Towards CIAO-Prolog - A Parallel Concurrent Constraint System

M. Hermenegildo

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

1 Introduction

We present an informal discussion on some methodological aspects regarding the efficient parallel implementation of (concurrent) (constraint) logic programming systems, as well as an overview of some of the current work performed by our group in the context of such 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.

CIAO can be in some ways seen as an evolution of the &-Prolog [17] system concepts: it builds on &-Prolog ideas such as parallelization and optimization heavily based on compile-time global analysis and efficient abstract machine design. On the other hand, CIAO is aimed at adding several important extensions, such as or-parallelism, constraints, more direct support for explicit concurrency in the source language, as well as other ideas inspired by proposals such as Muse [1] and Aurora [27], GHC [39], PNU-Prolog [30], IDIOM [16], DDAS [32], Andorra-I [31], AKL [20], and the extended Andorra model [40]. One of the objectives of CIAO is to offer at the same time all the user-level models provided by these systems.

More than a precisely defined design, at this point the CIAO system should be seen as a target which serves to motivate and direct our current research efforts. This impreciseness is purposely based on our belief that, in order to develop an efficient system with the characteristics that we desire, a number of technologies have to mature and others still have to be developed from scratch. Thus, our main focus at the moment is in the development of some of these technologies, which include, among others, improved memory management and scheduling techniques, development of parallelization technology for non-strict forms of independence, efficient combination of and- and or-parallelism, support of several programming paradigms via program transformation, and the extension of current parallelization theory and global analysis tools to deal with constraint-based languages.

We will start our discussion by dealing with some methodological issues. We will then introduce some of our recent work in the direction mentioned above. Given the space limitations the description will be aimed at providing an overall view of our recent progress and a set of pointers to some relevant recent publications and technical reports which describe our results more fully. We hope that in light of the objective of providing pointers, the reader will be kind enough to excuse the summarized descriptions and the predominance in the references of (at least recent) work of our group.

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 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 the concepts mentioned above, 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 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 [18].

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

[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.

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.

2.2 Abstracting Away the Granularity Level: The 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 [30].

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 [18]– 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 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.

[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.

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 [32]. 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.

With this view in mind we argue that there are essentially four fundamental principles which govern exploitation of parallelism:

- *independence*, which allows parallelism among non-deterministic threads,
- *determinacy*, which allows parallelism among dependent threads,
- *non-failure*, which allows guaranteeing non-speculativeness, and
- *granularity*, which allows guaranteeing speedup in the presence of overheads.

2.3 User-level Concurrency

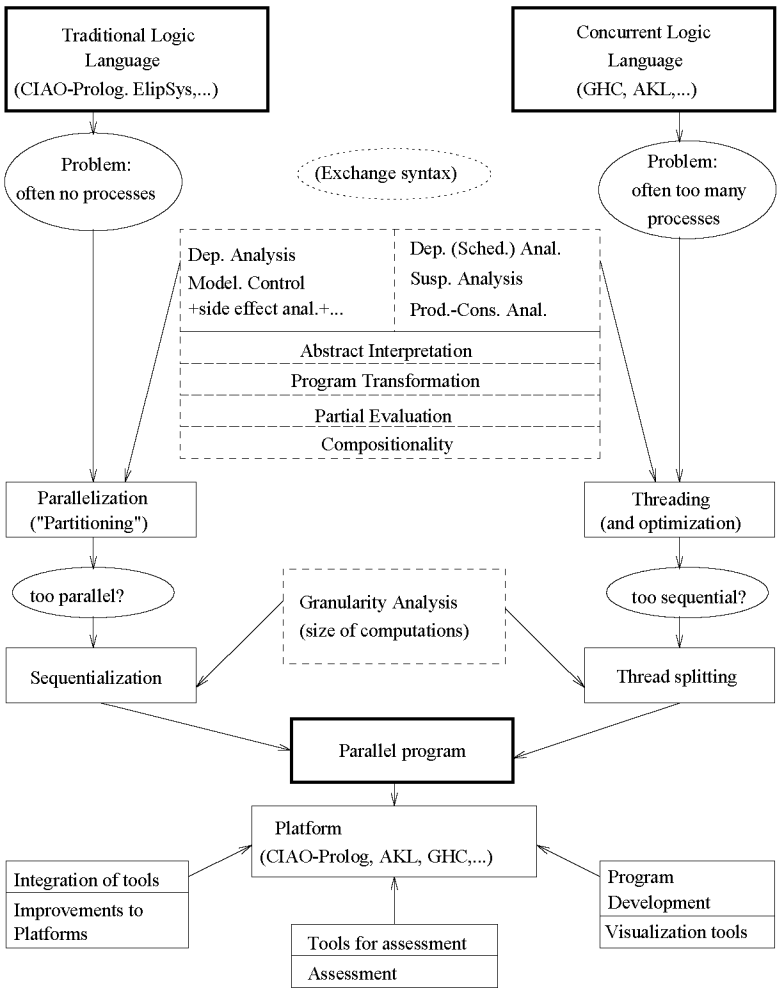
Similarly to the separations mentioned above (parallelism vs. computation rule and principles vs. granularity level of their application) we also believe in a separation of "concurrency" from both parallelism and computation rules. We believe concurrency is most useful when it is explicitly controlled by the user and should be separate from the implicit computation rules. This is in contrast with parallelism, which ideally should be transparent to the user, and with smart computation rules of which the user should only 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.

2.4 Towards a General-Purpose Implementation

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 implemenation techniques of systems that are in principle very different, such as the various parallel implementations of Prolog on one hand (see, for example, [17, 27, 2]) and the implementations of the various committed choice languages on the other (see, for example, [7, 8, 14, 19, 24, 35, 38, 39]). 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 [6], `when` declarations of NU-Prolog [36], 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 [5]. 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, [13, 37, 22]), 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 2.4 illustrates this (and in fact reflects the interactions among the partners of the ParForCE Esprit project, where some of these interactions are being investigated).



This convergence of trends 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 abstract machine to serve as a compilation target for a variety of 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 [12] 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 [11]. 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 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. 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.

3 Some of our recent work in this context

We now provide an overview of our recent work in filling some of the gaps that, in our understanding, are missing in order to fulfill the objectives outlined in the previous section.

3.1 Parallelism based on Non-Strict Independence

One of our starting steps is to improve the independence-based detection of parallelism based on information that can be obtained from global analysis using the current state of the art in abstract interpretation. We have had a quite successful experience using this technique for detecting the classical notion of "strict" independence. These results are summarized in [3], which compares the performance of several abstract interpretation domains and parallelization algorithms using the &-Prolog compiler and system.

While these results are quite encouraging there is another notion of independence – "non-strict" independence [18] – which ensures the same important "no slow down" properties than the traditional notion of strict independence and allows considerable more parallelism than strict independence [33]. The support of non-strict independence requires, however, a review of our compile-time parallelization technology which to date has been exclusively based on strict independence. In [4] we describe some of our recent work filling this gap. Rules and algorithms are provided for detecting and annotating non-strict independence at compile-time. We also propose algorithms for combined compile-time/run-time detection, including run-time checks for this type of parallelism, which in some cases turn out to be different from the traditional groundness and independence checks used for strict independence. The approach is based on the knowledge of certain properties about run-time instantiations of program variables —sharing, groundness, freeness, etc.— for which compile-time technology is available, with new approaches being currently proposed. Rather than dealing with the analysis itself, we present how the analysis results can be used to parallelize programs.

3.2 Parallelization in the Presence of Constraints: Independence / Stability

In the CIAO-Prolog system, from the language point of view, we assume a constraint-based, non-deterministic logic programming language. As such, and apart from the concurrency/coroutining primitives, the user language can be viewed as similar to Prolog when working on the Herbrand domain, and to systems such as CLP(R) or CHIP when working over other domains. This implies that the traditional notions of independence / stability need to be evaluated in this context and, if necessary, extended to deal with the fact that constraint solving is occurring in the actual execution in lieu of unification.

Previous work in the context of traditional Logic Programming languages has concentrated on defining independence in terms of preservation of search space, and such preservation has then been achieved by ensuring that either the goals do not share variables (*strict independence*) or if they share variables, that they do not "compete" for their bindings (*non-strict independence*).

In [10] we have shown (in collaboration with Monash University) that a naive extrapolation of the traditional notions of independence to Constraint Logic Programming is unsatisfactory (in fact, wrong) for two reasons. First, because interaction between variables through constraints is more complex than in the case of logic programming. Second, in order to ensure the efficiency of several optimizations not only must independence of the search space be considered, but also an orthogonal issue – "independence of constraint solving." We clarify these issues by proposing various types of search independence and constraint solver independence, and show how they can be combined to allow different independence-related optimizations, in particular parallelism. Sufficient conditions for independence which can be evaluated "a-priori" at run-time and are easier to identify at compile-time than the original definitions, are also proposed. Also, it has been shown how the concepts proposed, when applied to traditional Logic Programming, render the traditional notions and are thus a strict generalization of such notions.

3.3 Extending Global Analysis Technology to CLP

As mentioned before, since many optimizations, including independence / stability detection, are greatly aided by (and sometimes even require) global analysis, traditional global analysis techniques have to be extended to deal with the fact that constraint solving is occurring in the actual execution in lieu of unification. In [9] we present and illustrate with an implementation a practical approach to the dataflow analysis of programs written in constraint logic programming (CLP) languages using abstract interpretation. We argue that, from the framework point of view, it suffices to propose quite simple extensions to traditional analysis methods which have already been proved useful and practical and for which efficient fixpoint algorithms have been developed. This is shown by proposing a simple but quite general extension to the analysis of CLP programs of Bruynooghe's traditional framework, and describing its implementation – the "PLAI" system. As the original, the framework is parametric and we provide correctness conditions to be met by the abstract domain related functions to be provided. In this extension constraints are viewed not as "suspended goals" but rather as new information in the store, following the traditional view of CLP. Using this approach, and as an example of its use, a complete, constraint system independent, abstract analysis is presented for approximating definiteness information. The analysis is in fact of quite general applicability. It has been implemented and used in the analysis of CLP(R) and Prolog-III applications. Results from this implementation are also presented which show good efficiency and accuracy for the analysis.

This framework, combined with the ideas of [10] (and [29]) presented in the previous section, is the basis for our current development of automatic parallelization tools for CLP programs, and, in particular, of the parallelizer for the CIAO-Prolog system.

3.4 Extending Global Analysis Technology for Explicit Concurrency

Another step that has to be taken in adapting current compile-time technology to CIAO systems is to develop global analysis technology which can deal with the fact that the new computation rules allow the specification of concurrent executions. While there have been many approaches proposed in the literature to address this problem, in a first approach we focus on a class of languages (which includes modern Prologs with delay declarations) which provide both sequential and concurrent operators for composing goals. In this approach we concentrate on extending traditional abstract interpretation based global analysis techniques to incorporate these new computation rules. This gives a practical method for analyzing (constraint) logic programming languages with (explicit) dynamic scheduling policies, which is at the same time equally powerful as the older methods for traditional programs.

We have developed, in collaboration with the University of Melbourne, a framework for global dataflow analysis of this class of languages [28]. First, we give a denotational semantics for languages with dynamic scheduling which provides the semantic basis for our generic analysis. The main difference with denotational definitions for traditional Prolog is that sequences of delayed atoms must also be abstracted and are included in "calls" and "answers." Second, we give a generic global dataflow analysis algorithm which is based on the denotational semantics. Correctness is formalized in terms of abstract interpretation. The analysis gives information about call arguments and the delayed calls, as well as implicit information about possible call schedulings at runtime. The analysis is generic in the sense that it has a parametric domain and various parametric functions. Finally, we demonstrate the utility and practical importance of the dataflow analysis algorithm by presenting and implementing an example instantiation of the generic analysis which gives information about groundness and freeness of variables in the delayed and actual calls. Some preliminary test results are included in which the information provided the implemented analyzer is used to reduce the overhead of dynamic scheduling by removing unnecessary tests for delaying and awakening, to reorder goals so that atoms are not delayed, and to recognize calls which are "independent" and so allow the program to be run in parallel.

3.5 Granularity Analysis

While logic programming languages offer a great deal of scope for parallelism, there is usually some overhead associated with the execution of goals in parallel because of the work involved in task creation and scheduling. In practice, therefore, the "granularity" of a goal, i.e. an estimate of the work available under it, should be taken into account when deciding whether or not to execute a goal in parallel as a separate task. Building on the ideas first proposed in [13] we describe in [25] a proposal for an automatic granularity control system, which is based

on an accurate granularity analysis and program transformation techniques. The proposal covers granualrity control of both and-parallelism and or-parallelism. The system estimates the granularities of goals at compile time, but they are actually evaluated at runtime. The runtime overhead associated with our approach is usually quite small, and the performance improvements resulting from the incorporation of grain size control can be quite good. Moreover a static analysis of the overhead associated with granularity control process is performed in order to decide its convenience.

The method proposed requires among other things knowing the size of the terms to which program variables are bound at run-time (something which is useful in a class of optimizations which also include recursion elimination). Such size is difficult to even approximate at compile time and is thus generally computed at run-time by using (possibly predefined) predicates which traverse the terms involved. In [26] we propose a technique based on program transformation which has the potential of performing this computation much more efficiently. The technique is based on finding program procedures which are called before those in which knowledge regarding term sizes is needed and which traverse the terms whose size is to be determined, and transforming such procedures so that they compute term sizes "on the fly". We present a systematic way of determining whether a given program can be transformed in order to compute a given term size at a given program point without additional term traversal. Also, if several such transformations are possible our approach allows finding minimal transformations under certain criteria. We also discuss the advantages and applications of our technique and present some performance results.

3.6 Memory Management and Scheduling in Non-deterministic And-parallel Systems

From our experience with the &-Prolog system implementation [17], the results from the DDAS simulator [32], and from informal conversations with the Andorra-I developers, efficient memory management in systems which exploit and-parallelism is a problem for which current solutions are not completely satisfactory. This appears to be specially the case with and-parallel systems which support don't-know nondeterminism or deep guards. We believe *non-deterministic* and-parallel schemes to be highly interesting in that they present a relatively general set of problems to be solved (including most of those encountered in the memory management of or-parallel only systems) and have chosen to concentrate on their study.

In collaboration with U. of Bristol, we have developed a distributed stack memory management model which allows flexible scheduling of goals. Previously proposed models are lacking in that they impose restrictions on the selection of goals to be executed or they may require a large amount of virtual memory. Our measurements imply that the above mentioned shortcomings can have significant performance impacts, and that the extension that we propose of the "Marker Model" allows flexible scheduling of goals while keeping (virtual) memory consumption down. We also discuss methods for handling forward and backward execution, cut, and roll back. Also, we show that the mechanism proposed for flexible scheduling can be applied to the efficient handling of the very general form of suspension that can occur in systems which combine several types of non-deterministic and-parallelism and advanced computation rules, such as PNU-Prolog [30], IDIOM [16], DDAS [32], AKL [20], and, in general, those that can be seen as an instantiation of the extended Andorra model [40]. Thus, we believe that the results may be applicable to a whole class of and- and or-parallel systems. Our solutions and results are described more fully in [34].

3.7 Incorporating Or-Parallelism: The ACE Approach

Another important issue is the incorporation of Or-parallelism to an and-parallel system. This implies well known problems related to or-parallelism itself, such as the maintenance of several binding environments, as well as new problems such as the interactions of the multiplicity of binding environments and threads of or-parallel computation with the scoping and memory management requirements of and-parallelism. The stack copying approach, as exemplified by the MUSE system, has been shown to be a quite successful alternative for representing multiple environments during or-parallel execution of logic programs. In collaboration with the U. of New Mexico and U. of Bristol we have developed an approach for parallel implementation of logic programs, described more fully in [15], which we believe is capable of exploiting both or-parallelism and independent and-parallelism (as well as other types of and-parallelism) in an efficient way using stack copying ideas. This model combines such ideas with proven techniques in the implementation of independent and-parallelism, such

as those used in &-Prolog. We show how all solutions to non-deterministic and-parallel goals are found without repetitions. This is done through re-computation as in Prolog (and &-Prolog), i.e., solutions of and-parallel goals are not shared. We propose a scheme for the efficient management of the address space in a way that is compatible with the apparently incompatible requirements of both and- and or-parallelism. This scheme allows incorporating and combining the memory management techniques used in (non-deterministic) and-parallel systems, such as those mentioned in the previous section, and memory management techniques of or-parallel systems, such as incremental copying. We also show how the full Prolog language, with all its extra-logical features, can be supported in our and-or parallel system so that its sequential semantics is preserved. The resulting system retains the advantages of both purely or-parallel systems as well as purely and-parallel systems. The stack copying scheme together with our proposed memory management scheme can also be used to implement models that combine dependent and-parallelism and or-parallelism, such as Andorra and Prometheus.

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*. MIT Press, October 1990.

[2] P. Brand, S. Haridi, and D.H.D. Warren. Andorra Prolog—The Language and Application in Distributed Simulation. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.

[3] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. Technical Report TR Number CLIP7/93.0, T.U. of Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, October 1993.

[4] D. Cabeza and M. Hermenegildo. Towards Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. Technical Report TR Number CLIP5/92.1, U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, August 1993.

[5] 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.

[6] M. Carlsson. *Sicstus Prolog User's Manual*. Po Box 1263, S-16313 Spanga, Sweden, February 1988.

[7] K. L. Clark and S. Gregory. Notes on the Implementation of Parlog. *Journal of Logic Programming*, 2(1), April 1985.

[8] Jim Crammond. Scheduling and Variable Assignment in the Parallel Parlog Implementation. In *1990 North American Conference on Logic Programming*. MIT Press, 1990.

[9] M.J.García de la Banda and M. Hermenegildo. A Practical Approach to the Global Analysis of Constraint Logic Programs. In *1993 International Logic Programming Symposium*. MIT Press, Cambridge, MA, October 1993.

[10] M.J.García de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. In *1993 International Logic Programming Symposium*. MIT Press, Cambridge, MA, October 1993.

[11] 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.

[12] S. K. Debray. Qd-janus : A sequential implementation of janus in prolog. *Software—Practice and Experience*, 23(12):1337–1360, December 1993.

[13] 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.

[14] 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.

[15] G. Gupta, M. Hermenegildo, Enrico Pontelli, and Vítor Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *International Conference on Logic Programming*. MIT Press, June 1994. to appear.

[16] G. Gupta, V. Santos-Costa, R. Yang, and M. Hermenegildo. IDIOM: Integrating Dependent and-, Independent and-, and Or-parallelism. In *1991 International Logic Programming Symposium*, pages 152–166. MIT Press, October 1991.

[17] M. Hermenegildo and K. Greene. The &-prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[18] 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*, 1993. To appear.

[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] M. Korsloot and E. Tick. Compilation techniques for nondeterminate flat concurrent logic programming languages. In *1991 International Conference on Logic Programming*, pages 457–471. MIT Press, June 1991.

[25] P. López and M. Hermenegildo. An automatic sequentializer based on program transformation. Technical report, T.U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, April 1993.

[26] P. López and M. Hermenegildo. Dynamic Term Size Computation in Logic Programs via Program Transformation. Technical report, T.U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, April 1993. Presented at the 1993 COMPULOG Area Meeting on Parallelism and Implementation Technologies.

[27] E. Lusk et. al. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7(2,3), 1990.

[28] K. Marriott, M. Garcia de la Banda, and M. Hermenegildo. Analyzing Logic Programs with Dynamic Scheduling. In *Proceedings of the 20th. Annual ACM Conf. on Principles of Programming Languages.* ACM, January 1994.

[29] U. Montanari, F. Rossi, F. Bueno, M. García de la Banda, and M. Hermenegildo. Contextual Nets and Constraint Logic Programming: Towards a True Concurrent Semantics for CLP. Technical report, T.U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, January 1993. To be Presented at the ICLP'93 Post Conference Workshop on Concurrent Constraint Logic Programming.

[30] L. Naish. Parallelizing NU-Prolog. In *Fifth International Conference and Symposium on Logic Programming*, pages 1546–1564. University of Washington, MIT Press, August 1988.

[31] 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.

[32] 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.

[33] K. Shen and M. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. In *1991 International Logic Programming Symposium.* MIT Press, October 1991.

[34] K. Shen and M. Hermenegildo. A Flexible Scheduling and Memory Management Scheme for Non-Deterministic, And-parallel Execution of Logic Programs. Technical report, T.U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, April 1993. Presented at the ICLP'93 Post Conference Workshop on Logic Program Implementation.

[35] S. Taylor, S. Safra, and E. Shapiro. A parallel implementation of flat concurrent prolog. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 575–604, Cambridge MA, 1987. MIT Press.

[36] J. Thom and J. Zobel. *NU-Prolog Reference Manual.* Dept. of Computer Science, U. of Melbourne, May 1987.

[37] E. Tick. Compile-Time Granularity Analysis of Parallel Logic Programming Languages. In *International Conference on Fifth Generation Computer Systems.* Tokyo, November 1988.

[38] 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.

[39] K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 140–156. MIT Press, Cambridge MA, 1987.

[40] 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.