (approximately 500 Kbytes/second) we designed a ROM-based microcoded interface processor. This handles the protocol needed to transfer data to and from *DADO*.

SYSTEM SOFTWARE

The system software includes a kernel (described below) and two parallel languages developed at Columbia. Copies of the kernel are located at each PE. This coordinates use of the hardware and provides essential services that are critical to the operation of the machine, or are very frequently required by the high level languages. The various layers of software are shown in Figure 1.

The kernel supports the most fundamental operations of the *DADO* machine. This includes control of the hardware lines, synchronization of processors, and the transfer of data. The kernel functions are detailed in Reference 2.

This kernel was extensively used without modification for over a year. Minor modifications have been made to support the I/O chip of *DADO2*. Recently, the kernel has been restructured to provide the resources needed for the PPSL language described below.

The lowest level of the kernel actually manipulates logic levels; an intermediate level implements protocols with these signals. The highest level is accessible to the user, and provides the following basic functions:

1. *Broadcast* to send information to the descendant processors.
2. *Resolve* to select one processor from a candidate set.

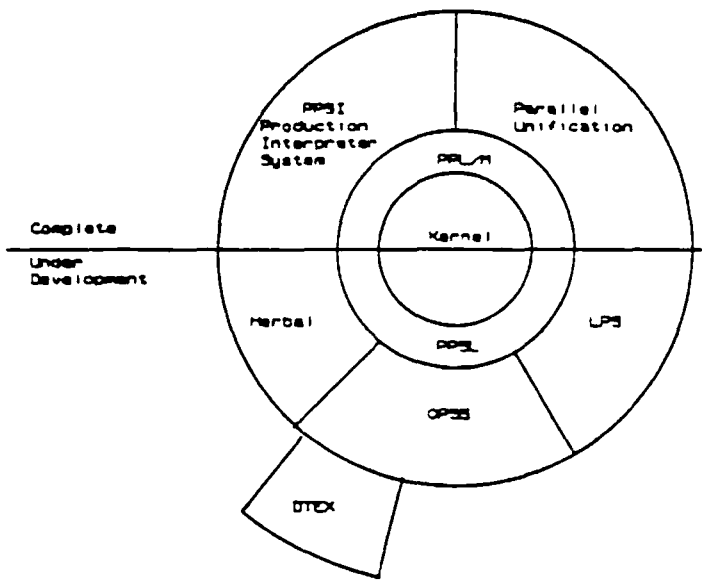

Figure 1—Layers of system software

3. *Report* to send information to the root processor.
4. *Send* to send data to a neighbor.
5. *Receive* to receive data from a neighbor.
6. *Enable* and *Disable* to make processors execute the instructions or to prevent them from doing anything except forward data.
7. *MIMD* and *Sync* to partition the tree into independently executing subtrees, and to resynchronize.

PPLM

Parallel programming requires the programmer to provide two kinds of specifications: (1) A control structure for the coordination of PEs. This may be viewed as a set of conditions that determine when a code segment should be executed. (2) The sequential code segments to be executed in particular processors. Communication and coordination between these segments may either be explicitly stated in the program, or implicit from the context in which they are used.

The first *DADO* language implemented, PPLM, uses explicit statements to synchronize the processors. PPLM provides direct access to the SIMD, MIMD, and communication features of the machine. Interspersed in the program are statements to be executed in parallel. These are designated via a syntactic construct, the DO SIMD block.

PPLM[13] translates source programs into Intel's PLM-51[23] language. PPLM checks for syntax errors, and in addition enforces the semantics needed to prevent deadlocks in a parallel computer. For example, if a processor expects to receive information from its parent, then the parent must place the information on the broadcast bus. The compiler determines whether these conditions are satisfied and generates an error message if they are not.

This program is currently executed on a VAX running UNIX. The translated output is sent to an Intel MDS with Kermit* and then compiled with Intel's PLM-51 compiler. PPLM is implemented via the standard UNIX tools (C, Lex, and Yacc) and can be easily transported to a variety of machines.

PPSL

The second language implemented is PPSL.[17] This is a LISP system which has been extended to permit the specification of parallel control. The most salient feature of the language is the declaration of functions as MIMD or SIMD, with semantic rules to determine how these forms are executed in *DADO*.

The language provides the ability to manipulate arbitrary LISP expressions (s-expressions, or forms) throughout the tree. The language constructs include:

1. (BROADCAST form1 form2), in which *form2* is evaluated in one processor and then stored into descendant processors as determined by their independent evaluations of *form1*
2. (SETQSLICE form1 form2), in which *form2* is evaluated in each descendant processor and stored into the *form1* of each processor.
3. (DOMIMD form), in which designated descendant processors disconnect themselves from the tree-structured broadcast bus, and begin execution of *form*.

The PPSL language can be executed on a conventional computer by use of a *DADO* simulator. This simplifies software development. Several programs have been written using this tool, including an OPS5 implementation, a parallel sort program, a spelling corrector, and numerous production system algorithms.

The PPSL language has the ability to *relocate* s-expressions (the fundamental LISP data structure). This is required to communicate the s-expressions between processors and the host. Storage structures have to be relocated when moved between processors. Moreover, communication of arbitrary length items is required for the language, and the language supports a protocol that is both robust and efficient.

The PPSL constructs are implemented primarily in LISP. The programmer need not be concerned with low level details. Instead, routines such as Broadcast, SetqSlice, and DoMimd provide implicit communication. In addition, the PPSL programmer has access to the complete *DADO* kernel (this may be needed, for example, in applications that utilize non-LISP segments of code such as assembler).

Static scoping is being implemented with a program analysis tool. This is a scope-checker, which will perform a static analysis to determine if scope rules are violated.

Compilation-time error checking and optimization will be essential. Diagnosis at compile time saves substantial effort at later phases. Common errors include misuse of scoping and use of the wrong type of data objects. (See Figure 2 for PPLM and PPSL code.)

LISP Compiler

A LISP compiler[19] based on other works[21, 25] has been developed for the *DADO* prototype machines. It translates LISP into assembly language for the Intel 8751 processor, which in turn is assembled and linked with commercially available tools.† This language can be used to program *DADO*; moreover, the high-level PPSL language runs on *DADO* by use of this underlying LISP structure.

In addition to the bare compiler, we have developed a run-time library and other software to support programming in PPSL. The most important parts of the work are a *kernel interface* and *garbage collector*. Other significant aspects include independent compilation, optimizations, interprocessor communication of s-expressions, and several kinds of diagnostics.

Independent compilation allows us to create object code libraries. The original design includes provision for the independent compilation by use of a linkage editor and a global symbol file.

* Kermit is a widely used file transfer and terminal emulation program developed at the Columbia University Center for Computing Activities.

† We use the Nu-data Cohware assembler and linkeditor.

```
DO_SIMD
   CALL ENABLE.                    /* Enable all processors */
   STRPTR=0.
END
DO I=1 TO LENGTH( INPUT)
   CALL BROADCAST(INPUT(I)).       /* Broadcast the byte */
   STRING(STRPTR)=ICS.             /* Store the byte in descendants */
   STRPTR=STRPTR+1.
END
```

(a)

```
(DE SENDDATA(INPUT)
   ENABLE)
   BROADCAST (QUOTE STRING) INPUT))
```

(b)

Figure 2—(a) PPLM code to store the string contained in INPUT; (b) PPSL code to store the string contained in INPUT

We are currently investigating optimization techniques to reduce code size. We have developed a preliminary data-compaction algorithm. This shows that code improvements are possible by two techniques. First, the algorithm is used as an analysis tool to show where compiler internals must be changed to improve the code. Second is common substring detection followed by automatic subroutine generation. Preliminary studies indicate that up to 40% compaction is possible.

The compiler will eventually provide direct support for parallel debugging. It will support four techniques:

1. Read out code or data from the tree to compare with that stored on the host
2. Break points, to gain control of a PE when it executes certain code
3. Rebinding of functions, to modify what a function does
4. Profiling, to determine how frequently functions are used

These will be incorporated into a graphics-based debugging system.

Graphics

We are currently using high resolution color graphics to debug and demonstrate programs. We have designed a graphics based debugging system that draws a picture of a binary tree. By use of two physical screens, we will be able to probe into processing elements as required. These graphic routines are used to demonstrate the operation of *DADO*.

ARTIFICIAL INTELLIGENCE SOFTWARE

Several artificial intelligence areas are being studied as application areas for *DADO*. We have made significant progress in three of these: a production system language (HerbAl), a logic programming language (LPS), and a generic expert system tool (DTEX). Although *DADO* was originally designed to be a production system machine, the additional uses attest to the flexibility of the architecture design.

A production system[26,27] consists of a number of production rules (PM), which are matched against a database of facts called *working memory* (WM). Rules have a left hand side (condition) and a right hand side (action). The system repeatedly executes the following cycle:

1. Match: Determine those productions with conditions that are true given the current state of the working memory database.
2. Select: choose one of the true productions.
3. Execute: Execute the actions specified by the right hand side of the production. Actions include additions, deletions, and modifications to the working memory.

Although production systems have been put to practical use on sequential machines with considerable success, there is a critical need for significantly faster execution speeds. We have demonstrated that production systems can be executed by use of parallel hardware, and expected to achieve dramatic performance improvements in the near future.

One major question in the design of the parallel hardware is the *granularity* question. This is concerned with the resources (functionality, and storage capacity) available at each processing element. When the size of RAM is increased in a coarse-grain approach, more rules and WM elements may be stored and processed by an individual PE. Since fewer PEs are available, however, less work may be performed in parallel. Conversely, by restricting the size of RAM in a finer-grain approach, fewer rules and WM elements may be located at a PE, but the additional PEs can possibly perform more operations in parallel.

Recent statistics reported for R1* indicate that of a total of 2000 rules and several hundred WM elements, on average, 30 to 50 rules need to be matched against WM on each cycle of operation. Thus, even if 2000 finer-grain PEs were available to process the rules (say, one rule per processor), only 30 to 50 PEs would perform useful work on each cycle of execution. If we used 30 to 50 coarser-grain processors storing many more rules, instead, all of the inherent production matching parallelism would be captured, making more effective use of the hardware.

This approach ignores the advantages of processing WM elements in parallel. In a manner analogous to partitioning rules to a set of PEs, WM elements may also be distributed to a set of independent PEs distinct from those storing rules.[4,28] The grain size of a PE may then directly affect the number of WM elements that may be processed concurrently. Thus, with a larger number of smaller PEs, WM may be operated upon more efficiently than with a smaller number of larger PEs. It follows that a "tug-of-war" between production-level and WM-level parallelism provides an interesting theoretical arena to study the tradeoffs involved between parallel processors of varying granularity.

A larger question has now arisen. The reported statistics for R1 are based on a problem-solving formalism that has been fine-tuned for fast execution on serial processors, namely OPS5. *Thus, the inherent parallelism in R1 may bear little resemblance to the inherent parallelism in the problem R1 solves; but it is, in our opinion, an artifact of current OPS5*

* R1 is an expert system written in OPS5 that configures DEC VAX computers

production system programming on serial machines. The OPS5 implementation of R1 provides little information about what subproblems are inherently parallelizable. Indeed, all subproblems are carefully handcrafted to be sequentially executed using "control elements." Our aim is to provide other formalisms that allow one to explore and implement much more parallelism than OPS5 encodes or encourages.

For example, two simple experiments have been performed on two small production system programs running at Columbia. A few additional parallel constructs were added to OPS5 which simulate the parallel manipulation of WM[29] and multiple rule applications on each cycle of execution. The statistics indicate that these slight enhancements to OPS5 produce a *factor of 6–10 fewer* production system cycles of execution to solve the same problem. Similar studies are underway using the ACE expert system.[30,31] Thus, we can expect that a formalism slightly different than OPS5 will admit much more opportunity for parallelism and concomitant speed-up.

Original DADO *PS Algorithm*

The original algorithm, implemented in the PPLM language, works as follows. The *DADO* machine is divided into subtrees, in which there is a distinguished level for production memory (PM-level) and a lower level for working memory (WM-level). During the match phase, each PM-level PE enters MIMD mode and uses its SIMD PEs as a content-addressable memory. No state is saved between cycles because the algorithm exploits large-scale parallelism in the match phase.

During the act phase, changes to WM are enforced when the winning production reports the actions to the *DADO* root. These actions are subsequently broadcast to all PM-level PEs, which in turn make the appropriate updates to their descendant PEs' storage.

One characteristic of production systems is temporal redundancy. Few changes are made to WM on each cycle, which restricts in practice the number of possible rule matchings on subsequent cycles. The above algorithm recomputes all matches before each production cycle, and does not save any state information between cycles. Thus, it does not exploit temporal redundancy. On the other hand, the OPS5 production system[32] saves state between production cycles and never recomputes comparisons. This exploits temporal redundancy, yet there may be excessive overhead, particularly when large changes are needed to WM.

The Treat Algorithm

A new algorithm, Temporally *RE*dundant *A*ssociative *Tree* algorithm (TREAT)[33] uses ideas from both of the above algorithms. It maintains the most useful state information, while less useful comparisons are recomputed as needed. This approach is based upon several observations.

When a new element is added to the WM, every rule that becomes eligible for execution will also contain the new WM element. A technique used by OPS5 (called alpha-memories) allows us to quickly compute matches for the new WM element. Similarly, when WM is deleted, the tree is examined in parallel and all conflicts are removed concurrently.

Significant performance improvements can be obtained by maintaining less state information. Less time is spent maintaining the state, and computations can be reordered without concern for maintaining state information. Consequently, heuristics may be used to select more optimal orderings. Finally, it should be noted that the dramatically increased computational ability of *DADO* can be exploited to rapidly compute matches when required—the importance of saving the prior matches is thereby reduced.

LOGIC PROGRAMMING

Logic programming is a formalism based upon symbolic logic. PROLOG is probably the best known example. According to this formalism, a clause consists of logical first-order terms that must be satisfied. Frequently, the clauses may be viewed as a sequence of conjunctions and disjunctions, and thus an AND/OR tree is commonly used to represent the programs. Logic programming allows a problem to be specified as general rules and data structures that will be expanded to solve a directive.

Logic programming has attracted a great deal of attention as a medium for the development of software for parallel execution. Two major factors contributing to this perception are the demonstrated suitability of logic programming for the expression of a wide variety of software tasks, and the identification of several sources of parallelism inherent in the logic formalism itself. Thus logic programming languages appear to offer a framework in which programs naturally lend themselves to efficient parallel execution, but in which the programmer need not be overly cognizant of this goal.

A Logic Programming Language (LPS)[16,15,18] is under development at Columbia. The key aspects of LPS are *unification* and *reconciliation*. Unification finds a substitution to transform two terms into identical terms. There is a most general unifier such that all other unifiers are instances of it. The unification processes can be performed in many processors, each one working on a particular goal.

Reconciliation is a process that determines if two substitutions are compatible, and produces the most general substitution if one exists. It is an essential part of the LPS algorithms, and makes the various unifiers consistent. In this manner the independent unifications give rise to one answer.

We may view the proof search as a perusal of goal lists. The LPS search tree, in comparison to the standard sequential algorithm used by Prolog, is characterized by

1. Shorter paths to leaves
2. Earlier termination of unproductive paths
3. Earlier consideration of most goals, causing earlier branching but not necessarily higher branching factors
4. A substantially reorganized leaf structure, resulting in a different order to the construction of solutions.

The unification phase of the LPS algorithm places different facts into each PE, and then transmits a goal list from the CP

into the PE network. Each PE unifies with many goals, producing unifiers that are tagged with a *level number* to identify the goal whose unification gave rise to the binding set.

The second phase is the reconciliation phase. It is also known as a "join phase" due to similarity with the operation of an equi-join over a set of relational database operations. A heuristic for ordering the join phase can, in most circumstances, keep join phase communication close to minimal. The join phase is coordinated by the CP, to allow communication of binding sets around the network.

The last task to be performed upon discovery of a successful proof is the composition of the various substitutions that were generated along the way. In the LPS implementation, the substitution phase is accomplished by transmitting prior reconciliation history to the PE network, and computing in each PE the composition of that substitution with any new reconciliation.

These algorithms have been implemented (on a sequential computer using PSL) in order to uncover problems in parallel execution of logic programs. Planning for an implementation of an LPS interpreter to be executed on the prototype *DADO2* machine is presently underway.

ARTIFICIAL INTELLIGENCE: DTEX GENERIC EXPERT SYSTEM TOOL

DTEX is a decision tree based expert system building tool. DTEX organizes knowledge into a decision graph, which is appropriate for problems that can be expressed as a hierarchy of relationships. It uses nodes to represent properties, and edges to represent the values corresponding to properties of the source node. This design represents conjunctions of disjunctions as appropriately structured graphs. Explicit AND/OR edges are not needed because the system forward-chains towards its decision.

DTEX[22] consists of three main modules: strategy, explanation, and knowledge acquisition. The strategy phase is divided into *ask*, *decide*, and *next* phases whereby the system can collect information, select an appropriate edge, and continue in the next phase.

Explanation provides a recapitulation of the inference processes. In addition, it responds by supplying the rationale behind a link as a useful technique to convince or teach the user.

The system adds new knowledge through interaction between DTEX and the specialist. The knowledge acquisition phase will perform extensive consistency checking in addition to syntactic checks (such as "*if* valid values are integers from 0 to 5, *but* the new node could only be reached if values are between 3 and 5, *and* the purpose is to consider the new node, *then* only allow edges with values between 3 and 5"). Semantic checking is performed by building frames with appropriate semantic information. This requires the expert to express meanings carefully and thereby prevents mistakes. A preliminary version of DTEX has been developed in the domain of recommending treatment for adolescent idiopathic scoliosis.

SUMMARY

A prototype machine, named *DADO1*, has been operational at Columbia University since April 25, 1983. As noted, *DADO1* consists of 15 Intel 8751 microprocessors. These 15 PEs, interconnected in a complete binary tree, serve as a testbed for the development of software for a larger prototype currently under construction. *DADO2* consists of 1023 Intel 8751-based PEs, which will provide in excess of 570 MIPS, and is expected to be completed in the coming months.

DADO2 is not viewed as a performance machine, but rather as a laboratory vehicle to investigate fine-grain processors. We expect *DADO2* to achieve significant performance improvements of AI software;[18, 20] but more important, it will provide a testbed for the next generation machine, which is expected to achieve the stated goals of DARPA's Strategic Computing Program for symbolic multiprocessors. Although the binary tree topology may cause severe communication bottlenecks for some parallel operations, we expect that *DADO* will achieve very impressive performance for applications where computation dominates over communication. We have found this to be the case for AI production systems, for example. Since tree structures offer many favorable advantages for hardware implementation, *DADO* will offer very attractive cost performance for many of these important applications.

ACKNOWLEDGMENTS

This research was conducted as part of the *DADO* project. It was supported in part by the Defense Advanced Research Projects Agency under contract N00039-84-C-0165, and the New York State Science and Technology Foundation NYSSTFCAT(84)-15, as well as by grants from AT&T, DEC, Hewlett Packard, IBM, Intel, and Valid Logic Systems.

The work was done by the following members of the *DADO* project: Michael van Biema, Jens Christensen, Andrew Comas, Eugene Dong, Beverly Dyer, Douglas Gordin, Mark D. Lerner, Andy Lowry, Janet Lustgarten, Gerald Q. Maguire Jr., Chris Maio, Russell Mills, Daniel P. Miranker, Alexander Pasik, Richard Reed, Pandora Setian, Salvatore Stolfo, Steve Taylor, Nancy Yee, Philip Yuen.

REFERENCES

1. Stolfo S. J., and D. E. Shaw. *Specialized Hardware for Production Systems.* Department of Computer Science, Columbia University, August, 1981.
2. Stolfo, S. J. "Knowledge Engineering: Theory and Practice." *Proceedings of IEEE Trends and Applications*, Gaithersburg, MD, 1983.
3. Stolfo, S. J., D. P. Miranker, and D. E. Shaw. "Architecture and Applications of DADO: A Large Scale Parallel Computer for Artificial Intelligence." *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, Karlsruhe, West Germany: IJCAI, August 1983.
4. Stolfo, S. J. *On the Design of Parallel Production System Machines: What's in a LIP?* Department of Computer Science, Columbia University, 1984.
5. Stolfo, S. J., and D. E. Shaw. "DADO: A Tree-Structured Machine Architecture for Production Systems." *Proceedings of AAAI-82*. Pittsburgh: Carnegie-Mellon University, 1982.
6. Stolfo, S. J. *The DADO Parallel Computer.* Technical report, Department of Computer Science, Columbia University, August, 1983.
7. Stolfo, S. J., and D. P. Miranker. "DADO: A Parallel Processor for Expert

Systems." *Proceedings of the 1984 International Parallel Processing Conference*. Michigan, IEEE Computer Society Press, 1984.

8. Shaw, D. E., S. J. Stolfo, H. Ibrahim, B. Hillyer, G. Wiederhold, and J. A. Andrews. "The NON-VON Database Machine: A Brief Overview." *Database Engineering* 4, 2 (December 1981).
9. Browning, S. "Hierarchically Organized Machines." In C. Mead and L. Conway (eds.), *Introduction to VLSI Systems*. Reading, Mass.: Addison-Wesley, 1978.
10. Flynn, M. J. "Some Computer Organizations and Their Effectiveness." *IEEE Transactions on Computers*, Vol. 25 (September 1972).
11. Miranker, D. P. *Performance Analysis of DADO by Queueing Network Analogy*. Department of Computer Science, Columbia University, November 1983.
12. Taylor, S., C. Maio, S. J. Stolfo, and D. E. Shaw. *PROLOG on the DADO Machine: A Parallel System for High Speed Logic Programming*. Department of Computer Science, Columbia University, 1983.
13. Stolfo, S. J., D. Miranker, and M. Lerner. *PPLM: The Systems Level Language for Programming the DADO Machine*. Department of Computer Science, Columbia University, 1984.
14. Taylor, S., D. Tzoar, and S. J. Stolfo. *Unification in a Parallel Environment*. Department of Computer Science, Columbia University, 1984.
15. Taylor, S., A. Lowry, G. Maguire, and S. J. Stolfo. "Logic Programming Using Parallel Associative Operations." *Proceedings of the International Logic Programming Symposium*. Atlantic City, N.J., IEEE Computer Society Press, February, 1984.
16. Taylor, S. *LPS, A Logic Programming System: Motivations and Goals*. Department of Computer Science, Columbia University, 1984. (In preparation)
17. van Biema, M., M. D. Lerner, G. Q. Maguire, and S. J. Stolfo. *PSL: A Parallel LISP for the DADO Machine*. Department of Computer Science, Columbia University, February, 1984.
18. Lowry, A., S. Taylor, and S. J. Stolfo. "LPS Algorithms: A Detailed Examination and Critical Analysis." *Proceedings of the International Conference on Fifth Generation Computer Systems*. Tokyo: Proc. Institute for New Generation Computer Technology, November 1984.
19. Lerner, M., M. van Biema, and G. Q. Maguire Jr. *A LISP Compiler for the DADO Parallel Computer*. Department of Computer Science, Columbia University, November 1984.
20. Gupta, A. *Implementing OPS5 Production Systems on DADO*. Department of Computer Science, Carnegie-Mellon University, 1984.
21. Miranker, D. *HeraAI, A Production System for the DADO Machine*. Department of Computer Science, Columbia University, 1984. (In preparation)
22. Pasik A., D. Gordin, L. Finkel, and J. Lustgarten. "XNET AIS: An Expert System that Recommends Treatment for Adolescent Idiopathic Scoliosis." Unpublished course project, Columbia University, Department of Computer Science, 1984.
23. Intel Corporation. *PL/M-51 User's Guide for the 8051 Based Development System*. Intel, 1982. Order Number 121966.
24. Griss M. L., and A. C. Hearn. "A Portable LISP compiler." *Software—Practice and Experience*, 11 (June 1981), pp. 541–605.
25. Griss, M. L., E. Benson, and G. Q. Maguire Jr. "PSL: A Portable LISP System." *Symposium on LISP and Functional Programming*. ACM, Pittsburgh, August 1982, pp. 88–97.
26. Newell, A. "Production Systems: Models of Control Structures." In W. Chase (ed.), *Visual Information Processing*. New York: Academic Press, 1973.
27. Rychener, M. *Production Systems as a Programming Language for Artificial Intelligence*. Ph.D. thesis, Carnegie-Mellon University, Department of Computer Science, 1976.
28. Stolfo, S. J. *A Note on Implementing OPS5 Production Systems on DADO*. Department of Computer Science, Columbia University, June 1984.
29. Pasik A., and M. Schorr. "Table-driven Rules in Expert Systems." *SIGART Newsletter*, 87 (January 1984), pp. 31–33.
30. Vesonder, G. T., S. J. Stolfo, J. Zalinski, F. Miller, and D. Copp. "ACE: An Expert System for Telephone Cable Maintenance." *Proceedings of the International Joint Conference on Artificial Intelligence*. Karlsruhe, West Germany: IJCAI, August 1983.
31. Vesonder, G. T., S. J. Stolfo, J. Zalinski, F. Miller, and J. Wright. "The Application of Artificial Intelligence Technology for Managing the Local Telephone Network." *Proceedings of the International Conference on Computer Communication*. Australia, November 1984.
32. Forgy, C. L. *OPS5 User's Manual*. Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, July 1981.
33. Miranker, D. P. "Performance Estimates for the DADO Machine: A Comparison of TREAT and RETE." *Proceedings of the International Conference on Fifth Generation Computer Systems*. Tokyo: Proc. Institute for New Generation Computer Technology, November 1984.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS: NONE
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: APPROVED FOR PUBLIC RELEASE. DISTRIBUTION UNLIMITED.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: COLUMBIA UNIVERSITY; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: NAVELEX
6c. ADDRESS (City, State, and ZIP Code): 450 Computer Science Building, Columbia University, New York, NY 10027	7b. ADDRESS (City, State, and ZIP Code): 2511 Jefferson Davis Highway, Arlington, VA 22202
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: DARPA; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code): 1400 Wilson Boulevard, Arlington, VA 22209	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO; PROJECT NO N00039-84-C-0165; TASK NO. 2; WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)

An Overview of the DADO Parallel Computer

12. PERSONAL AUTHOR(S)

Lerner, M. D., G. Q. Maguire Jr. and S. J. Stolfo

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
SPECIAL	FROM 3/85 TO 6/85	1985, June 12	8

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	DADO, Production Systems, Parallel Computer, Fifth Generation, Logic Programming, AI, LISP.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

DADO is a special purpose parallel computer designed for the rapid execution of artificial intelligence expert systems. This article discusses the DADO hardware and software systems, with emphasis on the question of *granularity*.

DADO is designed as a fine-grain machine constructed from many thousands of processing elements (PEs) interconnected in a complete binary tree. Two prototype systems, DADO1 and DADO2, are detailed. Each PE of these prototypes consists of a commercially available microprocessor chip, memory chips, and an additional semicustom I/O processor designed at Columbia University. The software includes a kernel and parallel languages. Under development are several artificial intelligence systems, including a production system interpreter, a logic programming language and an expert system building tool.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Salvatore J. Stolfo	22b. TELEPHONE (Include Area Code): (212) 280-8111; 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE