SICS/R-90/9007

Generating AND-parallel Execution Expressions

by

A. Thomas Sjöland

SICS research report
R90007
ISSN 0283-3638

Generating AND-parallel Execution Expressions

A. Thomas Sjöland

March 27, 1990

Swedish Institute of Computer Science, Box 1263, S-164 28 Kista, Sweden

Internet: alf@sics.se

Keywords: Parallel execution graphs, restricted AND-parallelism, logic programming, graph rewriting

Project: Declarative programming.

Abstract

An AND-parallel execution model for logic programs needs a representation of clauses that explicitly expresses the possible parallelism. It is natural to consider a clause body to be a set of indexed goals where the indices are partially ordered and the order represents the required execution ordering. This representation is not sufficient for an efficient parallel execution model, so we need another representation which expresses the parallelism. We define an algorithm to generate balanced execution expressions for parallel execution from the more general form with partially ordered indexed literals.

1 Introduction

Partially ordered goal literals are not a sufficient representation for an efficient AND/OR-parallel execution, since e.g. backtracking is difficult to implement based on this representation (see e.g. [6]). If we can guarantee that the body can be written as a balanced execution expression, more efficient schemes can be designed [9] giving e.g. intelligent backtracking or an AND-parallel scheme with reduced reexecution. This is of utmost importance since such schemes can reduce the complexity of the search, which compensates for the loss of parallelism we get when we introduce extra synchronisations [9].

After showing how partially ordered clauses are constructed, [1], I will here present a graph rewriting algorithm to convert partially ordered clauses into balanced execution expressions.

[1] The conditions for the partial order should be simplified through a program analysis step, e.g. formulated as an abstract interpretation, but that is not the topic here.

When talking about the clause graph I will mean the connected, directed acyclic graph formed for the body of the clause when letting the indexed literals be the goal nodes, and the order between indices be the arcs.

2 Direct construction of clause graphs

Without knowledge of the interdependence of variables in a clause given by a dataflow analysis there have to be several different versions of the execution order in order to guarantee independence between literals that execute in parallel. At run time the generated code should check the conditions for independence in the given environment to determine the current clause order before starting parallel resolutions. I will first give an algorithm to compute conditions for dependence expressed as the order relation $\prec$. The discriminating relation used as a run-time test is *reaches* holding for pairs of variables of a clause. This relation holds if the binding of one variable instance can possibly be restricted in a resolution sequence involving an instance of the other.

2.1 Algorithm for construction of clause graphs

A partially ordered clause, Cl, can be written:

$$Head \leftarrow Syncs : Body.$$

where $Head$ is a literal, $Body$ is a set of indexed literals $\{L_i\}_{i \in I}$ and $Syncs$ is a set $\{P_k \ if \ Cond_k\}$ where each P_k is a p.o. relation $\prec$ over I.

The conditions $Cond_k$ are of the form $\bigwedge\{reaches(v_i, v_j)\}$ for variables v_i, v_j in the clause and the conditions of the set $Syncs$ cover all possible data dependency situations. The generated clause code should check the conditions for the sequentialisation after head unification and then execute the clause using the clause graph implied by the conditions.

Given a clause containing a set of indexed literals $\mathbf{L}$ and a set of variables $\mathbf{V}$, the conditions for the clause graph are formed thus: ($Vars(l)$ gives the set of variables in a literal l.)

- for each pair of literals $l_m, l_n \in \mathbf{L}$ such that l_m is located syntactically before l_n and for each pair of variables $v_i, v_j \in \mathbf{V}$ such that $v_i \in Vars(l_m) \wedge v_j \in Vars(l_n)$: Assert $(m \prec n \ if \ reaches(v_i, v_j))$

The conditions are simplified by iterating the following rewrites on the set Syncs.

If $Cond_1 \rightarrow Cond_2$, $E_1 \equiv (P \ if \ Cond_1) \in Syncs$ and $E_2 \equiv (P \ if \ Cond_2) \in Syncs$ then $Syncs$ rewrites to $Syncs \setminus \{E_2\}$.

If $(E_1 \equiv (P_1\ if\ Cond) \in Syncs$ and $E_2 \equiv (P_2\ if\ Cond) \in Syncs)$ then $Syncs$ rewrites to $(Syncs \setminus \{E_1, E_2\}) \cup \{(P_1 \cup P_2)\ if\ Cond\}$.

□

2.2 Balanced clause graphs

A balanced clause graph is the body of a partially ordered definite clause where the order satisfies an additional criterion.

Note that $\prec$ is transitive (it is a p.o.).

We identify two important characteristics for goal nodes, *fork* and *sync*.

- A **fork** node is a goal node where there is more than one outgoing arc. The intended operational semantics is to allocate the control structures for a parallel proof and start a parallel resolution.
- A **sync** node is a goal node with more than one incoming arc. The intended operational semantics is to await answer substitutions from all preceding goals in the graph and then to produce a combined result.

Associate a left parenthesis to each fork node and a right parenthesis to each sync node.

Definition 1 *(Balanced graph) A directed acyclic graph is* balanced *iff for each node of the graph there is a path containing that node from a source node to a sink node, such that the sequence of fork nodes and sync nodes forms a balanced parenthesis expression.*

In the figure it is indicated how a loss of parallelism is sometimes unavoidable when we impose these demands on the clause order.

I name a balanced subgraph an *independent unit*, or shorter, IU.

The produced result substitutions of an independent unit are the composition of tuples of substitutions in the cartesian product of answer substitutions from its threads, produced lazily, e.g. by means of an extended backtracking. Preceding result substitutions of the threads are kept by the fork node, in order not to lose any combinations. Memory limitations can enforce reexecution. Failure of one thread means that ongoing proofs of the other threads in the IU can be discarded.

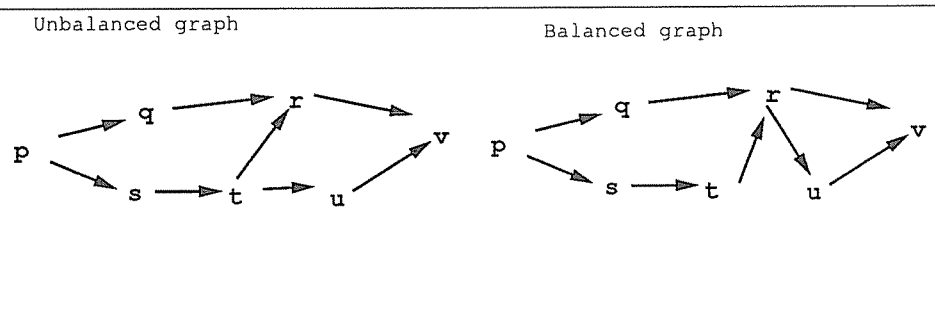

Figure showing how a loss of parallelism is inevitable when we introduce a balance between forks and syncs. The execution of u must wait for r.

2.3 Parallel execution expressions

A balanced clause graph can be written as a parallel execution expression:

$$Bal$$

where Bal is either a literal G, or, for a set of balanced clause graphs$\{Bal_i\}$, a parallel conjunction $\|\{Bal_i\}$ or a sequential conjunction $\&\{Bal_i\}$. The operators $\|$ and & are written infix for the binary case.

A *thread* is a Bal_i in a parallel conjunction.

This will be proven in a forthcoming publication.

3 An algorithm for computing a balanced graph from a clause graph

In order to utilise an efficient execution model for restricted AND-parallel resolution [6] the clauses of the program should be balanced. We will give an algorithm for transforming clause graphs to balanced clause graphs.

3.1 The algorithm for constructing balanced graphs from arbitrary clause graphs

We define an algorithm for generation of a system of balanced graphs with independent threads from a clause graph.

Given is an indexed set of literals and synchronisation conditions for these, expressed as the relation $\prec$ on pairs of indices, (i, k).

We construct a sequence of balanced subgraphs where each subgraph contains a number of independent threads without mutual synchronisation.

The criteria for start nodes are defined:

Definition 2 *(start node) A node is a* start node *if either (criterion a) there are 0 or more than 1 incoming arcs at the node or otherwise (criterion b) the node is located after a node with at least two outgoing arcs.*

3.1.1 The algorithm

The algorithm is stated as follows:
(A chain is a sequence of nodes ordered by the synchronisation relation).

1. Identify start nodes in the graph.

2. Build chains from the start nodes up to and including a node just before a start node. Determine for each chain thus formed the sets of in/out synchronisations.

3. Identify chains with common in/out synchronisations. Build an IU for each such group of chains and reduce the graph with respect to these. Iterate from point 1, if a new IU was formed.

4. Build an IU containing all chains with an empty set of incoming arcs (disregarding earlier IUs).

5. Reduce the graph with respect to the new IU and iterate from point 1 if it changed.

3.2 Properties of the algorithm

Any ordered clause graph will be mapped to a balanced graph by the algorithm.

Only redundant synchronisations are removed by the algorithm. Therefore the balanced graph formed by the algorithm is still a safe parallelisation (with respect to independence) if the original one was.

The drawback of added synchronisations is compensated for by the fact that reexecution will not so often be necessary.

3.3 Example

The algorithm here defined will behave as follows given the clause graph with the order shown in the figure:

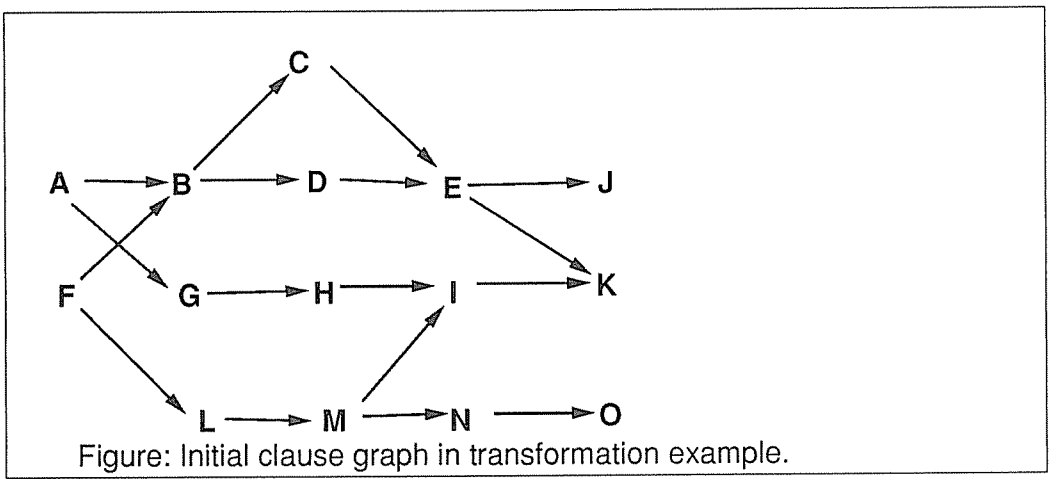

Figure: Initial clause graph in transformation example.

$A \prec B$, $A \prec G$, $B \prec C$, $B \prec D$, $C \prec E$, $D \prec E$, $E \prec J$, $E \prec K$, $F \prec B$, $F \prec L$, $G \prec H$, $H \prec I$, $I \prec K$, $L \prec M$, $M \prec I$, $M \prec N$, $N \prec O$

Step 1: Identify start nodes.

Criterion a holds for A,B,E,F,I,K of the initial clause in the figure.

Criterion *b* holds for N,G,D,C,L,J of the initial clause in the figure.

Step 2: Build chains.

We build chains with a start node followed by a maximal sequence of non start nodes.

	Node	Sync in	Sync out
1:	A	{}	{2,8}
2:	B	{1,7}	{3,4}
3:	C	{2}	{5}
4:	D	{2}	{5}
5:	E	{3,4}	{6,10}
6:	J	{5}	{}
7:	F	{}	{11,2}
8:	G → H	{1}	{9}
9:	I	{8,11}	{10}
10:	K	{9,5}	{}
11:	L → M	{7}	{9,12}
12:	N → O	{11}	{}

Step 3.

Build the balanced graph IU1 for chains with common in/out synchronisations.

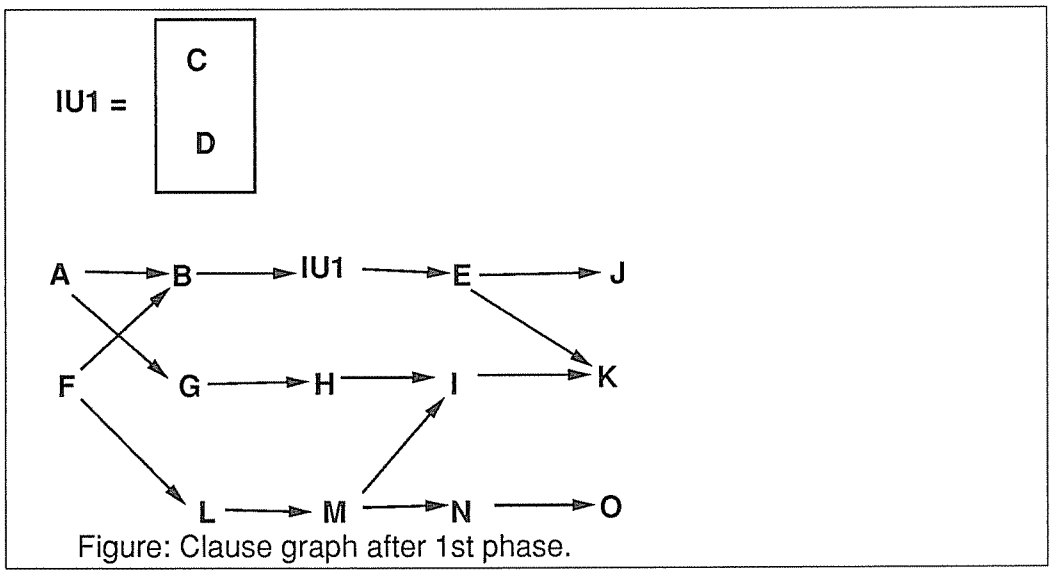

Figure: Clause graph after 1st phase.

Reduce the graph by substituting the balanced subgraph IU1 and taking away synchronisation links that have become redundant.

Iterate from step 1.

Identify start nodes according to criterion *a* [A,B,F,I,K]
Identify start nodes according to criterion *b* [N,G,L,J]

	Node	Sync in	Sync out
1:	A	{}	{2,7}
2:	B → IU1 → E	{1,3}	{5,9}
3:	F	{}	{2,8}
4:	I	{7,8}	{5}
5:	K	{4,2}	{}
6:	N → O	{8}	{}
7:	G → H	{1}	{4}
8:	L → M	{3}	{4,6}
9:	J	{2}	{}

Step 3.

There are no chains with common in/out synchronisations.

Step 4.

Build sequences of balanced subgraphs from the chains.

choose all chains with an empty set of in-synchronisations (except IU1). This corresponds to the set of literals without any particular synchronisation requirements.

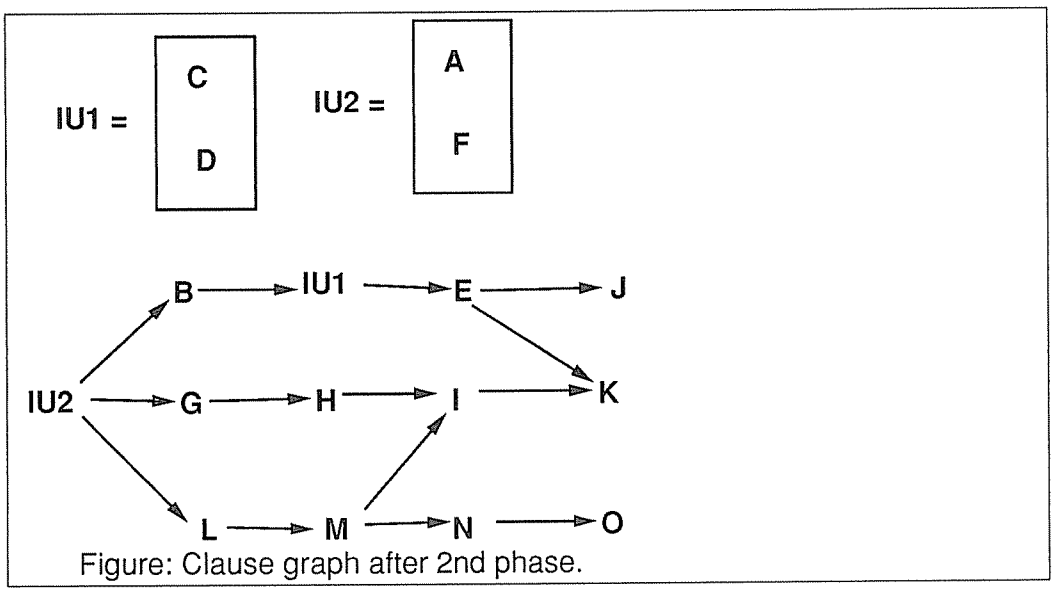

Figure: Clause graph after 2nd phase.

Chains no. (1,3) thus forms a balanced subgraph, IU2. The set of out-synchronisations for this new IU is the union of those for the components.

Iterate from step 1.

Identify start nodes according to criterion *a* [IU2,I,K]
Identify start nodes according to criterion *b* [B,N,G,L,J]

	Node	Sync in	Sync out
1:	IU2	{}	{2,5,6}
2:	B → IU1 → E	{1}	{4,7}
3:	I	{5,6}	{4}
4:	K	{3,2}	{}
5:	G → H	{1}	{3}
6:	L → M	{1}	{3,8}
7:	J	{2}	{}
8:	N → O	{6}	{}

Step 2,3

There are no chains with common in/out- synchronisations.

Step 4 Take away IU2 from the sets of in-synchronisations. Choose all chains with an empty set of in-synchronisations.

This gives us chains no. (2,5,6) which forms IU3.

Iterate from step 1

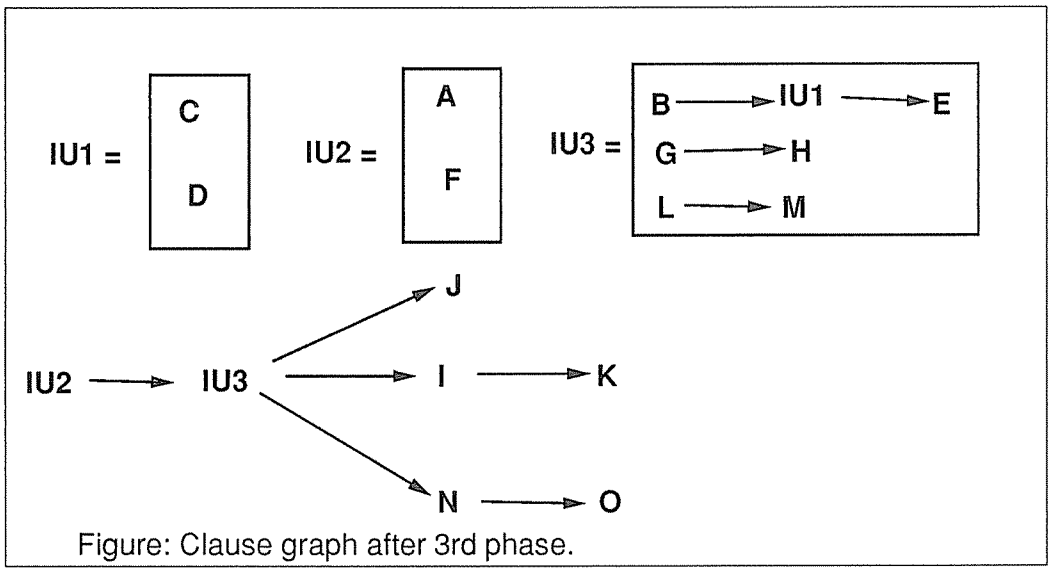

Figure: Clause graph after 3rd phase.

Identify start nodes according to criterion a [IU2]
Identify start nodes according to criterion *b* [N,J,I]

Step 2: Build chains

	Node	Sync in	Sync out
1:	IU2 → IU3	{}	{2,3,4}
2:	J	{1}	{}
3:	I → K	{1}	{}
4:	N → O	{1}	{}

Step 3.

The chains (2,3,4) have common in/out-synchronisations.

This gives the last balanced subgraph, IU4.

Step 5.

All nodes are covered by the chain IU2 → IU3 → IU4.

We can now build the execution expression by expanding the IUs.

We get:

$$(A\|F)\&((B\&(C\|D)\&E)\|(G\&H)\|(L\&M))\&(J\|(I\&K)\|(N\&O))$$

which corresponds to a graph with some added (true) goal nodes: IU1,IU2,IU3,IU4,END:

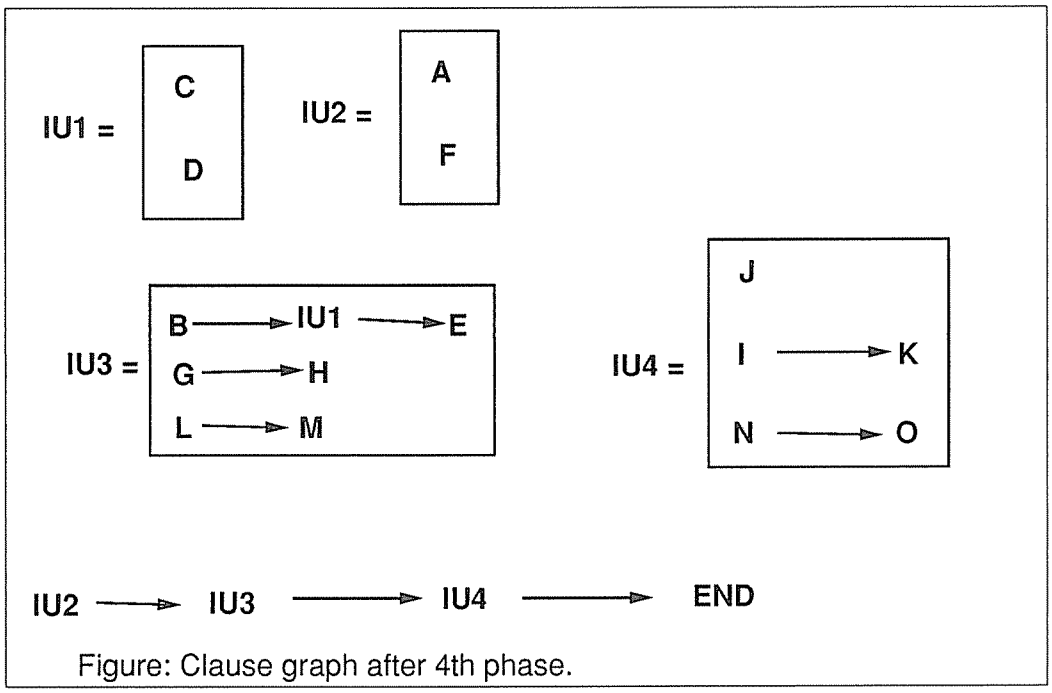

Figure: Clause graph after 4th phase.

$IU2 \prec A$, $IU2 \prec F$, $A \prec IU3$, $F \prec IU3$, $IU3 \prec B \prec IU1 \prec E$, $IU3 \prec G \prec H$, $IU3 \prec L \prec M$, $E \prec IU4$, $H \prec IU4$, $M \prec IU4$, $IU4 \prec J$, $IU4 \prec I \prec K$, $IU4 \prec N \prec O$, $J \prec END$, $K \prec END$, $O \prec END$

We can easily check that the generated graph is balanced and implies the original order relation $\prec$.

4 Qualities of this contribution

4.1 Originality

The graph property defining a balanced graph and the presented algorithm have to our knowledge not been elaborated elsewhere.

4.2 Efficiency

The efficiency of the algorithm depends on the choice of representations in the implementation. This aspect is not considered here.

4.3 Application

The primary application of the presented algorithm is as a part of a compiler for Horn clause logic programs where conjunctions which can safely be resolved in parallel should be explicitly given to an efficient run-time system like that presented in [6].

The algorithm is not restricted to logic programs, but lends itself to all systems where the input order is expressed as a directed acyclic graph and the criterion for balanced expressions is relevant. E.g. a parallelising compiler for functional programs or a planning program could utilise this algorithm.

4.4 Termination

The algorithm terminates. The argument is the following:

A rewrite results in the construction of a balanced subgraph. Either the constructed graph is a total order in which case we have a balanced graph, or we have a balanced subgraph, which is replaced by a node by the algorithm. Since the graph is finite it becomes gradually smaller and termination follows trivially.

4.5 Correctness

We would like to later give a formal proof that the algorithm generates balanced execution expressions for any ordered definite clause.

The informal sketchy argument we have so far is the following:

The order relation expressed by the graph is transitive. Any rewrite step will take away only redundant synchronisation arcs with respect to the transitivity and the introduced new arcs will construct new balanced subgraphs. Whenever a balanced subgraph is constructed the algorithm is iterated. Therefore it does not terminate prematurely.

5 Related Work

Several authors have written about AND-parallel execution of logic programs based on some form of parallel execution expressions [3, 5, 8, 1, 7, 2, 6, 10, 4]. E.g. the RAP execution strategy formulated in [6] can utilise these execution expressions to build the needed "conditional graph expressions".

References

[1] Jung-Herng Chang. *High Performance Execution of Prolog Programs Based on a Static Data Dependency Analysis.* Thesis, University of California, Berkeley, 1985. Report No. UCB/CSD 86/263.

[2] J.S. Conery. Implementing backward execution in non-deterministic and-parallel systems. In *Proceedings, Melbourne, Australia*, pages 633–653. The International Conference on Logic Programming, May 1987.

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

[4] Saumya Kanti Debray. Synthesizing control strategies for and-parallel logic programs. Technical report, Dept. of Comp. Sci. Univ. of Arizona, Tucson, 1988. Preliminary Draft.

[5] Doug deGroot. Restricted and-parallelism. Technical report, IBM Thomas J. Watson Research Center, Yorktown Heights, 1984. Proc of the International Conf. on 5th Gen. Comp. Systems, Tokyo.

[6] Manuel V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Computer Sciences, The University of Texas at Austin, August 1986. TR-86-20.

[7] Dean Jacobs and Anno Langen. Compilation of logic programs for multiprocessor systems. Technical report, Computer Science Dept., Univ. of South Western California, LA, 1987.

[8] K. Muthukumar and M. Hermenegildo. Methods for automatic compile-time parallelization of logic programs using independent/restricted and-parallelism. Technical report, Advanced Computer Architecture, MCC Dept. of Comp. Sci., June 1989. MCC Technical report ACT-ST-233-89.

[9] Francesca Rossi and M. Hermenegildo. On the correctness and efficiency of independent and-parallelism in logic programs. Technical report, Advanced Computer Architecture, MCC Dept. of Comp. Sci., July 1989. In Proceedings of the 1989 North American Conf. on Logic Programming.

[10] Richard Warren, Manuel Hermenegildo, and Saumya Debray. On the practicality of global flow analysis of logic programs. Technical report, Advanced Computer Architecture, MCC Dept. of Comp. Sci., 1988. Refereed for SLP88, Seattle.