In 1994 Principles and Practice of Constraint Programming, pp. 123–133, LNCS 874

Some Methodological Issues in the Design of CIAO, a Generic, Parallel Concurrent Constraint Logic Programming System

M. Hermenegildo

Facultad de Informática
Universidad Politécnica de Madrid (UPM)
28660-Boadilla del Monte, Madrid, Spain
herme@fi.upm.es

Abstract. We informally discuss several issues related to the parallel execution of logic programming systems and concurrent logic programming systems, and their generalization to constraint programming. We propose a new view of these systems, based on a particular definition of parallelism. We argue that, under this view, a large number of the actual systems and models can be explained through the application, at different levels of granularity, of only a few basic principles: determinism, non-failure, independence (also referred to as stability), granularity, etc. Also, and based on the convergence of concepts that this view brings, we sketch a model for the implementation of several parallel constraint logic programming source languages and models based on a common, generic abstract machine and an intermediate kernel language.

1 Introduction

We present an informal discussion on some methodological aspects regarding the efficient parallel implementation of (concurrent) (constraint) logic programming systems. These efforts represent our first steps towards the development of what we call the CIAO (Concurrent, Independence-based And/Or parallel) system – a platform which we expect will provide efficient implementations of a series of *non-deterministic, concurrent, constraint logic programming languages*, on sequential and multiprocessor machines. Because of broad-view nature of the discussion, in the following a certain familiarity with constraint logic programming and parallel logic programming theory. models, and actual systems such as Muse [1], Aurora [24], &-Prolog [13], GHC [32], PNU-Prolog [26], DDAS [28], Andorra-I [27], AKL [20], and the extended Andorra model [33] is assumed.

2 Separation of issues / Fundamental Principles

We begin our discussion with some very general observations regarding computation rules, concurrency, parallelism, and independence. We believe these observations to be instrumental in understanding our approach and its relationship to others. A motivation for the discussions that follow is the fact that many current proposals for parallel or concurrent logic programming languages and models are actually "bundled packages", in the sense that they offer a combined solution affecting a number of issues such as choice of computation rule, concurrency, exploitation of parallelism, etc. This is understandable since certainly a practical model has to offer solutions for all the problems involved. However, the bundled nature of (the description of) many models often makes it difficult to compare them with each other. It is our view that, in order to be able to perform such comparisons, a "separation analysis" of such models, isolating their fundamental principles in (at least) the coordinates proposed above must be performed. In fact, we also believe that such un-bundling brings the additional benefit of allowing the identification and study of the fundamental principles involved in a system independent manner and the transference of the valuable features of a system to another. In the following we present some ideas on how we believe the separation analysis mentioned above might be approached.

2.1 Separating Control Rules and Parallelism

We start by discussing the separation of parallelism and computation rules in logic programming systems. Of these two concepts, probably the best understood from the formal point of view is that of computation rules. Assuming for example an SLD resolution-based system the "computation rules" amount to a "selection rule" and a "search rule." The objective of such computation rules in general is to minimize work, i.e. to reduce the total amount of resolutions needed to obtain an answer. We believe it is useful, at least from the point of view of analyzing systems, to make a strict distinction between parallelism issues and computation-rule related issues. To this end, we define parallelism as the simultaneous execution of a number of *independent* sequences of resolutions, *taken from those which would have to be performed in any case as determined by the computation rules.* We call each such sequence a *thread* of execution. Note that as soon as there is an *actual* (i.e., run-time) dependency between two sequences, one has to wait for the other and therefore parallelism does not occur for some time. Thus, such sequences contain several threads. Exploiting parallelism means taking a fixed-size computation (determined by the computation rules), splitting it into independent threads related by dependencies (building a dependency graph), and assigning these segments to different agents. Both the partitioning and the agent assignment can be performed statically or dynamically. The objective of parallelism in this definition is simply to *perform the same amount of work in less time.*

We consider as an example a typical or-parallel system. Let us assume a finite tree, with no cuts or side-effects, and that all solutions are required. In a first approximation we could consider that the computation rules in such a system are the same as in Prolog and thus the same tree is explored and the number of resolution steps is the same. Exploiting (or-)parallelism then means taking branches of the resolution tree (which have no dependencies, given the assumptions) and giving them to different agents. The result is a performance gain that is independent of any performance implications of the computation rule. As is well known, however, if only (any) one solution is needed, then such a system can behave quite differently from Prolog: if the leftmost solution (the one Prolog would find) is deep in the tree, and there is another, shallower solution to its right, the or-parallel system may find this other solution first. Furthermore, it may do this after having explored a different portion of the tree which is potentially smaller (although also potentially bigger). The interesting thing to realize from our point of view is that part of the possible performance gain (which sometimes produces "super-linear" speedups) comes in a fundamental way from a change in the computation rule, rather than from parallel execution itself. It is not due to the fact that several agents are operating but to the different way in which the tree is being explored ("more breath-first").[1]

A similar phenomenon appears for example in independent and-parallel systems if they incorporate a certain amount of "intelligent failure": computation may be saved. We would like this to be seen as associated to a smarter computation rule that is taking advantage of the knowledge of the independence of some goals rather than having really anything to do with the parallelism. In contrast, also the possibility of performing additional work arises: unless non-failure can be proved ahead of time, and-parallel systems necessarily need to be speculative to a certain degree in order to obtain speedups. However such speculation can in fact be controlled so that no slow down occurs [14].

Another interesting example to consider is the Andorra-I system. The basic Andorra principle underlying this system states (informally) that deterministic reductions are performed ahead of time and possibly in parallel. This principle would be seen from our point of view as actually two principles, one related to the computation rules and another to parallelism. From the computation rule point of view the bottom line is that deterministic reductions are executed first. This is potentially very useful in practice since it can result in a change (generally a reduction, although the converse may also be true) of the number of resolutions needed to find a solution. Once the computation rule is isolated the remaining part of the rule is related to parallelism and can be seen simply as stating that deterministic reductions can be executed in parallel. Thus, the "parallelism part" of the basic Andorra principle, once isolated from the computation rule part, brings a basic principle to parallelism: that of the general convenience of parallel execution of deterministic threads.

We believe that the separation of computation rule and parallelism issues mentioned above allows enlarging the applicability of the interesting principles brought in by many current models.

---

[1] This can be observed for example by starting a Muse or an Aurora system with several "workers" on a uniprocessor machine. In this experiment it is possible sometimes to obtain a performance gain w.r.t. a sequential Prolog system even though there is no parallelism involved – just a *coroutining* computation rule, in this case implemented by the multitasking operating system.

2.2 Abstracting Away the Granularity Level: Fundamental Principles

Having argued for the separation of parallelism issues from those that are related to computation rules, we now concentrate on the fundamental principles governing parallelism in the different models proposed. We argue that moving a principle from one system to another can often be done quite easily if another such "separation" is performed: isolating the principle itself from the *level of granularity* at which it is applied. This means viewing the parallelizing principle involved as associated to a generic concept of thread, to be particularized for each system, according to the fundamental unit of parallelism used in such system.

As an example, and following these ideas, the fundamental principle of determinism used in the basic Andorra model can be applied to the &-Prolog system. The basic unit of parallelism considered when parallelizing programs in the classical &-Prolog tools is the subtree corresponding to the complete resolution of a given goal in the resolvent. If the basic Andorra principle is applied at this level of granularity its implications are that deterministic subtrees can and should be executed in parallel (even if they are "dependent" in the classical sense). Moving the notions of determinism in the other direction, i.e. towards a finer level of granularity, one can think of applying the principle at the level of bindings, rather than clauses, which yields the concept of "binding determinism" of PNU-Prolog [26].

In fact, the converse can also be done: the underlying principles of &-Prolog w.r.t. parallelism –basically its independence rules– can in fact be applied at the granularity level of the Andorra model. The concept of independence in the context of &-Prolog is defined informally as requiring that a part of the execution "will not be affected" by another. Sufficient conditions –strict and non-strict independence [14]– are then defined which are shown to ensure this property. We argue that applying these concepts at the granularity level of the Andorra model gives some new ways of understanding the model and some new solutions for its parallelization. In order to do this it is quite convenient to look at the basic operations in the light of David Warren's *extended* Andorra model.[2] The extended Andorra model brings in the first place the idea of presenting the execution of logic programs as a series of simple, low level operations on and-or trees. In addition to defining a lower level of granularity, the extended Andorra model incorporates some principles which are related in part to parallelism and in part to computation rule related issues such as the above mentioned basic Andorra principle and the avoidance of re-computation of goals.

On the other hand the extended Andorra model also leaves several other issues relatively more open. One example is that of when nondeterministic reductions may take place in parallel. One answer for this important and relatively open issue was given in the instantiation of the model in the AKL language. In AKL the concept of "stability" is defined as follows: a configuration (partial

[2] This is understandable, given that adding independent and-parallelism to the basic Andorra model was one of the objectives in the development of its extended version.

resolvent) is said to be stable if it cannot be affected by other sibling configurations. In that case the operational semantics of AKL allow the non-determinate promotion to proceed. Note that the definition is, not surprisingly, equivalent to that of independence, although applied at a different granularity level. Unfortunately stability/independence is in general an undecidable property. However, applying the work developed in the context of independent and-parallelism at this level of granularity provides sufficient conditions for it. The usefulness of this is underlined by the fact that the current version of AKL incorporates the relatively simple notion of strict independence (i.e. the absence of variable sharing) as its stability rule. However, the presentation above clearly marks the way for incorporating more advanced concepts, such as non-strict independence, as a sufficient condition for the independence/stability rule. As will be mentioned, we are actively working on compile-time detection of non-strict independence, which we believe will be instrumental in this context. Furthermore, and as we will show, when adding constraint support to a system the traditional notions of independence are no longer valid and both new definitions of independence and sufficient conditions for it need to be developed. We believe that the view proposed herein allows the direct application of general results concerning independence in constraint systems to several realms, such as the extended Andorra model and AKL.

Another way of moving the concept of independence to a finer level of granularity is to apply it at the binding level. This yields a rule which states that dependent bindings of variables should wait for their leftmost occurrences to complete (in the same way as subtrees wait for dependent subtrees to their left to complete in the standard independent and-parallelism model), which is essentially the underlying rule of the DDAS model [28]. In fact, one can imagine applying the principle of non-strict independence at the level of bindings, which would yield a "non-strict" version of DDAS which would not require dependent bindings to wait for bindings to their left which are guaranteed to never occur, or for bindings which are guaranteed to be compatible with them.

Recently, new concepts of independence have been proposed for constraint logic programming [6], since the traditional concepts are not valid in this context. It is our belief that these new concepts can also be applied at different granularity levels and thus render "constraint correct" versions of models such as DDAS (and also be used for defining sufficient conditions for stability in the context of constraints other than Herbrand). In order to do this, we have recently proposed a very fine grain, truly concurrent semantics for both CC-type languages with atomic tell and CLP-type languages [2]. Applications of this semantics are illustrated in [25].

With this view in mind we argue that, once thy are abstracted out from the control rules, and from the granularity level at which they are applied, there exist several common, fundamental principles which govern exploitation of parallelism. Our discussion has revolved around:

- *independence*, which allows parallelism among non-deterministic threads, provided they do not "affect" each other,

– *determinacy*, which allows parallelism among dependent threads.

We believe there are other such fundamental principles, among which we would like to mention *non-failure*, which allows avoiding speculativeness, and *granularity*, or thread size, which allows guaranteeing speedup in the presence of overheads. Space limitations prevent us from elaborating on these.

2.3 Parallelism vs. Concurrency

Similarly to the separations mentioned above (parallelism vs. computation rule and principles vs. granularity level of their application) we also believe in a separation of "concurrency" from both parallelism and computation rules. We believe that concurrency is most useful when explicitly controlled by the user and in that sense it should in some ways also be separate from the implicit computation rules, although this is more of a source language semantics choice. This is in contrast with parallelism, which ideally should be transparent to the user, and with smart computation rules of which the user should be aware, in the sense of being able to derive an upper bound on the amount of computation involved in running a program for a given query using that rule. Space limitations prevent us from elaborating more on this topic or that of the separation between concurrency and parallelism. However, an example of an application of the latter can be seen in *schedule analysis*, where the maximal essential components of concurrency are isolated and sequenced to allow the most efficient possible execution of the concurrent program by one agent [21]. Schedule analysis is, after all, an application of the concept of dependence (or, conversely, independence) at a certain level of granularity in order to "unparallelize" a program, and is thus based on the same principles as automatic parallelization.

Furthermore, we believe that there are actually at least two forms of concurrency based on whether or not there is a notion of a "computing agent" attached to the concurrent task. In other words, whether there is a notion of "computational gas" or "fairness" attached to each such task. The first form of concurrency is traditionally referred to as "coroutining". This is the concurrency obtained explicitly with Prolog's "freeze" (and, also, through &-Prolog's "&"), and implicitly in Gödel [15] or in Andorra-I [27] through the basic Andorra principle. The second one is associated with the explicit creation of an actual process and is generally not present in concurrent logic programming systems. The differences between these two forms of concurrency can easily been seen through an example: imagine a procedure "cube(X)" that opens a window in a display and shows a cube which rotates X times. Assume "&" to be the concurrent operator. Now consider the concurrent conjunction "cube(5) & cube(5)". Note that the two tasks are independent and thus need no synchronization. Thus, in the case of a coroutining interpretation of "&" one possible execution would be to execute the two calls sequentially. But it may be that what the programmer actually means is that two windows should be opened (more or less) at the same time and the cubes should rotate (more or less) simultaneously, in which case the second type of concurrency is meant. In any case, the two interpretations

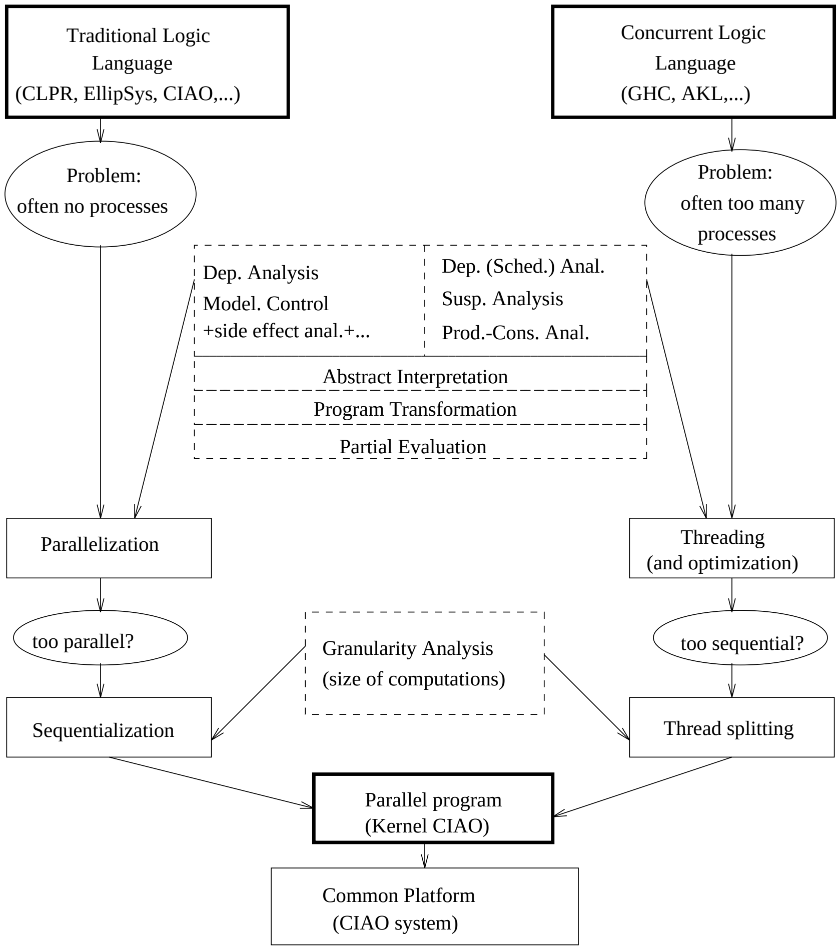

Fig. 1. Common Problems in Compilation

are clearly different. In the CIAO system we propose to support both forms of concurrency through different user-level primitives ("&" and "&&" – for more details, please see [12]).

3 Towards General-Purpose Implementations

We believe that the points regarding the separation of issues and fundamental principles sketched in the previous sections at the same time explain and are supported by the recent trend towards convergence in the analysis and implementation techniques of systems that are in principle very different, such as the various parallel implementations of Prolog on one hand (see, for example, [13, 24, 27]) and the implementations of the various committed choice languages on the other (see, for example, [5, 10, 19, 31, 32]). The former are based on schemes for parallelizing a sequential language; they tend to be stack-based, in the sense that (virtual) processors allocate environments on a stack and execute computations "locally" as far as possible until there is no more work to do, at which point they "steal" work from a busy processor. The latter, by contrast, are based on concurrent languages with dataflow synchronization; they tend to be heap-based, in the sense that environments are generally allocated on a heap, and there is (at least conceptually) a shared queue of active tasks.

The aforementioned convergence can be observed in that, on one hand, driven by the demonstrated utility of delay primitives in sequential Prolog systems (e.g., the `freeze` and `block` declarations of Sicstus Prolog [4], `when` declarations of NU-Prolog [29], etc.), parallel Prolog systems have been incorporating capabilities to deal with user-defined suspension and coroutining behaviors—for example, &-Prolog allows programmer-supplied *wait*-declarations, which can be used to express arbitrary control dependencies. In sequential Prolog systems with delay primitives, delayed goals are typically represented via heap-allocated "suspension records," and such goals are awakened when the variables they are suspended on get bindings [3]. Parallel Prolog systems inherit this architecture, leading to implementations where individual tasks are stack-oriented, together with support for heap-allocated suspensions and dataflow synchronization. On the other hand, driven by a growing consensus that some form of "sequentialization" is necessary to reduce the overhead of managing fine-grained parallel tasks on stock hardware (see, for example, [9, 30, 22, 11]), implementors of committed choice languages are investigating the use of compile-time analyses to coalesce fine-grained tasks into coarser-grained sequential threads that can be implemented more efficiently. This, again, leads to implementations where individual sequential threads execute in a stack-oriented manner, but where sets of such threads are represented via heap-allocated activation records that employ dataflow synchronization. Interestingly, and conversely, in the context of parallel Prolog systems, there is also a growing body of work trying to address the problem of automatic parallelizing compilers often "parallelizing too much" which appears if the target architecture is not capable of supporting fine grain parallelism. Figure 1 illustrates this.

This convergence of trends both at the compiler and the run-time system levels is exciting: it suggests that we are beginning to understand the essential implementation issues for these languages, and that from an implementor's perspective these languages are not as fundamentally different as was originally believed. It also opens up the possibility of having a general purpose kernel language and abstract machine to serve as a compilation target for a variety of user-level languages. As mentioned before this is precisely one of the objectives of the CIAO system. Encouraging initial results in this direction have been demonstrated in the sequential context by the QD-Janus system [8] of S. Debray and his group. QD-Janus, which compiles down to Sicstus Prolog and uses the delay primitives of the Prolog system to implement dataflow synchronization, turns out to be more than three times faster, on the average, than Kliger's customized implementation of FCP(:) [23] and requires two orders of magnitude less heap memory [7]. We believe that this point will also extend to parallel systems: as noted above, the &-Prolog system already supports stack-oriented parallel execution together with arbitrary control dependencies, suspension, and dataflow synchronization via user-supplied *wait*-declarations, all characteristics that CIAO inherits. This suggests that, with some enhancements, the dependence graphs and *wait*-declarations of &-Prolog/CIAO, can serve as a common intermediate language, and its runtime system can act as an appropriate common low-level implementation, for a variety of parallel logic programming implementations.

Along these lines, in [12] we have recently proposed a method for providing user-level access to such a generic implementation, based on the use of attributed variables [18, 3]. Incorporating the possibility of attaching attributes to variables in a logic programming system has been shown to allow the addition of general constraint solving capabilities to it [16, 17]. This approach is very attractive in that by adding a few primitives any logic programming system can be turned into a generic constraint logic programming system in which constraint solving can be user defined, and at source level – an extreme example of the "glass box" approach. In [12] we propose applying the concept of attributed variables to provide the same "glass box" flavor in a generic parallel/concurrent (constraint) logic programming system. We argue that a system which implements attributed variables and a few additional primitives (such as those present in &-Prolog/CIAO) can be easily customized at source level to implement many of the languages and execution models of parallelism and concurrency currently proposed, in both shared memory and distributed systems. We do not mean to suggest that the performance of such a system will be *optimal* for all possible logic programming languages: our claim is rather that it will provide a way to researchers in the community implement their languages with considerably less effort than has been possible to date, and yet attain reasonably good performance. We are currently exploring these points in collaboration with S. Debray, F. Rossi, and U. Montanari, by using the CIAO system as a generic implementation platform.

4 Acknowledgements

The work presented in this paper is funded in part by ESPRIT contracts 6707 "PARFORCE" and 7195 "ACCLAIM", and CICYT contract TIC93-TIC93-0737. This work is part of the joint effort of the CLIP (Computational Logic, Implementation, and Parallelism) group at UPM which includes Francisco Bueno, Daniel Cabeza, Manuel Carro, María José García de la Banda, Pedro López García, Germán de la Puebla, and Manuel Hermenegildo.

References

1. K.A.M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*, pages 757–

776. MIT Press, October 1990.
2. F. Bueno, M. Hermenegildo, U. Montanari, and F. Rossi. From Eventual to Atomic and Locally Atomic CC Programs: A Concurrent Semantics. In *Fourth International Conference on Algebraic and Logic Programming*, number 850 in LNCS, pages 114–132. Springer-Verlag, September 1994.
3. M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the Wam. In *Fourth International Conference on Logic Programming*, pages 40–58. University of Melbourne, MIT Press, May 1987.
4. M. Carlsson. *Sicstus Prolog User's Manual.* Po Box 1263, S-16313 Spanga, Sweden, February 1988.
5. Jim Crammond. Scheduling and Variable Assignment in the Parallel Parlog Implementation. In *1990 North American Conference on Logic Programming.* MIT Press, 1990.
6. M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. In *1993 International Logic Programming Symposium*, pages 130–146. MIT Press, Cambridge, MA, October 1993.
7. S. K. Debray. Implementing logic programming systems: The quiche-eating approach. In *ICLP '93 Workshop on Practical Implementations and Systems Experience in Logic Programming*, Budapest, Hungary, June 1993.
8. S. K. Debray. QD-Janus : A Sequential Implementation of Janus in Prolog. *Software—Practice and Experience*, 23(12):1337–1360, December 1993.
9. S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.
10. I. Foster and S. Taylor. *Strand* : A practical parallel programming tool. In *1989 North American Conference on Logic Programming*, pages 497–512. MIT Press, October 1989.
11. P. López García, M. Hermenegildo, and S.K. Debray. Towards Granularity Based Control of Parallelism in Logic Programs. In *Proc. of First International Symposium on Parallel Symbolic Computation, PASCO'94*, pages 133–144. World Scientific Publishing Company, September 1994.
12. M. Hermenegildo, D. Cabeza, and M. Carro. On The Uses of Attributed Variables in Parallel and Concurrent Logic Programming Systems. Technical report CLIP 5/94.0, School of Computer Science, Technical University of Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, June 1994.
13. M. Hermenegildo and K. Greene. The &-prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.
14. M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 1995. To appear.
15. P. Hill and J. Lloyd. *The Goedel Programming Language.* MIT Press, Cambridge MA, 1994.
16. C. Holzbaur. *Specification of Constraint Based Inference Mechanisms through Extended Unification.* PhD thesis, University of Vienna, 1990.
17. C. Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In *1992 International Symposium on Programming Language Implementation and Logic Programming*, pages 260–268. LNCS631, Springer Verlag, August 1992.
18. Serge Le Houitouze. A New Data Structure for Implementing Extensions to Prolog. In P. Deransart and J. Małuszyński, editors, *Proceedings of Programming Language*

Implementation and Logic Programming, number 456 in Lecture Notes in Computer Science, pages 136–150. Springer, August 1990.
19. A. Houri and E. Shapiro. A sequential abstract machine for flat concurrent prolog. Technical Report CS86-20, Dept. of Computer Science, The Weizmann Institute of Science, Rehovot 76100, Israel, July 1986.
20. S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.
21. A. King and P. Soper. Reducing scheduling overheads for concurrent logic programs. In *International Workshop on Processing Declarative Knowledge*, Kaiserslautern, Germany, (1991). Springer-Verlag.
22. Andy King and Paul Soper. Schedule Analysis of Concurrent Logic Programs. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 478–492, Washington, USA, 1992. The MIT Press.
23. S. Kliger. *Compiling Concurrent Logic Programming Languages*. PhD thesis, The Weizmann Institute of Science, Rehovot, Israel, October 1992.
24. E. Lusk et. al. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7(2,3), 1990.
25. U. Montanari, F. Rossi, F. Bueno, M. García de la Banda, and M. Hermenegildo. Towards a Concurrent Semantics based Analysis of CC and CLP. In *Principles and Practice of Constraint Programming*, LNCS 874, pages 151–161. Springer-Verlag, May 1994.
26. L. Naish. Parallelizing NU-Prolog. In *Fifth International Conference and Symposium on Logic Programming*, pages 1546–1564. University of Washington, MIT Press, August 1988.
27. 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 *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, April 1990.
28. K. Shen. Exploiting Dependent And-Parallelism in Prolog: The Dynamic, Dependent And-Parallel Scheme. In *Proc. Joint Int'l. Conf. and Symp. on Logic Prog.* MIT Press, 1992.
29. J. Thom and J. Zobel. *NU-Prolog Reference Manual*. Dept. of Computer Science, U. of Melbourne, May 1987.
30. E. Tick. Compile-Time Granularity Analysis of Parallel Logic Programming Languages. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.
31. E. Tick and C. Bannerjee. Performance evaluation of monaco compiler and runtime kernel. In *1993 International Conference on Logic Programming*, pages 757–773. MIT Press, June 1993.
32. K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 140–156. MIT Press, Cambridge MA, 1987.
33. D.H.D. Warren. The Extended Andorra Model with Implicit Control. In Sverker Jansson, editor, *Parallel Logic Programming Workshop*, Box 1263, S-163 13 Spanga, SWEDEN, June 1990. SICS.

This article was processed using the LaTeX macro package with LLNCS style