CUCS-113-84

LPS Algorithms: A Critical Analysis

Andy Lowry
Stephen Taylor
Salvatore J. Stolfo

Columbia University
Department of Computer Science

March 1984

Abstract

We are interested in high speed execution of logic programs using a particular model of parallel computing devices. In a companion paper we present a detailed explanation of current implementation strategies and develop an abstract proof procedure encompassing those algorithms as well as the standard sequential Prolog algorithm. Here we explore a number of alternative strategies and provide a comparative analysis in the context of that abstract procedure. We investigate issues of performance, storage requirements, and applicability of the algorithms under various assumptions concerning the program under execution.

We assume that the reader is familiar with the material developed in the companion paper.

This research is supported cooperatively by: Defense Advanced Research Projects Agency under contract N00039-82-C-0427, New York State Science and Technology Foundation, Intel Corporation, Digital Equipment Corporation, Valid Logic Systems Inc., Hewlett-Packard, AT&T Bell Laboratories and International Business Machines Corporation.

1. Introduction

We have proposed a computing model which we believe may be suitable for the parallel execution of logic programs [5, 7]. In a companion paper [2] we develop detailed algorithms by which this might be achieved. Our development is cast in the framework of an abstract proof procedure [2] which encompasses our algorithms (the *LPS algorithms*[a]) as well as the standard sequential Prolog algorithms [8]. In this paper we investigate several alternative parallel execution strategies and analyze them in the context of this same abstract proof procedure.

We include discussion of the trade-offs among various execution strategies in terms of performance, storage requirements, and appropriateness to various types of logic programs. Much of the analysis presented here is intuitive in nature, due to a lack of observed performance measurements. Meaningful measurements are difficult to obtain because:

- Our current implementation is in the form of a simulation on a sequential machine, so that sample execution of any but the tiniest programs is prohibitively expensive. Implementations on a functioning parallel machine are currently underway.

- The algorithms do not as yet provide for extensions to the Horn clause formalism such as negated condition elements, evaluable predicates, and goals with side-effects. These features are generally required by logic programs that attempt to do anything substantial and useful, so most existing programs cannot be executed in our current framework.

It is hoped that future work will remove these obstacles and allow for statistical analyses providing greater insight into the effects of the various strategies. This should in turn suggest opportunities for a more general mathematical analysis.

We assume an understanding on the part of the reader of the material developed in the companion paper [2]. No summary is attempted here.

---

[a] Named after LPS, the name we have chosen for our Logic Programming System

2. Unification Phase Strategies

Two strategies have been identified for the unification of goals in a goal list, which we call *asynchronous* and *synchronous unification*.

2.1 Asynchronous Unification

In the asynchronous case, a goal list is broadcast as a single unit to the PE network, and the PE's are instructed to go to work unifying the entire list of goals. The CP waits until all PE's have completed this task, at which point all possible unifications of the goals have taken place, and the resulting binding sets are resident in the PE network. This strategy allows overlapping of goal unification among the individual PE's. That is, each PE moves on to the next goal as soon as it has exhausted its own local supply of literals with which to attempt unification of the current goal, regardless of the state of progress in the other PE's.

As an example, consider the following somewhat idealized scenario:

Goals to be unified: a, b

Literals resident in PE 1: a_1.

Literals resident in PE 2: b_1.

The following sequence of events results:

The CP broadcasts the goal list '{a, b}' to the PE network.

PE 1 begins unifying $<a, a_1>$

At the same time PE 2 begins unifying $<a, b_1>$, fails quickly and progresses to unify $<b, b_1>$

PE 1 completes unifying $<a, a_1>$, attempts to unify $<b, a_1>$ and fails quickly

PE 2 completes unifying $<b, b_1>$

PE 1 and PE 2 have completed the unification phase

As we see, unification of goal a in PE 1 is overlapped in time with unification of goal b in PE 2. Beneficial overlapping occurs for two reasons:

- A successful unification generally requires more time than an unsuccessful attempt. Failure is usually detected long before the two literals have been completely scanned (indeed, failure is immediate in the case of different predicate symbols), whereas success is not recognized until the scan is complete. Furthermore, additional work is required after a successful unification, for the construction of a binding set.

- One PE may need to attempt unification of a particular goal with more literals than another PE. If we assume a very small number of literals resident in each PE (due to a large PE network), we can expect that most PE's will be unable to unify most goals, so this will be a high source of overlap. Even without this assumption, various strategies for distributing the literals and indexing each PE's local literal pool by predicate symbol can increase the likelihood of this type of overlap.

Again assuming a very small number of literals resident in each PE, we see that the entire unification phase takes time that is linear in the size of the broadcast goal list. Furthermore, we expect the entire process to be exceptionally fast due to a small constant factor in our linear complexity. Contributing components are: (1) the time required to transmit the goal list, and (2) the time required for the PE's to individually scan the goal list and create binding sets for successful unifiers. The second component is linear because the basic unification algorithm is linear in the size of the terms being matched, and the sum of those sizes is no larger than the size of the entire goal list.

Generally, we would expect a single literal to unify with at most one goal in a goal list, so that the constant factor in our linear time complexity will be heavily dominated by the time for failure, rather than the time for successful unification. This accounts for the high performance we expect from asynchronous unification, since failure time is quite small.

2.2 Synchronous Unification

In the synchronous unification strategy the goals in a goal list are broadcast one at a time rather than as a single unit. Unification of each goal is performed in the PE's before the next goal is broadcast, so that none of the overlapping that we witness in the asynchronous strategy can occur.

The synchronous strategy offers a potential benefit only in the case of the failure of a single goal throughout the entire PE network. In this case, the entire goal list can be thrown out immediately without attempting unification of the remaining goals.

Whether or not such opportunities arise with a frequency that merits adoption of a synchronous unification strategy is a question that will be investigated through statistical analyses of logic programs [6]. We hope also to develop methods for identifying local characteristics of a search space that may indicate an increased likelihood for global failure of a single goal. If this can be done, a dynamic selection mechanism may be implemented that is capable of using asynchronous or synchronous unification depending on the proof history and current state.

A hybrid strategy may also be envisioned, in which the goal list is partitioned according to some suitable heuristic, and each portion is broadcast as a unit for asynchronous unification, while unification of the overall goal list is synchronous among the portions.

3. Join Phase Strategies

We note that our parallel execution of a pair-wise relational join results in a log(n) improvement in the time required for this operation by a sequential algorithm [7]. Our alternative join strategies investigate methods for minimizing the number of pair-wise joins required in a single join phase, avoiding redundant computations, and controlling depletion of PE storage.

The reconciliation operation itself is performed as a series of refinements on a collection of bindings. The collection begins with the union of the bindings found in the two component substitutions. A refinement consists of identifying two bindings for the same variable and replacing one of them with the unifier of the two terms. When no such pair of bindings is left, reconciliation is complete.

Although pathological cases can be constructed, we believe that in practice the unification that takes place during reconciliation seldom produces new bindings for variables already bound, so that (again recalling that unification itself is linear in the size of the terms being combined) we will expect a performance that is roughly linear in the size of the component substitutions. We intend to investigate the validity of our assumption through statistical analysis.

3.1 Retaining Goals

It has been pointed out that our LPS implementation never retains a goal from one goal list to its successor. Instead, each goal is either removed or expanded. We expect this strategy to be quite beneficial in many applications, however there are at least two potential pitfalls, which we call the big-small problem and the *cartesian product* problem.

3.1.1 The Big - Small Problem

Consider the goal set {big(X), small(Y)} where, as its name suggests, big(X) is a goal that represents an extremely lengthy computation involving long chains of inference. Likewise, small(Y) is a goal that is very quickly satisfied, with multiple solutions. It turns out that a strict policy of non-retention of goals will perform substantially more work in locating multiple solutions of this directive than would a more flexible approach.

To see this, consider the very first step in the solution of our goal, and suppose that our database contains the two facts **small(flea)** and **small(pebble)**. These two facts will unify immediately and produce simple bindings **[Y/flea]** and **[Y/pebble]**, respectively. Meanwhile, **big(X)** unifies with a rule head somewhere in the database, producing a complex binding that represents a long computation in its infancy.

With these binding sets in place, our join phase will produce two complex bindings, both containing the status of the just-started big computation, and each containing one of the small solutions. One of these complex bindings will be selected during the next cycle, and that selection will begin the long computation of **big(X)**.

Meanwhile, the second complex binding will lie dormant in the PE network awaiting selection but not benefiting from the ongoing computation. When it is finally selected, the big computation will be repeated almost in its entirety. This is the duplication of effort that we would like to avoid.

To see a possible way out, consider the behavior of this goal in the standard sequential algorithm. Here the **big(X)** computation is carried out until a solution for it is achieved, all the while retaining the original **small(Y)** goal in the goal list. It is not until the big computation has terminated in a solution that the small goal is finally unified, resulting in its removal by the fact **small(flea)**. Next the algorithm backs up its computation to its last choice point, which was its choice of a unifying fact for **small(Y)**, and makes a new choice. This time the **small** goal is removed by the fact **small(pebble)** *without having to recompute the current solution for* **big(X)**.*

Thus we see that by *retaining* the small goal we have avoided a large redundant computation.

It is instructive here to consider the tree of goal lists generated by our proof procedure, in which the decision to retain a goal at a certain point has the effect of postponing whatever branching might be caused by the choice of clauses with which the goal may unify. If each of the resulting branches gives rise to a deep subtree, the postponement may turn out to be quite beneficial. The case of our example is diagrammed in figure 3-1.

---

*Note that the Prolog algorithm would encounter the big-small problem if the goal list were reversed, as in {small(Y),big(X)}.

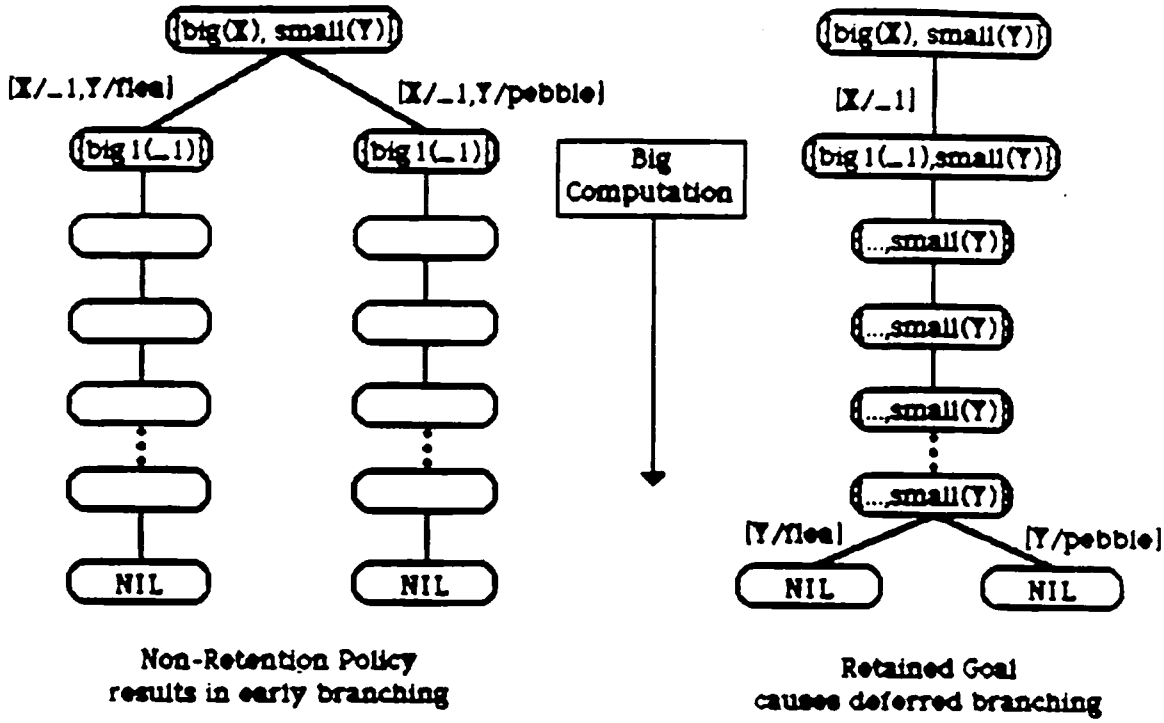

Figure 3-1: Goal retention postpones branching and may result in considerable savings of effort in the big - small problem

3.1.2 Cartesian Products

One other possible benefit of goal retention is containment of the potentially explosive growth in the number of binding sets resident in the PE network. As an example, suppose our goal list is {**plentiful1(X), plentiful2(Y)**}, where each goal unifies with a very large fact base (say M facts for plentiful1 and N facts for plentiful2). Thus in one proof step we achieve two large, independent relations, whose join is their complete cross product consisting of M x N binding sets. In such a situation it might be desirable (or even necessary) to limit the accumulation of binding sets by generating only one "slice" of the cartesian product at a time.

This might be accomplished by retaining **plentiful2(Y)** during the first cycle. The result will be M binding sets containing the solutions for **plentiful1(X)**, and each containing our retained goal as well. These binding sets are selected one by one for further elaboration, and each one gives rise to N bindings sets that are reported and discarded in turn. The maximum number of binding sets resident in the network is thus M + N - 1, rather than M x N.

3.1.3 Implementation Of Goal Retention

Two problems need to be addressed if goal retention is to be accommodated in our algorithms:

1. The actual mechanisms for retaining goals must be worked into the implementation. This requires a slight modification of the binding set representation so that actual goals can be represented, as well as mechanisms that allow the CP to identify to the PE's which of the broadcast goals are to be retained.

2. The means by which goals are selected for retention must be decided. Possibilities include automatic selection based on static and/or dynamic program analysis; marking of procedures, rules, or even individual condition elements by the programmer; and combinations of these two strategies. We prefer a completely automatic mechanism, consistent with the philosophy that logic programming offers opportunities for parallelism without burdening the programmer with this goal.

3.1.4 Benefits Of A Non-Retention Policy

It should be noted here that a policy of non-retention of goals provides at least two potential benefits.

First, the total path length for any successful proof is minimized by such a policy, generally translating into reduced effort for a single proof. As the big-small problem illustrates, however, situations may easily arise in which much greater benefits due to commonality of proof paths are missed by this eager strategy.

Second, a retained goal does not constrain the search space under consideration. One benefit of the reconciliation model over the depth-first search strategy of the Prolog algorithms is that a much larger range of interactions are possible among the goals in a single goal list. In the Prolog strategy, the effects of computations on a goal may only propagate forward in the goal list, whereas if several goals are unified in one step, constraining interactions are carried in both directions. The program presented below is an example where the Prolog algorithms will never terminate, whereas a non-retention strategy terminates quite quickly. Here the second goal in the goal list constrains the first goal so as to avoid the infinite search that the first goal produces on its own. Since this "backward" constraint is not possible in the Prolog algorithm, we find the unconstrained first goal generating an infinite sequence of results, all but the first of which are disallowed by the second goal.

Rule 1: append(cons(A,T1),L,cons(A,T2)) <- append(T1,L,T2)
Fact 1: append(NIL,L,L) <-

Directive: <- append(X1,X2,Y1) ' append(Y1,Y2,NIL)

3.2 Single Feed Joins

The cartesian product problem mentioned in the last section is just one particularly severe case of the general problem that our parallel execution model may tend to accumulate binding sets that are waiting for selection. Goal retention was seen as one strategy for alleviating this problem by expanding the search space one "slice" at a time.

Another strategy is to perform our join operations in small steps by broadcasting only one feeder relation member to the consumer relation at a time. The binding sets produced by that single feeder are processed one by one until they run out, at which point the next feeder from the suspended join is broadcast.

This single-feed strategy offers a second possible benefit aside from containment of the binding set population. In many cases, a query will be presented with the intention of producing only a single solution, rather than pursuing all possible solutions. In this case, much of the effort that goes into our join operations will be wasted since if a solution is encountered early in the search space, a large percentage of the binding sets generated from joins will be discarded. The single-feed strategy defers this effort until it is required in order to continue the search.

It is expected that the implementation of a single feed strategy will require considerably more complicated control mechanisms than are needed for the eager join strategy. At this point in time no such implementation has been attempted, nor has careful thought been given as to the exact control mechanisms that would be required.

3.3 Redistribution of Binding Sets

One final approach to the problem of explosive growth in the binding set population takes a more local view. Specifically, what can be done about the case where binding sets begin accumulating at a few "hot spots" in the PE network?

In such a situation it would be beneficial to have a mechanism available whereby heavily loaded PE's could export some of their binding sets to other PE's. Such a mechanism is difficult to imagine in our computing model since all communication must be funneled through the CP. If, however, some direct PE-PE communication mechanism is provided* efficient redistribution might be realizable. We may even envision redistribution within the PE network overlapping computational tasks within the CP, such as the construction of a new goal list from a reported binding set.

*Such a mechanism is available, for example, in the DADO binary tree architecture, in which tree neighbors may communicate without burdening the CP [3, 4].

3.4 Rule Layer Caching

We discuss one other join phase variant in which rule layers are stripped from binding sets whenever they pass through the CP and are replaced by unique tags. The rule layers are stored in the CP and are retrievable via their tags. The advantage of this scheme is reduced communication of feeder binding sets during the pair-wise join operation. The strategy is justifiable on the basis that no use is made of rule layers except in the CP, so that they are little more than "excess baggage" in the binding sets during the join phase.

One major drawback of this scheme, however, is that it precludes the removal of common layer bindings from binding sets during the join. This is impossible because any common layer binding might be needed in order to update the instantiators in the binding set, and instantiators that are start out identical may in this way end up differing in their final form. In order to ensure correct updating of instantiators before instantiation, then, the common layer bindings must be fully maintained and reported to the CP along with the binding set.

In addition, such a scheme would probably require some method for determining when a rule layer may be discarded by the CP owing to all referencing binding sets having been reported and elaborated. Such a mechanism seems feasible given the current sequence number scheme.

The trade-offs involved have not yet been studied, although there seems to be reason to suspect that overall communication costs will not be greatly affected, the two effects largely cancelling each other.

3.5 Multiple Independent Joins

A particularly intriguing prospect for optimization of the join phase is the idea of performing two or more pair-wise joins in unison in the PE network. Our standard join algorithms may be adapted for this purpose by extending our model of computation to include a facility for temporarily partitioning the PE network into independently functioning subnetworks. One PE in each subnetwork would act as CP for the subnetwork.* With such a facility, our pair-wise join may be migrated to the subnetworks, so that several pairs of relations may be joined simultaneously. This strategy requires that the each subnetwork contain each relation to be joined, in its totality.

As an example of how a multiple join strategy might be realized, and to illustrate the potential savings,

*The DADO architecture [3, 4], for example, allows for such a "multiple SIMD" execution mode.

we consider a rather "brute force" approach. The PE network is divided into two subnetworks, and each fact and rule head is stored once in each subnetwork.

The unification phase will produce twice as many binding sets as in our standard model, each binding set appearing in both subnetworks. We note that the effort expended during the unification will be doubled in worst case, since the concentration of literals in the PE's has doubled so that each PE's scan of literals will take twice as long.

The join phase proceeds in two stages. During the first stage, one of the PE subnetworks joins half of the relations resulting from the unification phase while the other subnetwork simultaneously joins the other relations. The second phase is a single pair-wise join performed by the CP in the standard fashion, combining the results of the subnetwork joins. The total effort required by the join phase starting with n relations is that required for $n/2$ pair-wise joins, as compared with $n-1$ pair-wise joins if multiple independent joins are not utilized.

If we consider the overall savings realized by multiple joins in the above scenario we see that while the time for unification has been doubled in worst case, we have halved the time required in the join phase. In the case of communication costs, we see that there has been no increase during unification, whereas costs have been halved during the join phase. The situation is promising, to say the least. Further analysis may be able to identify more intelligent partitioning strategies, possibly based on data dependency analyses similar to those under investigation by Ishida [1] in his work on parallel execution of production systems.

4. Substitution Phase Strategies

The only major alternative under consideration for the substitution phase is the postponement of the composition of individual reconciliations in the proof path. Rather than keeping a completely updated reconciliation in each binding set, common layers would represent only the substitution required to complete the last proof step. The overall substitution would be computed by the CP whenever a solution was encountered.

The only substantial benefit that may be obtained from this strategy is that the entire substitution history, along with a history of goal sets that could also be maintained by the CP, would allow the reconstruction of the entire proof for reporting purposes. The drawbacks are several:

- The history mechanism required in the CP appears substantially more complicated than what is presently required. Binding sets would need additional tagging information to identify depth in the search space, and the CP's history mechanism would have to monitor this information in order to know whether to stack a new component, replace the top component, or pop the stack.

- The history mechanism would seem to prevent much flexibility in the order of selection of binding sets. A predictable order of traversal through the search space is potentially beneficial to programmers. The history mechanism would fit well into the ordering imposed by our current sequence number scheme, but as indicated in [2], it is questionable whether this ordering is useful. We hope to be able to identify a different ordering that fits well into the algorithms, but a history mechanism would severely constrain our options.

- We would no longer be able to remove common layer bindings prior to reporting the binding set to the CP.

4.1 A Previous Implementation

Earlier published work on LPS described a substitution phase that is substantially different from those currently under consideration. In fact, in early implementations the substitution phase was probably the most complex phase of the algorithm. The current approach is a direct result of investigations prompted by discontent with the earlier techniques. For historical completeness we briefly discuss this approach and relate it to current work.

The task of the substitution phase can be regarded as pushing forward a *frontier set* of bindings. Prior to a proof cycle, we are equipped with a collection of bindings that relate our top-level variables to variables in the rules about to be fired, as well as various created variables. We call these variables *middle-level* variables. As a result of unification and reconciliation, we are left with another collection of bindings, this time between middle-level variables and *bottom-level* variables, which are variables from the facts and rule heads with which our goals unified. The substitution phase must resolve these two collections into a new collection of bindings relating the top-level variables to the bottom-level variables. During the next proof step, of course, those bottom-level variables play the role of the middle-level variables, and the frontier set is advanced one more level.

The innovation that has allowed us to discard our old substitution algorithm is the filling out of instantiators with "dummy bindings" for unbound rule variables. As a result of this operation, our new frontier set and instantiator fall directly out of the composition procedure. Previously our approach was as follows:

1. Classify bindings into five different categories, as follows:

- Upper level variable bound to lower level variable
- Upper level variable bound to lower level ground term
- Upper level variable bound to lower level non-ground term
- Lower level variable bound to upper level ground term
- Lower level variable bound to upper level non-ground term

2. List all possible combinations of a top-to-middle binding of one type and a middle-to-bottom binding of another type. The resulting set of twenty-five binding scenarios, along with the five cases where a top-to-middle binding is left by itself (unpaired with a middle-to-bottom binding) comprise all possible binding interactions.

3. Consider each binding interaction in turn and decide how it can be recognized and what contributions it can make to the resulting binding set.

4. Develop an algorithm to handle interactions according to the analysis just performed.

We do not intend to consider this approach further.

5. Conclusions

We have presented a number of alternatives for execution of logic programs on a particular model of parallel computation. Although we have been able to identify some trade-offs, it is apparent that no single choice of strategies will be optimal in all circumstances. Future research aims to further our understanding of these and other algorithms and to identify characteristics of logic programs that may be used as a criterion for strategy selection.

Statistical analysis of logic programs is currently underway [8]. We hope to identify search space characteristics that arise frequently in practice and which dictate particular strategy choices. This analysis will be further aided by the implementation of our algorithms on a functioning prototype machine based on the DADO architecture.

A factor that has not been considered in this paper but which will affect our algorithm choices to a great extent is the need to implement extensions to the horn clause formalism into LPS. Such extensions as negated condition elements, evaluable predicates, and goals causing I/O or other side effects will be necessary if LPS is to be useful as a programming language in a wide variety of applications.

References

1. Ishida, Toru, Salvatore J. Stolfo. Simultaneous Firing of Production Rules on Tree Structured Machines. Columbia University, New York City, March, 1984.

2. Lowry, Andy, Stephen Taylor, Salvatore J. Stolfo. LPS Algorithms: A Detailed Examination. Columbia University, New York, NY 10027, March, 1984.

3. Stolfo, S. J. and Shaw, D. E. "DADO: A Tree-Structured Machine Architecture For Production Systems." *Proceedings of the National Conference on Artificial Intelligence Vol. 1* (August 1982).

4. Stolfo, S. J., Miranker, D. and Shaw, D. E. Architecture and Applications of DADO, A Large-Scale Parallel Computer for Artificial Intelligence. Proceedings of the Eighth International Joint Conference on Artificial Intelligence, International Joint Conferences on Artificial Intelligence, Inc., Karlsruhe, West Germany, August, 1983, pp. 850-854.

5. Taylor, S., Lowry, A., Maguire, G. Q. Jr., Stolfo, S. J. Logic Programming using Parallel Associative Operations. 1984 International Symposium on Logic Programming, Atlantic City, February, 1984, pp. 58-68.

6. Taylor, Stephen, Andy Lowry, G. Q. Maguire Jr., Salvatore J. Stolfo. Analyzing Prolog Programs. Columbia University, New York, NY 10027, March, 1984.

7. Taylor, S., C. Maio, S.J. Stolfo, D.E. Shaw. Prolog On The DADO Machine: A Parallel System for High-Speed Logic Programming. Third Annual International Phoenix Conference On Computers And Communications, IEEE, March, 1984.

8. Warren, D. H. D. Implementing Prolog - Compiling Predicate Logic Programs. Tech. Rept. D.A.I. 39/40, Department of Artificial Intelligence, Edinburgh University, May, 1977.