CUCS-46-83

PROLOG on the DADO Machine:

A Parallel System For High-Speed Logic Programming*

Stephen Taylor
Christopher Maio
Salvatore J. Stolfo
David E. Shaw

Columbia University
Department of Computer Science
January 15, 1983

Abstract

DADO is a highly-parallel, VLSI-based, tree-structured machine designed to provide significant performance improvements in the execution of large *production system* programs. In this paper, we describe current research aimed at implementing PROLOG within the parallel framework which DADO provides. The implementation allows parallel satisfaction of both disjunctions and conjunctions which occur in the goal tree generated during the execution of a PROLOG program. Local unification routines in each processor allow parallel satisfaction of disjunctive goals while a parallel relational join operation provides a framework to solve conjunctive subgoals. An overview of the techniques currently being implemented and their relationship to the architecture is presented.

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

1. Introduction

PROLOG [2] [10] is a simple goal-oriented programming language, based on symbolic logic, which is well suited to execution on parallel hardware. PROLOG programs consist of a logic component, supplied by the programmer, and a distinct control component implicit in the semantics of the language. The logic component involves a set of statements in predicate calculus, called clauses, which describe the domain of the program. We shall concern ourselves with two types of clauses, informally referred to as *facts* and *rules*. In general, a fact consists of a single *atomic formula* of the first order predicate calculus which may be interpreted as a simple statement of truth. An atomic formula or *literal* consists of a predicate symbol applied to first-order terms which include constants, *universally-quantified* logical variables and skolem functions recursively applied to terms. For example,

likes(mary, logic). (1)

may be read declaratively as

mary likes logic.

A rule consists of a set of atomic formulae, which convey procedural information. The left-hand-side of the rule always consists of a single atomic formula, called the *head*, while the right-hand-side, or *body*, involves one or more formulae separated by logical connectives (conjunction is denoted by "," and disjunction by ";"). For example,

logician(X) :- human(X), likes(X, logic). (2)

is read declaratively as

X is a logician if X is human AND X likes logic.

Similarly,

logician(X) :- teaches(X, logic). (3)

is read declaratively as

X is a logician if X teaches logic.

PROLOG has a set of procedural semantics which define how a program is to be executed. Under this interpretation, terms in the body of a clause constitute subgoals which must be examined in order to satisfy the head goal. Thus, clauses 2 and 3 are read procedurally as

> *to satisfy the goal of finding if X is a logician, satisfy the goal of finding if X is human AND satisfy the the goal of finding if X likes logic, OR satisfy the goal of finding if X teaches logic.*

PROLOG employs an inference mechanism called *unification* [5] in order to satisfy goals. This may be viewed as a general pattern matching algorithm, with special significance given to the logical variable. The use of unification allows PROLOG to infer new facts from facts that already exist in the PROLOG database.

1.1 The PROLOG AND/OR Goal Tree

The control strategy employed by PROLOG has been referred to in the literature of artificial intelligence as *backward-chaining, goal-directed* execution. Given a goal to be satisfied, PROLOG examines all rules whose head unifies with the goal in question. The constituent elements of the bodies of the matching rules are then proposed as subgoals to be achieved. Thus, each rule contributes a conjunctive set of subgoals (AND goals) while the entire set of unified clauses collectively contribute a disjunctive set of subgoals (OR goals). Branches of the resulting AND/OR goal tree, which may be exhaustively generated in this fashion, are terminated by the presence or absence of unifiable facts in the initial PROLOG database. Many aspects of this control strategy lend themselves naturally to parallel execution; a PROLOG interpreter which exploits this characteristic of the language is presently under development at Columbia University for DADO, an experimental parallel computer system. We are also investigating the implementation of PROLOG on NON-VON [7], another highly parallel machine now under construction at Columbia; this work will not be discussed in this paper, however.

1.2 The DADO Architecture

DADO [8] is a highly parallel tree-structured architecture based on VLSI technology. Although originally intended as a special purpose device for rapid execution of *production systems* [4], it is our belief that DADO can provide significant performance improvements over sequential machines in a wide range of Artificial Intelligence applications. The DADO prototype now under construction comprises 1023 processing elements (PE's) inter-connected to form a *complete binary tree.* Currently, each PE is implemented using an Intel 8751 microcomputer chip, an Intel 2186 8Kx8 RAM chip, and a special combinational I/O switch, implemented as a custom integrated circuit, which provides high-speed tree communication facilities. The full-scale version of the system, implemented entirely in custom VLSI, is expected to contain on the order of hundreds of thousands of PE's.

Each PE may operate in either *single instruction, multiple data stream* (SIMD) or *multiple instruction, multiple data stream* (MIMD) mode [3]. In SIMD mode. a PE executes instructions broadcast by some ancestor PE in the tree. In MIMD mode, a PE may execute instructions stored in its local RAM, independently of other PE's. A PE in MIMD mode may utilize the subtree below as a SIMD device, provided its descendants have been switched into SIMD mode. DADO can be configured in such a way as to logically divide the tree into a number of smaller machines, each acting as independent subsystems. In the envisioned PROLOG implementation, however, the operation of the entire DADO tree will be supervised by the *Control Processor* (CP) adjacent to the root node.

A framework for the execution of production system programs on DADO [8] is currently under development. Its running time depends not on the number of rules within the production system program, but rather on the size of the largest rule. Using similar techniques, the PROLOG implementation outlined in this paper will provide significant asymptotic performance improvements over implementations based on von Neumann machines.

It should be noted that the binary tree organization of DADO was chosen for reasons related to efficient implementation in VLSI. As is the case with many highly parallel DADO algorithms, the DADO tree structure has no direct relevance to the PROLOG algorithm outlined in this paper.

Details of the DADO architecture have been reported in a companion paper [9].

2. A Parallel Implementation of PROLOG

For simplicity, we will assume in this paper that one PE in the DADO tree is allocated to each clause in the PROLOG database; using packing techniques analogous to those employed in the production system algorithm reported in [8], however, this assumption is easily relaxed at the expense of a modest cost in time.

During the evaluation of a PROLOG program, it is often necessary to communicate information between the Control Processor and various subsets of PE's within the DADO tree which are relevant to a particular computation. A high-speed, associatively-based mechanism which allows certain distinguished sets of PE's to be chosen and information to be broadcast to them is available [8]. The algorithms outlined in the following sections make extensive use of these facilities in order to effect the high-speed execution of PROLOG programs. As noted earlier, various features of the underlying control strategy employed by PROLOG lend themselves naturally to parallel execution. In the following sections, we outline the various parallel techniques utilized in the satisfaction of both OR- and AND-levels which occur within the PROLOG goal tree.

2.1 Parallel Computation at OR-levels within the Goal Tree

As noted earlier, given a goal to be satisfied, PROLOG examines all clauses whose head unifies with the goal in question. These clauses collectively form a disjunctive set of subgoals for the goal under consideration and occur at an OR-level in the PROLOG execution tree. The semantics of this operation provide the opportunity for the unification process to be carried out independently, and in parallel, for each clause. In the DADO implementation, a program resident within each PE is used to compute a set of variable bindings for a goal broadcast from the Control Processor.

After successful unification, PE's containing facts which unify with the goal in question may immediately be marked as contributing to the solution set for the goal. In order to maintain PROLOG's procedural semantics, this marking process is carried out in such a way as to preserve the order in which the clauses appear in the database. Rules whose head goal unified successfully must be considered in the same order, and will thus be executed *sequentially*.

Prior to the execution of successfully unified rules, variable instantiations made during unification are substituted for the corresponding variables in the rule body. This operation occurs independently in each of the rules in question and therefore can also be carried out in parallel. Execution of a rule involves transmitting the body of the rule, along with any variable instantiations made during unification, to the Control Processor. The rule body,

which may contain a subgoal or conjunction of subgoals, may then be satisfied recursively. Having computed all results for the body of a rule, it is necessary to convert the results to a form consistent with the rule head. This operation requires a back-substitution for variables occurring in the goal that originally unified with the rule. Since all results are converted in the same manner, this process may be carried out in parallel.

The solutions which result from the satisfaction of rules, together with those contributed by facts marked during the original unification, constitute all possible results for a given goal.

2.2 Parallel Computation at AND-levels within the Goal Tree

A conjunctive goal is solved by a relational *equi-join* operation [1] applied to the partial solutions, generated sequentially, of its constituent elements. Solutions to a given goal are represented by sets of bindings for variables occurring in the goal. When all solutions for each constituent subgoal have been accumulated, inconsistent bindings of variables which are common to subgoals are eliminated by this join process. Those solutions remaining form the result for the conjunctive goal.

While space does not permit a full exposition of the general theory which allows this elimination to be carried out efficiently, readers familiar with the literature of relational database systems, and in particular, database machines, may find the following brief comments illuminating. First, we note that the set of facts represented in the database may be regarded as comprising several *relations*, each the *extension* of some goal literal. Viewed in this way, the elimination of binding conflicts may be carried out by applying a relational join algorithm to sets of variable bindings for goals occurring in the conjunction under consideration.

On a conventional von Neumann architecture, this algorithm is expensive to compute; however, on DADO, it can be computed optimally, in time which is strictly proportional to:

- the size of the smallest set of results for any goal involved in the join,
- the number of common variables joined over, and
- the size of the result set.

The result of this operation is a new relation which embodies all consistent instantiations for variables occurring in the conjunction. This algorithm is based on a technique described in a doctoral dissertation by Shaw [6].

3. Current and Future Research

Although originally designed as a special purpose device for the execution of production system programs, DADO can support the high-speed execution of a considerably larger class of AI applications. Other applications currently under study include parallel implementations of frame-based systems, as well as methods for improving the performance of conventional AI languages such as LISP.

Presently, we are implementing the algorithms presented within this report on a small, fifteen-element prototype of DADO that will shortly become operational.

The authors are also interested in developing new languages, based on logic, which may better utilize parallelism. The reduction of the number of imperative constructs and the separation of logic and control components in programming languages seems to allow a natural transition toward the specification of highly parallel languages.

4. Conclusions

In the parallel algorithm outlined in this paper, the conventional procedural semantics of PROLOG have been preserved in order to support the inherently sequential constructs available to the programmer. The natural embedding of parallel constructs has been employed wherever possible without reducing the generality of the language. The complexity of the algorithm for finding all solutions for a goal is reduced in the following ways:

- Parallel unification of clauses (OR-levels).

- Parallel resolution of binding conflicts (AND-levels).

The algorithms outlined in this paper run in time dependent on the number and size of rules activated during execution of a given program.

Acknowledgments

The authors would like to extend their thanks to Rod Farrow and Monnett Hanvey, whose critical insight proved invaluable during the early stages of this research.

References

[1] **E. F. Codd.**
Relational Completeness of Data Base Sublanguages.
Courant Computer Science Symposium 6: Data Base Systems , 1972.
R. Rustin (ed.), Prentice-Hall, Inc.

[2] P. Roussel.
Prolog : Manuel de Reference et d'Utilisation.
Technical Report, Marseille-Luminy, 1975.

[3] M. Flynn.
Some Computer Organizations And Their Effectiveness.
IEEE Transactions On Computers vol. C-21:pp. 948-960, September, 1972.

[4] R. Davis and J. King.
An Overview of Production Systems.
Technical Report, Stanford University Computer Science Department, 1975.
AI Lab Memo, AIM-271.

[5] J. A. Robinson.
A Machine-Oriented Logic Based on the Resolution Principle.
JACM vol 12, pp. 23-44 , 1965.

[6] D. E. Shaw.
Knowledge-Based Retrieval on a Relational Database Machine.
PhD thesis, Department of Computer Science, Stanford University, 1980.

[7] D. E. Shaw.
The NON-VON Supercomputer.
Technical Report, Department of Computer Science, Columbia University, 1982.

[8] S. J. Stolfo and D. E. Shaw.
DADO: A Tree-Structured Machine Architecture For Production Systems.
Proceedings of the National Conference on Artificial Intelligence , August, 1982.

[9] S. J. Stolfo, D. Miranker and D. E. Shaw.
Architecture and Applications of DADO, A Large-Scale Parallel Computer for Artificial Intelligence.
Technical Report, Department of Computer Science, Columbia University, January, 1983.

[10] D. H. D. Warren.
Implementing Prolog - Compiling Predicate Logic Programs.
PhD thesis, Department of Artificial Intelligence, Edinburgh University, 1977.