Backward Execution in Nondeterministic AND-Parallel Systems

John S. Conery

CIS-TR-86-09
November, 1986

Abstract

A new algorithm for "backward execution" in AND-parallel logic programs is described, and new implementation techniques for this class of algorithm are introduced. The dataflow graph that determines forward and backward execution orders, the status of subgoals, and other state information in an AND process are represented by bitsets, and the steps of the algorithm are efficient operations on bitsets. Data from simulations show the implementation techniques are effective, and form the basis of several interesting future experiments.

Submitted to the
Fourth International Conference on Logic Programming

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

1 Introduction

Nondeterminism is one of the distinguishing features of logic programs. There may be more than one way to satisfy a user's goal statement, leading to more than one set of output values. Nondeterminism is achieved in sequential systems through *backtracking*; when the system has a choice in methods for solving a goal, it saves the current state of the computation on a stack so the other choices can be explored later.

This simple control algorithm is not applicable in AND-parallel systems. When the sequential left-to-right solution order is replaced by parallel control, in which more than one goal can be solved at a time, the notion of "most recently solved goal" cannot be used to select a backtrack literal if one of those goals fails. There are two general classes of techniques for generating multiple results in AND-parallel systems. The backtracking algorithm can be replaced by a *join* algorithm that combines streams of results from processes that solve individual goals (Taylor *et al* [11], Kalé [7], Li and Martin [8]), or a *backward execution* algorithm can accept results one at a time and coordinate re-solution after a goal fails. The first backward execution algorithm was designed and implemented in an interpreter based on the AND/OR Process Model (Conery [2]). Recent algorithms by Chang, Despain, and DeGroot [1], Lin, Kumar, and Leung [9], and Woo and Choe [13] are improvements on the original algorithm, designed either to avoid an error that caused that algorithm to miss some correct solutions, or to be more efficient by making some decisions at compile time. Hermenegildo and Nasr presented an algorithm for a model similar to RAP [5,6].

In this paper we present a new algorithm and its implementation. The new algorithm was developed independently as part of a project to study low level representations and implementation techniques for systems based on the AND/OR Process Model (Conery [3], More [10]). As is the case for other backward execution algorithms, the order of solution of goals and decisions about backtrack literals are based on directed acyclic graph (*dag*), containing one node per literal. In the new algorithm, the dag is represented by an adjacency matrix, and other state information is represented by bitsets. The algorithm was designed to be implemented efficiently as a series of simple set operations.

The new algorithm is similar to the Lin-Kumar-Leung and Woo-Choe algorithms, to the extent that all three select the same backtrack point given the same history of failed literals. The new algorithm allows for optimizations in subsequent steps, however, leading to fewer messages being sent to descendants and a higher percentage of independent work being saved. Another contribution of this paper is to implementation technology. We show how simple data structures and a few simple operations on them can form an efficient implementation of a backward execution algorithm.

2 Data Structures

In the abstract model, literals of a clause body are identified by terms of the form `#N`, where `N` is the lexical position of the literal in the body of the clause. Many runtime decisions are based on a *linear ordering* of literals, determined by the shape of the graph. For example, after literal `#N` is selected as a backtrack literal, generators that follow `#N` in the linear ordering are reset. In the original implementation, the linear ordering was represented by a list, and decisions based on position in the linear ordering required access to this list. In the new implementation, we dispense with an explicit list and represent a literal by an *index* which indicates its position in the linear ordering. Thus the decision "does i follow j in the linear ordering?" is reduced to the comparison $i > j$ without reference to an actual list.

The dag that controls both forward and backward execution is represented as a binary adjacency matrix. Row i of the graph can be considered a bit vector, with bit j set if node i is an immediate predecessor of node j, *i.e.* node i is a generator that binds variable(s) consumed by j. Many of the steps of the backward execution algorithm are based on relations derived from the dag. As a time/space tradeoff, these relations are represented explicitly, also as binary vectors. *predecessors*[i] is the set of predecessors of node i; bit j is set if literal j is an immediate or indirect predecessor of literal i. *successors*[i] is defined similarly. *candidates*[i] is the set of literals that are possible backtrack literals after node i fails (Section 3.3).

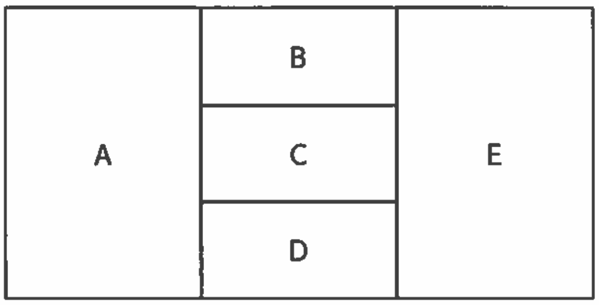
The head of the clause has a special status, and was treated as a special case in the original algorithm. In the new algorithm, the head is represented by two nodes, *HG* and *HC*. *HG* is the head literal in its role as generator of values for variables that are bound in the start messages that creates the AND process. *HC* is the head in its role as consumer; this node is a successor of the generator of every variable that is unbound in the head when the clause is invoked. *HG* is assigned the index 0 in every graph, and *HC* is given an index higher than the index of any body literal.

The newest implementation of the backward execution algorithm is based on a static ordering of the literals in the body of a clause. Like DeGroot, we will assume the graph is created at compile time and does not change during execution [5]. The bit vectors representing the dag and the predecessor, successor, and candidate relations are all created before the algorithm is applied. The steps in compiling information for an AND process are: apply the literal ordering algorithm to the clause to create a dag, obtain a linear ordering, label the literals according to their index in the linear ordering, and finally build the adjacency matrix and relations. The new representation may lead to efficient operations on dynamic graphs, so the restriction to static graphs may be eased; this is the subject of future work and will be explored further in Section 7.

The AND process maintains the status of each literal and updates it as necessary after each forward or backward step. A literal is *solved* if an OR process has been created for it and the OR process has returned a success message. A literal is *pending* if an OR process has been started for it but has not yet returned a result, and it is *blocked* if some of its predecessors in the dag are not yet solved. The AND process maintains three bitsets to represent these states, where $i \in S$ if literal i has status S. There are two reasons for representing status information this way, instead of as a vector of status indicators. After each step, we want to quickly ascertain the set of *enabled* literals so we can start OR processes for them. The decision as to whether or not a literal is enabled is easily made: a process can be started for literal i if $i \in$ *blocked* and *predecessors*$[i] \subseteq$ *solved*. Second, a literal may have the status of solved and pending at the same time, as a result of the way results are cached by the AND process (Section 3.2).

The final data structure is a set of *marks* associated with each generator literal. $i \in$ *marks*$[j]$ if j is potentially the cause of the failure of node i. Whenever a literal fails, its index is added to the mark set of each of its predecessors since each predecessor is potentially the cause of the failure. How the marks are used to select the backtrack literal is explained in the next section.

The dataflow graph for a map coloring problem used to illustrate many backward execution algorithms is given in Figure 1 along with the linear ordering of the literals, the index of each literal, and the sets of predecessors, successors, and candidates for each literal.

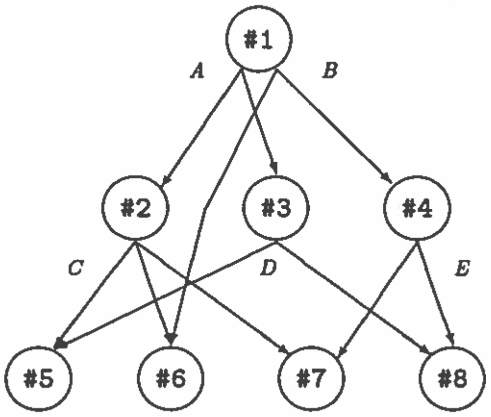


Call:

```
   ←
color(A,B,C,D,E).
```

Clause:

```
   color(A,B,C,D,E) ←
(1)  next(A,B) ∧
(5)  next(C,D) ∧
(2)  next(A,C) ∧
(3)  next(A,D) ∧
(6)  next(B,C) ∧
(4)  next(B,E) ∧
(7)  next(C,E) ∧
(8)  next(D,E).
```



```
next(green,yellow).
next(green,red).
next(green,blue).
next(yellow,green).
next(yellow,red).
next(yellow,blue).
next(red,green).
next(red,yellow).
next(red,blue).
next(blue,green).
next(blue,yellow).
next(blue,red).
```

Candidate Sets:

1:	{2,3,4}	2:	{1,3,4}
3:	{1,2,4}	4:	{1,2,3}
5:	{1,2,3}	6:	{1,2}
7:	{1,2,4}	8:	{1,3,4}

Figure 1: Example Dataflow Graph and Relations

3 The Backward Execution Algorithm

The solution of a goal statement by a parallel AND process is carried out as a series of indivisible steps. Each step is triggered by the arrival of a message from a descendant OR process. The receipt of a success message from the process for literal i leads to a forward step: i is removed from *pending* and added to *solved*, and OR processes are started for each newly enabled process. The receipt of a fail message leads to a backward step: one of the generators that contributed directly or indirectly to the failure of i must be sent a redo message, and other generators related to i might have to be reset.

The original backward execution algorithm of [2] was defined in terms of forward and backward execution *phases*. When an AND process received a fail message, it embarked on a backward path in the dag until the predecessors of the failed literal had created a new set of bindings. Any fail messages from processes for literals not on the backward path were postponed for later processing since they required the starting of a new path. The backward execution phase of the new algorithm lasts for just one step, the step that processes the fail message. After this backward step, the AND process is ready to accept and process additional fail messages. Situations that led to multiple failure paths and postponed message processing in the earlier algorithm are handled naturally by successive backward steps in the new algorithm, and backward and forward steps can be freely intermixed.

The rule for selecting the literal to send a redo message to, and the rules for deciding which generators have to be reset, will be explained in this section. A concise description of the algorithm is given in Figures 3 and 4 at the end of the paper.

3.1 Processing a Fail Message

The first operation performed by the AND process after literal i fails is to add i to the marks of every predecessor of i. The backtrack literal is determined by finding the literal latest in the linear ordering which has i or a successor of i in its set of marks. Since literals are indexed according to the linear order, this step consists of a linear scan of the marks vectors, working from the highest index to the lowest, looking for a literal j such that

$$marks[j] \cap (\{i\} \cup succ[i]) \neq \emptyset$$

If the backtrack literal has index 0, *i.e.* the backtrack literal is the head of the clause, the AND process fails.

The reason we must look for a successor of i as well as i itself can be explained by the example in Figure 1. Suppose the process for literal 4 fails as a result of the bindings produced by its predecessor, literal 1. The correct step is to send a redo message to the process for literal 1. Since literal 4 has not been solved, there can be no processes for its successors in the dag, and thus no marks from its predecessors in the mark vectors of literals 2 and 3, and the selected backtrack literal is literal 1. However, what would happen if literal 4 sends a fail message because the corresponding process has produced all possible solutions for its subgoal? In other words, suppose the process for literal 7 rejects every value produced by the process for literal 4. Literal 7 might be solvable with the combination of a different value from literal 2 and the original value from literal 4. The correct backtrack literal in this situation is 2. Thus, after literal 4 fails, the AND process searches for a generator with a mark of 4 or a successor of 4; in this case it finds 2 marked by the failure of 7 and selects 2 as the backtrack literal.

After the backtrack literal is identified, the AND process obtains another result from the corresponding OR process and resets the generators that follow the selected literal in the linear ordering. The reset operation is how the AND process creates the cross product of tuple values for consideration by consumers. By analogy with nested `FOR` loops, the backtrack literal is a generator that corresponds to one of the index statments, and the generators that are reset correspond to index statements closer to the body of the loop [4]. In a straightforward implementation of the redo and reset operation, a redo message is sent to the OR process for the backtrack literal, and the OR processes for the literals that are reset are replaced by new processes. Situations where a redo message can be avoided are described in the next section, and an optimization that leads to fewer resets is discussed in Section 3.4.

3.2 Result Cache

Resets can be made more efficient by introducing a "restart" message and timestamps so existing OR processes can be reused. Further savings are possible if the AND process maintains a cache of results from each generator and uses bindings from the cache during redo and reset operations. In the current implementation an AND process maintains two lists, named *old* and *new*, for bindings produced by each generator. *old* is the set of values already used by the AND process. When a success message arrives from a generator, the current bindings of its variables are placed in *old* and the variables are set to the values indicated by the success message. A reset consists of moving the current bindings plus all bindings from *old* to *new*, and then selecting a set from *new* to be the current bindings. From this point on, as long as *new* remains non-empty, the AND process takes bindings from *new* instead of sending a redo message to the OR process.

Some complexity is added to the message interface for the process by the use of a cache. A situation may arise where the *new* list for literal i is empty, but *old* is non-empty, and a redo message has been sent to the process for literal i. In this situation the status of i is pending, since the process is waiting for a result from the OR process for i. Next, suppose the processing of a fail message on behalf of another literal causes i to be reset. Since *old*$[i]$ is not empty, we can move all the bindings from *old* to *new* and consider i to be solved. However, the OR process for i may still be working on the redo message sent earlier, so we cannot simply remove i from the pending set. This is the situation mentioned earlier where a literal is in two status sets simultaneously: in this case i is both solved and pending. The message interface correctly handles cases where success or fail messages arrive for literals the AND process considers solved.

3.3 Candidate Sets

A data structure introduced earlier but not used so far is the set named *candidates* for each literal. *candidates*$[i]$ is the set of literals that are possible backtrack literals after the process for literal i fails. Obviously all predecessors of i should be included in this set. As a previous example showed, some literals that are not predecessors of i belong to this set, as well. In that example, literal 2 was selected as the backtrack literal after the failure of literal 4. *candidates[i]* is the set of predecessors of i and the predecessors of the successors of i, excluding i itself:

$$\mathit{predecessors}[i] \cup \mathit{cp}(i) - \{i\}$$

where

$$\mathit{cp}(i) = \bigcup_{x \in \mathit{succ}[i]} \mathit{predecessors}[x]$$

The candidate set is used in two situations. First, it helps when selecting the backtrack literal. The first description of the search for the backtrack literal indicated a search backward through the linear ordering for a literal with a mark set containing the failed literal i or one of its descendants. In fact, not all literals are considered; we only have to check the marks on the literals in *candidates[i]*. How this leads to a more efficient search is discussed in Section 4 below. The second application of the candidate set is related to deciding which generators to reset, which is discussed next.

3.4 Resetting Fewer Generators

In the original backward execution algorithm, when literal i was chosen as the backtrack literal, every generator following i in the linear ordering was reset. This is a waste, however, if these generators have nothing to do with the failure that caused i to be selected. Not only are innocent generators reset, but their consumers have to be canceled, even when these generators and consumers may have settled on an acceptable value. It is clearly worthwhile to reset as few generators as possible. Chang, Despain, and DeGroot used this idea in their backtracking algorithm [1].

What we need to do is reset only those literals which, along with the backtrack literal i, contribute to the solution of the successors of i. These are all the literals with i in their candidate set. However, $j \in$ *candidates*$[i]$ iff $i \in$ *candidates*$[j]$, so the set of generators that must be reset after selecting literal i as the backtrack literal can be restricted to those literals in *candidates*$[i]$ with index greater than i.

4 Notes on Efficient Implementation

The potential efficiencies of the new backward execution algorithm are based on the fact that all of the data structures can be represented as bitsets and corresponding operations are fast operations on sets. Many processors have instructions for manipulating bit fields that represent sets. The most obvious are bitwise OR for union and bitwise AND for intersection.

Some processors have a "find first set bit" instruction that will return the index of the first element in a set. This leads to an efficient enumeration of the elements of a set. For example, when looking for a backtrack literal after the failure of literal i, the algorithm selects as the backtrack literal the latest literal j in the linear ordering such that $i \in$ *marks*$[j]$. Instead of starting from the end of the array of mark vectors and working toward the front, testing to see if i is an element of each set of markings, we can use the definition of *candidates*$[i]$. The find first set bit instruction applied to *candidates*$[i]$ gives the index of the first set of marks to test. This use of find first bit set, combined with masking and loop indexing instructions, can be used in many places in the new algorithm to enumerate the elements of a set.

5 Comparison with Other Algorithms

The main problem to be solved by a backward execution algorithm, and the source of the error in the earlier algorithm, is how to remember all potential backtrack points after a failure and how to use these points in the context of any combination of intervening success and failure messages. In the following discussion we will be concerned with what happens after literal L_f fails. Assume the potential backtrack points are literals $B = \{L_i \ldots L_j\}$ and that $L_b \in B$ is selected as the backtrack literal. B_r is $B - \{L_b\}$, the set of remaining backtrack points. All three algorithms compared here select the same literal given the same failure history; what differs is how this history is represented and used. The basic problem is to remember all the elements of B_r so that if L_b later fails, the these literals can be considered for backtrack points along with the predecessors of L_b.

The algorithm by Lin, Kumar and Leung [9] stores the unused backtrack points B_r in a *B-list* associated with the backtrack literal L_b. Items of the B-list are sorted according to their placement in the linear ordering. Initially the B-list of a literal contains only its immediate predecessors. When L_b is chosen for backtracking, the elements of B_r are merged with this list. When L_b later fails, the new backtrack literal is taken from the B-list of L_b. The difference between this approach and our new algorithm is that in our algorithm, instead of saving B_r with the backtrack literal for use if and when that literal fails, we scatter the information that L_f failed among the mark sets of the elements of B_r. When L_b later fails we have to search these mark sets, but, as described in the previous section, this search can be done efficiently, and in general the first literal checked is the backtrack literal. Another difference is that our algorithm is based on sets and literal indices instead of lists and a linear ordering list. However, the techniques introduced here can be applied to the Lin-Kumar-Leung algorithm to make it more efficient. B-lists can be replaced by B-sets, the merge in step 3 of their algorithm can be done by setting B_j to $B_j \cup B_i - \{j\}$, and the backtrack literal identified by the first element of a B-set.

In the Woo and Choe algorithm, the equivalent of the B-list of the failed literal is reconstructed from information stored with the literal. Instead of

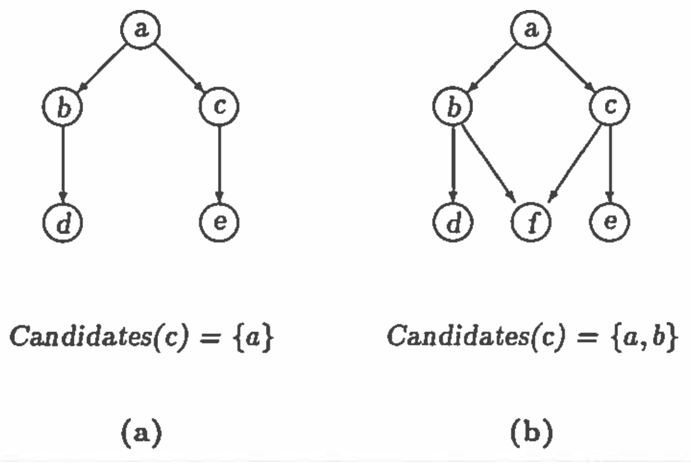


Figure 2: Candidate Sets

storing B_r with L_b, the identity of the failed literal L_f is stored in a *redo cause set (RCS)* associated with L_b. This is similar to the mark set of our algorithm; the difference is that Woo and Choe mark only the backtrack literal, not all predecessors of the failed literal. When L_b later fails, the RCS of this literal is used to construct, dynamically, the same information stored in the B-list for L_b. The elements of S3 after step 4 of the Woo and Choe algorithm are the same as the B-list of the failed literal in Lin, Kumar, and Leung's algorithm.

The biggest difference between the new algorithm and the two described above is the use of candidate sets when deciding which literals to reset after selection of a backtrack literal. The B-list or S3 of the other algorithms does not give us any information about which generators have to be reset. This is illustrated by the graphs in Figure 2. After the failure of literal 4, literal 2 is selected for backtracking. In the graph on the left, literal 3 should be reset, because it has a descendant in common with the backtrack literal. Such a descendant has to consider all combinations of values from its predecessors, and when an "outer variable" from a generator early in the linear ordering gets a new value, all "inner variables" must be reset. In the graph on the right, however, literal 3 does not have to be reset, since it has no descendants in common. This information is captured in the candidate set for literal 2; node 3 is in *candidates*[2] in the graph at the left but not in the graph on the right. In both other algorithms, literal 3 is reset after 2 is selected, regardless of whether it has descendants in common with the backtrack literal. There is no information about literal 3 and how it is related to literals 2 and 4 in either the B-list for literal 2 or the computed set S3.

6 Results of Experiments

The new algorithm was implemented in Modula-2 in a simulator that measured the number of steps required to solve a nondeterministic goal. In keeping with the goal of performing only simple set operations, Modula-2 bitsets were used to represent the dag and derived relations. The results of several simulations are presented in Table 1.

The simulator was used to test a number of aspects of the implementation. The independent variables of the tests were whether or not to use a result cache, whether or not to use the candidate set to decide which literals to reset, and variations on the order messages were processed. Messages in the AND process' input queue could be removed in serial order, random order, or giving preference to certain types of messages or messages from designated processes. The hypothesis was that by processing fail messages first, the chances were that some descendants would be canceled before their messages caused the AND process to take some useless steps [2]. This was almost always the case.

There are four groups of columns in Table 1, labeled `C/C`, `C/A`, `NC/C`, and `NC/A`. The letter(s) before the slash indicate whether or not a cache was simulated (`NC` means no cache), and the second letter indicates whether only candidate generators were reset (`C`) or if all generators were reset (`A`). The numbers in the columns are means, taken from three simulation runs, where each run used a different message order. For each test, the line marked `Desc` gives the total number of immediate descendant OR processes that were created. `Cons` is the number of descendants that were pure consumers, not generating any variable bindings. This is intended to be a measure of the efficiency of the AND process; a large number of consumer processes corresponds to a large number of tuples considered by the AND process. The most efficient AND process is one that would generate exactly those tuples that are solutions for the problem. The last line is the number of execution steps, corresponding to the total number of success and fail messages

Problem:		C/C	C/A	NC/C	NC/A
Map Coloring (72)					
	Desc	520	520	645	645
	Cons	483	483	464	464
	Step	713	713	982	982
Willow (9)					
	Desc	39	39	42	58
	Cons	25	25	25	31
	Step	68	68	76	92
MM (1)					
	Desc	30	32	33	36
	Cons	27	39	24	24
	Step	41	43	48	50
Tough (112)					
	Desc	1605	1605	4527	4527
	Cons	1016	1016	1412	1412
	Step	1899	1899	6022	6022

Table of results from simulation runs. Each number is a mean from three executions, each with a different message queue strategy. In the graphs, generators that are immediate predecessors of the head, *i.e.* generators that produce output values, are drawn as double circles. The number in parentheses by the name of the graph is the total number of solutions to the problem.

Table 12 Data

received.

The use of a result cache leads to significant savings in large programs, with fewer reset operations and execution steps. The use of candidate sets to decide which literals to reset does not always lead to more efficient execution. The reason is that in many clauses, the head is a consumer of the "last row" of generators, *i.e.* in most cases there are no literals that (1) produce values not exported to the calling procedure, and (2) follow the last generator that does produce exported values. When all generators are related because they have HC as a common descendant, every generator is in every other generator's candidate set, and there is no difference between the cases where all generators are reset and only candidates are reset. A graph in the Table 1 that does show a difference is the "double m." In another experiment, not shown, "tough" was modified so that only one variable was returned through the head. This problem showed a dramatic gain in efficiency when only items in the candidate set of the backtrack literal were reset.

7 Future Projects

There are a number of interesting future projects based on the new algorithm. The first is part of the definition of a virtual machine for programs of the AND/OR Process Model. The current implementations are all interpreter based, where the basic cycle of the interpreter is to select a message from a system wide queue, find the process that is the target of the message, apply the indicated state transition, and store the new process state and output messages. There is nothing in the system tailored to each individual clause, the way open-coded unification is tailored to compiled clauses of the Warren Abstract Machine [12]. The next stage in the implementation of backward execution is the definition of sequences of compiled code based on the operations of the new algorithm.

Another interesting project is to see if dynamic graphs can be manipulated efficiently by the new algorithm. When a generator can bind a variable to a nonground term, new dependences are introduced into the clause. The key to efficient handling of this situation is to notice that the new dependences are always between successors of the generator that introduces the dependence. It may be possible to refine the agorithm so that when a new dependence is created, bits are added to the representation of the dag to reflect the dependence, and the proper candidate sets are updated. When the AND process backs up to the generator, the dynamic dependences can be erased along with its mark set, pending the arrival of a new value.

A similar project involves creating the candidate sets dynamically as generators succeed. Suppose the head node *HC* is a successor of nodes 4 and 5 in Figure 2. This means the left and right "branches" in the tree are related, so generators of the left branch are candidates for the right branch, and vice versa. When the AND process is working on the first result, the two branches are independent. It is only when we are working on later results, in response to a redo message, that we have to worry about resetting the right branch for every new value from the left branch. If the candidate sets can be created dynamically, the dependence between branches will not be made until the first result is computed.

A fourth project involves keeping the bindings stored in the result cache in a strict order. In the current implementation values can be retrieved in any order, since taking a value from the cache is equivalent to receiving a success from a descendant OR process and the AND process is not allowed to depend on the order of results from descendants. Consider a case where a set of generators and consumers has settled on an acceptable set of values after a number of redo and reset operations. If one of the consumers also has as a predecessor that is selected as a backtrack literal by an unrelated failure, the generators of this group are all reset and the acceptable value will have to be recomputed. However, if the AND process can order its results, it may be possible to skip combinations that are known to fail. The definition of this ordering and its effect on the rest of the algorithm have to be explored in greater detail.

8 Summary

There are now a number of competing methods for generating multiple results in parallel AND processes. One class of methods, based on joining result streams from descendant processes, does not use any form of backtracking. Instead, the AND process retains results sent from each descendant and computes the stream of final results via a dynamic join operation. Examples are operations in the Reduce-OR Process Model (Kalé [7]) and the Sync Model (Li and Martin [8]). The other class of methods uses a generalized form of backtracking, called backward execution, where an AND process works on one result at a time and handles the failure of a descendant by re-solving a previously solved goal. Falling in this category are the algorithm presented here and the algorithms of Lin, Kumar, and Leung [9] and Woo and Choe [13].

There are optimizations to be made at the conceptual level for both class of methods. For example, in both the Reduce-OR Process Model and the Sync Model, the size of intermediate relations can be reduced considerably by using the structure of the solution order graph. In backward execution models, savings can be realized by a judicious choice of literals to reset [1]. Part of the decision of which style is preferable will depend on the application. If most queries are "setof" queries that require all results and there are sufficient resources to devote to the task, it makes sense to work on all results simultaneously. If few answers are needed, or resources are scarce, or the answers should be produced in a demand-driven fashion, working on a single result at a time might be better. Ultimately, however, a large part of the decision about which class of algorithm is preferable will depend on low level implementation. This paper introduced low level representations and operations that significantly speed up the operation of backward execution algorithms.

Procedure *back-up(FL):*

FL: The literal corresponding to the failed process.

1. Add *FL* to the marks of each literal in *pred(FL)*.
2. Let *BL* be the latest literal in the linear ordering with a set of marks containing a literal in $\{FL\} \cup$ *succ(FL)*. If there is no such set of marks, *BL* is `HG`.
3. If *BL* is `HG`, the AND process fails, otherwise continue.
4. Call *next-result(BL)*; if it fails, make the recursive call *back-up(BL)*, otherwise continue. (*next-result* is defined in Figure 4)
5. Initialize a set *MV* to be the set of variables generated by *BL*, and let *marks(BL)* $= \emptyset$.
6. Work toward the end of the literal ordering, starting from *BL*, and do the following to each literal *L*:
   (a) If *L* consumes a variable in *MV*, cancel the OR process for *L*, set *marks(L)* to $\emptyset$, and move *L* to the set of blocked literals. If *L* is a generator, add the variables generated by *L* to *MV*.
   (b) If *L* is a generator in *candidates(BL)* and does not consume a variable in *MV*, set *marks(L)* to $\emptyset$ and call *reset(L)* (Figure 4). If the values of the variables generated by *L* change as a result of the reset, add them to *MV*.
7. For each literal $\{L \mid L \in$ *Blocked*, *pred(L)* $\subseteq$ *Solved*$\}$ start an OR process and move *L* from *Blocked* to *Pending*.

After receiving a fail message from process for *L*: Call *back-up(L)*.

After receiving a redo message: Call *back-up*(`HC`).

Figure 3: Backward Execution Algorithm

Result Cache Algorithms

Old For each generator, the list of results sent from the OR process and used to set the value of the corresponding variables.

New For each generator, the list of future bindings for its variables. Generally these are "recycled" values, not values just arrived from the OR process and waiting to be applied.

Procedure *next-result(L):* Return *OK* if the variables generated by *L* can be given new bindings, or if the process for *L* can potentially send new bindings.

1. If *New(L)* $\neq \emptyset$, add the current bindings to *Old(L)*, and remove a set of bindings from *New(L)* and make them the current bindings. Return *OK*.
2. If $L \in$ *Pending*, add the current bindings to *Old(L)*, remove *L* from *Solved*, and return *OK*.
3. If the process for *L* has failed, return *FAIL*.
4. Add the current bindings to *Old(L)*, send the process for *L* a redo message, move *L* from *Solved* to *Pending*, and return *OK*.

Procedure *reset(L):* Return *TRUE* if the variables generated by *L* change values as a result of the reset.

1. If $L \in$ *Blocked*, do nothing, return *FALSE*.
2. If *Old(L)* $= \emptyset$, do nothing, return *FALSE*.
3. Move all bindings to *New(L)*, setting *Old(L)* to the empty list. Remove a set of bindings from *New(L)* and make them the current bindings. Add *L* to *Solved* and return *TRUE*.

Figure 4: Maintaining a Cache of Results

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. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, Boston, MA, 1986.

[4] Conery, J.S. and Kibler, D.F. AND parallelism and nondeterminism in logic programs. *New Generation Computing 3*, (1985), 43–70.

[5] DeGroot, D. Restricted AND-parallelism. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, (Tokyo, Japan), 1984, pp. 471–478.

[6] Hermenegildo, M.V. and Nasr, R.I. Efficient management of backtracking in AND-parallelism. In *Proceedings of the Third International Conference on Logic Programming*, (London, England, Jul. 14–18), Springer-Verlag, 1986, pp. 40–54.

[7] Kalé, L.V. *Parallel Architectures for Problem Solving.* PhD thesis, SUNY Stony Brook, Dec. 1985. (Univ. of Illinois at Urbana-Champaign Tech. Rep. UIUCDCS-R-85-1237).

[8] Li, P. and Martin, A.J. The Sync model for parallel execution of logic programming. In *Proceedings of the 1986 Symposium on Logic Programming*, (Salt Lake City, UT, Sep. 22–25), 1986, pp. 223–234.

[9] 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.

[10] More, N. *Implementing the AND/OR Process Model.* Master's thesis, Univ. of Oregon, 1986.

[11] Taylor, S., Lowry, A., Maguire, G.Q., and Stolfo, S.J. Logic programming using parallel associative operations. In *Proceedings of the 1984*

International Symposium on Logic Programming, (Atlantic City, NJ, Feb. 6–9), 1984, pp. 58–68.

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

[13] 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.