ELSEVIER

Information Processing Letters 52 (1994) 191–199

Information Processing Letters

Path for AND-parallel execution of logic programs

Su-Hyun Lee [a,*], Do-Hyung Kim [b], Kwang-Moo Choe [a]

[a] *Programming Languages Laboratory, Department of Computer Science, Korea Advanced Institute of Science and Technology, 373-1, Kusung-Dong, Yusung-Gu, Taejon 305-701, South Korea*

[b] *Department of Computer Science, Sungshin Women's University, 249-1, 3-Ga, Dongsun-Dong, Sung-buk-Gu, Seoul 136-742, South Korea*

Communicated by K. Ikeda; received 9 September 1993; revised 12 May 1994

Abstract

An efficient backward execution method for AND-parallelism in logic programs is proposed. It is devised through a close examination of some relation over literals, which can be represented by paths in the data dependency graphs. The scheme is more efficient than other works in the sense that it performs less unfruitful resetting (and consequently, canceling) operations. Its rationale is also intuitively clear and natural. Furthermore, the relevant analyses needed for the proposal are largely achieved at compile time, which alludes to actual efficiency.

Keywords: Applicative (logic) programming; Parallel evaluation of logic programs; AND/OR Process Model; AND parallelism; Data dependency graph; Path

1. Introduction

It now seems well known that the need for parallel execution of programs becomes essential to make "smarter" computers. The *logic programming* paradigm is attractive especially in the afore-mentioned aspect, due to (massive) parallelism inherent in logic programs. The parallelism is often classified into four types: *AND, OR, stream*, and *search* parallelism [4], among which AND and OR parallelism are the most obvious ones, and the former is dealt with in this paper.

Whereas OR parallelism is conceptually simple (when neglecting the difficulty of handling multiple environments efficiently), we must solve complex problems concerning variables occurring in more than one body literal of a clause to exploit AND parallelism. Since those variables are shared among literals, *variable binding conflicts* can occur if we allow each literal to be resolved unconstrainedly. In order to prevent this problem, in many schemes to achieve AND parallelism, a partial ordering according to *data dependency analysis* is imposed upon body literals [2–4,6,10,12]. This ordering tells which literal is the *generator* of each shared variable (consequently, the other literals containing the variable become the *consumers* of it). Every consumer cannot be invoked until all the variables it consumes are bound. These generator–consumer relationships among body literals are generally represented by a directed acyclic graph, called *data dependency graph* (DDG) [4].

* Corresponding author. Email: suhyun@plhae.kaist.ac.kr.

SSDI 0020-0190(94)00144-8

The main issues of AND-parallel evaluation of logic programs have been the completeness and efficiency of methods to handle *failure* during execution of a literal, a situation where the literal cannot be resolved away with the current given (if any) bindings. We observe here that semantically clear solutions to these issues might be systematically devised by closely examining "some" relations among literals.

In this paper, we propose another scheme to manage failure in AND-parallel execution of logic programs. The scheme is formulated by analyzing the foregoing relations among literals, which are represented by paths in a DDG. Its rationale is rather simple and straightforward. Furthermore, the scheme also exhibits more efficient behavior than existing methods.

In the next section, we briefly review a scheme for AND parallelism as our working framework and DDGs in more detail. The DDGs are refined and incrementally subdivided into three kinds. Some preliminary definitions are also given. Section 3 describes our proposal, a new scheme for handling problems in AND parallelism. A comparison of our proposal with other works is presented in Section 4. A few further optimizations of the proposal are discussed in Section 5, followed by concluding remarks.

2. Background and definitions

We describe a representative method for parallelism in logic programs, upon which our study is based, to identify the problems tackled in the paper. Moreover, definitions necessary for presenting our proposal in Section 3 are described.

2.1. The AND / OR process model

Among many models ever proposed to exploit AND parallelism, the *AND / OR Process Model* [4] is much widely known since the computation structures of logic programs are naturally reflected in the model.

AND-parallel execution in the AND/OR Process Model consists of two phases: *forward* and *backward execution*. Forward execution activates literals with no unsolved generators, according to the pre-constructed generator–consumer relation. On the other hand, backward execution is triggered at the time of failure. In the phase, some generator literal, called a backtrack literal, is selected and redone to alleviate the failure. In general, there exist several candidates for the backtrack literal, thus a certain rule is needed to deterministically choose it. This rule is dictated by the *backtrack literal selection algorithm*, which is a part of the backward execution algorithm. Most backtrack literal selection algorithms adopts a straightforward and natural method, called *nested loop model* [4] for tuple generation, to choose a unique backtrack literal. This model imposes a linear ordering on body literals, which is made by traversing a DDG (denoting the partial ordering) in a breadth-first manner. Then the rightmost literal in the linear ordering among candidate backtrack literals is always selected as the backtrack literal. After the backtrack literal is redone, some generator literals succeeding the backtrack literal in the linear ordering must be re-started from the beginning, lest we miss any possible solutions. This operation is named *resetting* and handled by the *resetting algorithm*, which is also run during backward execution. Our new resetting algorithm is more efficient than others as you can see in Section 4, in the sense that fewer literals are reset. For further details about the AND/OR Process Model and problems regarding backward execution, the reader is referred to [4].

2.2. Data dependency graphs

To begin, we present some basic definitions for graphs. For definitions not stated here, we assume the same meaning as those in [1]. In our notation, an edge is denoted by the form (u, v) for undirected graphs and by the form $\langle u, v \rangle$ for digraphs. With each digraph D we can associate an undirected graph G on the same vertex set; corresponding to each edge of D there is an edge of G with the same ends. This graph G is the *underlying (undirected) graph* of D. Conversely, we can obtain a digraph from an undirected graph G by specifying an order on the ends of

each edge. Such a digraph is called an *orientation* of G. For a graph, an edge (u, v) (or $\langle u, v\rangle$) can be associated with a label X, denoted by the form $(u, v; X)$ (or $\langle u, v; X\rangle$). Such a graph is called a *labeled graph (or labeled digraph)*. A *subgraph* of G is a graph obtained by the removal of some edges of G. When all members of the vertex set have a specific ordering, the graph is called an *ordered graph*.

The basis of data dependency is variable sharing. In the first place we define a graph that reflects variable sharing.

Definition 2.1 (Variable-sharing graph). A labeled multi-graph $G_v = (V, E_v)$ is called a *variable-sharing graph* for a clause, where V is the set of all literals in the clause and an edge $(u, v; X)$ in E_v denotes that both literals u and v contain a variable X.

In constructing the generator–consumer relation, the main task is to determine generators. This selection is done either during or before forward execution. There is nondeterminism in selecting the generators. This is a kind of *don't care* [7] nondeterminism. That is, whichever literal is selected as the generator, the results of the logic program are equal. For efficiency, however, there can be some heuristics on selection of generators, such as in [4]. Another selection method is *competition* [10]. Since this method gives equal chance to each literal, the literals compete for becoming the generator. For flexibility to programmers, *variable annotation* can also be adopted as the selection rule. This method is used in [5,11]. When we fix the generator for each variable, then we get a data dependency graph.

Definition 2.2 (Data dependency graph). A labeled digraph $G_d = (V, E_d)$ called a *data dependency graph* for a clause, is an orientation of a subgraph of G_v, where an edge $\langle u, v; X\rangle$ in E_d denotes that u is the generator of a variable X and v is a consumer of X.

Once G_d is constructed, there exists a partial ordering on literals. It reflects the generator–consumer relation. Then forward execution proceeds according to this ordering.

When backward execution is initiated, the algorithm must select a backtrack literal. Since we do not analyze the cause of failure, we do not know which one(s) gave bad bindings. Thus there is nondeterminism in selecting the backtrack literal (this is a kind of *don't know* [7] nondeterminism). For safe execution, we need to fix a rule of selection. This is done by ordering generators in G_d as mentioned in Section 2.1. Now we get a total ordering on literals and another graph.

Definition 2.3 (Ordered data dependency graph). A simple undirected graph $G_o = (V, E_o)$, called an *ordered data dependency graph* for a clause, is an ordered underlying graph of G_d.

Hereafter, if not stated explicitly, DDG means G_o.

Definition 2.4 (Literal ordering). Let p and q be literals in a clause. If p precedes q in the vertex ordering in G_o then we say "p is *smaller* than q (or equivalently, q is *larger* than p)" and write $p <_l q$.

2.3. Path

Now we define some basic definitions about "paths," which are heavily used in presenting our new backward execution scheme in Section 3. For convenience of notation, we use P, Q for paths, S for a set, and p, q, r, f for literals.

Definition 2.5 (Path/Valid path). A sequence of literals $P = r_1 r_2 \ldots r_n$ $(n \geqslant 2)$ is a (simple) *path* if $(r_i, r_{i+1}) \in E_o$ for $1 \leqslant i < n$ and $r_i \neq r_j$ if $i \neq j$. The path P is called *valid* if $r_1 <_l r_i$ for $1 < i \leqslant n$. The length of P is n.

The valid paths are the same with the undirected (affection) paths in *reduced data dependency graphs* [3].

Definition 2.6 (Interior set/Start/End). Let $P = r_1 r_2 \ldots r_n$ be a path. The *interior set* of P is $\{r_2, \ldots, r_{n-1}\}$, denoted by $\text{IN}(P)$. The *start* of P is

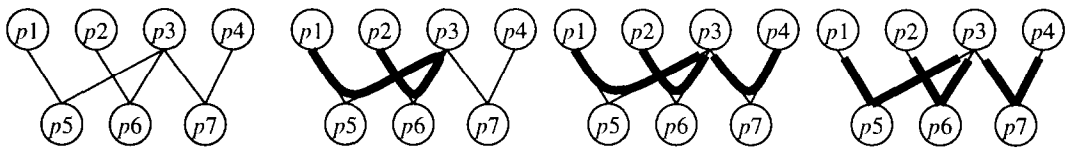

Fig. 1. An example of DDG and B-paths.

r_1 and the *end* of P is r_n. A path Q whose start is q and end is r can also be denoted by ${}_qQ_r$ for clarity.

3. Backward execution method

We first present the rationale of our backward execution method. Then relevant definitions and algorithms follow.

3.1. Rationale

Operations in backward execution can be explained in terms of a relation over literals, called *affection* [1] [6]. According to this relation, it is intuitively clear that literals which have affected a failed one are candidates for a backtrack literal. After redoing the backtrack literal, it is also obvious that we must reset some generator literals, which have been affected by the backtrack literal. (We further must *cancel* non-generator literals which have been affected by this backtracking.)

Affection is subdivided into two types: "forwarding values" and "causing redoing" [6]. The two types of affection are potentially represented in DDGs; "generator literals forward values to consumer ones" and "the failure of consumer literals causes redoing some generator ones". An affection relation is the transitive closure of these two types of subrelations i.e., values-forwarding affection and redo-causing affection. Under this observation, we represent the relation by a set of (start/end pairs of) paths between two literals in a DDG; the affecting literal is the start of the path and the affected one is the end of the path.

[1] Used in this paper synonymously with "effect".

In this setting, when a literal fails, the start of a path whose end is the failed literal is a candidate for the backtrack literal. Under the nested loop model, of course, this backtrack candidate must be smaller than the failed literal and its redo-causing ones (see Definition 2.5). After the backtrack literal is redone, the end of a path whose start is the backtrack literal is a candidate for resetting. It must also be a generator literal. For every such a path, it seems intuitively clear that the interior set is the set of "possible" redo-causing literals for the end of the path, so the end must be smaller than all members of the interior set according to the nested loop model, which constitutes the basis of the notion "B-path" (see Definition 3.1).

We thought that the above reasoning can be further refined as follows: for a path of concern, if its interior set is not included in the set of literals which have "actually" caused redoing its end, the path is not relevant to the "current" backtrack literal selection or resetting, since the path does not reflect the failure situation which "really" happened. This leads us to our new backward execution scheme in Section 3.3. As shown in Section 4, this observation makes our scheme more efficient than existing ones.

3.2. B-path

We refine valid paths so that the end of such a path is smaller than each member of its interior set.

Definition 3.1 (B-path). Let $P = r_1 r_2 \ldots r_n$ be a valid path. The path P is called a *B-path* if $r_n <_l r_i$ for $1 < i < n$.

Example 3.2. A DDG for the following query is shown in Fig. 1(a).

$:\text{-}\ p1(A), p2(B), p3(C), p4(D),$
$p5(A, C), p6(B, C), p7(C, D)?$

All valid paths and B-paths of the DDG are listed below. (Henceforth,if there is no danger of confusion, we shall abbreviate a literal to its "index," such as 1 for $p1(A)$ in the example. The index is an ordinal number of total ordering of literals.) The B-paths are underlined:

15, 26, 35, 36, 37, 47, 153, 263, 374, 1536,
1537, 2635, 2637, 15362, 15374, and 26374.

In Figs. 1(b), (c), and (d), we visualize the B-paths. Fig. 1(b) contains one B-path, Fig. 1(c) contains three B-paths, and Fig. 1(d) contains six B-paths. In the figures thick lines denote B-paths. As said before, B-paths represent affection. For example, the meaning of the B-path 15362 is that literal $p1$ affects literal $p2$ via literals $p5$, $p3$, and $p6$.

3.3. Backward execution

When a literal which consumes two or more variables fails, we cannot know the reason of the failure because we assume no cause analysis. While one candidate literal is selected as the backtrack literal (the rightmost literal in nested loop model), the other candidates should still remain as candidates for (possible) future failure. We keep them in the "failure history" of the backtrack literal. (Reducing the number of possible failure histories is discussed in Section 5.4.)

Definition 3.3 (Failure history). For a literal r, its *failure history* $\text{FH}(r)$ is:
(1) Initially, $\text{FH}(r) := \emptyset$.
(2) When r is redone by the failure of f, $\text{FH}(r) := \text{FH}(r) \cup \text{FH}(f) \cup \{f\}$.
(3) When r is reset or canceled, $\text{FH}(r) := \emptyset$.

Now we are in a position to present our proposal for backtrack literal selection and resetting.

Algorithm 3.4 (Backtrack literal selection).
When f fails, r is the backtrack literal,
where r is the largest among literals such that ${}_rP_f$ is a B-path and $\text{IN}(P) \subseteq \text{FH}(f)$.

Algorithm 3.5 (Resetting/Canceling).
When r is re-started (redone, reset, or canceled), for q such that ${}_rP_q$ is a B-path,
(1) cancel q, if $\text{IN}(P) = \emptyset$ and
(2) reset q, if $\text{IN}(P) \neq \emptyset$ and $\text{IN}(P) \subseteq \text{FH}(q)$.

4. Comparison with other works

In this section, our scheme will be compared with other backward execution methods: (1) Woo and Choe's method [12] for backtrack literal selection and (2) Kim and Choe's method [6] for resetting. (The proofs of the lemmas and theorems in the paper appear in [8].)

4.1. Redo cause set

Woo and Choe [12] proposed a backtrack literal selection algorithm which uses *redo cause set* (RCS). The RCS of a literal p, $\text{RCS}(p)$, is a set of literals that captures the failure history of p. RCS is the same with FH of our scheme. (The rightmost of a set of literals S is the largest literal in S, denoted by $\text{RIGHTMOST}(S)$. The parent set of a literal is all generators of the literal, i.e. $\text{PARENT}(q) = \{p \mid (p, q) \in E_0,\ p <_l q\}$.) Their algorithm is as follows.

Algorithm 4.1 (Backtrack literal selection in the RCS method [12]).
1. When f fails, collect the RCS of f, i.e.
   $S_1 := \text{RCS}(f) \cup \{f\}$
2. Obtain the parent set of S_1, i.e.
   $S_2 := \text{PARENT}(S_1)$
3. Select the rightmost literal in S_2 except for S_1, i.e.
   $r := \text{RIGHTMOST}(S_2 - S_1)$
4. Update RCS of r, i.e.
   $\text{RCS}(r) := \text{RCS}(r) \cup S_1$

The next lemma gives an important property of the RCS.

Lemma 4.2. *For a literal r, all members in* $\text{RCS}(r)$ *are connected in the DDG, and* $r <_l p$ *for any* $p \in \text{RCS}(r)$.

The backtrack literal selection in the RCS method is rephrased in the following theorem using B-paths.

Theorem 4.3. *A literal r is selected as the backtrack literal for the failure off in the RCS method iff there exists a B-path ${}_rP_f$ such that* $\text{IN}(P) \subseteq \text{RCS}(f)$ *and there is no B-path ${}_{r'}P'_f$ such that* $r <_l r'$ *and* $\text{IN}(P') \subseteq \text{RCS}(f)$.

From the above theorem, we can conclude that the intelligence of the two methods are equal for backtrack literal selection.

4.2. Dynamic affecting set

Kim and Choe [6] proposed an efficient resetting method which uses *dynamic affecting set* (DAS). For the purpose of reference and comparison, the key definitions of their method are contained here.

Definition 4.4 (Dynamic affecting set [6]). Let p and q be literals. Then "p DIRECTLY_D_AFFECT q" iff $p <_l q$ and either (1) $q \in \text{CHILDREN}(p)$ or (2) there is a literal r such that $r \in \text{CHILDREN}(p)$ and r caused redoing q. (CHILDREN(p) denotes p's consumers in the DDG.) The transitive closure of DIRECTLY_D_AFFECT is denoted by D_AFFECT. Then,

$$\text{D\_AFFECTING}(p) = \{q \mid p \text{ D\_YAFFECT } q\}.$$

Definition 4.5 (Dynamic canceled/reset literals set [6]). Let p and q be literals.

(1) $\text{D\_CANCEL}(p) = \{q \mid q \in \text{CHILDREN}(r),\ r \in \{p\} \cup \text{D\_AFFECTING}(p)\}$.

(2) $\text{D\_RESET}(p) = \text{D\_AFFECTING}(p) - \text{D\_CANCEL}(p)$.

The following lemma says about the relationship between the DAS and the B-path.

Lemma 4.6. *Let p and q be literals. If there exists a B-path ${}_pP_q$ such that* $\text{IN}(P) \subseteq \text{FH}(q)$, *then p* DIRECTLY_D_AFFECT q.

Since when a literal is reset in our scheme, so do DAS method, the set of reset literals in our scheme is a subset of that in the DAS method (in fact, our set is a proper subset). Now we show that our scheme is more intelligent than the DAS method by an example. Let us consider Fig. 1(a) and the following scenario:

(1) $p1$, $p2$, $p3$, and $p4$ succeed with {A/a0}. {B/b0}, {C/c0}, and {D/d0}, respectively.
(2) $p6$ fails; $p3$ is the backtrack literal.
(3) $p3$ succeeds with {C/c1}.
(4) $p5$ fails; $p3$ is the backtrack literal.
(5) $p3$ fails; $p2$ is the backtrack literal, $p3$ is reset.
(6) $p7$ fails; $p4$ is the backtrack literal.
(6-1) $p4$ succeeds with {D/d1}.
(6-2) $p7$ fails; $p4$ is the backtrack literal.
(7) $p4$ fails; $p3$ is the backtrack literal, $p4$ is reset.
(7-1) $p3$ succeeds with {C/c1}.
(7-2) $p7$ fails; $p4$ is the backtrack literal.
(7-3) $p4$ succeeds with {D/d1}.
(8) $p2$ fails; $p1$ is the backtrack literal, $p2$ is reset.

At the stage (8), $p3$ is not reset because $\text{FH}(p3) = \{p4, p7\}$ in our scheme. In the DAS method, however, $p3$ is reset because $\text{D\_RESET}(p1) = \{p2, p3\}$. The failure and redo in the step (7) are independent of the failure of $p2$, thus there is no need to reset $p3$. Our scheme is more faithful to the principle that "there is no need to reset literals which are not affected by the backtrack literal."

5. Some optimizations

In this section, we describe various analyses on path, which can be used to optimize our scheme.

5.1. Reducing the number of edges in DDGs

We describe deletion of some edges in a DDG with no effect on backward execution. The idea of the deletion is based on the transitivity of affection: if a literal p affects a literal q and q affects a literal r then p affects r. A DDG shows generator–consumer relation. As stated in Section 3.1, the relation can be viewed as "direct"

affection from generators to consumers. From transitivity of affection, we can say that a generator affects all the descendants of it. (The descendants of a literal is recursively defined as the literal itself or the consumers of its descendants.) If there exist two or more values-forwarding relationships between a generator and a consumer, the affection is redundant. Then we can remove redundancy in affection relation.

Definition 5.1 (D-edge). An edge (r_1, r_n) in a DDG is a *D-edge* if there exists a path $r_1 r_2 \ldots r_n$ such that $n > 2$ and $r_i <_l r_{i+1}$ for $1 \leqslant i < n$.

Theorem 5.2. *A D-edge can be deleted from the DDG without any effect on backward execution.*

An example of a D-edge is shown in the DDG of Fig. 2(a). The edge $(p1, p4)$ is useless, because it is overridden by $(p1, p3)$ and $(p3, p4)$.

5.2. Reducing the length of B-paths

When a path is used for backward execution, some information in the path may be redundant. After removing this redundancy, we can also represent the path in a shorter form with no effect on backward execution.

Now we consider the example in Fig. 2(b). Let us suppose $p5$ and $p4$ failed in that order and $p3$ was selected as the backtrack literal. Then the failure history of $p3$ is $\{p5, p4\}$. When $p3$ fails, $p2$ will be selected as the backtrack literal. In this situation, we can infer that if the failure history of $p3$ contains $p4$, then it contains $p5$, too. In other words, without the failure of $p5$, the failure history of $p3$ cannot contain $p4$. Therefore, the information that the failure history of $p3$ contains $p5$ does not contribute to the selection of a backtrack literal for the failure of $p3$.

When a B-path P is used in backward execution, the interior set IN(P) is compared with the failure history (Algorithms 3.4 and 3.5). Since some literals in the failure history do not contribute to backward execution, there is no need to keep such literals in IN(P). Thus a B-path can be shortened by removing non-contributing literals. The next theorem says the contributing literal is only the second element of a B-path.

Theorem 5.3. *Let $P = r_1 r_2 \ldots r_n$ be a B-path. We can shorten P as $r_1 r_2 r_n$ without any effect on backward execution.*

The B-paths of the DDG of Fig. 2(b) are 13, 24, 35, 45, 354, 1354, 2453, and 13542. In the above example, the path 2453 is used for backtracking. As a result of the above theorem, the path can be shortened to 243.

By storing critical information only, selecting a backtrack literal and resetting become much simpler. The backtracking and resetting algorithms can use membership test instead of set-inclusion test. Furthermore, we can save the space for paths. According to Theorem 5.3, the length of a B-path is at most three.

Theorem 5.4. *The number of B-paths for a clause is at most $O(n^3)$, where n is the number of the body literals.*

5.3. Refinement of B-paths

In our scheme, affection is represented by paths. From the transitivity of affection, if there exist paths from p to q and from q to r, then

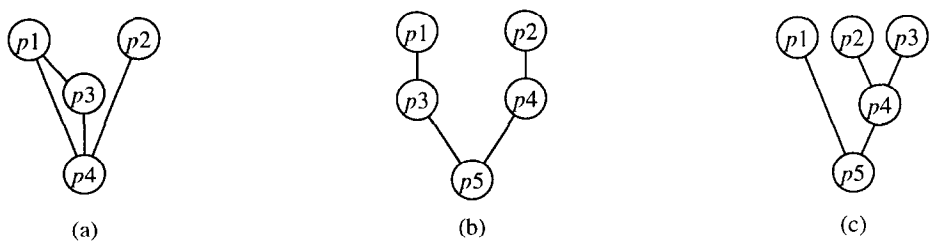

Fig. 2. Examples of DDGs.

there is no need to retain a path from p to r. The next lemma shows the transitivity of paths.

Lemma 5.5. *If there are two B-paths* ${}_pP_r$ *and* ${}_qQ_r$ *such that* $p <_l q$ *and* $\text{IN}(Q) \subset \text{IN}(P)$, *then there exists a B-path* ${}_pP'_q$.

We incorporate the above observation into RB-paths.

Definition 5.6 (RB-path). Let ${}_pP_r$ be a B-path. The path P is called an *RB-path* if there is no B-path ${}_qQ_r$ such that $p <_l q$ and $\text{IN}(Q) \subset \text{IN}(P)$.

When the backtrack literal selection algorithm uses RB-paths, the number of paths to be tested is reduced.

5.4. Refinement of failure history

The possible number of failure histories is the number of all subsets of all literals. However, some failure history is useless because it does not contribute to the selection of backtrack/reset literals. Thus the set of failure histories for a literal can be restricted.

Definition 5.7 (Set of failure histories). For a literal r, the set of possible failure histories is: $\text{SFH}(r) = \{\bigcup_{P \in S} \text{IN}(P) \mid S$ is a non-empty subset of the set of all the B-paths, each of which end is $r\}$.

By restricting failure history, in general, a literal has a small number of failure histories. For the efficiency of backward execution, we can construct a *finite state automaton* for a literal. Since the status of a literal during AND-parallel execution can be defined by means of its failure history, a state of the automaton is a failure history. The operations of the backward execution (backtrack literal selection, resetting, and canceling) change a state of the literal. At each state of the automaton, there is a unique backtrack literal because we can determine the backtrack literal if the failure history of a failed literal is given.

Example 5.8. The various paths for the DDG in Fig. 2(c) are shown in Table 1.

6. Concluding remarks

In the paper, we proposed another backtrack literal selection and resetting algorithm for AND parallelism in logic programs. The suggested scheme exhibits more efficient behavior in resetting than other existing ones, with the same efficiency in backtrack literal selection. Thus, the scheme can be said to be more efficient than others in overall execution. Some possible further optimizations for the proposal are also addressed.

The scheme has been built up from a simple and intuitively clear notion called affection [6] and compared with others somewhat formally. We thought that the intuitive justification of the method might be sufficient, but a rigorous correctness proof to it is also in progress [9].

As for the degree of the effectiveness/generality of the method (like any selective resetting scheme), it may be said to be rather dependent on programs of concern. If a program is much nondeterministic and its DDG has larger width compared with its depth, the gain via selective resetting (e.g., using our method) will become considerable. Additionally, we would like to point out that most part of the analysis required for the scheme can be performed at compile time, which implies practical efficiency.

Table 1

r	1	2	3	4	5
Valid paths whose end is r		1542	1543, 243	154, 24, 34	15, 245, 345, 45
B-paths whose end is r		1542	1543, 243	154, 24, 34	15, 45
RB-paths whose end is r		1542	243	34	45
SFH(r)		{{4, 5}}	{{4, 5}, {4}}	{{5}, ∅}	{∅}

References

[1] J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications* (Macmillan, London, 1976).

[2] J.-H. Chang, A.M. Despain and D. DeGroot, AND parallelism of logic programs based on static data dependency analysis, in: *Proc. CompCon Spring* (1985) 218–225.

[3] K.-M. Choe, M.J. Lee and N.S. Woo, Multiple backtracking in the AND process of logic programming, in: *Proc. TENCON 87 – IEEE Reg. 10 Conf.* (1987) 826–829.

[4] J.S. Conery and D.F. Kibler, AND parallelism and nondeterminism in logic programs, *New Generation Comput.* **3** (1) (1985) 43–70.

[5] S. Gregory, *Parallel Logic Programming in PARLOG: The Language and Its Implementation* (Addison-Wesley, Reading, MA, 1987).

[6] D.-H. Kim and K.-M. Choe, Yet another efficient backward execution algorithm in the AND/OR Process Model, *Inform. Process. Lett.* **40** (4) (1991) 201–211.

[7] R. Kowalski, *Logic for Problem Solving* (North-Holland, Amsterdam, 1979).

[8] S.-H. Lee, D.-H. Kim and K.-M. Choe, The path model for parallel evaluation of logic programs: Method and analysis, Tech. Rept. CS-TR-93-79, Dept. of Computer Science, KAIST, 1993.

[9] S.-H. Lee, D.-H. Kim and K.-M. Choe, Correctness proof of the path model, Tech. Mem., Prog. Lang. Lab., Dept. of Computer Science, KAIST, in preparation.

[10] K.W. Ng and H.F. Leung, Competition: A model of AND-parallel execution of logic programs, *Comput. J.* **33** (3) (1990) 215–218.

[11] E. Shapiro, A subset of Concurrent Prolog and its interpreter, Tech. Rept. TR-003, ICOT, 1983.

[12] N.S. Woo and K.-M. Choe, Selecting the backtracking literal in the AND/OR Process Model, in: *Proc. 3rd Internat. Symp. on Logic Programming* (1986) 200–210.