The Basic OASys Model: Preliminary Results

I. Vlahavas[1], P. Kefalas[2], I. Sakellariou[1] and C. Halatsis[3]

(1) Dept. of Informatics, Aristotle University of Thessaloniki,
54006 Thessaloniki, Greece.
e-mail:vlahavas@csd.auth.gr, esakelar@athena.auth.gr
(2) Dept. of Computer Science, City Liberal Studies, 13 Tsimiski Street,
546 24 Thessaloniki, Greece.
e-mail:kefalas@hyper.gr
(3) Dept. of Informatics, University of Athens, 15771 Athens, Greece.
e-mail:halatsis@di.uoa.gr

Research Paper

keywords: Prolog, AND/OR-Parallelism, Abstract Machine

Abstract

The exploitation of parallelism that is naturally embedded in the declarative semantics of logic programs has been a major issue of the logic programming community in the past decade. Various models and architectures have been proposed in order to achieve the above mentioned task. This paper describes the basic OASys (And/Or System), a computational model designed for parallel execution of logic programs. OASys performs OR parallelism by assigning each independent OR path to a different processing element, and AND-parallelism by executing the conjunctive determinate subgoals simultaneously. The model is designed in such way that all communication between the processing units is limited to only the necessary scheduling of different tasks that are generated during the program execution. This paper also presents preliminary results of the model that where obtained from a Prolog simulation.

1 Introduction

The most promising approach for running programs in parallel seems to be that of exploiting the parallelism that is naturally embedded in the semantics of the declarative languages [1]. This type of languages facilitate parallelisation without the use of explicit annotations from the programmer and thus leading to an effective parallel computational model.

The parallelisation of declarative logic languages such as Prolog, has been an extensive area of research in the logic programming community for the last two decades. A number of methods have been proposed in order to implement parallel execution of Prolog programs while not affecting their semantics. The semantics of sequential Prolog involve a fixed top-down, left-to-right search of the AND-OR tree which is generated while solving the user query. The parallel execution of a logic program involves traversing the execution tree in a parallel manner. The

alternative OR-branches of this tree can be explored simultaneously since they are independent, while the conjunctions of subgoals that belong in an AND-branch can also be simultaneously explored if there exists a safe and preferably efficient method of maintaining the consistency of the bindings of the shared variables in the conjunction of subgoals.

The research in the last decade has been concentrated on either exploiting the OR-parallelism [2, 3, 4] or various forms of AND-parallelism [5, 6] and reported very promising results. Practically though, a model that will exploit both AND/OR-parallelism is too complex to implement. Lately, efforts have been concentrated in combining the two types of parallelisation by exploiting each type when it is efficient to do so [7].

The basic OASys (Or/And SYStem) [8, 9] model, presented in this paper, is an approach for implementing full AND/OR-parallelism. In OASys OR-parallelism is implemented by assigning each OR-branch in a distinct processing element and thus exploring the different branches in parallel and at the same time investigating simultaneously each subgoal of a conjunction in parallel, thus implementing the AND-parallelism. In effect this model leads to an architecture where the processing elements that perform the OR-parallelism own their individual address space while the processing units that perform the AND parallelism share a common one. In such a model, the communication between the processors is limited to only the basic scheduling operations, ie. work allocation to the idle processing elements.

In the next sections a more detailed description of the model and the proposed architecture is presented as well as some performance results obtained by a Prolog simulation of the above mentioned model.

2 The Basic OASys Computational Model

The search space generated during the execution of a Prolog program can be represented by an AND/OR tree in which the OR-branching points represent the matching of a subgoal with multiple heads of program clauses, and the AND-branching points represent the subgoals that are contained in the body of a program clause (figure 1).

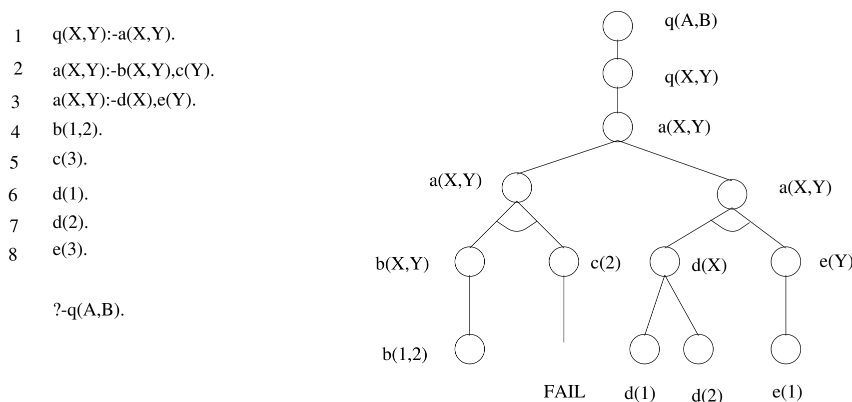

Figure 1: The Prolog search tree of a simple example

In OASys, the search space is represented by a set of independent branches in which each of the nodes denotes the matching of a subgoal with the head of a program clause. Each one of these branches represents a path from the root of the search either to a solution or a failed node. In figure 2 the corresponding search tree to the previous Prolog example is presented.

New branches are generated when a non-deterministic clause is encountered in the search; one of the matching predicates becomes member of the current branch and the alternative matching

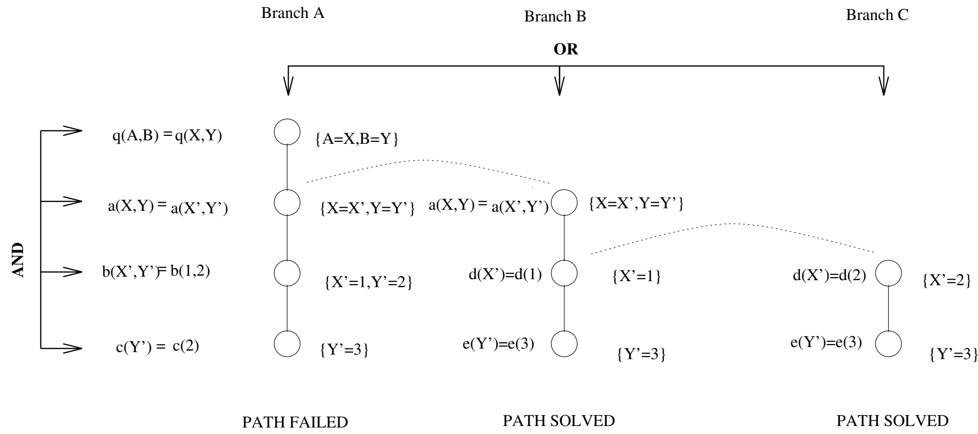

Figure 2: The OASys search tree

predicates become members of new branches that are generated by doublicating the "parent" branch to the point of the non deterministic clause. This parent child relationship though ends here; the newly generated branches are now totally independent and the search in each one of them continues regardless of what happens to the parent branch. This search process in a branch is terminated either when:

- a solution is found, ie all nodes are successful, which practically means that all nodes in the branch were explored and the variable bindings produced were consistent
- the path fails, either because a node is not successful, or there was a conflict in the bindings of the shared, between the nodes of the same branch, variables.

The unifications of the subgoals with the heads of program clauses that belong to the same branch are executed in parallel; this has the effect that some of the nodes generated may be speculative i.e. might not have existed in the corresponding Prolog execution.

For example, in the program shown in figure 1, the search will start in the first branch of the OASys tree (Branch A) and continues until subgoal a(X,Y) is encountered. Since the later can match with two program clauses (clause 2 and 3 in the code) a new branch is generated (branch B) in which the subgoal a(X,Y) will be unified with the head of clause 3, while the search continues in the current branch by performing the unification with clause 2. The search carries on parallely and independently in both branches. Branch A finally fails because the bindings produced by the unifications in b(X,Y)-b(1,2) and c(Y)-c(3) are inconsistent. Branch B encounters another non deterministic subgoal, d(X) and a third branch is generated (branch C) as before. Branch B and C succeed and represent the two solutions found of query q(A,B).

3 Abstract Architecture

The architecture that naturally derives from the previously described computational model is a group of *Processing Elements* (PEs), each being a shared memory multiprocessor consisting of three main parts: the preprocessor, the scheduler and the engine (figure 3).

3.1 Execution Scheme

Initially the compiled program is distributed to all PEs and the execution starts in one of them, which is declared busy while the rest are declared idle.

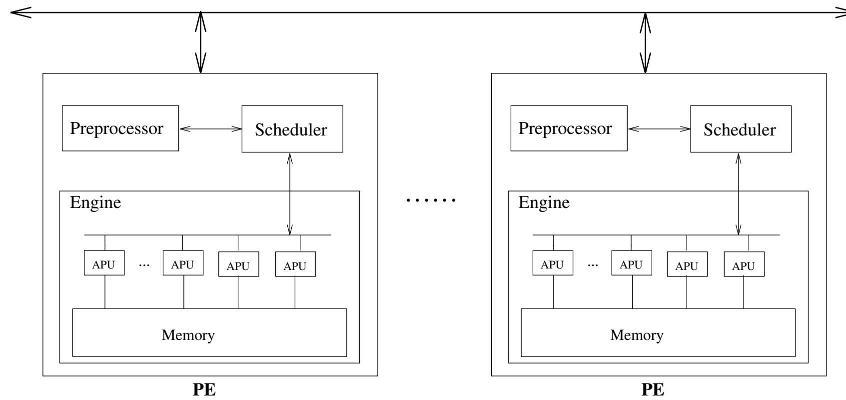

Figure 3: Overview of the OASys Architecture

In brief, there are three phases in the operation of each PE: preprocessing, scheduling and execution. Choice points along a path of the search are discovered by the **preprocessor** and then passed to the scheduler (Preprocessing Phase). A choice point is generated when a subgoal is non deterministic, that is when it matches more than one program clauses. The choices are actually indices that map the subgoal to the matching program clauses in the original Prolog code.

The **scheduler** decides whether the available work should be rescheduled to other possibly idle PEs (Scheduling Phase) or executed by its local engine and sends the choices to the engine for execution. A table of PE-ids is maintained in each scheduler that states whether these have unexplored paths to share. Each PE broadcasts a single byte that indicates its current state and updates the mentioned table. When a PE becomes idle, it consults this table in order to find a PE with paths to share. When such a PE is found, rescheduling of paths takes place. Under this scheme,the scheduling operation is distributed to all PEs avoiding the problems that would appear if one central scheduler was used, ie. delays in the rescheduling of tasks, etc. The model's modularity offers the ability to apply various scheduling strategies; for example schedule first jobs that are closer to the root, or the opposite, or use information obtained by analysis during precompilation to assign different priorities to scheduled tasks, or even employ user annotations to indicate parts of the program that should be kept private to a PE.

The **engine** executes the machine instructions and uses the choices provided by the scheduler as directives in order to construct pairs of addresses of matching predicates with heads of program clauses. Each such pair is assigned to an APU, that performs the unification efficiently. Since the APUs have access to one common memory, the variable bindings produced by the unifications are shared. Any conflicts to those bindings lead to a failed path. It is worth noticing here that since a path explored by one PE is deterministic the order in which the unifications are performed does not affect the completeness and soundness of the execution, and thus the above mentioned unifications can be executed in parallel.

This scheme supports both AND and OR parallelism:

- *OR-parallelism* is achieved by assigning different paths of the OASys search tree (figure 2) to different PEs. The distribution of work depends on the scheduler and the availability of PEs.

- *AND-parallelism* is handled by executing the nodes of the same path, that is processed in a PE, in parallel by assigning each node to an APU. Since those operate in parallel, the conjunction of predicates corresponding the specific path is executed in parallel.

For example in the case presented in figure 2, each one of the three branches (paths) is

assigned to a PE, and each node of each path is assigned to an APU in the specific PE that it belongs.

4 Performance Measurements

In order to test the feasibility of the model and test its performance, a simulator has been developed in Prolog. The simulator was implemented in Eclipse Prolog and tested on a Sun Sparcstation LX. In this section the results from this simulation are presented.

The problems selected in order to run the tests were:

- nrev140: Naive reverse of a list of 140 elements
- merge200: Mergesort of a list of 200 elements
- map: map coloring problem
- queens-8: The 8-queens problem

Results were taken for each of the above problems for all the combinations of one PE to ten PEs (1,2,4,6,8,10) and 1 APU/PE to 10 APUs/PE (again 1,2,4,6,8,10).

The relative speedup is calculated as the ratio of the number of resolution cycles of each run over the same number of a "sequential Prolog" run. In a resolution cycle all the PEs of the OASys machine continue the execution in the branch that is currently assigned to them until all their APUs have executed one unification; for example if a PE with four APUs is searching a branch of ten nodes, in a resolution cycle it will execute the unifications in the four first nodes. The execution of the rest of the branch will continue in the next cycle. By "sequential Prolog" run we refer to the first run with 1 PE and 1 APU/PE. In the later case the number of resolution cycles is equal to the number of Prolog resolutions since no OR/AND- parallelism takes place. Since the PEs work in parallel, we argue that this ratio is indicative of the speedup obtained in a true parallel system. Although in a real parallel implementation, there are overheads involved, such as scheduling operations, communication delays, synchronization, etc, these overheads are small since the proposed model is highly distributed.

Figure 4 (a), shows the effectiveness of the model obtained by AND-parallelism in the four problems that were measured. The x-axis indicates the number of APUs in a PE and the y-axis the ratio explained before. All measurements are taken using one PE.

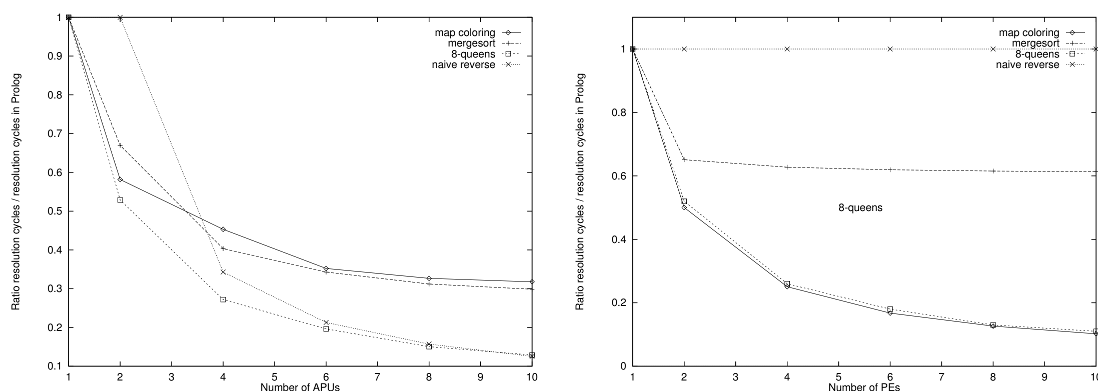

Figure 4: (a) Results of AND-parallelism (b) Results of OR-parallelism

The above measurements indicate clearly that the performance of OASys is improved as the number of APUs/PE increases in all the problems measured. It is worth noticing that the

improvement in the performance is not linear to the number of APUs/PE; for instance the relative speedup between 8 and 10 APUs is small compared with the improvement that occured when the APUs increased from 2 to 4 APUs in all the problems. This indicates that there is an upper limit in this number after which efficiency gains are trivial.

Figure 4 (b), shows the results obtained by exploiting OR-parallelism. The x-axis represents the number of PEs in the 'machine' tested while the y-axis presents the ratio as in the previous diagram. In this case all measurements are taken using 1 APU/PE.

As we see the performance improves in three cases as the number of PEs increases. The only case in which there is almost no improvement in the performance is that of the naive reverse problem but this is because the problem is purely deterministic.

In figures 5 and 6 that follow, results of runs of all combinations of PEs - APUs/PE for each of the problems are presented.

Each curve in the diagram represents the number of PEs used in the specific run. The x-axis represents the number of APUs and the y-axis the same ratio used to all diagrams up till now.

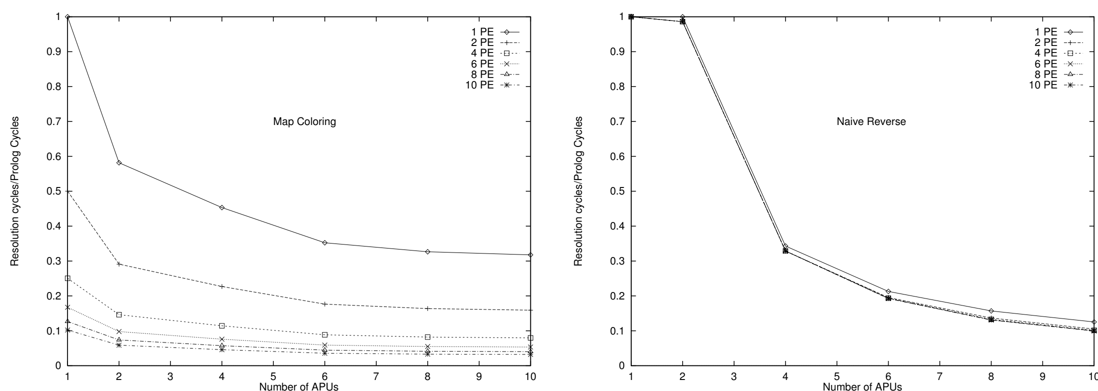

Figure 5: Results for the mapping and naive reverse problems

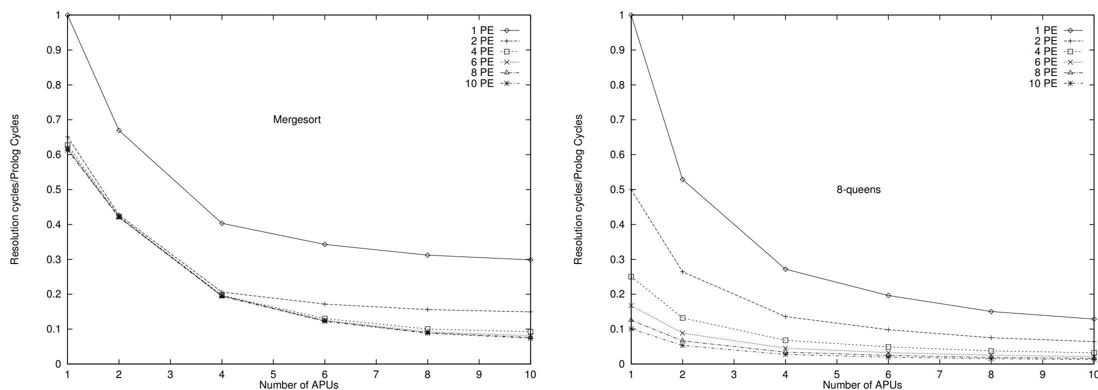

Figure 6: Results for mergesort and the 8-queens problems

5 Related Work

The two types of parallelism of logic programming languages present different problems in their implementation. In OR-parallel implementations the problem of handling multiple environments

occurs, while in the main problem in implementing AND parallelism is the binding of common variables between subgoals.

Concerning the multiple environment representation problem in OR parallelism three main approaches have been introduced:

- Shared environment, where bindings to variables are recorded to the same address space by different processors, employing appropriate data structures [2, 10, 11, 12].
- Environment copying, where each processor uses his own copy of the environment for variable bindings [13, 14].
- Recomputation, where processors view the search space as totally independent computations without the need of communicating with each other [15, 16, 17, 18].

For handling the problem of common variables that occurs when implementing AND parallelism two main approaches have been introduced:

- Independent AND-parallelism, where processors work in parallel only when the runtime bindings of the variables of a set of subgoals are such that the later are independent [5, 6, 19].
- Dependent AND-parallelism where processors work in parallel, but when a common variable is introduced, it is accessed by following a priority scheme [20], or by applying a producer-consumer model between the different processors.

Most of the models proposed till recently involved only the exploitation of either OR parallelism or different types of AND parallelism. A few implementations involved the simultaneous exploitation of OR parallelism and one of the two types of AND parallelism.

The OASys model that was presented in this paper offers a simple computational model that aims to provide an efficient way of implementing full AND-OR parallelism in a single framework. The described model is capable of supporting both environment copying as well as recomputation for implementing the OR parallel part and offers a scheme for exploiting both dependent and independent AND parallelism without the need of performing complicated determinacy checking and non-determinate goals suspension. This reduces significantly reduces the complexity of the proposed model, and we believe, that will lead to an efficient AND-OR parallel Prolog implementation.

6 Implementation Issues

The OASys execution scheme seems to adopt naturally to a hybrid multiprocessor architecture, in which parts of the address space are shared among subsets of processors, as for example a system that contains multiple shared memory multiprocessors connected by a message passing local network. OASys can be also considered as a SPDM (Single Program Multiple Data) machine, in which the processors execute independently different parts of the data.

Besides the above ideal architecture, the described model can also be implemented in the following architectures:

- A network of multiprocessor workstations: In such a implementation each multiprocessor workstation corresponds to a PE and each processor in the workstation to its different pasts (ie. engine, scheduler,etc).

- A message passing scalable multiprocessor architecture that provides a shared virtual address space on essentially physically distributed hardware [21]. The address space is truly shared but is also virtual in the sense that memory is physically distributed across different processors.

- A shared memory multiprocessor machine, in which the processors will be grouped in teams, each team corresponding to a PE, and each processor in the team to a component of the PE. The shared memory will be partitioned to as many independent areas as the number of teams

7 Conclusion

The described basic OASys model can perform AND/OR-parallelism of Prolog programs. OR-parallelism is achieved by assigning the independent OR-branches of the conventional Prolog tree to different processing elements (PEs) while AND-parallelism is achieved by assigning the nodes of the same branch to different APUs inside a PE. The modularity of the previous scheme allows it to expand easily and leads to a higly distributed system in which communication is limited only to the necessary scheduling operations.

Ideal architecture for the OASys would be a hybrid multiprocessor containing multiple shared memory multiprocessors connected by a message passing network.

We are currently working on an OASys implementation that will be executed over a network of Unix workstations, in order to be able to test the model more thoroughly and compare its performance with other parallel logic programming languages. Our work is also concentrated on testing various scheduling techniques such as copying and recomputation on the OASys model.

References

[1] Gopal Gupta, "Multiprocessor Execution of Logic Programs", Kluwer Academic Publishers, 1994.

[2] E.Lusk, R.Butler, T.Disz, R.Olson, R.Overbeek, R.Stevens, D.H.D Warren, A Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A Ciepielewski and B. Hausman. "The Aurora OR-parallel Prolog system". New Generation Computing, 7 (2,3) pp. 243-271, 1990.

[3] K.Ali and R.Carlsson. "The Muse Or-Parallel Prolog model and its performance, In Proc. 1990 North American Conf. on Logic Programming, Austin, USA, Oct. 1990.

[4] T.J.Reynolds and P.Kefalas, "OR-parallel Porlog and search problems in AI applications". In Proc. of the 7th International Conference on Logic Programming eds. D.H.D Warren and P.Szeredi, Jerusalem, pp.340-354,1990.

[5] M.Hermenegildo and K.Green, "&-Prolog and Its Performance: Exploiting Independent And-Parallelism" In Proc. of International Conference on Logic Programming, ICLP 90, MIT Press, pp. 253-268, 1990.

[6] Y.Lin and V. Kumar. "AND parallel execution of Logic Programs on a Shared-Memory Multiprocessor". The journal of Logic Porgramming, 10, pp. 155-178,1991.

[7] R.Yang, T. Beaumont, V.Santos Costa, D.H.D. Warren, "Performance of the compiler-baseed Andorra-I System". In Proc. of the 10th International Conference in Logic Programming, ed. D.S.Warren, MIT Press, pp. 150-166, 1993.

[8] I.Vlahavas and P.Kefalas, "A Parallel Prolog Resolution Based on Multiple Unifications". Parallel Computing, North-Holland, vol 18, pp.1275-1283, 1992.

[9] I.Vlahavas and P.Kefalas, "The AND.OR Parallel Prolog Machine APIM: Execution Model and Abstract Design". The Journal of Programming Languages 1, pp. 245-261, 1993.

[10] A. Ciepielewski and S. Haridi, "A formal Model for OR-Parallel execution of Logic Programs". Proc. in Format Processing, IFIP, pp. 299-305, 1983.

[11] D.S. Warren, "Efficient Prolog Memory Management for Flexible Control Strategy", Int Symp. om Logic Programming, pp 198-202, 1984.

[12] Sergio Delgado-Rannauro, M. Dorochevsky, K. Schucrman, A. Veron, and J. Xu, "A Shared Environment Parallel Logic Programming System on Distributed memory Architectures", Proc of 2nd European Distributed Memory Computing Conference, Munich, April 1991.

[13] K. Ali and R. Karlson " Scheduling Speculative Work in MUSE and Performance Results", Int. J. of Parallel Programming, vol 21 (6), pp 449-476, 1992.

[14] G. Gupta, M. Hermenegildo, E. Pontelli, V.S. Costa, "ACE: And-Or Parallel Copying-Based Execution of Logic Programs", Proc. of the 11th ICLP, MIT Press, 1994.

[15] W.F. Clocksin, "Principles of the Delphi Parallel Inference Machine", The computer Journal, vol. 30 (5), pp 386-392, 1987.

[16] L. Araujo and J.J. Ruz, "PDP: Prolog Distributed Processor for Independent AND/OR Parallel Execution of Prolog", Proc of the 11th ICLP, MIT Press, 1994.

[17] S. Mudambi and J. Schimpf, "Parallel CLP on Heteregeneous Networks", Proc of the 11th ICLP, MIT Press, 1994.

[18] G. Gupta and M. Hermenegildo, "And-Or Parallel Prolog: A Recomputation Based Approach", New Generation Computing, vol. 11, pp 297-321, 1993.

[19] E. Pontelli, G. Gupta and M. Hermenegildo, "&ACE: A High Performance Parallel Prolog System", in Proc. of IPPS, IEEE Computer Society, 1995.

[20] K. Shen, "Exploiting Dependent And-parallelism in prolog: the Dynamic Dependent And-parallel Scheme (DDAS)", in Proc of the 9th JICSLP, MIT Press, 1992.

[21] D.H.D. Warren and S. Haridi. "Data Diffusion Machine - A Scalable Shared Virual Memory Multiprocessor." In Proc. of the International Conference on Fifth Generation Computer Systems, 1988.