Using Compile-Time Granularity Information to Support Dynamic Work Distribution in Parallel Logic Programming Systems

Inês de Castro Dutra[1], Vítor Santos Costa[2], Jorge L. V. Barbosa[3], and Claudio F. R. Geyer[4]

[1] COPPE/Systems Engineering and Computer Science
Federal University of Rio de Janeiro
Rio de Janeiro, RJ, Brazil
ines@cos.ufrj.br

[2] LIACC and DCC-FCUP
4150 Porto, Portugal
vsc@ncc.up.pt

[3] Computer Science Department
Catholic University of Pelotas
Pelotas, RS, Brazil
barbosa@atlas.ucpel.tche.br

[4] Informatics Institute
Federal University of Rio Grande do Sul
Porto Alegre, RS, Brazil
geyer@inf.ufrgs.br

Abstract—

A very important component of a parallel system that generates irregular computational patterns is its *work distribution strategy*. Scheduling strategies for these kinds of systems must be smart enough in order to dynamically balance workload while not incurring a very high overhead.

Logic programs running on parallel logic programming systems are examples of irregular parallel computations. The two main forms of parallelism exploited by parallel logic programming systems are: and-parallelism, that arises when several literals in the body of a clause can execute in parallel, and or-parallelism, that arises when several alternative clauses in the database program can be selected in parallel.

In this work we show that scheduling strategies for distributing and-work and or-work in parallel logic programming systems must combine information obtained at compile-time with runtime information whenever possible, in order to obtain better performance.

The information obtained at compile-time has two advantages over current implemented systems that use only runtime information: (1) the user does not need to adjust parameters in order to estimate sizes of and-work and or-work for the programs; (2) the schedulers can use more accurate estimates of sizes of and-work and or-work to make better decisions at runtime.

We did our experiments with Andorra-I, a parallel logic programming system that exploits both *determinate* and-parallelism and or-parallelism. In order to obtain compile-time granularity information we used the ORCA tool.

Our benchmark set ranges from programs containing and-parallelism only, or-parallelism only and a combination of both and-, and or-parallelism. Our results show that, when well designed, scheduling strategies can actually benefit from compile-time granularity information.

Keywords— **logic programming, parallelism, granularity analysis.**

I. Introduction

A very important component of a parallel system that generates **irregular** computational patterns is its *work distribution strategy*. Scheduling strategies for these kinds of systems must be smart enough in order to dynamically balance workload while not incurring a very high overhead.

Parallel logic programming systems are examples of parallel systems that generate irregular computational patterns. There are two main sources of parallelism in logic programs, namely, and-parallelism (ANDP) and or-parallelism (ORP). ANDP is exploited when the system allows for several goals to be executed simultaneously. ORP arises from the parallel execution of several candidate clauses to a goal. Some systems exploit only or-parallelism [23], [1], [7], [26] while some others exploit only and-parallelism [19], [18], [31]. More sophisticated systems exploit both kinds of parallelism in the same framework [41], [25], [24], [17], [8], [22].

The scheduling strategies used in parallel logic programming systems concentrate on the problem of how to distribute or-parallel work [6], and-parallel work [38], and in the presence of both kinds of work, which kind of work to choose or give preference [11], [13].

There are at least three possible approaches to design a dynamic scheduling strategy for parallel systems. The first approach consists of using information generated at compile-time to guide scheduling decisions. In this case the task assignment is done statically before running the system. The second approach consists of postponing work distribution strategy to runtime and relying only on runtime information. The third approach consists of making decisions at runtime, but guided by information, not necessarily very precise, generated at compile-time.

The first approach has three main disadvantages:

- A fixed configuration of processors/tasks is set forever in the beginning of the computation and *never* changes. This may lead to loss of parallelism, since the amount of work in irregular computations varies along the execution time.
- The process of collecting information about the amount of parallelism in a program at compile-time is very complicated and sometimes does not produce the precise and expected results. First, because the computation of a logic program application varies greatly with different inputs and different data sizes. Second, because some variable dependencies in a logic program are only solved at runtime, which makes the task of generating *precise* information (for example, sizes of parallel tasks) or even *useful* information more difficult.
- Usually, the process of obtaining **precise** and **useful** information through compile-time analysis is very slow, therefore the overall gain in running the application in a parallel system may not be justified.

The second approach has some advantages over doing task assignment at compile-time. First, the system does not waste much time in the compilation phase. Second, dynamic distribution of work allows processors to react to the runtime system, and therefore have a chance to take new tasks according to the workload that varies at runtime. However, this approach has also some disadvantages. The first one is that completely dynamic strategies, in general, do not find global optima results. The second disadvantage is that processors have the overhead of re-scheduling at several stages of the computation. The re-scheduling may deteriorate performance if scheduling overhead costs and frequency of task switching are very high.

In previous works, we showed that dynamic scheduling strategies for distributing and-work and or-work in parallel logic programming systems yielded better performance than a fixed assignment of processors to exploit and-parallelism and or-parallelism. We then suggested that a combination of compile-time granularity information with dynamic scheduling strategies could improve the performance of the system even further [12], [11], [14].

In this work we show that a combination of compile-time granularity information with dynamic scheduling strategies (the third approach) can in fact improve the performance of parallel logic programming systems that use completely dynamic scheduling strategies.

Our target system is Andorra-I [41], [29], [40], [11], a parallel logic programming system that exploits both ORP and ANDP. ANDP in Andorra-I is exploited *determinately*, ie., only goals that match at most one clause in the program are allowed to proceed in parallel.

In order to obtain compile-time granularity information, we used the ORCA system [4], that generates simplified granularity information based on Tick's algorithm [34] for goals and clauses of a Prolog-like program.

We modified the Andorra-I system to understand the ORCA outputs. Our results show that dynamic scheduling strategies can actually benefit from compile-time granularity information.

The paper is organised as follows. Section II describes the ORCA tool and the information generated at compile-time. Section III describes the Andorra-I system and its scheduling algorithms. Section IV describes the applications used in our experiments. Section V presents our results and compares our experiments with a version of Andorra-I that does not use any kind of compile-time information to guide scheduling decisions. Finally, section VI draws some conclusions and presents directions for future work.

II. The ORCA System

The ORCA system is responsible for generating an annotated Prolog program that is later compiled to an abstract code, the Andorra-I VRAM code [30], and provides granularity information about clauses and goals to the Andorra-I engine.

The ORCA system consists of three main parts. The first part reads the Prolog program and creates a table with names of all procedures and their respective clauses. It also generates a list of calls to clauses. The table is organised in a way that each entry contains complexity of the procedure and of the clauses belonging to that procedure. The second part of the model computes the complexity measures for the clauses and procedures. The third part annotates the source program with complexity information.

The algorithm used is based on Tick's algorithm[33] with some modifications to increase precision. Complexity is measured in terms of number of resolutions. In that case, the complexity of a fact in the database is considered as 1. The complexity of a clause is obtained by the sum of the complexities of each goal in the clause plus one call (considering the head call). The complexity of a procedure is given by the sum of the complexities of the clauses belonging to that procedure. The

```
'__c_d/2'(1983,1,[1983],0,[]).
'__c_samples/2'(35,2,[2,33],18,[]).
```

Fig. 1. ORCA OUTPUT EXAMPLE FOR PROGRAM scanner

```
d(Data,Mode):-
        scannerdata(Data,R,C,D1,D2),
        scanner(Mode,R,C,D1,D2,Image).

samples([],Samples):- !, Samples = [].
samples([S|Spec],Samples):- !,
        scannerdata(S,R,C,D1,D2),
        Samples = [sample(R,C,D1,D2)|Smpls],
        samples(Spec,Smpls).
```

Fig. 2. SOME CLAUSES OF THE scanner PROGRAM

complexity of a recursive call is computed as the complexity of the corresponding procedure by considering each recursive call as having weight 1.

As an example, figure 1 shows the annotation for the clauses shown in figure 2.

The complexity of `d/2` is given by annotation `'__c_d/2'`. This annotation employs the Prolog syntax. The complexity of `samples` is given by `'__c_samples/2'`. The first argument of this annotated code corresponds to the complexity of the procedure, the second represents the number of clauses of the procedure which determines the size of the list used as third argument. The third argument represents a list with the complexity of each clause in the procedure, the fourth argument corresponds to the recursive value of the procedure (if it has a recursive call), and, finally, the last argument corresponds to a list of complexities for mutually recursive calls. As no one of the clauses shown in figure 2 have mutually recursive calls, this last argument is represented as an empty list.

Obviously, the numbers represented in each argument are computed based on the analysis of the whole program that we do not show here.

III. THE ANDORRA-I SYSTEM

Andorra-I is a parallel logic programming system that exploits ANDP and ORP. A processing element in Andorra-I is called a *worker*. ANDP in Andorra-I is exploited according to the Basic Andorra Model [37] where goals can only be executed in parallel if they match at most one clause in the program. ORP is exploited in Andorra-I as in Aurora, where each worker owns a special structure (the binding array [36]) to allocate and bind conditional variables.

Workers in the system are organised into *teams*. Each team has a *master* with possibly some *slaves*. Workers inside a team cooperate to exploit and-work in a or-branch of the execution tree. In that way, teams exploit or-parallelism while workers in teams exploit and-parallelism.

Andorra-I is composed of three main sub-systems: (1) the Andorra-I compiler that generates code to the VRAM abstract machine [27], (2) the engine [41], [28], that is responsible for executing VRAM code, and (3) the schedulers, that are responsible for distributing and-work and or-work during execution.

The schedulers are subdivided into three main components: (1) the or-scheduler [5], responsible for choosing the best choicepoint for the worker to move to, (2) the and-scheduler, responsible for choosing the best goal to execute, and (3) the reconfigurer[11], [13], responsible for choosing between the two kinds of work available, and-work or or-work, by reconfiguring workers into teams.

This work will concentrate on the reconfigurer, since its algorithms already assume that some information is available from compile-time analysis. We did not modify the or-scheduler and and-scheduler.

In order to add the information provided by ORCA to Andorra-I, we had to make some modifications to the Andorra-I compiler, to the Andorra-I engine and some minor modifications to the reconfigurer.

A. Modifying the Andorra-I compiler

ORCA gives information on clauses and on goal invocations. This information finds a simple match in the process of Andorra-I compilation:

- The compiler already maintains clause information in order to support the or-scheduler. Currently, information is maintained through the `or_sched_info` instruction, which details how many cuts and commits exist in a clause, and whether the clause terminates in a `fail`. To insert the ORCA extension we just extended the information with the fields found in the annotated code for each procedure.
- Each sub-goal invocation in Andorra-I corresponds to a `create` instruction. This instruction is generated by the compiler for each goal in a body clause. At runtime, this instruction is responsible for pushing a new goal onto the goal list used in Andorra-I. It is therefore natural to associate ORCA's annotation for each call in the clause to the `create` instruction.

B. Modifying the Engine

The engine was adapted to access granularity information. This mainly consisted of adapting the loader and supporting the new instruction fields.

C. Modifying the existing Andorra-I schedulers

As this work concentrates on the reconfigurer, no modification was made in the and-scheduler and or-scheduler related to the work distribution algorithms. Modifications were done only to provide information to the reconfigurer.

D. The Reconfigurer

The reconfigurer was implemented using two different strategies [12]. One, the efficiency-guided strategy, makes decisions based on past information and on *efficiency* of workers in a team [14]. The other, work-guided strategy, makes decisions based on instant information about the sizes of and-work and or-work available [13].

In this work we will concentrate only on the work-guided strategy. The information used by the work-guided strategy assumes some estimates to work sizes. In order to collect amount of and-work, the reconfigurer takes from the and-scheduler the width of the run queue of goals assuming that each goal *depth* has the same value. This depth value is given by the user as a parameter in the command line. This means that we keep a very imprecise information about the actual predicted size of the work below that subtree, which may lead to a decision that allocates workers to a team that is working on a short branch of the execution tree. This may cause load imbalance, since other sources of work may need more workers. In order to collect the amount of or-work available in the execution tree, the reconfigurer takes from the or-scheduler, the total number of alternatives available in the execution tree. The depth of these choicepoints are given by the user in the command line. The default value we use in our Andorra-I version without support to ORCA is 1.

ORCA gives to Andorra-I a kind of information that is very important to guide scheduling decisions: more accurate sizes of goals and clauses. This way, instead of taking width of run queues of goals in order to find size of and-parallel work, the reconfigurer can take estimated sizes of goals in the run queue of goals and estimated sizes of alternatives in each choicepoint.

The reconfigurer has two main objectives:

a) to decide when an idle worker will change its type (master/slave), i.e, what kind of work is preferable: and- or or-, and
b) to choose the preferred team for a slave. The preferred team is the one with more and-work per worker, i.e., the team that has the greatest sum of run queues for all workers in the team.

As regards (a), in order to find the preferred kind of work *currently* on the tree, the reconfigurer takes, from the and-scheduler, the total number of goals available in the run queues of all teams as an estimate of and-work. The reconfigurer also takes, from the or-scheduler, the total number of alternatives in the execution tree as an estimate of or-work.

From now on, we will refer to two versions of Andorra-I: (1) one that does not support ORCA, AI) and one that supports ORCA, AI-O. For both versions of Andorra-I, AI and AI-O, the strategy to choose between and-work and or-work remains the same. The only difference is that the AI-O deals with more precise sizes of and-work and or-work. For more details about the strategy, please refer to [12].

AI takes from the and-scheduler the sum of the run queues of goals as amount of and-parallel work. It considers the depth of each goal as being 1 (this is the value that gives the best performance overall for all our applications). AI-O takes the same information from the and-scheduler, but the sum already contains the predicted depth of each goal. Regarding collecting amount of or-work, AI already takes from the or-scheduler the number of alternatives available in the execution tree (the Andorra-I compiler already supported this information).

One obvious advantage of using more accurate information is that the user does not need to use his/her own estimated sizes of work in the command line as it can be done at the moment.

IV. Benchmarks

All programs used as the benchmark set were selected according to their degree of parallelism. One group of programs has predominantly and-parallelism, another has predominantly or-parallelism, another has both kinds of parallelism in different phases of the computation, and another has both kinds of parallelism appearing at the same computational phase.

A. Benchmarks with predominantly or-parallelism

bqu10
This is a program to solve the Queens problem written using a Pandora programming technique [3]. The problem is to place queens in a board (NxN) in order that no queen attacks each other in any column, row or diagonal. Or-parallelism arises when trying to solve the `cell` predicate. The board size tested was 10x10. The program has mainly or-parallelism with a small amount of and-parallelism.

cypher
This is a simple substitution decoding system developed by Rong Yang [39] in our Andorra-I group. The system reads an encrypted text and decodes it (assuming the original message is in English). According to the statistics, if one knows about 100 of the most common words, one can respectively understand on average 60% of most text. Now, instead of having an entire English dictionary, we can

solve a cipher using only a very small dictionary of common words (about 150 words). With this dictionary, the Andorra-I preprocessor can generate a reasonably small determinate tree for each word of a set length from the dictionary. Then the program performs a lot of letter matching, first determinately in and-parallel, then, when only non-determinate matching is left, the program creates several choices (giving rise to or-parallelism). An or-branch fails when the number of unmatchable words exceeds a certain limit. The algorithm takes advantage of the basic Andorra model to greatly reduce the search space.
The length of the ciphertext used in the benchmark is 56 letters, and the maximum number of unmatchable words is set to 2. This program has mainly or-parallelism with a small amount of and-parallelism.

B. *Benchmarks with predominantly and-parallelism*

fibonacci
This program computes the n-th element of the well-known fibonacci series.

flypan2
This is a program to generate naval flight allocations, based on a system developed by Software Sciences and the University of Leeds for the Royal Navy. It is an example of a real life resource allocation problem. The program allocates airborne resources (such as aircraft) whilst taking into account a number of constraints. The problem is solved by using the technique of active constraints as first implemented for Pandora [2]. In this technique, the co-routining inherent in the Andorra model is used to activate constraints as soon as possible. The program has both or-parallelism, arising from the different possible choices, and and-parallelism, arising from the parallel evaluation of different constraints. We used three input data for testing the program. The first one consists of 11 aircraft, 36 crew members and 10 flights needed to be scheduled. The degree of and- and or-parallelism in this program varies according to the queries, but all the queries give rise to more and-parallelism than to or-parallelism.

bt_cluster
This is a clustering algorithm for network management from British Telecom [9]. The program receives a set of points in a three dimensional space and groups these points into *clusters*. Basically, three points belong to the same cluster if the distance between them is smaller than a certain limit. To obtain best performance, we rewrote the original application to become a determinate-only computation. And-parallelism only in this case naturally stems from running the calculations for each point in parallel. The test program uses a cluster of 400 points as input data. This program has no or-parallelism.

scanner
This is a scanner program to reveal the contents of a bitmap. The program is an AKL [20] benchmark developed at SICS. The problem was described in [10]. It reveals the contents of a box (bitmap). The input is the number of dots on each row, column, left diagonal, and right diagonal. The bitmap used in the benchmark is a picture of a star. The program contains both or-parallelism corresponding to different choices for the unknown bits (either a dot or a blank), and and-parallelism corresponding to parallel propagation of determinate bits and evaluation of constraints. This program gives rise to reasonable amounts of both and- and or-parallelism in interleaved phases.

V. Results

Our experiments were done on a Sun SPARCstation-20 with 4 processors. We used only three processors and left one processor for OS duties. We ran the 2 versions of Andorra-I mentioned in section III: (1) AI, an original version that does not use any kind of compile-time information, and (2) AI-O, a version using compile-time information generated by ORCA. The benchmarks used are described in section IV.

Table I shows runtime executions (in miliseconds) for the benchmarks with the two versions of Andorra-I: (1) the original one without using ORCA information (`AI` column) and (2) the new one using the ORCA information (`AI-O` column), for 3 processors. We ran each application 5 times and computed the average runtime. Speedup numbers are shown to the right of each runtime number. The sequential times of Andorra with ORCA and Andorra without ORCA are similar.

TABLE I
Runtime in milliseconds and speedups, all applications

	Seq. time	AI	AI-O
flypan2	41305	22321 *(1.85)*	13701 *(3.00)*
cypher	15788	15064 *(1.05)*	5613 *(2.81)*
scanner	3098	2195 *(1.41)*	1760 *(1.76)*
bt_cluster	7761	2954 *(2.63)*	2848 *(2.73)*
fibonacci	1312	566 *(2.32)*	652 *(2.01)*
bqu10	542	220 *(2.47)*	217 *(2.50)*

From the results shown we can observe that in all cases, but one, the utilisation of granularity information can help in improving performance. Our improvement ranges from 1% (`bqu10`) to 168% (`cypher`). The program `fibonacci` does not have improvement in performance, because it is a deterministic application that contains only and-parallelism. As we do not use the granularity information in the and-scheduler, we see only the effect of the overhead of supporting ORCA information.

VI. Conclusions and Future Work

This work presented a methodology for incorporating compile time information in the Andorra-I system in order to improve scheduling decisions. The information is provided by the ORCA system. This information was incorporated to the Andorra-I compiler, and consulted by the schedulers. We described the reconfiguring algorithms and explained which parameters and estimates of sizes of and-work and or-work are used by the new version of Andorra-I that supports ORCA information. Our results show that a combination of simple compile-time granularity analysis information with runtime scheduling decisions can produce better performance than using only runtime scheduling decisions. Our modifications introduced very little overhead and the utilization of the compile time information was implemented in a very simple way.

Our next step is to detail the execution of each application in order to pinpoint the actual contribution of the information provided by ORCA. We also would like to repat our experiments with other applications. Another step further is to modify the and-scheduler and the or-scheduler in order to study the impact of the ORCA information on their decisions. Another step forward is to try other scheduling and reconfiguring strategies, and other parallel logic programming systems.

Other scheduling techniques have been proposed that aims at using compile-time granularity analysis, but either have not been fully implemented or have only been simulated. Wai-Keong [16] reports an or-scheduler strategy that uses a heuristic task distribution by assigning "weights" to the alternative clauses. Its algorithm to assign the weights is similar to Tick's granularity size algorithm [34]. In this algorithm each call to a goal is counted as having weight 1 and each recursive call has also weight 1. These weights are assigned to the clauses at compile-time by annotating the Prolog program. The scheduler then uses this information in order to select tasks to spawn. This technique was originally used by Tick [34] to control the scheduling of and-parallel goals in FGHC [35], but it was used by Wai-Keong to control the spawning of or-parallel alternatives.

Another strategy technique is reported in [21] where a method to remove structural imbalance of the programs by global analysis (basically unfolding/flattening recursive predicates) is proposed. By removing structure imbalance the author assumes that the or-work is evenly distributed during the execution. The following rules are applied to distribute work:

- *eager-splitting strategy*: at each choicepoint where m processors are present, assume there are n valid choices. m tasks are created and assigned evenly to m processors. If $n \geq m$, each task contains $\frac{n}{m}$ choices, and the remaining choices are randomly included in some of the tasks. If $n < m$, each processor randomly picks one choice.
- *lazy-splitting strategy*: at each choicepoint, two tasks are created and assigned to each of the half of the processors. In case of choices being not evenly divisible, remaining choices are treated in a way similar to that in the eager-splitting rule.

Work by King et al [32] also try to control granularity by minimising the number of CGEs (conditional graph expressions) generated in systems that exploit independent and-parallelism.

Ferrari et al [15] have been doing the same kind of work as ours, by applying the ORCA information to the or-parallel distributed system Plosys [26].

To the best of our knowledge, this is the first work on applying compile-time granularity analysis to parallel logic programming systems that exploit both and-parallelism and or-parallelism, with successful results.

References

[1] Khayri A. M. Ali and Roland Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *Proceedings of the 1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[2] Reem Bahgat. Solving Resource Allocation Problems in Pandora. Technical report, Imperial College, Department of Computing, 1990.

[3] Reem Bahgat. *Non-Deterministic Concurrent Logic Programming in Pandora*, volume 37. World Scientific, Singapore, 1993. Series in Computer Science.

[4] J. L. V. Barbosa and C. F. R. Geyer. Análise de Complexidade na Programação em Lógica: Taxonomia, Modelo Granlog e Análise OU. In *XXIII Conferência Latino Americana de Informática, V Encontro Chileno de Computação da Sociedade Chilena de Computação (CLEI – PANEL'97)*, November 1997.

[5] Anthony Beaumont, S. Muthu Raman, and Péter Szeredi. Flexible Scheduling of Or-Parallelism in Aurora: The Bristol Scheduler. In Aarts, E. H. L. and van Leeuwen, J. and Rem, M., editor, *PARLE91: Conference on Parallel Architectures and Languages Europe*, volume 2, pages 403–420. Springer Verlag, June 1991. Lecture Notes in Computer Science 506.

[6] Anthony Beaumont and David H. D. Warren. Scheduling Speculative Work in Or-Parallel Prolog Systems. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 135–149. MIT Press, June 1993.

[7] J. Briat, M. Favre, C. Geyer, and J. Chassin. Scheduling of Or-parallel Prolog on a Scalable, Reconfigurable, Distributed-Memory Multiprocessor. In *Proceedings of Parallel Architecture and Languages Europe*. Springer Verlag, 1991.

[8] Manuel Eduardo Correia, F. M. A. Silva, , and V. Santos Costa. The SBA: Exploiting orthogonality in OR-AND Parallel Systems. In *Proceedings of the 1997 International Logic*

Programming Symposium, October 1997. also published as Technical Report DCC-97-3, DCC - FC & LIACC, Universidade do Porto, April, 1997.

[9] Barry Crabtree. A clustering system to network control, British Telecom, March 1991.

[10] A. K. Dewdney. Mathematical Recreations – A Compendium of Math Abuse from around the World. *Scientific American*, 263(5):142–145, September 1990.

[11] I. C. Dutra. Strategies for Scheduling And- and Or-Work in Parallel Logic Programming Systems. In *Proceedings of the 1994 International Logic Programming Symposium*, pages 289–304. MIT Press, 1994. Also available as technical report CSTR-94-09, from the Department of Computer Science, University of Bristol, England.

[12] I. C. Dutra. *Distributing And- and Or-Work in the Andorra-I Parallel Logic Programming System.* PhD thesis, University of Bristol, Department of Computer Science, February 1995. available at http://www.cos.ufrj.br/∼ines.

[13] I. C. Dutra. Performance Analysis of a Strategy to Distribute And-work and Or-work in Parallel Logic Programming Systems. In *Proceedings of the VIII Brazilian Symposium on Computer Architecture and High Performance Processing – SBAC-PAD*, pages 449–463, 1995.

[14] I. C. Dutra. Distributing And-Work and Or-Work in Parallel Logic Programming Systems. In *Proceedings of the 29th Hawaii International Conference on System Sciences*, pages 646–655. IEEE, 1996.

[15] FERRARI, D. N. O uso e implementação de informações de granulosidade no plosys. Trabalho individual, Universidade Federal Do Rio Grande Do Sul, 1998.

[16] Wai-Keong Foong. Or-Parallel Prolog with Heuristic Task Distribution. In *Lecture Notes in Artificial Intelligence 592, Logic Programming Russian Conference*, pages 193–200, 1991.

[17] Gopal Gupta, M. V. Hermenegildo, E. Pontelli, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proceedings of the Eleventh International Conference on Logic Programming*, Italy, June 1994.

[18] Gopal Gupta, Enrico Pontelli, and Manuel Hermenegildo. &ACE: A High Performance Parallel Prolog System. In *Proceedings of the First International Symposium on Parallel Symbolic Computation, PASCO'94*, 1994.

[19] Manuel Hermenegildo. An Abstract Machine for Restricted And-Parallel Execution of Logic Programs. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 25–39. Springer-Verlag, 1986.

[20] Sverker Jansson and Seif Haridi. Programming Paradigms of the Andorra Kernel Language. In *Proceedings of the 1991 International Logic Programming Symposium*, pages 167–186. MIT Press, October 1991.

[21] Zheng Lin. Self-Organising Task Scheduling for Parallel Execution of Logic Programs. In *Proceedings of the 1992 International Conference on Fifth Generation Computer Systems*, pages 859–868. ICOT, 1992.

[22] Ricardo Lopes and V. Santos Costa. The BEAM: Towards a first EAM Implementation. In *Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, October 1997. Port Jefferson.

[23] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora Or-parallel Prolog System. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, pages 819–830. ICOT, Tokyo, Japan, November 1988.

[24] Johan Montelius. *Penny, A Parallel Implementation of AKL.* PhD thesis, Swedish Institute for Computer Science, SICS, Sweden, May 1997.

[25] Remco Moolenaar and Bart Demoen. Optimization Techniques for nondeterministic promotion in the Andorra Kernel Language. In *Proceedings of the Compulog-Net, Madrid*, May 1993.

[26] E. Morel, J. Briat, J. Chassin de Kergommeaux, and C. Geyer. Side-Effects in PloSys OR-parallel Prolog on Distributed Memory Machines. In *JICSLP'96 Post-Conference Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages, Bonn, Germany*, September 1996.

[27] V. Santos Costa. *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I.* PhD thesis, Department of Computer Science, University of Bristol, August 1993.

[28] V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 83–93. ACM press, April 1991. SIGPLAN Notices vol 26(7), July 1991.

[29] Vítor Santos Costa, David H. D. Warren, and Rong Yang. The Andorra-I Preprocessor: Supporting full Prolog on the Basic Andorra model. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 443–456. MIT Press, 1991.

[30] Vítor Santos Costa and Rong Yang. Andorra-I User's Guide and reference manual. Technical report, University of Bristol, Computer Science Department, Sept 1990. Internal Report, Gigalips Project.

[31] Kish Shen. Overview of DASWAM: Exploitation of Dependent And-parallelism. *J. of Logic Prog.*, 29(1–3), 1996.

[32] Kish Shen, Vítor Santos Costa, and Andy King. A New Metric for Controlling Granularity for Parallel Execution. In *Joint International Conference and Symposium on Logic Programming*, Manchester, UK, June 1998.

[33] Evan Tick. Compile Time Granularity Analysis for Parallel Logic Programming Systems. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems.* ICOT, 1988.

[34] Evan Tick. Compile Time Granularity Analysis for Parallel Logic Programming Systems. *New Generation Computing*, 7(2,3):325–337, 1990.

[35] Kazunori Ueda and Masao Morita. A New Implementation Technique for Flat GHC. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 3–17. MIT Press, June 1990.

[36] David H. D. Warren. The SRI Model for Or-Parallel Execution of Prolog—Abstract Design and Implementation Issues. In *Proceedings of the 1987 International Logic Programming Symposium*, pages 92–102, 1987.

[37] David H. D. Warren. The Andorra model. Presented at Gigalips Project workshop, University of Manchester, March 1988.

[38] Rong Yang. Implementation Notes on the Andorra Model. Technical report, University of Bristol, Computer Science Department, Sept 1989. Internal Report, Gigalips Project.

[39] Rong Yang. Solving simple substitution ciphers in Andorra-I. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 113–128. MIT Press, June 1989.

[40] Rong Yang, Tony Beaumont, Inês Dutra, Vítor Santos Costa, and David H. D. Warren. Performance of the Compiler-Based Andorra-I System. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 150–166. MIT Press, June 1993.

[41] Rong Yang, Vítor Santos Costa, and David H. D. Warren. The Andorra-I Engine: A parallel implementation of the Basic Andorra model. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 825–839. MIT Press, 1991.