Information Processing Letters 29 (1988) 191–198
North-Holland

12 November 1988

EFFICIENT BACKWARD EXECUTION IN AND/OR PROCESS MODEL

Chan-Ik PARK, Kyu Ho PARK and Myunghwan KIM

Department of Electrical Engineering, Korea Advanced Institute of Science and Technology, Cheongryang, P.O. Box 150, Seoul 130-650, Korea

Communicated by S. Yajima
Received 11 March 1987
Revised 24 August 1987, 9 April 1988 and 10 May 1988

A method of generating n-tuples for a clause with n variables as its argument under the AND/OR process model (Conery, 1983) of logic programs is considered. The AND/OR process model uses the nested loop model for it, which incurs much inefficiencies because it does not consider the relationship between literals. The improvement based on this point is made from static analysis of the data dependency graph which represents the relationship between literals in a clause. It is shown that our improved method for generating tuples is better than the AND/OR process model. Adoption of the improved method makes multiple failure situation handled more efficiently. We also show that several backtrack literals can be found for multiple failures by analyzing the sources of failures when the improved method is used for generating tuples. Our method is illustrated by several examples.

Keywords: Logic programming, AND/OR process model, backward execution algorithm, nested loop model, multiple failure

1. Introduction

Logic programs contain various kinds of parallelism such as AND parallelism, OR parallelism, stream parallelism and search parallelism [2,4]. Among these parallelisms, AND and OR parallelism are the most important ones in executing logic programs efficiently.

There have been many computational models which exploit those parallelisms [1,3,6,7]. All these models utilize the OR parallelism which is natural in the execution of logic programs. For AND parallelism, the models in [3,6,7] serialize the execution of the body literals in a clause. In [1], independent literals among the body literals in a clause are found by checking the dependency of variables in the clause so that the independent literals can be processed concurrently. Hence, the AND/OR process model proposed by Conery [1] is prominent because it represents the behavior of logic program evaluation naturally [8]. In the AND/OR process model [1], two kinds of processes are used, one of which is the AND process and the other the OR process. The OR process is created to solve exactly one literal and it generates the values of variables in the literal. The OR process tries more than one candidate clause simultaneously to solve a given literal; it is known as OR parallelism. The AND process is created for each rule clause and constructs a consistent solution that satisfies all literals in the rule clause. The AND process can solve more than one literal by creating more than one OR process simultaneously (AND parallelism). The AND process coordinates response message from these descendent OR processes until all literals in a given clause have been fully solved.

For this AND parallelism, three major parts are used in the AND/OR process model: ordering part, forward execution part, and backward execution part. The ordering part constructs a data dependency graph representing the generator–consumer relationship between literals and determines the linear order of all the literals by level order traversing of the data dependency graph. The forward execution part determines which literals should be evaluated next by examining the data dependency graph, and creates the corresponding OR processes. The backward execution part determines which literal in a clause should be re-evaluated when a literal fails. The case in which one literal fails is called *single failure case* and, when more than one literal fails, it is called *multiple failure case* [8]. The linear ordering obtained by the ordering algorithm is used as the order of generating the values of variables in a clause, i.e., it determines the **for**-loop nesting sequence in the nested loop model. The nested loop model is described in detail in Section 2.

In this paper we present an improved method of generating tuples for the backward execution, which removes inefficiency and redundancy of the nested loop model. It is also shown that this improved method can reduce the number of evaluation steps as well as the number of messages in communication when compared with the AND/OR process model. In addition, the multiple backtrack literal selection algorithm is presented for the multiple failure case based on our improved method.

2. Nested loop model by Conery

If there are n variables in the head of a clause, the AND process of the clause is expected to construct as many n-tuples of terms as possible. A subset of these n-tuples may belong to the relation defined by the clause of the AND process.

In Conery's model [1], a straightforward method for generating tuples is used. It is specified as the nested loop model in the PASCAL programming language (see Fig. 1). Suppose a clause is $p(x_1, x_2, \ldots, x_n)$ and each variable can be mapped into one element of the set $\{n_1, n_2, \ldots, n_m\}$.

In order to adopt the nested loop model in generating tuples for a clause, the AND process has a linear ordering of literals and the *reset* operation. The linear ordering is actually an ordering of all literals in a clause, i.e., it is used as an ordering of all variables in the clause which is the **for**-loop sequence in Fig. 1. The only constraint on the relative order of any two literals is that a generator literal must always precede all literals which consume its variable. The linear ordering is obtained from level order traversing of the data dependency graph. The *reset* operation restarts a generator literal so that the variable of the generator takes on the same set of values once again (for the linear ordering and *reset*, see [1]). These operations are included in the backward execution part.

The backward execution part is divided into two parts: *selection of a backtrack literal* and *construction of the nested loop model* for the selected backtrack literal. Woo [8] and Lin [5] found out, however, that the first part of Conery's method is incorrect. Assuming that the first part is correct throughout this paper, we only consider the second part because the nested loop model behaves inefficiently from the viewpoint of backtracking.

The backward execution part starts when the OR process for a literal sends a fail message to the parent AND process. The backward execution part first selects a backtrack literal which is responsible for resolving the current failure. After selecting the backtrack literal, the backward execution constructs the nested loop model for the selected backtrack literal by sending two types of

```
$p(x_1, x_2, \ldots, x_n) :- p_1(x_1, x_2), \ldots, p_{n-1}(x_{n-1}, x_n)$
for $x_1 := n_1$ to $n_m$ do
    for $x_2 := n_1$ to $n_m$ do
        ...
            for $x_n := n_1$ to $n_m$ do
                writeln("$Success(x_1, x_2, \ldots, x_n)$");
```

Fig. 1. Nested loop model of generating n-tuples for a clause [1].

```
1. //Selection of a backtrack literal//:
   Select the backtrack literal, say $L_b$, for a failed literal, $L_f$;
2. //Construction of the nested loop model for $L_b$//:
   2a. Send a redo message to the OR process of $L_b$;
   2b. For each generator later than $L_b$ in the linear ordering,
       //which are called reset literals//
       {
        perform reset operation;
       }
       //The variables generated by reset literals and $L_b$//
       //are called modified variables.//
   2c. For each literal later than $L_b$ in the linear ordering which consumes modified variables,
       //which are called cancel literals//
       {
        send cancel message to the corresponding OR process;
       }
```

Fig. 2. Backward execution algorithm in the AND/OR process model [1].

messages to the descendent OR processes, which are *redo* and *cancel*, and by performing the *reset* operation for some generator literals. Literals for which the *reset* operations and *cancel* messages are needed in the backward execution part are called *reset* literals and *cancel* literals, respectively. Fig. 2 shows Conery's backward execution method [1].

3. Improvements in the nested loop model

In this section we describe an improved method of generating tuples for a clause. Let us assume that the data dependency graph (DDG) of a clause is constructed by the ordering algorithm in the AND/OR process model. To begin with, some notations in DDG are given as follows.

– *CON* represents the set of all consumer literals, i.e., all leaf nodes in the DDG.

– The parent set of a literal L, denoted $Parent(L)$, is given as

$$Parent(L) = \{L' \mid \text{there is a directed edge from } L' \text{ to } L\}.$$

– The Directly Reachable Literal Set of a literal L, denoted $DRLS(L)$, is given as

$$DRLS(L) = \{L' \mid \text{there is a directed path from } L \text{ to } L'\}.$$

– The Indirectly Reachable Literal Set of a literal L, denoted $IDRLS(L)$, is given as

$$IDRLS(L) = ID(L) - DRLS(L)$$

where

$$ID(L) = \{L' \mid \text{there is a path between } L \text{ and } L' \text{ ignoring the direction of edges, such that each literal on the path (including } L'\text{) is later than } L \text{ in the linear ordering}\}.$$

– The Related Reset Literal Set of a literal L, denoted $RRLS(L)$, is given as

$$RRLS(L) = \{L' \mid L' \in IDRLS(L) \text{ such that } Parent(L') \cap IDRLS(L) = \emptyset\}.$$

– The Cancel Literal Set of a literal L, denoted $CLS(L)$, is given as

$$CLS(L) = \bigcup_{L_i \in (RRLS(L) \cup \{L\})} DRLS(L_i).$$

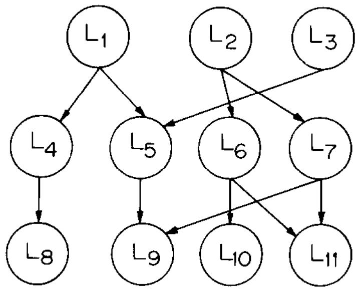

Fig. 3. Example of notations.

– The independent generator: Given two generator literals, L_1 and L_2, L_1 and L_2 are said to be *independent* iff

$$\{RRLS(L_1) \cup CLS(L_1)\} \cap \{RRLS(L_2) \cup CLS(L_2)\} = \emptyset.$$

Without loss of generality, we begin with the structure of the data dependency graph for a clause, not the whole logic program. For the example in Fig. 3, we get the following:

(1) $CON = \{L_8, L_9, L_{10}, L_{11}\}$,
(2) $Parent(L_5) = \{L_1, L_3\}$,
(3) $DRLS(L_2) = \{L_6, L_7, L_9, L_{10}, L_{11}\}$,
(4) $IDRLS(L_2) = \{L_3, L_5\}$,
(5) $RRLS(L_2) = \{L_3\}$,
(6) $CLS(L_2) = \{L_5, L_6, L_7, L_9, L_{10}, L_{11}\}$,
(7) L_4 and L_5 are independent.

For $DRLS(L)$, $IDRLS(L)$, $RRLS(L)$, and $CLS(L)$, the following lemma holds.

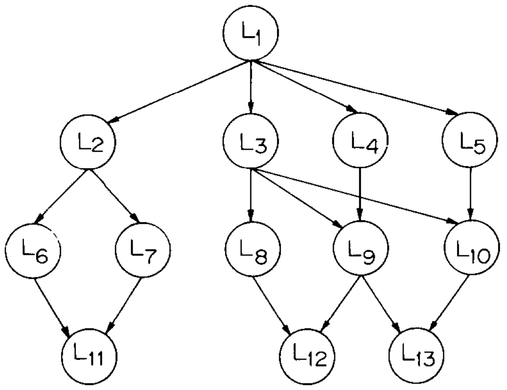

Fig. 5. Data dependency graph: An example.

Lemma. *For a literal L,*
(a) $RRLS(L) \cap CON = \emptyset$,
(b) $RRLS(L) \cap CLS(L) = \emptyset$,
(c) $DRLS(L) \cup IDRLS(L) = RRLS(L) \cup CLS(L)$.

An improved backward execution part using the above notations is shown in Fig. 4. We can easily see that there are common literals between *reset* literals and *cancel* literals in the backward execution of the AND/OR process model (see Fig. 2), whereas the improved method does not have common literals between *reset* and *cancel* literal sets because *RRLS* and *CLS* are disjoint.

From the example in Fig. 5, it can be seen that our method can reduce the number of evaluation

```
1. //Selection of a backtrack literal (same as Fig. 2)//
   Select the backtrack literal, $L_b$, of a failed literal $L_f$;
2. //Construction of the improved method for $L_b$//
   2a. Send a redo message to the OR process of $L_b$;
   2b. For each literal in $RRLS(L_b)$,
       {
        perform reset operation;
       }
   2c. For each literal in $CLS(L_b)$,
       {
        send cancel message to the corresponding OR process;
       }
```

Fig. 4. Improved backward execution algorithm.

Table 1
Comparison between Conery's and our method in the backward execution

Backtrack literal selected	Reset literals		Cancel literals	
	Conery's	Ours	Conery's	Ours
L_1	$\{L_2,\ldots,L_{10}\}$	$\emptyset$	$\{L_2,\ldots,L_{13}\}$	$\{L_2,\ldots,L_{13}\}$
L_2	$\{L_3,\ldots,L_{10}\}$	$\emptyset$	$\{L_6,\ldots,L_{13}\}$	$\{L_6, L_7, L_{11}\}$
L_3	$\{L_4,\ldots,L_{10}\}$	$\{L_4, L_5\}$	$\{L_8,\ldots,L_{13}\}$	$\{L_8, L_9, L_{10}, L_{12}, L_{13}\}$
L_4	$\{L_5,\ldots,L_{10}\}$	$\{L_5, L_8\}$	$\{L_9,\ldots,L_{13}\}$	$\{L_9, L_{10}, L_{12}, L_{13}\}$
L_5	$\{L_6,\ldots,L_{10}\}$	$\{L_8, L_9\}$	$\{L_{10},\ldots,L_{13}\}$	$\{L_{10}, L_{12}, L_{13}\}$
L_6	$\{L_7,\ldots,L_{10}\}$	$\{L_7\}$	$\{L_{11}, L_{12}, L_{13}\}$	$\{L_{11}\}$
L_7	$\{L_8, L_9, L_{10}\}$	$\emptyset$	$\{L_{11}, L_{12}, L_{13}\}$	$\{L_{11}\}$
L_8	$\{L_9, L_{10}\}$	$\{L_9, L_{10}\}$	$\{L_{12}, L_{13}\}$	$\{L_{12}, L_{13}\}$
L_9	$\{L_{10}\}$	$\{L_{10}\}$	$\{L_{12}, L_{13}\}$	$\{L_{12}, L_{13}\}$
L_{10}	$\emptyset$	$\emptyset$	$\{L_{13}\}$	$\{L_{13}\}$

steps as well as the number of messages in communication. Note that the improvement is made from the static analysis of DDG. The comparison between Conery's and our method is shown in Table 1 from the viewpoint of *cancel* literals and *reset* literals. For example, when L_3 is selected as a backtrack literal, Conery's method performs the *reset* operations for the *reset* literals $\{L_4, L_5,\ldots, L_{10}\}$ and sends the *cancel* messages to the OR processes for the *cancel* literals $\{L_8, L_9,\ldots,L_{13}\}$. For $\{L_8, L_9, L_{10}\}$, both *reset* and *cancel* occur. But, our method performs the *reset* operation for the *reset* literals $\{L_4, L_5\}$ and sends the *cancel* massages to the OR processes for the *cancel* literals $\{L_8, L_9, L_{10}, L_{12}, L_{13}\}$. There are no literals for which both *reset* and *cancel* occur in this case. In addition, five steps are reduced in performing the *reset* operation and one step in sending the *cancel* message.

For comparision from the viewpoint of backtracking, we consider a logic program:

$p(x_1, x_2,\ldots, x_6) :- p_1(x_1, x_3), p_2(x_3, x_4),$
$\quad p_3(x_3, x_5), p_4(x_2, x_3), p_5(x_4, x_5), p_6(x_2, x_6).$
$p_1(b, a).\ p_1(c, a).\ p_1(c, b).$
$p_2(a, b).$
$p_3(a, b).\ p_3(a, c).$
$p_4(b, a).\ p_4(c, a).\ p_4(c, b).$
$p_5(a, b).\ p_5(b, c).$
$p_6(c, a).$

The DDG of the logic program is shown in Fig. 6. The partial order of variables for generating a 6-tuple is determined from the corresponding DDG. It becomes x_3, x_4, x_5, x_2 in sequence. We exclude x_1 and x_6 from the ordering because there are no literals which consume the values of x_1 or x_6. Hence, only a 4-tuple is enough to generate a complete solution of the clause because the remaining two variables can be set appropriately according to the 4-tuple.

Assume that $\langle a, b, b, c\rangle$ is generated according to the variable ordering. At this time, only p_5 fails so that the backtrack literal is p_3, the *reset* literal is p_4, and the *cancel* literals are p_5, p_6 according to Conery's model. Hence, the 4-tuple which is generated next becomes $\langle a, b, c, b\rangle$. This makes p_6 fail, which succeeds at the previous tuple, $\langle a, b, b, c\rangle$. One more backtracking step is required to resolve the failure of p_6. Our improved model does not perform the *reset* operation for p_4 because p_3 and p_4 are independent literals, result-

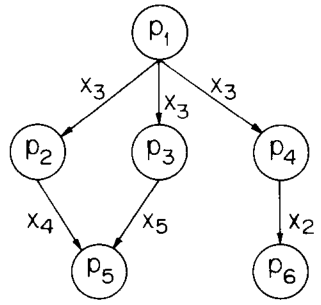

Fig. 6. A logic program and its data dependency graph.

ing in $\langle a, b, c, b\rangle$ which makes all literals a success.

4. Backtrack literal selection for multiple failure case

In this section we present a backtrack literal selection algorithm for the multiple failure case defined in [8].

During the backward execution for a failed literal, several OR processes can report failures to the parent AND process. Because the processing of the backward execution is nonpreemptive, these literals which report failures are stored into the *failed literals set* [8]. The multiple failure situation occurs when more than one OR process reports failures while the AND process is in the backward execution. In that case, the failed literals set contains more than one literal. This definition of multiple failure situation is different from Conery's in that Conery's definition depends on the current path of the backward execution while this definition which comes originally from [8] depends on the timing of the backward execution. Therefore, this definition is more realistic than Conery's in parallel processing environment.

The backward execution algorithm for the multiple failure case proposed by Woo and Choe [8], called Algorithm M, is not efficient because their algorithm is inherently based on the nested loop model. Because we devise the improved method of generating tuples for a clause, a more efficient backward execution algorithm for the multiple failure case can be sought for by using the improved method.

Because the backtrack literal selection part is not our concern, we simply denote the backtrack literal selection part as Algorithm BLS. In other words, Algorithm BLS covers the single failure case.

Fig. 7 describes Algorithm M for the multiple failure case in [8]. It gathers a set of backtrack literals for each literal in the failed literals set and then selects the leftmost literal among them to the linear ordering. The reason why it must select only one leftmost literal as a backtrack literal is that it inherently uses the nested loop model in generating tuples.

Fig. 8 represents our algorithm for the multiple failure case based on the improved method of generating tuples, which is called Algorithm I. Steps 1 and 2 are the same as those of Algorithm M, whereas the subsequent steps are different. From the backtrack literals set (i.e., *EL* in Fig. 8) obtained by executing Algorithm BLS for each literal in the failed literals set, Algorithm I constructs a set of groups such that each group consists of only dependent literals. The partitioning can be done simply by executing three steps for each literal, say L_i, in *EL* (note that *EL* is already arranged to the linear ordering): finding out the literals in *EL* which have a *dependent* relation with L_i, constructing a group with L_i and all those dependent literals, and deleting all the literals in that group from *EL*. The set of groups obtained by this scheme is unique for a given *EL*. Those groups represent the disjoint sources of the multiple failures so that Algorithm I selects one backtrack literal for each group. This is possible because the improved method of generating tuples is adopted.

```
1. For each literal $L_{f_i}$ $(i = 1, 2, \dots, n)$ in the failed literals set
   {
    1a. Invoke Algorithm BLS to find the backtrack literal $L_{b_i}$ for $L_{f_i}$;
    1b. If the previous step returns "AND process fails" as a result
        then exit this algorithm with "AND process fails"
        else continue;
   }
2. Enumerate $L_{b_i}$ $(i = 1, 2, \dots, n)$ to the linear order;
3. Select the leftmost literal (say, $L_b$) as the backtrack literal;
```

Fig. 7. Algorithm M [8].

```
1. For each literal $L_{f_i}$ $(i = 1, 2, \ldots, n)$ in the failed literals set
   {
   1a. Invoke Algorithm BLS to find the backtrack literal $L_{b_i}$ for $L_{f_i}$;
   1b. If the previous step returns "AND process fails" as a result
       then exit this algorithm with "AND process fails"
       else continue;
   }
2. Enumerate $L_{b_i}$ $(i = 1, 2, \ldots, n)$ to the linear order;
   //this list is denoted as EL//
3. Partition EL into a set of groups, $G_1, G_2, \ldots, G_k$, such that for $L_i \in G_m$ and $L_j \in G_n$, $m \neq n$, $L_i$ and $L_j$
   are independent.
4. For each group $G_i$ $(i = 1, 2, \ldots, k)$
   {
   4a. Select the leftmost literal, say $L'_{b_i}$, as a backtrack literal;
   4b. Do the backward execution for $L'_{b_i}$;
   {
```

Fig. 8. Algorithm I.

Let us compare Algorithm I with Algorithm M through the DDG in Fig. 5. Suppose the OR process for L_{13} reports failure so that the backward execution for the failure of L_{13} starts and during it the OR processes for L_{11} and L_{12} report failures. The backward executions for L_{11} and L_{12} are delayed until the backward execution for L_{13} is completed. After that, the algorithm for the multiple failures is invoked and finds the backtrack literal. Algorithm M selects only one backtrack literal, L_7, in this case, while Algorithm I selects two backtrack literals, L_7 and L_9, because there are two independent failure sources in this multiple failure case. The comparison between Algorithm M and Algorithm I is described in Table 2. For the example in Fig. 6, consider the following multiple failure case: $\langle a, b, b, b\rangle$ is generated at which both p_5 and p_6 fail. According to Algorithm M, p_3 is selected as a backtrack literal so that the 4-tuple generated next becomes $\langle a, b, c, b\rangle$ at which p_6 still fails. Algorithm I selects two literals, p_3 and p_4, as backtrack literals because p_3 and p_4 are independent. Therefore, the next 4-tuple becomes $\langle a, b, c, c\rangle$ at which all literals succeed.

Table 2
Comparison between Algorithm M and Algorithm I

Operation step	Failed literals set = $\{L_{11}, L_{12}\}$	
	Algorithm M	Algorithm I
1	$L_{b_1} = L_7$, $L_{b_2} = L_9$	$L_{b_1} = L_7$, $L_{b_2} = L_9$
2	$\{L_7, L_9\}$	$\{L_7, L_9\}$
3	Backtrack literal, $L_b = L_7$	Group G_1 and G_2, $G_1 = \{L_7\}$ $G_2 = \{L_9\}$
4		Backtrack literals, $L'_{b_1} = L_7$ $L'_{b_2} = L_9$

5. Concluding remarks and discussion

The backward execution algorithm based on the nested loop model in the AND/OR process model [1] produced many redundant evaluation steps and unnecessary message communications because it did not consider the relationship between literals. We have presented an improved backward execution algorithm using the static analysis of the data dependency graph. Several examples have shown that the improved method reduces the inefficiency of the backward execution algorithm in the AND/OR process model.

The multiple failure situation may occur very frequently under the AND/OR process model because so many OR processes are running in parallel. Algorithm I, which exploits the improved method of generating tuples in handling the multiple failure case, is presented and compared with Algorithm M proposed in [8]. Algorithm I determines several backtrack literals to resolve the multiple failure situation, while Algorithm M determines only one backtrack literal.

Our method can be used in the intelligent backtracking mechanism. Especially the selection of multiple backtrack literals is thought to play an important role in reducing backtrack steps in the parallel processing environment of the AND/OR process model. We are currently investigating its applicability to the intelligent backtracking scheme.

Acknowledgment

The authors wish to thank the anonymous referees and the communicating editor for their assistance in improving this paper.

References

[1] J. Conery, *The AND/OR Process Model for Parallel Interpretation of Logic Programs*, Ph.D. Dissertation, Univ. of California at Urvine, Tech. Rept. 204, 1983.
[2] J. Conery and D. Kiber, Parallel interpretation of logic programs, *Proc. Conf. on Functional Programming Languages and Computer Architecture* (1981) 163–170.
[3] N. Ito, N. Shimizu, M. Kishi, E. Kuno and K. Rokusawa, Data-flow based execution mechanisms of parallel and concurrent Prolog, *New Generation Computing, Vol. 3* (Springer, Berlin, 1985) 15–41.
[4] R. Kowalski, Algorithm = Logic + Control, *Comm. ACM* **22** (1979) 424–436.
[5] Y.J. Lin and V. Kumar, An intelligent backtracking algorithm for parallel execution of logic programs, *Proc. 3rd Internat. Conf. on Logic Programming* (1986) 55–68.
[6] G. Lindstrom and P. Panangaden, Stream-based execution of logic programs, *Proc. 1984 Symp. on Logic Programming* (1984) 168–176.
[7] S. Umeyama and K. Tamura, A parallel execution model of logic programs, *Proc. 10th Internat. Symp. on Computer Architecture* (1983) 349–355.
[8] N.S. Woo and K. Choe, Selecting the backtrack literal in the AND/OR process model, *Proc. 1986 Symp. on Logic Programming* (1986) 200–210.