In ILPS 1995 WS on Visions for the Future of Logic Programming

The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems

*M. Hermenegildo** *F. Bueno** *M. García de la Banda†* *G. Puebla**

Extended Abstract

1 Introduction

In [HtCg93, HtCg94] we discussed several methodological aspects regarding the design and efficiency of a class of future logic programming systems. In particular, we proposed a somewhat novel view of concurrent logic programming systems, based on a particular definition of parallelism. We argued that, under this view, a large number of the actual systems and models can be explained (and implemented) through the application of only a few basic principles. They include determinism, non-failure, independence (also referred to as stability), and task granularity. We also argued for a separation between those principles which have to do with the computation rule (i.e., to performing the least work possible) and those directly related to parallelism (i.e., to performing such work in the smallest amount of time by splitting it among several processors). Finally, and basing our discussion on the convergence of concepts that this view brought, we sketched the design of the CIAO (Concurrent, Independence-based And/Or parallel) system, a platform for the implementation of several source languages and models based on a common, generic abstract machine and using an intermediate kernel language.

The purpose of this paper is to report on recent progress in the design and implementation of the CIAO system itself, with special emphasis on the capabilities of the compiler and the techniques used for supporting such capabilities. CIAO is a multi-dialect compiler, run-time, and program development system. It is based on a versatile kernel language which has extensive sequential, parallel, concurrent, and distributed execution capabilities. The CIAO system is *generic* in the sense that the different source-level languages are supported by compilation via *source to source* transformations into the kernel language (which is also the native CIAO language). The system subsumes standard left-to-right SLD resolution and the determinate-first principle (as in the Andorra model [SCWY90, dMSC93]). The kernel language is directly supported by a comparatively simple abstract machine, mainly based on the parallelism and concurrency capabilities of the PWAM/&-Prolog [Her86, HG91]. The analysis and transformation techniques used in the compiler are based on novel semantic modeling of CLP and CC program behavior and on the exploitation of fundamental optimization principles (independence/stability and determinism), and techniques based on global analysis (program specialization and abstract executability).

Given the characteristics mentioned above, CIAO can be used quite effectively for developing applications. However, one of its fundamental objectives is to be a tool for easily experimenting with and evaluating language design issues, including program analysis and transformation methods and lower-level implementation techniques. In particular, CIAO can be used to provide efficient implementations of a range of LP, CLP, and CC programming languages, on sequential and multiprocessor machines.

*Computer Science Department, Technical University of Madrid {herme,bueno,german}@fi.upm.es
†Dept. of Computer Science, Monash University, Clayton 3168, Australia. mbanda@bruce.cs.monash.oz.au

This is achieved thanks to a powerful compiler strongly based on program analysis and transformation. This compiler provides the required support for the different programming paradigms and their optimization. The compilation process can be viewed as a translation process from the input language to (kernel) CIAO. Some optimizations are already performed during this translation. However, most optimizations are performed to the already translated code, via source to source transformation. Therefore, most analysis phases are performed at the kernel language level, so that the same analyzer can be used for several models. Optimizations include parallelization, reduction of concurrency and synchronization, reordering of goals, code simplification, and specialization.

2 The CIAO Kernel Language

The CIAO kernel language can be seen as a superset of Prolog (and, having the capability to support constraints, it also subsumes CLP), including the meta-programming and extra-logical facilities. The versatility of this kernel language is mainly due to explicit control primitives. This makes it possible to perform many control-related optimizations at the source level. Such explicit control is performed via operators which include:

- *Sequential Operator:* “`,/2`”. Allows the sequential execution of the goals involved (as in the traditional Prolog style).
- *Parallel Operators:* “`&/2`”, “`&>/2`”, and “`&</1`”. These operators indicate points where parallelism can be exploited. Their behavior is otherwise equivalent to that of the sequential operator. During the parallel execution the communication of bindings is not guaranteed until the join, i.e., until the parallel goals have finished. Therefore, no variable locking is needed and full backtracking is implemented. More concretely:
  - `A &> H` – Sends out goal `A`, to be executed potentially by another agent, returning in the variable `H` a handle to the goal sent.
  - `H <&` – “Joins” the results of the goal whose handle is `H`, or executes it locally if it has not been executed yet. This will also be the point at which backtracking of the goal will be performed during backwards execution.
  - `A & B` – Performs a parallel “fork” of the two goals involved and waits for the execution of both to finish. This is the parallel conjunction operator used, for example, by the &-Prolog parallelizing compiler [BGH94]. If no agents are idle, then the two goals may be executed by the same agent and sequentially, i.e., one after the other. This primitive can be implemented using the previous two:

    ```
    A & B :- B &> H, call(A), H <& .
    ```

    An example of a simple parallel loop is:

    ```
    process_list([]).
    process_list([H|T]) :-
            process(H) & process_list(T).
    ```

- *Concurrency Operator:* “`&/1`”. It allows concurrent programming in the style of CC languages (also, the parallelism in such concurrent execution may be exploited if resources are available). In particular, `A &` sends out the goal `A` to be executed potentially by another agent. No waiting for its return is performed. Bindings in the variables of `A` (tells) will be seen by other agents sharing such variables. Backtracking is limited to allow a relatively straightforward implementation. In particular, in the current version of the CIAO system no “active shared binding” is allowed to be undone via backtracking. An active shared binding is a binding to a variable that is shared among active (i.e., non-finished) processes. Variable communication (and locking) is performed.

  A certain level of encapsulation of search is often necessary in order to ensure that the conditions above hold. This is done explicitly in the kernel language, but can be supported at a higher level in the source languages using constructs such as the search operator of Oz [Smo94] or the deep guard mechanism of AKL.

- *Explicit And-Fairness Operator:* "`&&/1`". This operator is a "fair" version of the `&/1` operator. It explicitly requests the (efficient) association of computational resources (e.g., an operating system thread) to a goal. In particular, if there is no idle agent `A &&` , creates one to execute the goal `A`. Fairness among concurrent threads is ensured. Note that this leaves open the possibility of implementing a fair source language that compiles efficiently into this and the above operators (perhaps via an analysis which can determine the program points where fairness is really needed – to ensure, for example, termination).

- *Explicit Synchronization:* "`wait/1`", "`d_wait/1`, and "`ask/1`'. These operators can be augmented with some meta-tests on the variables (such as `ground/1` or `nonvar/1`). More concretely:

  - `wait(X)` – Suspends the execution until `X` is bound.
  - `d_wait(X)` – Suspends the execution until `X` is deterministically bound.
  - `ask(`C`)` – Suspends the execution until the constraint C is satisfied. Whether this is a primitive or compiles into the previous two primitives depends on how constraint solving is implemented for the particular domain – see later.

  A simple producer-consumer can be programmed as follows:

```
main(L) :- producer(10,L) & , consumer(L).

producer(0,T) :- !, T = [].
producer(N,T) :- N > 0, T = [N|Ns], N1 is N-1, producer(N1,Ns).

consumer(L)   :- wait(L), consumer_body(L).

consumer_body([]).
consumer_body([H|T]) :- consumer(T).
```

  With only one agent active the program above will first produce the whole list before consuming it. If actual interleaving of the producer and consumer is desired, then "`&&`" can be used instead of "`&`".

- *Explicit Placement Operator:* an explicit placement operator ("`@`") allows control of task placement in distributed execution. This operator can be combined with any of the parallelism and concurrency operators mentioned before. These and other primitives for controlling distributed execution, and to implement the concept of active modules or active objects, are described in [CH95]. Such capabilities can be used for example when accessing remote resources such as knowledge bases. They can also be used for performance improvement through parallelism.

The kernel language described above is supported by a comparatively simple abstract machine. The design of the abstract machine is based on the parallelism and concurrency capabilities of the PWAM/&-Prolog abstract machine [Her86, HG91] and recent work on extending its capabilities and efficiency [PGT95a, PGH95, PGT+95b]. The abstract machine includes native support for *attributed variables* [Hol92, Hui90, Neu90] which are used extensively in the implementation of constraint solvers (as in other systems such as Eclipse [Eur93] and SICStus 3 [Swe95]) and in supporting communication among concurrent tasks [HCC95]. While the current abstract machine supports only ("dependent" and "independent") and-parallelism, it is expected that combination with or-parallelism will be possible by applying the techniques developed in [GC92, GHPC94, GSCYH91].

3 The CIAO Compiler

The CIAO compiler provides the required support for the different programming paradigms and their optimization. As mentioned before, it is strongly based on program analysis and transformation. The compilation process can be viewed as a translation process from the input language to (kernel) CIAO.

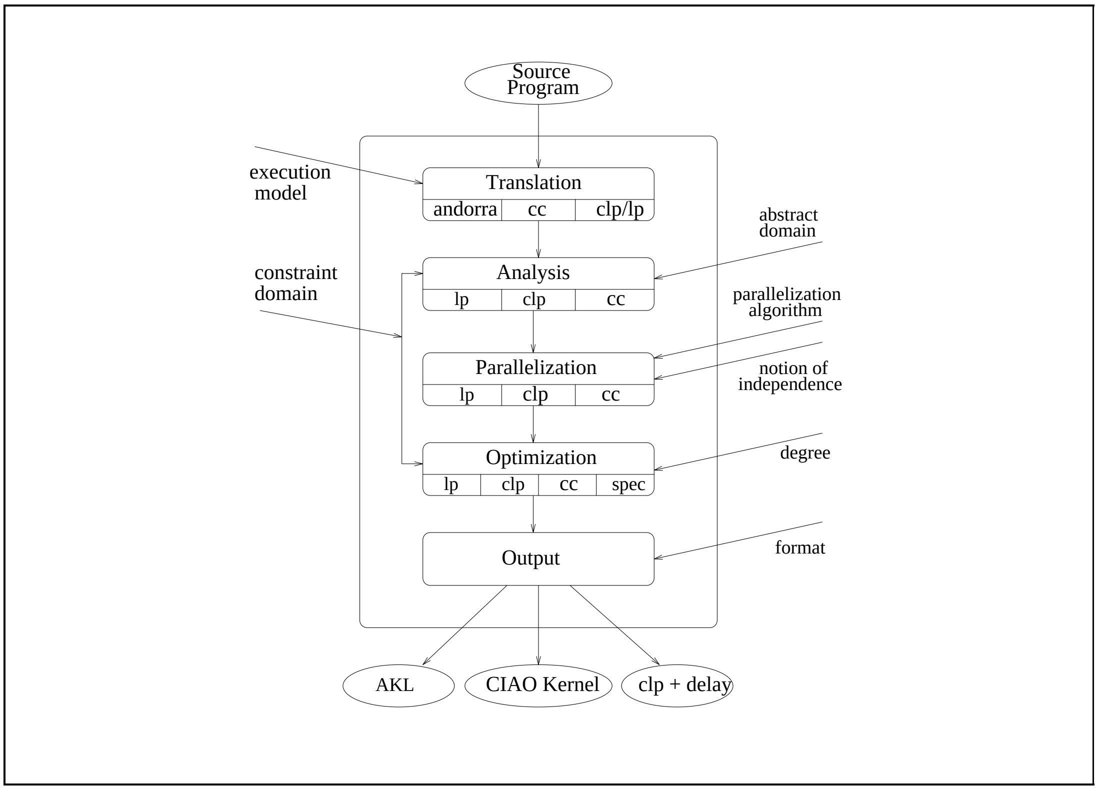

Figure 1: CIAO Compiler

The system is able to translate the input source, automatically extracting parallelism, compiling synchronization, and optimizing the final program. Optimizations include simplifying the code to avoid run-time tests and suspensions, and specializing predicates in order to generate much simpler and efficient code in the back end.

This compilation process is depicted in Figure 1, which illustrates the inputs and outputs, as well as the compilation options, which are selected via either menus or program flags. The compilation process is structured into several steps. First, a module in a given input language is translated into the kernel language. Then, analysis is performed if required to support the rest of the compilation process. In some cases some degree of analysis may also be performed in the translation step to aid in the translation. After analysis, the program is optionally annotated for parallel execution, simplified and specialized.

The output can then be loaded for execution on the abstract machine. As an alternative, and using the transformational approach, most of the capability of the system is also supported (with sometimes somewhat lower efficiency) on any Prolog with delay declarations and attributed variables (e.g., SICStus Prolog Version 3 [Swe95]). In that sense, the CIAO compiler can also be viewed as a library package for Prolog systems with these capabilities.

The compiler steps and options are discussed in the following sections. Given the space limitations the aim is to offer a general description and provide references for publications or technical reports where the techniques used are described. An extended description of the capabilities of the compiler can be found in the User's Manual [Bue95].

3.1 Source Languages Supported and Transformations Performed

The compiler can deal with several languages and computation rules simultaneously and perform several translations among them. Currently, there are three languages supported: the CIAO kernel language, languages based on the basic Andorra model, and basic CC languages. Also, for each of these languages several constraint domains can be chosen. Currently, the system supports those of Prolog, CLP(R), and CLP(Q).

The mode of the system can be changed by typing at the top level the commands `ciao(Domain)`, `andorra(Domain)`, or `cc(Domain)`, where the variable `Domain` has to be instantiated to either `h`, `q`, or `r`, indicating the desired constraint domain, i.e. Herbrand, Q, or R, respectively. Programs read from then on will be assumed to have the characteristics associated to the new mode:

- `ciao(Domain)`: CIAO full syntax (backwards compatible with Prolog, plus the specific CIAO primitives) language; left-to-right and (encapsulated) concurrency.
- `andorra(Domain)`: Prolog language; computation rule based on the basic Andorra principle.
- `cc(Domain)`: basic CC language; concurrent computation rule.

Alternatively, the mode can be directly included in the program. This is done in the module declaration, which has one additional argument available for the specification of the mode.

Program transformations bridge the semantic gaps between the different programming paradigms supported. The methods used for translating programs based on the (Basic) Andorra model to CIAO are described in [BDGH95]. The methods used for translating CC languages are an extension of those of [DGB94, Deb93] and are described in [BH95c].

3.2 Analysis

The CIAO compiler includes both local and global analysis of programs. Local analysis of program clauses is usually very simple but not accurate. Nonetheless, it is sometimes useful in some optimizations, as in program parallelization [BGH94]. Global analysis is performed in the context of abstract interpretation [CC77, Deb92, CC92]. The underlying framework of analysis is that of PLAI [HWD92, MH90, MH92]. PLAI implements a generic (goal-dependent and goal-independent) top-down driven abstract interpreter. The whole computation is domain–independent. This allows plugging in different abstract domains, provided suitable interfacing functions are defined. PLAI also incorporates incremental analysis [HMPS95] in order to deal with large programs and is capable of analyzing full languages (in particular, full standard Prolog [BCHP96, CRH94]).

A modification of the PLAI framework capable of analyzing dynamically scheduled programs is also provided in order to support the concurrent models. Note that, thanks to the transformational approach, only two frameworks are used (one for simple, left-to-right execution and another for the case when there are dynamically scheduled goals). The compiler automatically decides the framework being used. This decision is based both in the language being analyzed and the presence of dynamically scheduled goals in the program.

The CIAO analyzer incorporates the following domains, which are briefly explained below: *Sh*, *Sh+Fr*, *ASub*, *Sh+ASub*, and *Sh+Fr+ASub*, which are used in logic programming, and *Def*, *Fr*, *Fd*, which can be used either in logic or constraint logic programming, and *LSign*, which is more specific to constraint logic programming.[1] Analysis of dynamically scheduled languages can be carried out with the *Sh+Fr* and *Def* domains.

3.2.1 HERBRAND For the analysis of (classical) logic programs (over the Herbrand domain) the CIAO compiler includes a number of traditional domains proposed in the literature for capturing properties such as variable groundness, freeness, sharing, and linearity information. This includes the set sharing *Sh* [JL89, MH89], set sharing and freeness Sh+Fr [MH91], and pair sharing *ASub* [Son86] domains. Combinations of the *Sh* and *Sh+Fr* domains with *ASub* are also supported, resulting in the *Sh+ASub* and *Sh+Fr+ASub* domains. The combination is done in such a way that the original domains and operations of the analyzer over them are re-used, instead of redefining the domains for the combination [CC79, CMB$^+$95]. Two other domains, a modified version of *Path sharing* [KS95] and *Aeqns* (abstract equations) [MSJB95] are currently being incorporated to the system.

[1] Some of these domains have been implemented by other users of the PLAI system, notably the K. U. Leuven, Monash University, and the U. of Melbourne.

3.2.2 Constraint Programming The abstract domain *Def* [GH93, Gar94] determines whether program variables are definite, i.e. constrained to a unique value. In doing this it keeps track of *definite* dependencies among variables. The abstract domain *Fr* [DJBC93, DJ94, Dum94] determines which variables act as *degrees of freedom* with respect to the satisfiability of the constraint store in which they occur. In doing this it keeps track of *possible* dependencies among variables. The definite and possible dependencies are used to perform accurate definiteness and non-freeness propagation, respectively, and are also useful in their own right to perform several program optimizations. A combined domain *Fd* which infers both definiteness and freeness is also integrated.

A preliminary version of the domain *LSign* [MS94] is also supported. This domain is aimed at inferring accurate information about possible interaction between linear arithmetic equalities and inequalities. The key idea is to abstract the actual coefficients and constants in constraints by their "sign". A preliminary implementation of this domain shows very promising accuracy, although at a cost in efficiency.

3.2.3 Dynamically Scheduled Programs CIAO also includes a version of the PLAI framework which is capable of accurately analyzing (constraint) programs with dynamic scheduling (e.g., including delay declarations [eA82, Car87]). Being able to analyze constraint languages with dynamic scheduling also allows analyzing CC languages with angelic nondeterminism.[2] This is based on the observation that most implementations of the concurrent paradigm can be viewed as a computation which proceeds with a fixed, sequential scheduling rule but in which some goals suspend and their execution is postponed until some condition wakes them. Initial studies showed that accurate analysis in such programs is possible [MGH94], although this technique involves relatively large cost in analysis time. The analysis integrated into the CIAO compiler uses a novel method which improves on the previous one by increasing the efficiency without significant loss of accuracy [GMS95]. The approach is based on approximating the delayed atoms by a closure operator.

A direct method for analysis of CC programs has also been developed and is currently being integrated into the compiler. This method is an extension of previous work of Debray [DGB94, Deb93]. It is based on the observation that for certain properties, it is possible to extend existing analysis technology for the underlying fixed computation rule in order to deal with such programs [BH95b]. In particular, this idea has been applied using as starting point the original framework for the analysis of sequential programs. The resulting analysis can deal with programs where concurrency is governed by the Andorra model as well as standard CC models. The advantage with respect to the method above is lower analysis time, in exchange for a certain loss of accuracy.

3.3 Parallelization

The information inferred during the analysis phase is used for independence detection, which is the core of the parallelization process [BGH94, GBH95]. The compile-time parallelization module is currently aimed at uncovering goal-level, restricted (i.e., fork and join), independent and-parallelism (IAP). Independence has the very desirable properties of correct and efficient execution w.r.t. standard sequential execution of Prolog or CLP. In the context of LP, parallelization is performed based on the well-understood concepts of *strict* and *non-strict* independence [HR95], using the information provided by the abstract domains. While the notions of independence used in LP are not directly applicable to CLP, specific definitions for CLP (and constraint programming with dynamic scheduling) have been recently proposed [GHM93, Gar94] and they have been incorporated in the CIAO compiler in order to parallelize CLP and CC programs [GHM95]. Additionally, the compiler has side-effect and granularity analyzers (not depicted in Figure 1) which infer information which can yield the sequentialization of goals (even when they are independent) based on efficiency or maintenance of observable behavior.

The actual automatic parallelization of the source program is performed in CIAO during compilation of the program by the so called *annotation* algorithms. The algorithms currently implemented are: `mel`, `cdg`, `udg` [Mut91, Bue94], and `urlp` [CH94]. To our knowledge, the CIAO system is the first one to perform automatic compile-time (And-)parallelization of CLP programs [GBH95].

[2] This is a kind of nondeterminism which does not give rise to an arbitrary choice when applying a search rule.

3.4 Optimization

The CIAO compiler performs several forms of code optimization by means of source to source transformations. The information obtained during the analysis phase is not only useful in automatic program parallelization, but also in this program specialization and simplification phase.

The CIAO compiler can optimize programs to different degrees, as indicated by the user. It can just simplify the program, where simplification amounts to reducing literals and predicates which are known to always succeed, fail, or lead to error. This can speed up the program at run-time, and also be useful to detect errors at compile-time. It can also specialize the program using the versions generated during analysis [PH95a]. This may involve generating different versions of a predicate for different *abstract call patterns*, thus increasing the program size whenever this allows more optimizations. In order to keep the size of the specialized program as reduced as possible, the number of versions of each predicate is minimized attaining the same results as with Winsborough's algorithm [Win92].

As well as handling sequential code, the optimization module of the CIAO compiler contains what we believe is the first automatic optimizer for languages with dynamic scheduling [PH95b]. The potential benefits of the optimization of this type of programs were already shown in [MGH94], but they can now be obtained automatically. These kinds of optimizations include simplification and elimination of suspension conditions and elimination of concurrency primitives (sequentialization).

3.5 Output

The back end of the compiler takes the result of the previous program transformations and generates a number of final output formats. Normally, the result of the compiler is intended for the CIAO/&-Prolog abstract machine. Output possibilities are then byte-code ("`.ql`") files, stand-alone executables, and incore compilation (when the compiler is running inside the system rather than as a stand-alone application). As mentioned before, and as an alternative output, most of the capability of the system can also be handled by any Prolog which supports delay declarations and attributed variables. Alternatively, also AKL [JH91] can be used as a target, using the techniques described in [BH95a].

Finally, it is possible to obtain the results of each of the intermediate compilation phases. This allows visualizing and affecting the transformation, analysis, parallelization, and optimization processes. Because of the source to source nature of the compiler, this output is always a (possibly annotated) kernel CIAO program.

4 The Future: Work and Visions

We have briefly described the current status of the CIAO system. The current main objective of the system is to be an experimentation and evaluation vehicle for programming constructs and optimization and implementation techniques for the programming paradigms of LP, CLP, CC, and their combinations. Current versions of the system are often available for experimentation (please contact the authors; further information can be obtained from `http://www.clip.dia.fi.upm.es/`).

The system has already shown itself useful in illustrating the power (or lack thereof) of a number of analysis and optimization techniques (see the referenced papers for details). We are continuing to improve the system. Additionally, we are developing pilot applications with the system which should provide valuable feedback regarding its capabilities.

Much work remains to be done in several areas. We are planning on including support for programming in the functional style, both in terms of syntax (in order to allow nesting of calls via a designated output argument in relations) and of improved support for meta-programming (higher-order), both at the source language level and also at the implementation level. Again, functions will simply be compiled into the kernel language.

While the CIAO system illustrates that analysis and optimization of concurrent programs is possible, much work remains in improving the efficiency and accuracy of the analysis and in improving the performance gains obtained with the resulting optimizations.

As mentioned in Section 3.3, the automatic parallelization currently performed in the CIAO system is at the goal level. However, it is possible to parallelize at finer granularity levels, thus obtaining greater degrees of parallelism. The concept of *local independence* [MRB$^+$94, BHMR94] can be used for this purpose. Although some promising progress has been made in this direction [HCC95], it remains as future work to implement a system fully capable of efficiently exploiting this very fine grained level of parallelism.

Granularity control is a very important issue in both parallelization of sequential programs and sequentialization of concurrent ones. As mentioned in Section 3.3, the CIAO compiler already has some granularity control capabilities [DLH90, KS90, LHD94, DLHL94, LH95], but much work also remains to be done in this important area.

While our work in detection of parallelism in the CIAO compiler concentrates on compile-time detection of parallelism, run-time detection also needs to be explored. Significant progress has been made in this area by models and systems such as DDAS [She92], Andorra-I, and AKL.

Finally, there remains the issue of what is the ideal, future source language for LP/CLP/CC. Much promising work has been done in this direction in the design of languages such as ALF, AKL, CCP, CLP, CORAL, Escher, Goedel, LDL, LIFE, LambdaProlog, Lolli, Lygon, Mercury, NAIL, NuProlog, Oz, XSB, etc. In some ways, the kernel CIAO language also offers a (simplistic, but effective) solution to the problem, which is backwards compatible with Prolog and CLP. On the other hand, the overall CIAO design in some ways sidesteps this issue by attempting to support several languages (including those that combine several paradigms). This allows concentrating on the implementation issues and developing basic techniques for analysis and optimization that, in the belief that the underlying principles are quite common to the approaches being explored, will hopefully be applicable to future languages.

Regarding the underlying principles mentioned above, we propose to use the following two guidelines, which we have tried to follow in our design [HtCg94]: (1) separating computation rules on one hand, and optimizations of program execution, on the other, and (2) incorporating, in an orthogonal way, as many of both as possible in a single computational framework. Computation rules may include SLD resolution, best/breadth-first search, the determinate-first principle, etc. Optimizations may include parallel execution, reduction of synchronization, reordering of goals, code simplification, etc. Another crucial point to our approach is separating the issues concerning concepts such as computation, parallelism, concurrency, and optimization principles, from the granularity level at which such concepts are applied (e.g., a parallelization principle, such as independence or determinacy, can be applied at several levels, such as the goal level or the binding level).

In the belief that there is much in common at the abstract machine level among many of the LP, CLP, and CC models that we may be looking after integrating, we also argue that the support for multiple models and paradigms can be implemented via compilation into a simple kernel language, requiring a comparatively simple, generic abstract machine. Also, in the belief that many distributed applications are a good target for computational logic systems, we propose to include extensive distributed execution capabilities. We also argue that it is useful for the kernel language supporting all this machinery to use explicit control. Explicit control makes it possible to easily reason about the computational characteristics of the program and to perform many optimizations at the (kernel) source level. Compile-time analyses of the kernel language can then be implemented at the back-end. This, plus extensive compile-time optimization, may very probably make it feasible to obtain competitive performance for our languages.

Acknowledgments

The development of the CIAO system is a collaborative effort of several groups and includes work developed at the U. of Arizona (S. Debray's group), New Mexico State University (G. Gupta and E. Pontelli), K. U. Leuven (M. Bruynooghe's group), and Monash and Melbourne U. (M. García de la Banda, K. Marriott, and P. Stuckey), in addition to the CLIP (Computational Logic, Implementation, and Parallelism) group at the Technical University of Madrid, which includes Francisco Bueno, Daniel

Cabeza, Manuel Carro, M. García de la Banda, Pedro López García, Germán Puebla, and Manuel Hermenegildo. Parts of this work have been funded by ESPRIT projects PRINCE, PARFORCE, and ACCLAIM, and CICYT project IPL-D. The authors would like to thank the anonymous referees for their useful and encouraging comments. Given the space and time limitations a few of the more involved modifications could not be included in this version, but we hope to include them in future versions of the paper.

References

[BCHP96] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Data–flow Analysis of Standard Prolog Programs. In *European Symposium on Programming*, Sweden, April 1996.

[BDGH95] F. Bueno, S. Debray, M. García de la Banda, and M. Hermenegildo. Transformation-based Implementation and Optimization of Programs Exploiting the Basic Andorra Model. Technical Report CLIP11/95.0, Facultad de Informática, UPM, May 1995.

[BGH94] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, November 1994.

[BH95a] F. Bueno and M. Hermenegildo. An Automatic Translation Scheme from CLP to AKL. Technical Report CLIP7/95.0, Facultad de Informática, UPM, June 1995.

[BH95b] F. Bueno and M. Hermenegildo. Analysis of Concurrent Constraint Logic Programs with a Fixed Scheduling Rule. In *ICLP95 WS on Abstract Interpretation of Logic Languages*, Japan, June 1995.

[BH95c] F. Bueno and M. Hermenegildo. Compiling Concurrency into a Sequential Logic Language. Technical Report CLIP15/95.0, Facultad de Informática, UPM, June 1995.

[BHMR94] 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.

[Bue94] Francisco J. Bueno Carrillo. *Automatic Optimisation and Parallelisation of Logic Programs through Program Transformation*. PhD thesis, Universidad Politécnica de Madrid (UPM), October 1994.

[Bue95] F. Bueno. The CIAO Multiparadigm Compiler: A User's Manual. Technical Report CLIP8/95.0, Facultad de Informática, UPM, June 1995.

[Car87] 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.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CC79] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Sixth ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979.

[CC92] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2 and 3):103–179, July 1992.

[CH94] D. Cabeza and M. Hermenegildo. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In Springer-Verlag, editor, *1994 International Static Analysis Symposium*, number 864 in LNCS, pages 297–313, Namur, Belgium, September 1994.

[CH95] D. Cabeza and M. Hermenegildo. Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the 1995 COMPULOG-NET Workshop Parallelism and Implementation Technologies*, Utrecht, NL, September 1995. U. Utrecht / T.U. Madrid.

[CMB+95] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains. *ACM Transactions on Programming Languages and Systems*, pages 28–44, January 1995.

[CRH94] B. Le Charlier, S. Rossi, and P. Van Hentenryck. An Abstract Interpretation Framework Which Accurately Handles Prolog Search–Rule and the Cut. In *International Symposium on Logic Programming*, pages 157–171. MIT Press, November 1994.

[Deb92] S. Debray, editor. *Journal of Logic Programming, Special Issue: Abstract Interpretation*, volume 13(1–2). North-Holland, July 1992.

[Deb93] 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.

[DGB94] S. Debray, D. Gudeman, and P. Bigot. Detection and Optimization of Suspension-free Logic Programs. In *1994 International Symposium on Logic Programming*, pages 487–501. MIT Press, November 1994.

[DJ94] V. Dumortier and G. Janssens. Towards a practical full mode inference system for CLP(H,N). In Pascal Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming*, pages 569–583, Italy, June 1994. MIT Press.

[DJBC93] V. Dumortier, G. Janssens, M. Bruynooghe, and M. Codish. Freeness analysis in the presence of numerical constraints. In David S. Warren, editor, *Proceedings of the 10th International Conference on Logic Programming*, pages 100–115, Budapest, Hungary, June 1993. MIT Press.

[DLH90] 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.

[DLHL94] S.K. Debray, P. López García, M. Hermenegildo, and N.W. Lin. Estimating the Computational Cost of Logic Programs. In *1994 International Static Analysis Symposium*, pages 255–266, Namur, Belgium, September 1994.

[dMSC93] Vítor Manuel de Morais Santos Costa. *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I.* PhD thesis, University of Bristol, August 1993.

[Dum94] Veroniek Dumortier. *Freeness and Related Analyses of Constraint Logic Programs using Abstract Interpretation.* PhD thesis, K.U.Leuven, Dept. of Computer Science, October 1994.

[eA82] A. Colmerauer et Al. *Prolog II: Reference Manual and Theoretical Model.* Groupe D'intelligence Artificielle, Faculté Des Sciences De Luminy, Marseilles, 1982.

[Eur93] European Computer Research Center. *Eclipse User's Guide*, 1993.

[Gar94] María José García de la Banda García. *Independence, Global Analysis, and Parallelism in Dynamically Scheduled Constraint Logic Programming.* PhD thesis, Universidad Politécnica de Madrid (UPM), July 1994.

[GBH95] M. García de la Banda, F. Bueno, and M. Hermenegildo. Automatic Compile-Time Parallelization of CLP Programs by Analysis and Transformation to a Concurrent Constraint Language. Technical Report CLIP3/95.0, Facultad de Informática, UPM, June 1995. Also in ILPS'95 WS on Parallel Logic Programming Systems.

[GC92] G. Gupta and V. Santos Costa. And-Or Parallelism in Full Prolog based on Paged Binding Array. In *Parallel Architectures and Languages Europe '92*. Springer Verlag, June 1992. to appear.

[GH93] M. García de la Banda and M. Hermenegildo. A Practical Approach to the Global Analysis of CLP Programs. In Dale Miller, editor, *Proceedings of the 10th International Logic Programming Symposium*, pages 437–455, Vancouver, Canada, October 1993. MIT Press.

[GHM93] 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.

[GHM95] M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence and Search Space Preservation in Dynamically Scheduled Constraint Logic Languages. Technical Report CLIP10/95.0, Facultad de Informática, UPM, February 1995.

[GHPC94] G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *International Conference on Logic Programming*, pages 93–110. MIT Press, June 1994.

[GMS95] M. García de la Banda, K. Marriott, and P. Stuckey. Efficient Analysis of Constraint Logic Programs with Dynamic Scheduling. In *1995 International Logic Programming Symposium*, Portland, Oregon, December 1995. MIT Press, Cambridge, MA.

[GSCYH91] 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.

[HCC95] M. Hermenegildo, D. Cabeza, and M. Carro. Using Attributed Variables in the Implementation of Concurrent and Parallel Logic Programming Systems. In *Proc. of the Twelfth International Conference on Logic Programming*, pages 631–645. MIT Press, June 1995.

[Her86] M. V. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 25–40. Imperial College, Springer-Verlag, July 1986.

[HG91] M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[HMPS95] M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. Incremental Analysis of Logic Programs. In *International Conference on Logic Programming.* MIT Press, June 1995.

[Hol92] 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.

[HR95] 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*, 22(1):1–45, 1995.

[HtCg93] M. Hermenegildo and the CLIP group. Towards CIAO-Prolog – A Parallel Concurrent Constraint System. In *Proc. of the Compulog Net Area Workshop on Parallelism and Implementation Technologies.* Technical University of Madrid, June 1993.

[HtCg94] M. Hermenegildo and the CLIP group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel Concurrent Constraint System. In *Principles and Practice of Constraint Programming*, LNCS 874, pages 123–133. Springer-Verlag, May 1994.

[Hui90] S. Le Huitouze. 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.

[HWD92] M. Hermenegildo, R. Warren, and S. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, August 1992.

[JH91] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

[JL89] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming.* MIT Press, October 1989.

[KS90] A. King and P. Soper. Granularity analysis of concurrent logic programs. In *The Fifth International Symposium on Computer and Information Sciences*, Nevsehir, Cappadocia, Turkey, October (1990).

[KS95] A. King and P. Soper. Depth-k Sharing and Freeness. In *International Conference on Logic Programming.* MIT Press, June 1995.

[LH95] P. López and M. Hermenegildo. Efficient Term Size Computation for Granularity Control. In *Proc. of the Twelfth International Conference on Logic Programming.* The MIT Press, June 1995.

[LHD94] 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.

[MGH94] K. Marriott, M. García de la Banda, and M. Hermenegildo. Analyzing Logic Programs with Dynamic Scheduling. In *20th. Annual ACM Conf. on Principles of Programming Languages*, pages 240–254. ACM, January 1994.

[MH89] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.

[MH90] K. Muthukumar and M. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990.

[MH91] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347, July 1992.

[MRB+94] 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.

[MS94] K. Marriott and P. Stuckey. Approximating Interaction Between Linear Arithmetic Constraints. In *1994 International Symposium on Logic Programming*, pages 571–585. MIT Press, 1994.

[MSJB95] A. Mulkers, W. Simoens, G. Janssens, and M. Bruynooghe. On the Practicality of Abstract Equation Systems. In *International Conference on Logic Programming*. MIT Press, June 1995.

[Mut91] Kalyan Muthukumar. *Compile-time Algorithms for Efficient Parallel Implementation of Logic Programs*. PhD thesis, University of Texas at Austin, August 1991.

[Neu90] U. Neumerkel. Extensible Unification by Metastructures. In *Proceeding of the META'90 workshop*, 1990.

[PGH95] E. Pontelli, G. Gupta, and M. Hermenegildo. &ACE: A High-Performance Parallel Prolog System. In *International Parallel Processing Symposium*. IEEE Computer Society Technical Committee on Parallel Processing, IEEE Computer Society, April 1995.

[PGT95a] E. Pontelli, G. Gupta, and D. Tang. Determinacy Driven Optimizations of And-Parallel Prolog Implementations. In *Proc. of the Twelfth International Conference on Logic Programming*. MIT Press, June 1995.

[PGT+95b] E. Pontelli, G. Gupta, D. Tang, M. Hermenegildo, and M. Carro. Efficient Implementation of And–parallel Prolog Systems. *Computer Languages*, 1995. Accepted for publication in the special issue on *Parallel Logic Programming*.

[PH95a] G. Puebla and M. Hermenegildo. Implementation of Multiple Specialization in Logic Programs. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*. ACM, June 1995.

[PH95b] G. Puebla and M. Hermenegildo. Specialization and Optimization of Constraint Programs with Dynamic Scheduling. Technical Report CLIP12/95.0, Facultad de Informática, UPM, September 1995. Presented at the 1995 COMPULOG Meeting on Program Development.

[SCWY90] 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.

[She92] 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.

[Smo94] G. Smolka. The Definition of Kernel Oz. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), November 1994.

[Son86] H. Sondergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

[Swe95] Swedish Institute of Computer Science, P.O. Box 1263, S-16313 Spanga, Sweden. *Sicstus Prolog V3.0 User's Manual*, 1995.

[Win92] W. Winsborough. Multiple Specialization using Minimal-Function Graph Semantics. *Journal of Logic Programming*, 13(2 and 3):259–290, July 1992.