Architected Failure Handling for AND-Parallel Logic Programs

David M. Meyer
John S. Conery

CIS-TR-89-05
March 24, 1989

Department of Computer and Information Science
University of Oregon

Architected Failure Handling for AND-Parallel Logic Programs*

David M. Meyer
John S. Conery

University of Oregon
Eugene, Oregon

CIS-TR-89-05
March 24, 1989

Abstract

In this paper, we present an *architected* approach to failure handling for independent AND parallel logic programs. That is, the architecture presented here represents its failure handling algorithm as a sequence of simple abstract machine instructions, rather than as a built-in function. Information about data dependencies is used by a compiler to generate special purpose fail routines on a clause-by-clause basis. We also present two simple optimizations that further specialize the handling of failures for each clause.

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

*Supported by NSF Grant CCR-8707177

1 Introduction

One of the major features of logic programming languages is their ability to express nondeterministic computations. Standard sequential logic languages (*e.g.* Prolog) implement nondeterminism through chronological backtracking: when a subgoal fails, the system backtracks to the most recent subgoal with untried alternatives.

Chronological backtracking is not effective in AND-parallel systems, where the predecessors of a failed goal are solved in parallel, and backtracking to the most recently solved goal may not correct the conditions that led to the failure. There are several closely related methods for exploiting nondeterminism in independent AND parallel systems. Known as *semi-intelligent* backtracking schemes, they rely on a combination of static and dynamic information about producer/consumer relationships within a clause to determine how to retry a previously solved goal after a failure[1,2,3,7,12]. The algorithms are all very complex. When a literal consumes bindings created by more than one predecessor, and the predecessors operate in parallel, the backtracking scheme must coordinate *retry* and *reset* operations so the consumer literal sees all possible combinations of bindings from the predecessors.

As a result of this complexity, abstract architectures for AND parallelism have implemented semi-intelligent backtracking as a built-in operation, analogous to the way backtracking is done in sequential machines for Prolog[11]. These machines use extensive compile time analysis to create carefully optimized sequences of instructions for forward execution (unification and procedure calls or process creation), but treat all failures uniformly, by handling them with a common, built-in fail procedure implemented "inside" the architecture. We call this *non-architected* failure handling.

There are two problems with this approach. First, failure handling may not be as efficient as it can be. The general purpose failure mechanism is not subject to the same kind of compile time analyses and optimizations that are applied to other aspects of the architecture. Second, the general purpose failure routine is a barrier to efficient VLSI implementation of the architecture. The more complex the backtracking algorithm, the more difficult it will be to implement it efficiently on the processor chip, and the entire machine will be less efficient (the classic RISC argument).

In this paper, we present a method for *architected failure handling*, which does semi-intelligent backtracking for independent AND parallel logic programs. In our model, no general purpose failure handling algorithm is required. Instead, failures are handled by visible parts of the architecture, in the form of special purpose failure routines compiled for each clause. We compile a sequence of simple abstract machine instructions for each literal in a clause, to be invoked when the literal fails. The sequence of failure handling instructions is specialized for each goal, in the same way the combination of `put` and `get` instructions is unique for each clause during forward execution. The backward execution

code can be further optimized by some simple yet powerful optimizations, leading to very efficient implementation of semi-intelligent backtracking.

The rest of this paper is organized as follows: Section 2 gives a brief overview of the backward execution algorithm used by the new architecture. Section 3 describes the architectural features that implement the algorithm, along with compilation strategies and optimizations. Section 4 discusses the tradeoffs implicit in the architected failure handling model and future work.

2 The Backward Execution Algorithm

An AND process invokes its backward execution algorithm when it receives a *fail* message from a descendant, or a *redo* message from its parent. At this point, the AND process must select a body goal to retry, and resume forward execution. The selected goal is called the *backtrack literal.* If no appropriate backtrack literal can be found, the AND process fails. The purpose of a backward execution algorithm, then, is to select an appropriate backtrack literal, cancel or reset its successors, and set up for resumed forward execution.

The backtrack literal selection algorithm used by the architecture described in this paper was originally presented in [3]. This algorithm is based on a static data dependency graph, in which there is a node for each literal in a clause body, and an arc between literals i and j if i must be executed before j because the solution of i binds a variable occurring in j. We assume the graph can be derived from precise modes [2,10] or other compilation techniques.

Given a data dependency graph G that describes the producer/consumer relationship among literals in the body of a clause, the backward execution algorithm is described in terms of the following sets:

- $succ[i]$ – The projection of the second element of the transitive closure of the *successor* relation for literal i in G.
- $pred[i]$ – Defined analogously to $succ[i]$.
- $candidates[i]$ – The set of literals that are possible semi-intelligent backtrack points after literal i fails. This set is computed as follows:

$$candidates[i] = pred[i] \bigcup cp[i], \text{ where} \quad (1)$$

$$cp[i] = \bigcup_{x \in succ[i]} \{l \mid l \in pred[x] \wedge l < i\} \quad (2)$$

Intuitively, we can understand the structure of the candidate sets by considering how a literal may fail. Literal failures can be classified into two types. First, a literal may reject the bindings supplied by its predecessors; we call this *consumer failure*. Consumer failure is cured by backtracking to a predecessor, which will presumably bind the consumed variable to a different value, hence the first term of the candidates expression.

The second type of failure, called *generator failure*, occurs when one or more of the successors of a literal L reject all of the bindings it generates. When this happens, L will fail after it receives a redo message after it has generated its last binding. In this case, the rejecting successor(s) may be solvable by the next value from a different predecessor and a value previously returned by L. The AND process tries to cure generator failure by backtracking to one of the other predecessors of the rejecting successor. These predecessors are described by Equation (2) in the candidates expression (literals with index greater than the failed literal are removed from the candidate set to insure that the backtrack literal is "to the left" of the failed literal).

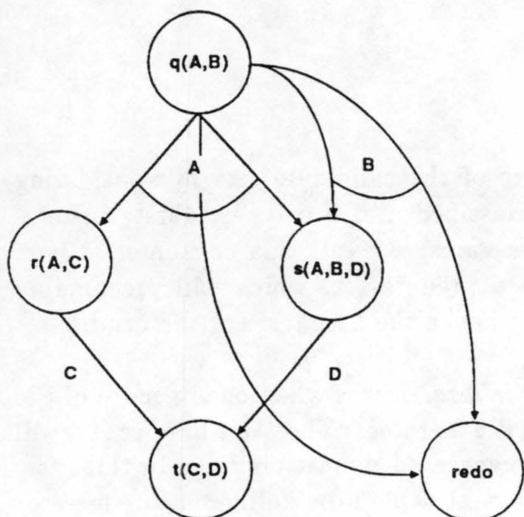
Finally, to distinguish between failure types, the AND process maintains a set of failed successors for each literal in its body. We use $marks[i]$ to denote the set of failed successors of literal i.

The backward execution algorithm operates as follows. When an AND process receives a *fail* message from literal i, it records this information by adding the index i to the marks set of each of i's predecessors. The appropriate backtrack literal is then selected by finding the candidate literal latest in the linear ordering which has i or a successor of i in its marks set. That is, the backtrack literal j has the property that

$$j = max\{c \mid c \in candidates[i] \ \wedge \ (marks[c] \cap (\{i\} \bigcup succ[i]) \neq \emptyset)\}$$

If no such literal exists, the AND process fails. Otherwise, the backward step is effected by canceling or resetting the literals with index larger than j, untrailing the appropriate variables, and sending a *redo* message to the process corresponding to literal j.

The use of failure history is illustrated by the example in Figure 1. In this example, suppose that `s` rejects the bindings provided it by `q` (consumer failure). In this case, the failure of `s` will cause `q` to be marked with the index of `s`, and `q` will be subsequently selected as the backtrack literal. Next suppose `s` accepts the bindings from `q`, but `t` rejects all values of `D` generated by `s` (generator failure). In this case, the failure of `t` will cause `r` and `s` to be marked. When `s` subsequently fails, `r` will be selected as the backtrack literal. This selection reflects the fact that, while `t` had previously failed with all values of values of `D` generated by `s`, it may be solvable with the next value of `C` from `r` and some previous value of `D` from `s`.



index	literal	candidates
1	q	{}
2	r	{1}
3	s	{1,2}
4	t	{1,2,3}
5	redo	{1}

The psuedo-literal "redo" is used to coodinate backtracking activities when a redo message is received

Figure 1: Graph for `p(A,B) :- q(A,B),r(A,C),s(A,B,D),t(C,D)`

3 The Backward Execution Architecture

The backward execution architecture is introduced in this section. The instructions and other machine structures presented here are extensions to the OPAL Machine (OM) [4], but the technique may well apply to RAP-WAM and other AND-parallel architectures.

We associate a *failure continuation* with each literal in a clause body. The continuation is a sequence of machine instructions that will be executed when the AND process receives a fail message from the OR process created to solve the literal.

3.1 Instructions

The only addition to the state vector of an AND process is a marks set for each body literal. The following two instructions are sufficient to implement the algorithm described in the previous section:[1]

set_mark i, j

Add the index i to the marks set of literal j.

check_mark $LSet, c, Label$

[1] There are also instructions to reset variables to the equivalent of unbound, and other control instructions that are not important to this discussion.

Branch to *Label* if literal *c* has been marked by any of the literals in *LSet*.

When `check_mark` is compiled as part of the failure continuation of literal i, *LSet* will be a suitable representation of $\{i\} \bigcup succ[i]$, c will be an element of $candidates[i]$, and *Label* will be the address of the routine that carries out the required functions when literal c is determined to be the backtrack literal (*e.g.* untrailing and canceling descendants). The implementation of `check_mark` is straightforward: if the marks set and *LSet* are represented as bitsets (unsigned integers), this instruction performs a bitwise AND of the two words, and branches if the result is nonzero.

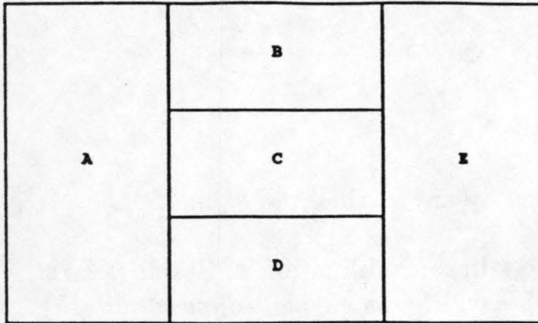
3.2 Compilation

Given a data dependency graph, compilation of the backward execution algorithm into `set_mark` and `check_mark` instructions is trivial. We implement the algorithm by compiling, for each literal, a failure continuation which consists of a sequence of `set_mark` instructions followed by a sequence of `check_mark` instructions. One `set_mark` instruction instruction is compiled for each predecessor of the failed literal. The second part of the backtracking algorithm, the search for the backtrack literal, is implemented by a sequence of `check_mark` instructions, one for each element of the failed literal's candidate set. These instructions are generated so that the failed literal's candidates are inspected in descending order. Finally, since the clause fails if none of the failed literal's candidates are appropriately marked, a `fail` instruction is compiled following the last `check_mark` instruction.

The compilation technique is illustrated using the standard map coloring example [3] shown in Figure 2. The data dependency graph shown in the figure is used by a compiler to generate the backtracking code shown in Figure 3. In this figure, the label on the failure continuation for literal i has the form `next2_i_FC`. Labels of the form `next2_i_IsBl` represent the addresses of the routines which effect the backward step when literal i is selected as the backtrack literal. The code labeled `redo` is the redo continuation of the clause, *i.e.* the code that is executed when the AND process is sent a redo message from its parent.

3.3 Optimization

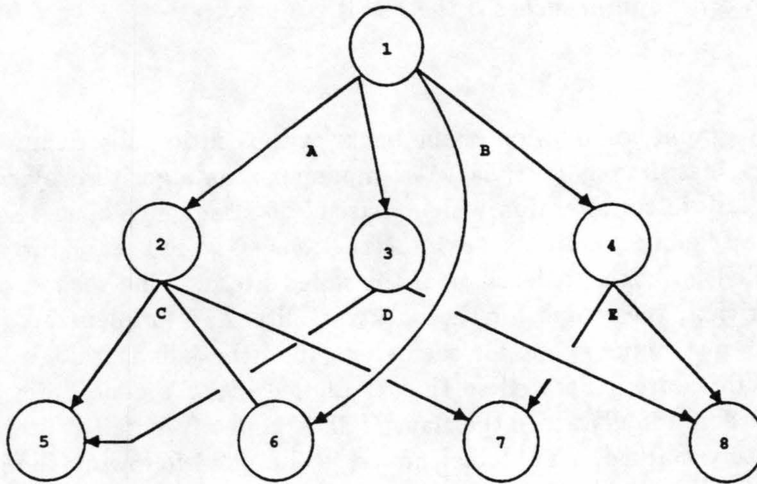
In this section we describe two simple but powerful optimizations. The first optimization, called the "set_mark/check_mark" optimization, replaces a `check_mark` instruction with an unconditional branch when it is known that the `check_mark` instruction will take its branch, *i.e.* it is known that the appropriate mark will always be set by the time the `check_mark` instruction is executed. The optimization can be characterized abstractly



Clause

```
color(A,B,C,D,E) :-
    next(A,B),
    next(A,C),
    next(A,D),
    next(B,E),
    next(C,D),
    next(B,C),
    next(C,E),
    next(D,E).
```

Static Analysis



Index	Literal	Candidate Set
1	next(A,B)	{}
2	next(A,C)	{1}
3	next(A,D)	{1,2}
4	next(B,E)	{1,2,3}
5	next(C,D)	{1,2,3}
6	next(B,C)	{1,2}
7	next(C,E)	{1,2,4}
8	next(D,E)	{1,3,4}
9	redo	{1,2,3,4}

```
next2_1_FC:   fail
next2_2_FC:   set_mark     2,1
              check_mark   [2,5,6,7,9],1,next2_1_IsBl
              fail
next2_3_FC:   set_mark     3,1
              check_mark   [3,5,8,9],2,next2_2_IsBl
              check_mark   [3,5,8,9],1,next2_1_IsBl
              fail
next2_4_FC:   set_mark     4,1
              check_mark   [4,7,8,9],3,next2_3_IsBl
              check_mark   [4,7,8,9],2,next2_2_IsBl
              check_mark   [4,7,8,9],1,next2_1_IsBl
              fail
next2_5_FC:   set_mark     5,1
              set_mark     5,2
              set_mark     5,3
              check_mark   [5],3,next2_3_IsBl
              check_mark   [5],2,next2_2_IsBl
              check_mark   [5],1,next2_1_IsBl
              fail
next2_6_FC:   set_mark     6,1
              set_mark     6,2
              check_mark   [6],2,next2_2_IsBl
              check_mark   [6],1,next2_1_IsBl
              fail
next2_7_FC:   set_mark     7,1
              set_mark     7,2
              set_mark     7,4
              check_mark   [7],4,next2_4_IsBl
              check_mark   [7],2,next2_2_IsBl
              check_mark   [7],1,next2_1_IsBl
              fail
next2_8_FC:   set_mark     8,1
              set_mark     8,3
              set_mark     8,4
              check_mark   [8],4,next2_4_IsBl
              check_mark   [8],3,next2_3_IsBl
              check_mark   [8],1,next2_1_IsBl
              fail
redo:         set_mark     9,1
              set_mark     9,2
              set_mark     9,3
              set_mark     9,4
              check_mark   [9],4,next2_4_IsBl
              check_mark   [9],3,next2_3_IsBl
              check_mark   [9],2,next2_2_IsBl
              check_mark   [9],1,next2_1_IsBl
              fail
```


Figure 3: Unoptimized Backward Execution Code for the Map Coloring Problem

by considering the structure of the code generated for a given failure continuation. Consider the failure of a literal i, where $pred[i] = \{j_1, \ldots, j_n\}$, with $j_1 < j_2 < \ldots < j_n$, and $candidates[i] = \{c_1, c_2, \ldots, c_m\}$, with $c_1 < c_2 < \ldots < c_m$. The failure continuation for this literal will have the following form:

```
set_mark      i, j1
...
set_mark      i, jn
check_mark    [...], cm, cmIsBL
...
check_mark    [...], c1, c1IsBL
fail
```

An analysis of a failure continuation almost always reveals pairs of **set_mark** and **check_mark** instructions (possibly with other instructions in between) of the form:

$$\texttt{set_mark}\ i,k \quad \ldots \quad \texttt{check_mark}\ [\ldots,i,\ldots],k,L$$

This pair of instructions marks literal k with i and then later checks to see if k is marked by i. Recall from Equation (1) that the definition of the candidates set of i includes the predecessors of i. Since we generate a **set_mark** instruction for each predecessor, and a **check_mark** for each candidate, and predecessors are by definition part of the candidate set, we should expect to see many such pairs. In fact, every literal that has a predecessor in the data dependency graph will have a pair of instructions of this form. Internal nodes in the graph (corresponding to generators) may have additional **check_mark** instructions, corresponding to the candidates that are not predecessors.

For each pair of instructions with the form shown above, we know that execution of the **check_mark** instruction will cause the branch to be taken, since the preceding **set_mark** instruction added i to the mark set of the predecessor, and no intervening instruction removes a mark. Thus can we replace the **check_mark** instruction with an unconditional branch to the target. Furthermore, since the code following the branch is now unreachable, it can be removed.

The second optimization, called "Redundant Marks Elimination", is based on the observation that some marks will never be inspected, since the **check_mark** instructions that examine them have been converted to unconditional branches or pruned as dead code. Thus instructions that set marks that are never inspected can also be removed from the failure continuations. This is the case for any instruction of the form **set_mark** i, j for which no corresponding instruction of the form **check_mark** $[\ldots, i, \ldots], j, L$ exists in any of the failure continuations for the clause. In this case, the **set_mark** instruction is redundant, and can be removed.

```
next2_1_FC:   fail
next2_2_FC:   jump next2_1_IsBl
next2_3_FC:   check_mark      [3,5,8,9],2,next2_2_IsBl
              jump next2_1_IsBl
next2_4_FC:   check_mark      [4,7,8,9],3,next2_3_IsBl
              check_mark      [4,7,8,9],2,next2_2_IsBl
              jump            next2_1_IsBl
next2_5_FC:   set_mark        5,2
              jump            next2_3_IsBl
next2_6_FC:   jump            next2_2_IsBl
next2_7_FC:   set_mark        7,2
              jump            next2_4_IsBl
next2_8_FC:   set_mark        8,3
              jump            next2_4_IsBl
redo:         set_mark        9,2
              set_mark        9,3
              jump            next2_4_IsBl
```

Figure 4: Optimized Backward Execution Code for the Map Coloring Problem

The result of applying the set_mark/check_mark and redundant marks elimination optimizations to the code in Figure 3 is shown in Figure 4. The reduction in complexity of the code is dramatic. Approximately 65% of the total number of backward execution instructions have been removed. 8 of the `check_mark` instructions were converted to unconditional branches, and 13 of the remaining `check_mark` instructions were deleted as unreachable code, leaving only 3 of the original 21 `check_mark` instructions. 13 of the 18 `set_mark` instructions were removed by the redundant marks optimization.

This example also illustrates that more space efficient representations of the marks set are typically available. The straightforward representation of the marks and candidates sets for a clause with n body literals requires $\mathcal{O}(n)$ bits per set, or $\mathcal{O}(n^2)$ bits per clause. After optimization, we may find some marks are never checked, so the corresponding sets can be represented in fewer bits. In the map coloring example, the optimized code requires only five bits to represent the marks set. A compiler will be able chose an efficient representation for both the marks sets and the associated masks for the `set_mark` and `check_mark` instructions.

4 Discussion and Summary

The goal of this research has been to show how efficiency and flexibility can be gained by making control of backward execution a "first-class" operation in AND parallel abstract

machines. The model has been implemented as part of our heap-based AND/OR parallel abstract machine [4]. We have also looked at the possibility of using the algorithm in an extension to the RAP-WAM model [6,8].

One can view the architected approach described here as a tradeoff between instruction bandwidth requirements and host complexity. Architectures in which failure is non-architected have lower instruction bandwidth requirements, but require more complex hosts. On the other hand, the architected failure approach has higher instruction bandwidth requirements, but requires a less complex host. In particular, no general purpose fail routine is required. Instead, by exposing the failure handling algorithms to compile time analyses and providing appropriate instruction set functionality, efficient clause-specific failure handling routines can be constructed.

The costs incurred by the architected model derive from the overhead involved in manipulating the marks sets. This overhead has two components: the cost of individual instruction execution, and cost of the increased instruction bandwidth required by the model. Notice, however, that instruction execution can be made very efficient. In particular, efficient bit-encoding of the marks set allows each instruction to be implemented as a short sequence of simple operations. For example, `check_mark LSet,c,Label` can be implemented on the Motorola 68020 as follows. First, suppose a marks set is represented as a bit string, where the ith bit is 1 if literal i is in the set (assuming we are using unoptimized representation of sets). If we assume an address register points to the state vector of the current process, three instructions will load the marks set of `c` into a register and check to see if elements of `LSet` are contained in it:

```
move.l   cmarks(Ai),Dj     ; load marks of literal c to Dj
andi     LSet,Dj           ; any marks in common?
bne      Label
```

Thus, since the cost of executing the individual backtracking instructions is anticipated to be small, the overall cost of the architected model will be dominated by its additional instruction bandwidth requirements. These costs can be mitigated, however, by detecting determinate computations [5] (bypassing the marking algorithm in these cases), and optimizations such as those described above.

An interesting question is whether the increased instruction bandwidth *vs.* lower host complexity will pay off for a sequential Prolog machine. For software implementations of the WAM, it is highly unlikely, as the fail routine simply has to work its way through the trail, checking to see if trailed variables are still on the stack or heap, resetting them to unbound if they are, and there is little to optimize in handling the failure of a procedure call. Architected failure handling might be worth investigating for VLSI implementations of machines with more complex trails. For example, SICStus Prolog pushes descriptors of goals to be executed on backtracking and previous values

of structures modified with **setarg** onto the trail. For a single-chip Prolog processor, it may well be worthwhile to recover the space devoted to a complex general purpose fail routine, devoting some of it to control for failure continuation instructions.

The instructions presented here implement backward execution in programs with fixed data dependencies, determined at compile time. The resulting data dependency graph is represented in data structures (such as the marks sets) of the AND process state vector, and as arguments of instructions that operate on the structures. We have also studied backward execution in systems that allow dynamic data dependencies. A companion paper describes a different data structure, known as a *generator inheritance graph*, which efficiently represents the set of possible dynamic data dependencies [9]. In that paper we show that the instructions presented here can operate as efficiently on a generator inheritance graph, for efficient backward execution in programs with dynamic data dependencies.

Areas for further study include detailed analysis of the memory referencing behavior of the model, and further development of compiler technology along the lines described in [9]. Another interesting question regards the applicability of the architected failure model to other execution models and languages. For example, architected failure handling may be advantageous to sequential logic programming execution models, or for other non-logic languages such as Icon that support "don't-know" nondeterminism through backtracking.

References

[1] Chang, J., Despain, A.M., and DeGroot, D. AND-parallelism of logic programs based on static data dependency analysis. In *COMPCON Spring 85*, (Feb.), IEEE, 1985, pp. 218–225.

[2] Conery, J.S. *The AND/OR Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, Univ. of California, Irvine, 1983. (Computer and Information Science Tech. Rep. 204).

[3] Conery, J.S. Implementing backward execution in nondeterministic AND-parallel systems. In *Proceedings of the Fourth International Conference on Logic Programming*, (Melbourne, Australia, May 25–29), 1987, pp. 633–653.

[4] Conery, J.S. and Meyer, D.M. *OM: A Virtual Processor for Parallel Logic Programs.* Tech. Rep. 87-01, University of Oregon, 1987.

[5] Debray, S.K. and Warren, D.S. Dectection and optimization of functional computations in Prolog. In *Proceedings of the Third International Conference on Logic*

Programming, (London, United Kingdom, July 14–18), 1986, pp. 490–504.

[6] Hermenegildo, M. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, University of Texas, Austin, TX, 1986.

[7] Lin, Y., Kumar, V., and Leung, C. An intelligent backtracking algorithm for parallel execution of logic programs. In *Proceedings of the Third International Conference on Logic Programming*, (London, England, Jul. 14–18), Springer-Verlag, 1986, pp. 55–68.

[8] Meyer, D.M. *Architected Failure Handling for the RAP-WAM Model.* OM note, Department of Computer and Information Science, University of Oregon, Eugene, OR 97403, March 1989.

[9] Meyer, D.M. *Backward Execution Based on Dynamic Data Dependencies.* OM note 89-01, Department of Computer and Information Science, University of Oregon, Eugene, OR 97403, 1989. Submitted to NACLP '89.

[10] Somogyi, Z. A System of Precise Modes for Logic Programs. In *Proceedings of the Fourth International Conference on Logic Programming*, (Melbourne, Australia, May 25–29), 1987, pp. 769–787.

[11] Warren, D.H.D. *An Abstract Prolog Instruction Set.* Tech. Note 309, SRI International, Oct. 1983.

[12] Woo, N.S. and Choe, K. Selecting the backtrack literal in the AND process of the AND/OR Process Model. In *Proceedings of the 1986 Symposium on Logic Programming*, (Salt Lake City, UT, Sep. 22–25). 1986, pp. 200–210.