Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming

FRANCISCO BUENO
Universidad Politécnica de Madrid
MARÍA GARCÍA DE LA BANDA
Monash University
and
MANUEL HERMENEGILDO
Universidad Politécnica de Madrid

We report on a detailed study of the application and effectiveness of program analysis based on abstract interpretation to automatic program parallelization. We study the case of parallelizing logic programs using the notion of strict independence. We first propose and prove correct a methodology for the application in the parallelization task of the information inferred by abstract interpretation, using a parametric domain. The methodology is generic in the sense of allowing the use of different analysis domains. A number of well-known approximation domains are then studied and the transformation into the parametric domain defined. The transformation directly illustrates the relevance and applicability of each abstract domain for the application. Both local and global analyzers are then built using these domains and embedded in a complete parallelizing compiler. Then, the performance of the domains in this context is assessed through a number of experiments. A comparatively wide range of aspects is studied, from the resources needed by the analyzers in terms of time and memory to the actual benefits obtained from the information inferred. Such benefits are evaluated both in terms of the characteristics of the parallelized code and of the actual speedups obtained from it. The results show that data flow analysis plays an important role in achieving efficient parallelizations, and that the cost of such analysis can be reasonable even for quite sophisticated abstract domains. Furthermore, the results also offer significant insight into the characteristics of the domains, the demands of the application, and the trade-offs involved.

Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Automatic Programming—*automatic analysis of algorithms; program transformation*; D.1.3 [**Programming Techniques**]: Parallel Programming; D.1.6 [**Programming Techniques**]: Logic Programming; D.3.4 [**Programming Languages**]: Compilers; F.3.1 [**Logics and Meanings of Programs**]: Spec-

A preliminary version of this article was presented at the 1994 International Logic Programming Symposium, pages 320–336, November 1994, MIT Press.
This research was supported in part by CEC DGXIII ESPRIT Project "PARFORCE" and CICYT TIC projects IPL-D and ELLA. Francisco Bueno is supported in part by a grant from the Spanish Ministry of Education. María José García de la Banda is supported in part by a grant from the Australian Research Council.
Authors' addresses: F. Bueno and M. Hermenegildo, Universidad Politécnica de Madrid, Facultad de Informática, Madrid, Spain; email: {bueno; herme}@fi.upm.es; M. García de la Banda, Monash University, Computer Science Department, Melbourne, Australia; email: mbanda@cs.monash.edu.au.

ifying and Verifying and Reasoning about programs—*logics of programs*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*automatic analysis of algorithms; program transformation*

General Terms: Languages, Performance

Additional Key Words and Phrases: Abstract interpretation, automatic parallelization, data flow analysis, logic programming, parallelism

1. INTRODUCTION

Program analysis is the process of statically—i.e., at compile-time— inferring information about run-time properties of programs. Generally, the purpose of the process is to use this information to perform optimizations which improve some characteristics of the program or its execution. Abstract interpretation [Cousot and Cousot 1977] allows the systematic design of correct data flow analyses through formalization of the relationship between analysis and semantics. The idea is to view data flow analysis as a nonstandard semantics defined over an abstract domain in which the usual domain of values has been replaced by a domain of descriptions of values, and the operations are replaced by corresponding abstract operations defined on the new domain of descriptions.

This paper studies the application of abstract interpretation to the problem of automatic program parallelization, in the context of logic programs. We believe that logic programming offers a particularly interesting case study for automatic parallelization. On one hand, this programming paradigm poses significant challenges to the parallelization task, which relate closely to the more difficult challenges faced in imperative language parallelization [Bacon et al. 1994]. Such challenges include highly irregular computations and dynamic control flow (due to the symbolic nature of many of their applications), nontrivial notions of (semantic) independence, the presence of dynamically allocated, complex data structures containing pointers (logical variables), and having to deal with speculation. On the other hand, this paradigm also facilitates the study of parallelization issues. Logical variables are actually a quite "well-behaved" version of pointers, in the sense that no castings or pointer arithmetic (other than array indexing) is allowed. Thus, pointers in these languages are not unlike those allowed, for example, in "clean" versions of C. In addition, similarly to functional languages, logic languages allow coding the desired algorithm in a way that reflects more directly the structure of the problem. This makes the parallelism available in the problem more accessible to the compiler. The relatively clean semantics of logic programming languages also makes it comparatively easy to use formal methods, such as the abstract interpretation approach subject of this paper, and prove the transformations performed by the parallelizing compiler both correct and efficient.[1] This paper proposes and proves correct a methodology for the application of the results of abstract interpretation-based data flow analysis in automatic parallelization of logic programs (using the "independent and-parallelism" model and *strict* independence—see, for example, [Conery 1983; DeGroot 1984; Hermenegildo and Rossi 1995] and their references). It also

[1]See [Hermenegildo 1997] for a more extended discussion of the relationships between parallelization techniques used for logic programs and those used for other programming paradigms.

reports on the implementation of the methodology and a number of analyzers, their integration in a compiler, and the study of their effectiveness for the application considered.

Much work has been done using the abstract interpretation technique in the context of logic programs (e.g., [Mellish 1986; Bruynooghe and Janssens 1988; Debray 1989; Marriott and Søndergaard 1989; Bruynooghe 1991; Debray 1992] and their references). However, only a few studies have been reported which examine the performance of analyzers in the actual optimization task they were designed for (notable exceptions are [Warren et al. 1988; Marien et al. 1989; Van Roy and Despain 1990; Taylor 1990; Santos Costa 1993; Getzinger 1993]). This article also contributes to fill this gap.

Data flow analysis seems to be crucial in the context of automatic parallelization of programs within the independent and-parallel model. Unfortunately, little work has been reported on the complete task of global analysis-based compile-time automatic parallelization of logic programs within that model. In a previous study [Warren et al. 1988; Hermenegildo et al. 1992], we have reported on a first set of experiments in abstract interpretation-based program parallelization. This study was interesting in that, to the best of our knowledge, it represented the first actual application of abstract interpretation reported within logic programming. However, being essentially a feasibility study for the abstract interpretation technique, that work had several shortcomings: it included only one domain (a simple depth-K/sharing domain); it used a relatively simple basic parallelizer and analyzer; it presented the results only in terms of program simplifications; and the correctness of the approach was never shown. Furthermore, it was not explained how and when the information inferred by the analyzers was used. Since then, several new parallelization algorithms [Muthukumar and Hermenegildo 1990b] and sophisticated abstract analyzers (i.e., domains and the associated abstract functions) relevant to the application have been proposed [Sondergaard 1986; Jacobs and Langen 1989; Codish et al. 1991; Muthukumar and Hermenegildo 1991; 1992; Codish et al. 1995]. Furthermore, a complete parallel platform [Hermenegildo and Greene 1991], a set of performance evaluation tools [Fernández et al. 1996], and a second-generation analysis framework [Muthukumar and Hermenegildo 1990a; 1992] have become available to us for experimentation.

We report on a detailed study of the application and effectiveness of program analysis using abstract interpretation in the parallelization of logic programs. We first propose and prove correct a novel methodology for the application of the inferred information to the parallelization task, via a parametric domain. A number of well-known approximation domains are then selected and the transformation into the parametric domain defined. The transformation directly illustrates the relevance and applicability of each abstract domain for the application. Both local and global analyzers are then built using these domains and embedded in a complete parallelizing compiler. Then, the performance of the domains in this context is assessed through a number of experiments. A comparatively wide range of aspects is studied, from the resources needed by the analyzers in terms of time and memory to the actual benefits obtained from the information inferred. Such benefits of analysis are evaluated not only in terms of accuracy, i.e., the ability to determine the actual dependencies among the program variables, but also of effectiveness, measured in terms of code reduction and also in terms of the ultimate performance of the parallelized programs, i.e., the actual speedup obtainable with respect to the sequential version.

The analyzers we use were defined to track classical properties of logic program variables: groundness, freeness, linearity, and sharing, which are interesting in several optimizations, including parallelization. A program variable is said to be ground if it is bound (at run-time) to a term which has no variables—a ground term. It is said to be free if it is bound to a variable term (or unbound). It is linear if it is bound to a term in which any variable occurs only once—a linear term. Finally, two (or a set of) program variables are said to share if they are bound to terms which have variables in common. While groundness, freeness, and sharing are useful for detecting independence, linearity improves the accuracy of sharing and, therefore, the propagation of groundness and freeness.

Five relatively sophisticated abstract domains are used in our evaluation: the ASub domain defined by Söndergaard [1986] for inferring groundness, (pair) sharing and linearity information, the Sharing domain defined by Jacobs and Langen [1992] for inferring groundness and (set) sharing information, the Sharing+Freeness domain defined by Muthukumar and Hermenegildo [1991] for inferring groundness, (set) sharing and freeness information, and the domains resulting from the combination of the ASub and Sharing domains, and also ASub and Sharing+Freeness domains, as presented by Codish et al. [1995]. The Sharing and Sharing+Freeness domains were defined specifically for the parallelization application. One of the main objectives of ASub was to accurately infer variable aliasing information for garbage collection optimization, which is also useful in parallelization. Finally, the combined domains were defined in order to further improve the behavior of their components.

We would like to point out that our main aim is not to perform a comparison among different global analyzers, but rather to devise a flexible methodology for applying "state-of-the-art" abstract analyzers in the automatic parallelization of logic programs, and evaluating the results. The use of several global analyzers based on different abstract domains with different levels of accuracy and complexity is motivated by the fact that this allows us to also study the relationship between accuracy, efficiency, and usefulness.

The rest of the article is structured as follows. Section 2 summarizes the application in hand: and-parallelization of logic programs. Section 3 describes the structure and task of the "annotators"—the actual parallelizers which interface with the analyzers—and proposes the actual interface itself. Section 4 proves the correctness of the general framework described in the previous section. Section 5 then presents the different domains, and proposes a method for casting the information encoded by each domain in terms of the defined interface for the parallelization process. Section 6 introduces global and local analyzers based on these domains, describing the framework they are embedded in. Section 7 briefly describes the evaluation environment—i.e., the parallelizing compiler and the evaluation tools used for performing some of the experiments, and Section 8 describes the experiments and presents the results obtained from those experiments. Finally, Section 9 presents our conclusions and suggestions for future work.

2. THE APPLICATION: AUTOMATIC AND-PARALLELIZATION

The two main types of parallelism which can be exploited in logic programs are well known [Conery 1983; Chassin and Codognet 1994]: or-parallelism and and-parallelism. Or-parallelism refers to the parallel exploration of branches in the derivation tree corresponding to different clauses which match a given literal. This kind of parallelism is specially useful when solving nondeterministic problems, i.e., problems whose solution involves a large amount of search. And-parallelism refers to the parallel execution of (sequences of) literals (referred to as goals) in the body of a clause. And-parallelism may appear in both nondeterministic and deterministic programs, and it corresponds directly to the more "conventional" view of parallelism present in other languages, as solving simultaneously *independent* parts of the problem. Several models have been proposed to take advantage of such opportunities (e.g., [DeGroot 1984; Hermenegildo 1986a; Westphal and Robert 1987; Warren 1987b; Biswas et al. 1988; Lin 1988; Ramkumar and Kale 1989; Gupta and Jayaraman 1989; Szeredi 1989; Ali and Karlsson 1990] and their references).

Given the high potential for parallelism exhibited by logic programs, it is tempting to directly make use of all available parallelism by maximizing the number of tasks scheduled for parallel execution. However, this can in some cases result in a very significant slowdown, which is clearly in conflict with the final aim of parallelism: to achieve the maximum speed (effectiveness) while computing the same solution (correctness). In practice, "maximum speed" may be difficult to achieve or even define. Therefore, other concepts, such as ensuring "no slowdown" (guaranteeing that the parallel execution will be no slower than the sequential one) are often used instead [Hermenegildo and Rossi 1995].

The objective can then be seen as determining which of all the opportunities for parallelism available are not only correct but also profitable from the efficiency point of view. It turns out that when or-parallelism is considered, and if all solutions to the problem are desired, both correctness and efficiency can be guaranteed.[2] This directly results from the inherent independence among the different branches which are executed in parallel [Hermenegildo and Rossi 1995]. This simplicity has contributed to the rapid development of parallel frameworks based on the or-parallel models (e.g., [Warren 1987a; Lusk et al. 1988; Szeredi 1989; Lusk et al. 1990; Ali and Karlsson 1990] and their references). Furthermore, such models can be easily extended to constraint logic programming (CLP), as shown by the implementations of Van Hentenryck [1989] and McAloon and Tretkoff [1989]. The associated performance studies have shown good performance for nondeterministic programs in a number of practical implementations [Szeredi 1989; Ali and Karlsson 1990].

On the other hand, guaranteeing correctness and efficiency in and-parallelism is less straightforward. The main problems are due to the fact that dependencies may exist among the goals to be executed in parallel, because of the presence of shared variables at run-time. It turns out that when these dependencies are present, exploitation of and-parallelism might not guarantee efficiency. Furthermore, if certain "impure" predicates that are relatively common in Prolog programs are used, even

[2]Care also has to be taken in the case of programs containing impure literals like cuts and side-effects.

correctness cannot be guaranteed.

Assuming such a *dependency* among the set of goals to be executed in parallel is detected, different alternatives have been proposed to solve the problem, resulting in two main models. The *stream* and-parallelism model [Clark and Gregory 1986; Ueda 1987; Taylor et al. 1987; Shapiro 1989; Saraswat 1989], runs dependent goals in parallel. The variables that are shared by the parallel goals are used as communication channels, and partial bindings of these variables are *incrementally* passed as streams from the producer to the consumers. The main drawbacks of this model are the overhead introduced by fine-grained variable-based synchronization and the problems appearing in the implementation of backtracking in the presence of nondeterminism. Therefore, many systems which exploit this type of parallelism give up the built-in search capabilities (*"don't know" nondeterminism*) present in logic programming.

The *independent* and-parallelism model sequentializes dependent goals [Conery 1983]. Thus, only goals which are independent are allowed to execute in parallel, the rest being executed sequentially, preserving the order assumed by the sequential model. The main drawback associated with this model is that a certain amount of existing parallelism might be lost if independent goals cannot be detected accurately. However, as shown in [Hermenegildo and Rossi 1995], such "independent and-parallelism" has the advantages of fulfilling both the correctness and the efficiency requirements and being amenable to an efficient implementation if combined with compile-time analysis. Furthermore, it has been argued [Hermenegildo and The CLIP Group 1994; Bueno Carrillo 1994; Bueno et al. 1998] that all forms of and-parallelism in logic programs can be seen as independent and-parallelism, provided the definition of independence is applied at the appropriate granularity level. For example, stream and-parallelism can be seen as independent and-parallelism if the independence of "bindings" rather than goals is considered.

At this point it should be apparent that, in and-parallelism, automatic parallelization is related closely to the detection of some notion of independence. Unfortunately, detecting the dependencies among the different goals implies in general a parallelization overhead. It is vital that such overhead remains reasonable. Several models have been proposed in the literature, mainly differing in whether the dependencies are detected exclusively at run-time [Conery 1983], exclusively at compile-time [Chang et al. 1985], or by marking at compile-time selected literals and, when independence cannot be determined statically, generating a reduced set of efficient parallelization tests, to be checked at run-time [DeGroot 1984; Hermenegildo 1986a; Lin and Kumar 1988; Hermenegildo and Rossi 1995].

The last approach reduces the independence checking overhead with respect to fully dynamic models and achieves more parallelism than fully static models. However, the conditions generated if the approaches above are used as proposed are still often too costly to compute at run-time. This is mainly due to the lack of adequate analysis and optimization technology: simplification of expressions is done in an "ad hoc" way; global data flow analysis is scarcely used; and thus usually only local analyzers are considered. A more effective approach, proposed initially by R. Warren et al. [Warren et al. 1988; Hermenegildo et al. 1992] and developed further in [Muthukumar and Hermenegildo 1990b], and [Cabeza and Hermenegildo 1994], is to combine local analysis and run-time checking with global data flow analysis

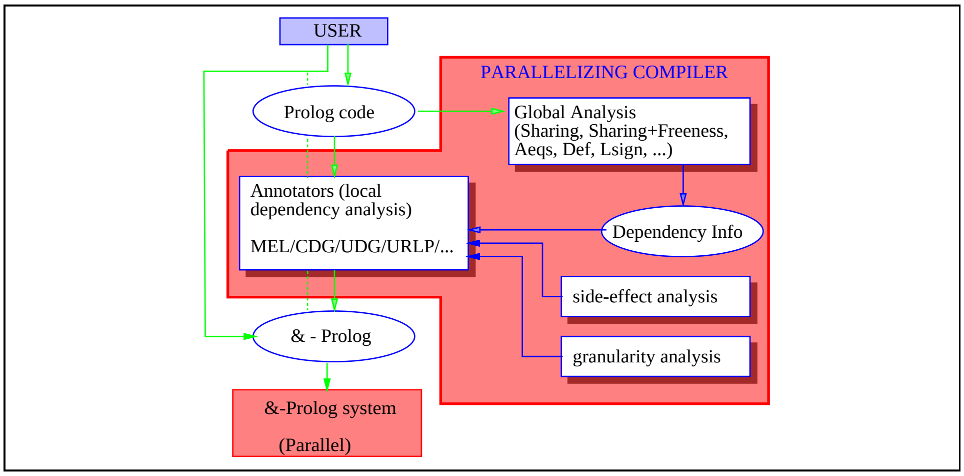
USER
PARALLELIZING COMPILER
Prolog code
Global Analysis (Sharing, Sharing+Freeness, Aeqs, Def, Lsign, ...)
Annotators (local dependency analysis)
MEL/CDG/UDG/URLP/...
Dependency Info
side-effect analysis
granularity analysis
& - Prolog
&-Prolog system (Parallel)

Fig. 1. Parallelization process.

based on the technique of abstract interpretation [Cousot and Cousot 1977]. This is the approach that we will study.

3. THE ANNOTATION PROCESS

In our approach (see also Section 7), the automatic parallelization process is performed as follows (see Figure 1). First, if required by the user, the Prolog program is analyzed using one or more global analyzers, whose purpose is to infer information that is relevant for identifying independence. Second, since side-effects in general cannot be allowed to execute freely in parallel, the original program is analyzed using a simple global analyzer which propagates the side-effect characteristics of built-ins determining the scope of each side-effect, such as the one described in [Muthukumar and Hermenegildo 1989a].[3] Finally, the *annotators* perform a source-to-source transformation of the program in which each clause is annotated with parallel expressions where conditions which encode the notion of independence are used. This source-to-source transformation is referred to as the *annotation process*. This process uses the information provided by the global analyzers mentioned before. Additionally, while annotating each clause, the annotators can also invoke local analyzers in order to infer further information regarding the literals in the clause.

The annotation process is divided into three subtasks (see Figure 2). The first one is concerned with identifying the dependencies between each two literals in a clause and generating the conditions (*iconds*) which ensure their independence. The second task aims at simplifying such conditions by means of the information inferred by the local or global analyzers. In other words, transforming the conditions into the minimum number of tests which, when evaluated at run-time, ensure the independence of the goals involved. Finally, the third task is concerned with the core of the annotation process, namely the application of a particular strategy to obtain

[3]See also [Chang and Chiang 1989; DeGroot 1987; Gupta and Costa 1992] for other methods of handling side-effects in the context of logic program parallelization.

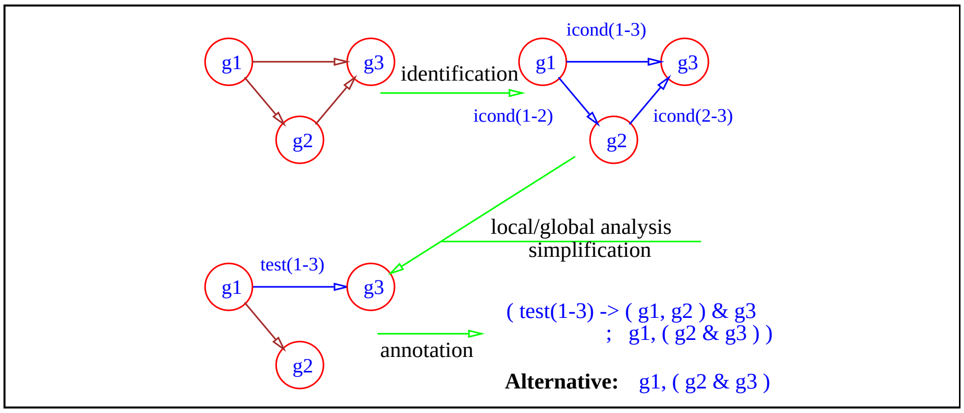

Fig. 2. Annotation process.

an optimal (under such a strategy) parallel expression among all the possibilities detected in the previous step, hopefully further optimizing the number of tests. If a parallel language is considered, as will be our case, it is possible to view the parallelization process as a source-to-source transformation. Such a transformation is called an *annotation*.

In the following we will briefly explain those steps in more detail, for the particular context of strict independence. Note that while the first and third steps have been already studied in [Muthukumar and Hermenegildo 1990b; Muthukumar et al. 1999] in terms of the parameterized notion of independence, the second, and most important step from the point of view of practicality, has never been formally described, and it has been usually performed in an "ad hoc" way. The work in [Muthukumar and Hermenegildo 1990b; Muthukumar et al. 1999] studied different heuristics for flattening dependency graphs into expressions suitable for fork/join parallelism[4] but did not describe the test simplification process or the impact of abstract interpretation, which is our focus here.

3.1 Identifying Dependencies

As mentioned before, the first step in the annotation process aims at identifying the dependencies between each two literals in a clause and generating the conditions which ensure their independence. The dependencies between literals can be represented as a dependency graph [Conery 1983; Chang et al. 1985; Kalé 1987; Jacobs and Langen 1988; Lin 1988; Sarkar 1989; 1990; Muthukumar and Hermenegildo 1990b]. Informally, a dependency graph is a directed acyclic graph where each node represents a literal, and each edge represents in some way the dependency between the connected literals. A conditional dependency graph (CDG) is one in which the edges are labeled with independence conditions. If those conditions are satisfied, the dependency does not hold. In an unconditional dependency graph (UDG) dependencies always hold, i.e., conditions are always "false."

[4]In this kind of parallelism, also referred to as "Restricted And-Parallelism," once the execution of the selected parallel goals is initiated (fork), the execution of the rest of the goals is delayed until all parallel goals are finished (join). Then, the execution proceeds as usual.

Strict independence is arguably the most commonly used notion in independent and-parallelism. The importance of strict independence lies in the fact that it allows a priori detection of independence, i.e., it is decidable at run-time *before* executing the goals. Thus, it can be used even when no information is provided by global analysis. Furthermore, it can be translated into simple tests which can be evaluated at run-time more efficiently than the tests obtained by more general independence notions. For this reason the edges of the CDGs obtained are directly labeled with such tests. In this section we will briefly present how to derive them. Many of these concepts are already well understood and are present in different ways in [Conery 1983; DeGroot 1984; Chang et al. 1985; Hermenegildo 1986a; Kalé 1987; Lin 1988; Jacobs and Langen 1988] and others. Here we will mainly follow the presentation of [Hermenegildo and Rossi 1995], which offered the first formal results regarding correctness and efficiency of the parallelization.

Two goals g_1 and g_2 are said to be strictly independent for a given substitution θ iff $\mathrm{vars}(g_1\theta) \cap \mathrm{vars}(g_2\theta) = \emptyset$, where $vars$ returns the set of variables occurring in a syntactic object.[5] A collection of goals is said to be strictly independent for a given θ iff they are pairwise strictly independent for θ. Also, a collection of goals is said to be strictly independent for a set of substitutions Θ iff they are strictly independent for every $\theta \in \Theta$. Finally, a collection of goals is said to be simply strictly independent if they are strictly independent for the set of all possible substitutions. This same definition can also be applied to terms and substitutions without any change.

Example 3.1. Let us consider the two literals $p(x)$ and $q(y)$. Given $\theta = \{x/y\}$, we have $p(x)\theta = p(y)$ and $q(y)\theta = q(y)$, so $p(x)$ and $q(y)$ are not strictly independent for this substitution. However, given $\theta = \{x/w, y/v\}$, we have $p(x)\theta = p(w)$ and $q(y)\theta = q(v)$, so $p(x)$ and $q(y)$ are strictly independent for the given θ because $p(w)$ and $q(v)$ do not share any variable.

Note that if an object (term, goal, substitution, etc) is ground, then it is strictly independent of any other object. Also, note that strict independence is symmetric, but not transitive.

Given a collection of goals, we would then like to be able to generate at compile-time a condition *i_cond* which, when evaluated at run-time, would guarantee their strict independence. Furthermore, we would like that condition to be as efficient as possible, hopefully being more economical than the straightforward application of the definition. Consider the set of conditions which includes $true$, $false$, or any set, interpreted as a conjunction, of one or more of the tests $ground(x)$, and $indep(x, y)$, where x and y can be goals, variables, or terms in general. Let $ground(x)$ be true when x is ground and false otherwise. Let $indep(x, y)$ be true when x and y do not share variables and false otherwise.

Consider the goals $g_1, \ldots, g_n$. If no global information is provided, an example of such a correct *i_cond* is $\{ground(x) \mid x \in SVG\} \cup \{indep(x, y) \mid (x, y) \in SVI\}$, where SVG and SVI are defined as follows:

[5]From a more implementation-oriented point of view, this corresponds to the following intuition: two procedure calls or statements g_1 and g_2 are strictly independent if the data structures that g_1 has access to do not contain any pointers to the data structures that g_2 has access to, and vice-versa (see [Hermenegildo 1997] for a more extended discussion of this issue).

Table I. Example Parallel Goals and Associated Conditions

Goals	SVG	SVI	i_cond
`a(w), b(x,y)`	$\emptyset$	$\{(\mathtt{w},\mathtt{x}),(\mathtt{w},\mathtt{y})\}$	$\{indep(\mathtt{w},\mathtt{x}), indep(\mathtt{w},\mathtt{y})\}$
`a(w), c(z,y)`	$\emptyset$	$\{(\mathtt{w},\mathtt{y}),(\mathtt{w},\mathtt{z})\}$	$\{indep(\mathtt{w},\mathtt{y}), indep(\mathtt{w},\mathtt{z})\})$
`b(x,y), c(z,y)`	$\{\mathtt{y}\}$	$\{(\mathtt{x},\mathtt{z})\}$	$\{ground(\mathtt{y}), indep(\mathtt{x},\mathtt{z})\}$
`a(w), b(x,y), c(z,y)`	$\{\mathtt{y}\}$	$\{(\mathtt{w},\mathtt{x}),(\mathtt{w},\mathtt{z}),(\mathtt{x},\mathtt{z})\}$	$\{ground(\mathtt{y}), indep(\mathtt{w},\mathtt{x}),$ $indep(\mathtt{w},\mathtt{z}), indep(\mathtt{x},\mathtt{z})\}$

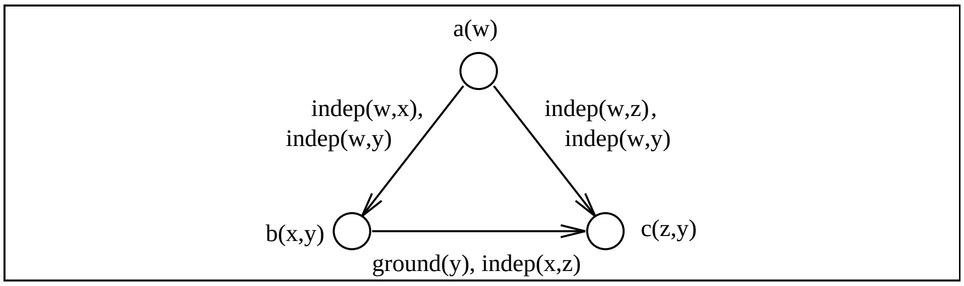

Fig. 3. Conditional dependency graph.

—$SVG = \{v \mid \exists i, j(i \neq j, v \in vars(g_i) \cap vars(g_j))\}$;

—$SVI = \{(v, w) \mid v, w \notin SVG, \exists i, j(i < j, v \in vars(g_i), w \in vars(g_j))\}$.

If the above condition is satisfied for substitution θ, then the goals are strictly independent for θ.

Example 3.2. Consider the following sequence of literals in a program clause: `a(w), b(x,y), c(z,y)`. Table I lists all possible goals that can be considered for parallel execution, their associated SVG and SVI sets, and a correct local *i_cond* with respect to strict independence.

The left-to-right precedence relation for these literals can be represented using a directed, acyclic graph in which we associate with each edge which connects a pair of literals the tests for their strict independence, thus resulting in the conditional dependency graph illustrated in Figure 3. This illustrates in a concrete example the first step of Figure 2.

It is easy to see that, in general, a groundness check is less expensive than an independence check. Thus, a condition, such as the one given, where some independence checks are replaced by groundness checks is obviously preferable.

Note that, for efficiency reasons, we can improve the conditions further by grouping pairs in SVI which share a variable x, such as $(x, y_1), \ldots, (x, y_n)$, by writing only one pair of the form $(x, [y_1, \ldots, y_n])$. By pursuing this idea further SVI can be defined in a more compact way as a set of pairs of sets: $SVI = \{(V, W) \mid \exists i, j(i < j, V = vars(g_i) \setminus SVG, W = vars(g_j) \setminus SVG)\}$. In many implementations this "compacted" set of pairs is less expensive to check than that generated by the previous definition of SVI. However, in our experiments, and for simplicity, when counting the number of independence checks generated statically we will use the previous definitions of SVI.

3.2 Simplifying Dependencies

The tests generated in the process described above imply the strict independence of the goals for any substitution in which the tests are satisfied. However, when considering the literals involved as part of a clause and within a program, the tests can be simplified, since then strict independence only needs to be ensured for those substitutions that can appear in that clause in the execution of the program. For example, if the groundness of a variable x is known to be satisfied in all substitutions appearing at some program point, then the test $\mathit{ground}(x)$ is known to succeed at run-time and can be eliminated from the set of run-time tests. Information about the set of substitutions can often be determined (or, at least, safely approximated) through local or global analysis. The second step in the annotation process aims at performing such simplification by identifying tests which are ensured to either fail or succeed with respect to the analysis information: if a test is guaranteed to succeed, it can be reduced to true, thus eliminating the edge (i.e., the dependency); if a test is guaranteed to fail, it can be reduced to false, yielding an unconditional edge. Note that, once the set of tests has been simplified, its satisfaction for a substitution θ will ensure the strict independence of the goals only if the substitution belongs to the set of substitutions that can appear at that program point.

Although this step is critical for the effectiveness of the approach, it has never been studied in detail: while in the case of [Hermenegildo and Rossi 1995] the simplification is explained by means of a simple example, [Warren et al. 1988; Hermenegildo et al. 1992] just mention that it is based on the groundness and independence mode information provided by the analyzer. As a result, and in the particular case of strict independence, it has been traditionally assumed that the simplification step just implies the following:

(1) translating the compile-time information into the independence and (possibly negated) ground facts which are known to hold with respect to such information,

(2) eliminating from the condition any ground test which explicitly appears in the translated information,

(3) reducing to false the condition whenever a ground test appears explicitly negated in the translated information, and

(4) eliminating from the condition any independence test whenever either this independence test or the groundness of at least one of the variables explicitly appears in the translated information.

This is, for example, the simplification procedure applied in [Warren et al. 1988; Hermenegildo et al. 1992]. However, we argue that this simple procedure can yield a loss of accuracy whenever the compile-time information is able to approximate more complex relationships among the groundness and independence characteristics of the program variables. Let us illustrate this by means of an example.

Example 3.3. Consider the sequence of literals in Example 3.2. Assume that our compile-time information is able to approximate that $\mathit{indep}(w,x) \rightarrow \mathit{indep}(w,y)$, but it cannot ensure whether $\mathit{indep}(w,x)$ and $\mathit{indep}(w,y)$ definitely hold or not. If the simplification follows the process described above, the condition $\{\mathit{indep}(w,x),$

indep$(w, y)\}$ labeling the edge from `a(w)` to `b(x,y)` cannot be simplified. However, it is clear that only *indep*(w, x) need be evaluated at run-time.

In this section we will formally define a more powerful procedure for the simplification process. This general procedure is based on a richer domain (referred to bellow as the *domain of interpretation*) to which the compile-time information is translated, and a function (called *improve*) which exploits the relationships approximated by the domain.

Let W be a set of variables and $ST(W)$ the set of tests over W that an independence condition can consist of. For any clause C, the information known at a program point i in C can be expressed in what we call a *domain of interpretation*: a subset of first-order logic,[6] such that each element κ of the domain defined over the variables in C is a set of formulae (interpreted as their conjunction) containing only tests of $ST(vars(C))$ and such that $\kappa \not\vdash false$. Additionally, some axioms may be present in the domain, which state relationships among the elements of $ST(W)$.

Example 3.4. In the case of strict independence, we have $ST(W) = \{ground(x) \mid x \in W\} \cup \{indep(x, y) \mid \{x, y\} \subseteq W\}$. The domain of interpretation for each clause with variables W will be denoted by GI ("groundness and independence") and will contain sets of formulae made from tests in $ST(W)$ with the classical connectives of logic. Additionally, there will be axioms which are assumed to be part of every $\kappa \in GI$:

$$\kappa \supseteq \{ground(x) \rightarrow indep(x, y) | \{x, y\} \subseteq W\} \cup \{ground(x) \leftrightarrow indep(x, x) | x \in W\}.$$

For the sake of simplicity, in the rest of the article this formula will be assumed to be part of any $\kappa \in GI$, although not explicitly written down. Thus, any time we write $\kappa = K$, κ should be interpreted as K augmented with the above set.

For any program point i of a clause C where a set of tests T_i on the independence of the clause variables is checked, the simplification of such test, based on an element κ_i of the domain of interpretation over the variables of C, is defined as the refinement of T_i to yield $T_i' = improve(T_i, \kappa_i)$, where

$$improve(T_i, \kappa_i) = \begin{cases} if & \exists t \in T_i \ s.t.\ \kappa_i \vdash \neg t \quad then \quad false \\ elseif & \kappa_i \vdash T_i \qquad\qquad\quad then \quad true \\ else & for\ some\ t \in T_i \quad \{t\} \cup improve(T_i \setminus \{t\}, \kappa_i \cup \{t\}). \end{cases}$$

Note that there is an implicit choice on the selection of $t \in T_i$ in the above definition. This choice can influence the result of *improve*. Consider GI, $\kappa_i = \{ground(x) \rightarrow ground(y)\}$, and $T_i = \{ground(x), ground(y)\}$. By first selecting $ground(y)$ the final result is $T_i' = \{ground(x), ground(y)\} = T_i$, whereas by selecting $ground(x)$ first the final result is $T_i' = \{ground(x)\} \subset T_i$, which is simpler. We will avoid such nondeterministic behavior by selecting first the tests which do not appear in a consequent in any atomic formula of κ_i, then those with lower cost at run-time (groundness in the case of GI), and then the rest. This will be done following a left-to-right selection rule.

[6]Though the domain is here defined over a first-order language, its "variables" W, which are the program clause variables, can be regarded as constants. Thus, the "first-order" formulation is mere syntactic sugar for a truly propositional language.

Table II. Example Simplified Conditions

κ	T	$improve(T,\kappa)$
$ground(x)$	$ground(x)$	$true$
$\neg ground(x)$	$ground(x)$	$false$
$ground(x) \rightarrow ground(y)$	$ground(x), ground(y)$	$ground(x)$
$ground(x) \rightarrow ground(y)$	$ground(y)$	$ground(y)$
$indep(x,y) \rightarrow ground(x)$	$indep(x,y), ground(x)$	$indep(x,y)$
$indep(x,y) \rightarrow \neg ground(x)$	$indep(x,y), ground(x)$	$false$

Example 3.5. For the following κ and set of tests T defined over the same sets of variables the simplified set resulting from *improve* is shown in Table II.

The accuracy and the size (the number of atomic formulae for simple facts) of each κ depend on the kind of program analysis performed. In the next section we will explain how to build this formula in the particular case of GI from the domains of analysis used in our experiments.

The simplest kind of information which can be derived is that obtained by a local analysis performed over each clause in isolation. This can be done based on knowledge about the semantics of the built-ins and the free nature of the first occurrences of variables. In the case of built-ins, their semantics implies certain knowledge about substitutions occurring at the points just before and after their execution. For first occurrences, it can always be ensured that the variable concerned is not ground and does not share with others, up to the point where it first appears.

Example 3.6. The information derived by the above-mentioned sources can be directly expressed in terms of elements of the GI domain.[7] The analysis of clause C starts with Fv_1, the set of variables not occurring in $head(C)$, and the formulae for first occurrences of variables, thus

$$\kappa_1 = \{\neg ground(x) \mid x \in Fv_1\} \cup \{indep(x,y) \mid x \in Fv_1, x \neq y, y \in vars(C)\}$$

and proceeds left to right with the body of C, $g_1, \cdots, g_n$. Assume we have obtained κ_i; then κ_{i+1} will be obtained from κ_i and g_i in the following way:

—$Fv_{i+1} = Fv_i \setminus vars(g_i)$

—if g_i is not a built-in $\kappa_{i+1} = \kappa_i \setminus (\{\neg ground(x) \mid x \in vars(g_i)\} \cup \{indep(x,y) \mid \{x,y\} \cap vars(g_i) \neq \emptyset, \{x,y\} \cap Fv_{i+1} = \emptyset\})$

—if g_i is a built-in, let κ_{g_i} be the denotation of g_i in GI. Then $\kappa_{i+1} = (\kappa_i \setminus Incons) \cup \kappa_{g_i}$ where $Incons$ is the minimum formula[8] such that $(\kappa_i \setminus Incons) \cup \kappa_{g_i} \not\vdash false$

Consider the sequence of literals in Example 3.2, augmented with a built-in: `w is x+1, a(w), b(x,y), c(z,y)`. The semantics of `is/2` ensures that both `x` and `w` are ground after the execution of this built-in. Since this information is downwards closed (i.e., once satisfied it will continue to hold for the remainder of the forward computation), the local analysis will be able to derive that this holds not only just

[7]As we will see, local analysis can also be performed by applying the abstract operations for the domains that will be introduced in the following sections, but only within the scope of the clause. For the sake of simplicity we have preferred to describe it here independently of the domains.
[8]This minimum inconsistent formula is computed specifically for each built-in.

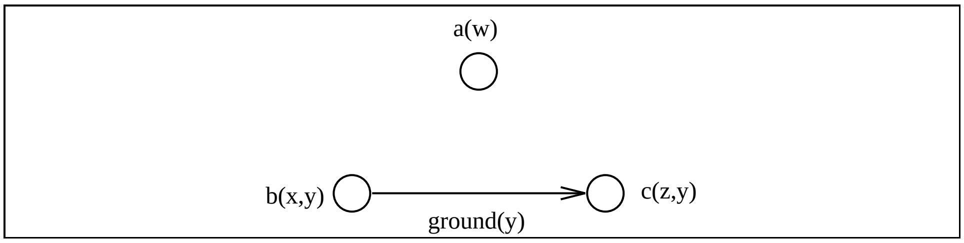

Fig. 4. Simplified dependency graph.

after the execution of the built-in, but also at every point in the clause to the right of it. Thus $\kappa_i \supseteq \{ground(x), ground(w)\}$ for all points $i > 1$. If the above sequence is the body of a clause whose head has only variable x, the information derived at each clause point is the following:

—$Fv_1 = \{w, y, z\}$, $\kappa_1 = \{\neg ground(w), \neg ground(y), \neg ground(z), indep(w, [x, y, z]), indep(y, [x, z]), indep(z, x)\}$,

—$\kappa_{\texttt{w is x+1}} = \{ground(w), ground(x)\}$, $Incons = \{\neg ground(w)\}$, $Fv_2 = \{y, z\}$, $\kappa_2 = \{ground(w), ground(x), \neg ground(y), \neg ground(z), indep(w, [x, y, z]), indep(y, [x, z]), indep(z, x)\}$,

—$Fv_3 = \{y, z\}$, $\kappa_3 = \{ground(w), ground(x), \neg ground(y), \neg ground(z), indep(w, [y, z]), indep(y, [x, z]), indep(z, x)\}$ (note that although $indep(w, x)$ has been dropped at this point, it still follows directly from $ground(x)$ or $ground(w)$; thus κ_3 is in fact equivalent to κ_2),

—$Fv_4 = \{z\}$, $\kappa_4 = \{ground(w), ground(x), \neg ground(z), indep(w, z), indep(y, z), indep(z, x)\}$ (as before, $indep(w, y)$ and $indep(y, x)$, which have been dropped at this point, follow directly from $ground(x)$ and $ground(w)$, respectively, but $\neg ground(y)$ is definitely dropped; thus κ_4 is not equivalent to κ_3),

—$Fv_5 = \emptyset$, $\kappa_5 = \{ground(x), ground(w)\}$ (as before, $indep(w, z)$ and $indep(z, x)$ follow directly from $ground(x)$ and $ground(w)$, respectively, but $\neg ground(z)$ and $indep(y, z)$ are definitely dropped).

The CDG in Example 3.2 becomes, by applying the *improve* function with this information, the one shown in Figure 4. Recall that edges labeled with *true* are eliminated since the dependency is known not to hold.

Thus, from the information of an analysis, the dependencies previously identified can now be simplified, as shown in the example, by applying *improve* to all edges in the CDG. For each edge (g_i, g_j) labeled l in the graph, l is substituted by $improve(l, \kappa_i)$. With this simplified CDG the second subtask, i.e., building the parallel expression, is simpler. Furthermore, the *improve* function will also be applied during this second subtask, as we will see.

3.3 Building Parallel Expressions

The third step in the annotation process aims at obtaining an optimal parallel expression among all the possibilities detected in the previous step, by applying a particular strategy, and further optimizing the number of tests if possible. Different heuristic algorithms implement different strategies to select among all possible parallel expressions for a given clause graph.

Parallel expressions will be built from a language capable of expressing and implementing independent and-parallelism, such as the &-Prolog language [Hermenegildo and Greene 1991]. &-Prolog is essentially Prolog, with the addition of the parallel conjunction operator "`&`" (used, when goals are to be executed concurrently, in place of "," —comma—, and binding stronger than it), a set of parallelism-related built-ins, which include the groundness and independence tests described in the previous section, and a number of synchronization primitives which allow expressing both restricted and nonrestricted parallelism. Combining these primitives with the usual Prolog constructs, such as "`->`" (if-then-else), users can conditionally trigger parallel execution of goals. For syntactic convenience an additional construct is also provided: the *Conditional Graph Expression (CGE)*. A CGE has the general form `(`*i_cond* `=>` $goal_1$ `&` $goal_2$ `& ... &` $goal_N$`)` where *i_cond* is a sufficient condition for running $goal_i$ in parallel under the appropriate notion of independence, in our case strict independence. &-Prolog if-then-else expressions and CGEs can be nested to create richer execution graphs. As mentioned before, if a parallel language as &-Prolog is considered, the parallelization process can be viewed as a source-to-source transformation called annotation. Given a clause, several annotations are possible.

Example 3.7. Consider again the sequence of literals `a(w), b(x,y), c(z,y)` in Example 3.2, whose graph appears in Figure 3. A possible CGE would be

```
a(w), (ground(y),indep(x,z)) => b(x,y) & c(z,y)
```

An alternative would be

```
(indep(w,x),indep(w,z),indep(x,z),ground(y)) =>
                                          a(w) & b(x,y) & c(z,y)
```

Another alternative would be

```
indep(w,[x,y]) -> a(w) & b(x,y), c(z,y)
                ; a(w), (ground(y),indep(x,z)) => b(x,y) & c(z,y)
```

and so on.

Three different heuristic algorithms (annotators) are embedded in our system, namely **CDG**, **UDG**, and **MEL** [Muthukumar and Hermenegildo 1990b].[9] The **CDG** algorithm seeks to maximize the amount of parallelism available in a clause, without being concerned with the size of the resulting parallel expression. In doing this, the annotators may switch the positions of independent goals. **UDG** does essentially the same as **CDG** except that only unconditional parallelism is exploited, i.e., only goals which can be determined to be independent at compile-time are run in parallel. **MEL** tries to find points in the body of a clause where it can be split into different parallel expressions (i.e., where edges labeled "false" appear) without changing the order given by the original clause and without building nested parallel expressions. At such points the clause body is broken into two, a CGE is built for

[9]**CDG** stands for Conditional Dependency Graph; **UDG** stands for Unconditional Dependency Graph; and **MEL** stands for Maximal Expression Length.

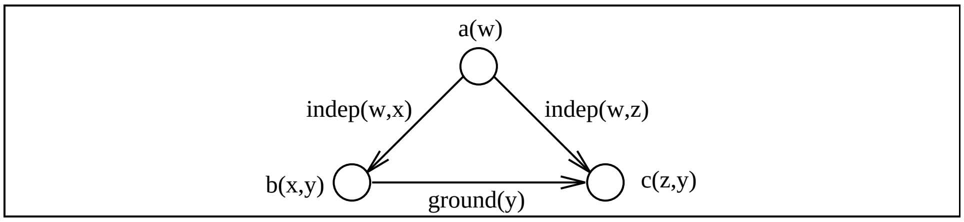

Fig. 5. Simplified conditional dependency graph.

the right part of the split sequence, and the process continues with the left part. The correctness of these three algorithms has been proved and their effectiveness experimentally evaluated in [Muthukumar et al. 1999]. In the following we will focus on the **MEL** algorithm, in the particular context of strict independence. The reason for choosing the **MEL** algorithm is that it is the simplest, and therefore the influence of the analysis information in the resulting parallel expressions is easier to understand.

Once an expression has been built, it can be further simplified, unless it is unconditional. Based on local or global information, the overall condition built by the annotation algorithm can possibly be reduced again following the same general procedure introduced in the previous section.

Example 3.8. Consider once more the sequence of literals in Example 3.2, augmented this time with a different built-in (a unification): `y = f(x,z), a(w), b(x,y), c(z,y)`. We will not consider small built-ins such as `y = f(x,z)` for parallelization (as normally done by the local granularity control). Now the analysis can derive $\{(ground(x) \wedge ground(z)) \leftrightarrow ground(y)\} \subseteq \kappa_i$ for all points $i > 1$.

Let the literals be the body of a clause whose head has variables w, x, and z. After identifying dependencies, the CDG is that of Figure 3, but after simplifying them with a local analysis it will become the one in Figure 5.

If, for instance, the parallel expression obtained for the simplified CDG involves all three literals, it will have the following condition:

```
y=f(x,z),(indep(w,x),indep(w,z),ground(y)) =>
                                        a(w) & b(x,y) & c(z,y)
```

But since the groundness of y implies the groundness of both x and z and thus their independence from any other variable, the condition can be further simplified to

```
y = f(x,z), ground(y) => a(w) & b(x,y) & c(z,y)
```

Example 3.9. Figure 6 summarizes the three steps of the annotation process for clause `h(x,y,z):- p(x,y), q(x,z), s(z,w)`, assuming that global analysis obtains $k_1 = \{ground(x), \neg ground(z), indep(y,z)\}$.

Note that more than the two alternatives shown in the figure are possible.

4. CORRECTNESS OF THE PARALLELIZATION FRAMEWORK

After having introduced the three steps of the parallelization process, we now show its correctness. We start by recalling the operational semantics of CDGs, which has

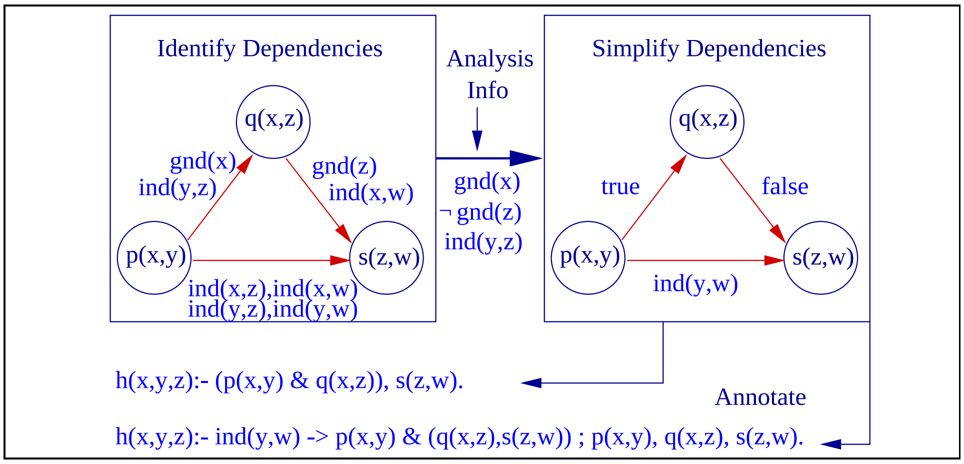

Fig. 6. Annotation process: A complete example.

been proven correct with respect to the sequential semantics of the corresponding clause in [Muthukumar et al. 1999]. Then we show that the CDG resulting from the simplification process is correct with respect to this semantics.

The (nonsimplified) conditional dependency graph of a clause expresses all independent and-parallelism available in the clause at the goal level. Thus, it is possible to define goal-level and-parallel models which work directly with such a graph. The general model MEIAP ("Maximal Efficient Independent And-Parallelism") considered in [Muthukumar et al. 1999] is based on this approach. It works as follows. Let a node be *ready* if it has no incoming edges. Parallel execution of the goals to which the graph nodes correspond is achieved by repeated application of the following rules:

—*Goal initiation*: Consider nodes whose incoming edges have source nodes which are ready. If the tests labeling all these edges are satisfied for the current store c, remove them all. Repeat until no edges are removed. Initiate all goals g in ready nodes by executing $\langle g, c\rangle$ in different environments.

—*Node removal*: Remove from the graph all nodes whose corresponding goals have finished executing, and their outgoing edges. Add the associated answers to the current store c.

The model is "maximal" in the sense that goals are run in parallel *as soon as* they become independent of all nodes to the left (i.e., ready). However, this requirement can be dropped, allowing a more general model in which ready goals are ensured to be independent if run in parallel, but they are not actually required to be run in parallel. In fact, even sequential execution is allowed. In particular, given a nonsimplified graph, a CGE simply corresponds to a particular subset of the allowed executions. This subset is precisely the one obtained following the particular heuristic used by the annotator that built the CGE.

The MEIAP model has been proved correct (and efficient) with respect to sequential execution [Muthukumar et al. 1999] as long as ready nodes are pairwise independent for the stores they are being executed in. This is true if the tests labeling the edges ensure the independence of the literals in the stores in which these tests are satisfied. Note that correctness is preserved even if the maximality condition is dropped: ready nodes remain independent in the stores they are executed in, even if these stores differ from the stores the tests were evaluated. The proof of correctness is based on the special properties of the notion of strict independence and, in particular, on the characteristics of the constraints that can be added to the store by strictly independent goals.

We first show how the strict independence properties allow the correctness proof, and then extend the correctness result to simplified graphs. As a consequence, the CGEs obtained by the annotators from both nonsimplified and simplified graphs are also guaranteed to be correct.

The following result is instrumental. It ensures pairwise strict independence of a node g_{n+1} and all (combinations of) nodes to its left for a store c, with the only condition that for g_{n+1} to become ready it must be strictly independent for c of each node to its left, i.e., the nodes to the left of g_{n+1} are not required to be also pairwise strictly independent.

LEMMA 4.1. *Consider the goals $g_1, \cdots, g_n, g_{n+1}$. If g_{n+1} is strictly independent of any $g_j, 1 \leq j \leq n$ for store c, then g_{n+1} is strictly independent of G for c, where G is any possible sequence formed from $g_1, \cdots, g_n$.*

PROOF. Comes directly from the definition of strict independence. □

The following result ensures that if the tests for the pairwise strict independence of a set of goals are satisfied in store c, they will also be satisfied in any subsequent store c' resulting from the execution of any of those goals in c, i.e., the goals are also pairwise strictly independent for any c'. As a result, all ready goals remain pairwise strictly independent even if they are not executed in parallel as soon as they become ready.

LEMMA 4.2. *Let $g_1, \cdots, g_n$ be a set of goals and t_{ij} be the tests needed for ensuring the strict independence of goals g_i and g_j, $i \neq j$. If the goals are pairwise strictly independent for store c, they remain independent for any store c' resulting from the execution of $\langle G, c\rangle$, where G is any possible sequence formed from $\{g_1, \cdots, g_n\}$, i.e., if every t_{ij} is satisfied for c they will also be satisfied for c'.*

PROOF. Let us reason by contradiction. Assume there is at least a test t in some t_{ij} which is not satisfied in some store c' which results from the execution of $\langle G, c\rangle$. If $t \equiv ground(x)$, and t was satisfied for c, it must also be satisfied for any further instantiated store. Thus t_{ij} must contain at least a test like $t \equiv indep(x, y)$. If either x or y are ground in c, t will also be satisfied in c'. Therefore, the variables must be nonground in c. Since $indep(x, y)$ fails in c', some goal in G must have introduced a dependency between them. Let us assume, without loss of generality, that x belongs to g_i and y belongs to g_j. From pairwise strict independence and the fact that x and y are nonground in c, x must only appear in g_i, y must only appear in g_j (if x appears in other g_k, t_{ik} must contain the test $ground(x)$, and it has to be true in c; the same reasoning applies for y), and no other goal can share with x or y. Thus, no dependency between x and y can be introduced by any other goal, and t_{ij} must be satisfied. □

The following result is directly obtained from the above two lemmas.

COROLLARY 4.3. *Consider the set of goals $g_1, \cdots, g_n$ and the store c. Let Ready be the set of goals g_j which are strictly independent of any $g_i, 1 \leq i < j$ for c. Then, for every $g_j \in Ready$, the execution of g_j in c is the same as the execution of g_j in any store obtained from the execution of $\langle G, c\rangle$, where G is any possible sequence formed from elements of $(\{g_1, \cdots g_{j-1}\} \cup Ready) \setminus \{g_j\}$. And, in particular, the answers are the same as in the corresponding sequential execution, i.e., the execution of g_j in the store obtained from $\langle g_1 ... g_{j-1}, c\rangle$.*

This result provides the condition on which the proof of correctness of the MEIAP model for nonsimplified graphs for *a priori* notions of independence is parameterized. As a consequence, ready goals can be correctly executed following any strategy, and, in particular, that one which a particular annotator chooses. However, for simplified graphs we have to go a step further. The difference now is that conditions between goals g_i and g_j, $i < j$, might have been simplified with respect to the information known to hold for the store obtained in the sequential execution right before executing g_i. Thus, the simplification might have been performed with respect to the information inferred for a store very different to the one the goals are determined as ready.

Example 4.4. Consider again the sequence of literals `w is x+1,a(w),b(x,y),` `c(z,y)` of Example 3.6. The simplified graph is that of Figure 4. In this case, the original independence test $\{indep(x, z), ground(y)\}$ labeling the edge between `b(x,y)` and `c(z,y)`, has been simplified to $\{ground(y)\}$. This has been possible because `x` was known to be ground before the execution of the goals. However, it is clear that although $ground(y)$ is satisfied, for example, in store $x = z \wedge y = 3$, the goals are not strictly independent for that store.

We have to ensure that the tests would be also satisfied in the stores the nodes are determined as ready. The solution is provided by the following lemma.

LEMMA 4.5. *Consider the goals $g_1, \cdots, g_n, g_{n+1}$ and the store c. If for each $g_j, 1 \leq j \leq n$, the tests t_j needed for ensuring the strict independence of goals g_j and g_{n+1} can be divided into t^c and $t^{c'}$, such that t^c is satisfied in c and $t^{c'}$ is satisfied in the store c' obtained from $\langle g_1 ... g_{j-1}, c\rangle$, then g_{n+1} is strictly independent of each goal $g_j, 1 \leq j \leq n$, for c, i.e., for each such g_j the associated $t^{c'}$ is also satisfied in c.*

PROOF. Let us reason by induction. The base case is obvious: since c and c' are the same, t_1 must then be satisfied in c, and thus g_1 and g_{n+1} are strictly independent for c. Assume that the lemma is satisfied for goals $g_1, \cdots, g_{j-1}$, and let us now prove the induction step. By assumption of the hypothesis we have that g_{n+1} is strictly independent of each goal $g_i, 1 \leq i < j$ for c, and that t_j can be divided into t^c and $t^{c'}$, such that t^c is satisfied in c and $t^{c'}$ is satisfied in the store c' obtained from $\langle g_1 ... g_{j-1}, c\rangle$. Now we have to prove that g_j and g_{n+1} are strictly independent for c. Given the previous assumptions, this will be true if $t^{c'}$ is true in c. Let us reason by contradiction. If $t^{c'}$ is not satisfied by c, there must be at least a test t in $t^{c'}$ which is true in c' but not in c. If $t \equiv ground(x)$, there must be a goal g_i, $1 \leq i < j$, which further instantiates x. If x appears in g_i, $ground(x)$

should also appear in t_i and, by induction hypothesis, $ground(x)$ must be satisfied in c. If x does not appear in g_i, by induction hypothesis, x must be independent of any variable in g_i, and thus g_i cannot further instantiate x. Therefore, t must be $indep(x,y)$. Since t is satisfied in c' but not in c, x and y must be nonground in c, and some goal g_i, $1 \leq i < j$, must have removed the dependency involved. This is only possible by further instantiating the variables shared by x and y. Let us assume, without loss of generality, that x belongs to g_{n+1} and y belongs to g_j. Following the same reasoning as before, x cannot be further instantiated by any g_i, and thus the dependency cannot be removed. Hence, $t^{c'}$ should be true in c. □

Example 4.6. Consider again the sequence of literals `p(x,y),q(x,z),s(z,w)` of Example 3.9. As illustrated in the leftmost part of Figure 6, the tests $t_1 = \{indep(x,z), indep(x,w), indep(y,z), indep(y,w)\}$ are needed for ensuring the strict independence of goals `p(x,y)` and `s(z,w)`, and the tests $t_2 = \{ground(z), indep(x,w)\}$ are needed for ensuring the strict independence of goals `q(x,z)` and `s(z,w)`. If we assume that before executing any literal the store c satisfies the tests in $k_1 = \{ground(x), \neg ground(z), indep(y,z)\}$, then we can ensure that those in $t_1^c = \{indep(x,z), indep(x,w), indep(y,z)\}$ and $t_2^c = \{indep(x,w)\}$ are also satisfied by c. Thus, according to 4.5, if the rest of the tests in t_1 (i.e., $t_1^{c'} = \{indep(y,w)\}$) are satisfied in any store c' obtained from $\langle nil, c\rangle$ (i.e., $c' \equiv c$), then they are also satisfied in c, and thus `p(x,y)` and `s(z,w)` are strictly independent for c. This is used for the generation of the second parallel expression in Figure 6. Also, if the rest of the tests in t_2 (i.e., $t_2^{c'} = \{ground(z)\}$) are satisfied in any store c' obtained from $\langle p(x,y), c\rangle$, then they would also be satisfied in c, and `q(x,z)` and `s(z,w)` would also be strictly independent for c. However, $ground(z)$ is known not to hold for c (or any c'), and thus we can ensure that `q(x,z)` and `s(z,w)` are not strictly independent for c. This is also used in the generation of both parallel expressions in Figure 6 by never considering the parallelization of `q(x,z)` and `s(z,w)`.

As a result of the above lemma, we are able to ensure that if simplification of a test occurs, then parallel execution of the goals involved can only occur in a store in which the test would have been satisfied anyway. Thus, we can now prove that the following result holds.

THEOREM 4.7 (CORRECTNESS OF IMPROVE). *Consider the CDG obtained by applying the notion of strict independence between goals $\{g_1 ... g_n\}$, possibly simplified by improve with respect to information valid for the sequential execution of state $\langle g_1 ... g_n, c\rangle$. Any execution obtained by applying the MEIAP model to the CDG with initial store c is correct with respect to the sequential execution of $\langle g_1 ... g_n, c\rangle$.*

PROOF. Let us prove it by induction:

—Base case: Some goals in ready nodes are going to be executed in parallel in the initial store c. Let t_{ij} be the original conditions to be satisfied for the strict independence of ready goals g_i and $g_j, i \neq j$. By definition of the model, for each ready goal g_j and each $g_i, 1 \leq i < j$, the remaining tests after simplifying t_{ij} have been satisfied for c, and the tests eliminated by simplification are satisfied in the corresponding sequential store. By 4.5, such eliminated conditions are also satisfied in c. Thus, each ready goal g_j is strictly independent of each goal $g_i, 1 \leq i < j$ for c. In particular, ready goals are pairwise strictly independent for c. Therefore by 4.3, their parallel execution in c is correct with respect to the sequential execution.

—Induction hypothesis: In any store c' ready goals are pairwise strictly independent for any store obtained by conjoining with c' the constraints introduced by the execution of any ready goal already running. This is satisfied after initiation of the model (in the base case), thanks to 4.2.

—Induction step: Assume that at some point of the execution the current store is c', and the set of ready goals is G_{old}. Then, after goal initiation, a new set of ready goals G_{new} is found. From the induction hypothesis, all goals in G_{old} are pairwise strictly independent for c'. By definition of the model, for each ready goal $g_j \in G_{new}$ and each goal $g_i, i < j$, the remaining tests after simplifying t_{ij} have been satisfied for c', and the tests eliminated by simplification are satisfied in the corresponding sequential store. By 4.5, such eliminated conditions are also satisfied in c'. Thus, ready goals in $G_{old} \cup G_{new}$ are pairwise strictly independent for c'. Therefore, by 4.3, their parallel execution in c' is correct with respect to the sequential execution. Also, by 4.2 they will also be pairwise strictly independent for any store obtained by conjoining with c' the constraints introduced by the execution of any ready goal now initiated or already running, so the induction hypothesis holds in any new store (after execution of some goals now initiated), too. $\square$

It is important to point out that the particular characteristics of strict independence which allow these results are not shared by all independence notions. In particular, although in traditional Logic Programming they are shared by all *a priori* notions, they are not present in more general *a posteriori* notions. Let us illustrate this point with an example.

Example 4.8. Consider the program

```
p(x,y) :- x=y.
q(y) :- y=3.
r(x) :- x=5.
```

and the notion of *search independence* [García de la Banda et al. 1993]. This is an a posteriori notion, which states that two goals are independent for store c iff the partial answers of the goals for c are consistent. It is clear that the above goals are pairwise search independent for store *true*. However, their parallel execution in store *true* is not correct, since `p(x,y)` introduces a dependency that will affect the parallel execution of the rest of the goals. This is directly related with 4.2, since although the conditions for independence are satisfied for *true*, they are not satisfied for the store obtained from the execution of `p(x,y)`. The core of the problem is that pairwise independence is not enough for ensuring the independence when considering a posteriori notions. The independence has to be ensured not only between each two goals, but also between each goal and the goal formed by concatenating the rest of the goals to be executed in parallel.

Beyond correctness, efficiency of the MEIAP model can also be proven. Such a proof is beyond the scope of this article. However, we will discuss some of the issues involved. It turns out that a theoretical result on efficiency needs some additional assumptions. In the context of parallel execution of logic programs the theoretical notion of efficiency used is that the parallel execution causes no slowdown with respect to the sequential one, measured in terms of number of resolution steps (see, e.g., [Hermenegildo and Rossi 1995; García de la Banda 1994]). The first condition to achieve this is that the independence conditions (*icond*s) guarantee that the execution of an individual goal run in parallel does not produce more work than if it was run sequentially. Beyond this, failure and backtracking over goals in parallel expressions introduce additional complications. In particular, the proof assumes that either the parallel goals can be proved not to fail or, if a parallel goal fails, all other parallel goals which are executing at the same time and correspond to literals appearing in the clause to the right of the literal of the failing goal, are immediately killed. Also, it is necessary to assume that the execution scheduler is “left-biased,” i.e., parallel goals are scheduled for execution in a left-to-right fashion (see, e.g., [Hermenegildo and Rossi 1995] for details). Third, the (implicit) reordering of goals which may incidentally happen when goals are run in parallel has also subtle consequences on efficiency. Even in the case of a left-biased scheduler, backtracking over parallel expressions which have been reordered and which have multiple solutions may cause more work to be performed than in the sequential case (see, e.g., [García de la Banda et al. 1993]). Thus, the proof assumes that only goals which do not have multiple solutions are reordered. Summarizing, and simplifying a bit the discussion, parallel execution is guaranteed to incur no slowdown, provided that the above-mentioned assumptions hold.

5. DOMAINS OF ANALYSIS

In this section we will briefly introduce the definition of each abstract domain used in the experimental evaluation and its concretization function. Then we will discuss in terms of the interface defined in the previous section the particular ways in which each domain captures the information needed to simplify the conditions in the parallel expressions.

5.1 ASub Domain

The domain ASub [Sondergaard 1986] was defined for inferring *groundness*, *sharing*, and *linearity* information. The abstract domain approximates this information by combining two components: definite groundness information is described by means of a set of program variables $D_1 = 2^{\mathrm{PVar}}$; possible (pair) sharing information is described by symmetric binary relations on PVar $D_2 = 2^{(\mathrm{PVar} \times \mathrm{PVar})}$.

The concretization function, $\gamma_{ASub} : \mathsf{ASub} \rightarrow 2^{Sub}$, where Sub is the set of idempotent substitutions, is defined for an abstract substitution $(G, R) \in \mathsf{ASub}$ as follows: $\gamma_{ASub}(G, R)$ approximates all concrete substitutions θ such that for every $(x, y) \in \mathsf{PVar}^2$, $x \in G \Rightarrow ground(x\theta)$; $x \neq y \wedge vars(x\theta) \cap vars(y\theta) \neq \emptyset \Rightarrow x\ R\ y$, and $x \not R\ x \Rightarrow linear(x\theta)$.

Note that the second condition implies that whenever $x \neq y$ if $x \not R\ y$ then we have that $vars(x\theta) \cap vars(y\theta) = \emptyset$, and thus x and y are independent.

Let us now present the relation between ASub and the domain GI. Consider an abstract substitution $\lambda_i \in \mathsf{ASub}$ for program point i of a clause C. The contents of κ_i follow from the following properties of $\lambda_i = (G, R)$ over $vars(C)$:

—$ground(x)$ if $x \in G$

—$indep(x,y)$ if $x \neq y, x \not R\, y$

Note that in this case κ_i contains neither $\neg ground(x)$ nor $\neg indep(x,y)$ for every $\{x,y\} \subseteq vars(C)$, and thus no tests in the CDG can ever be reduced to false with only this information.

Example 5.1. Consider a clause C such that $vars(C) = \{x,y,z,w,v\}$ and an abstract substitution $\lambda = (\{x\}, \{(z,w),(z,v)\})$. The corresponding κ will be the set $\{ground(x), indep(y,z), indep(y,w), indep(y,v), indep(w,v)\}$.

5.2 Sharing Domain

The Sharing domain [Jacobs and Langen 1989] was proposed for inferring *groundness* and *sharing* information. The abstract domain, $\mathsf{Sharing} = 2^{2^{\mathrm{PVar}}}$, keeps track of *set* sharing. The concretization function is defined in terms of the occurrences of a variable U in a substitution θ: $occs(\theta,U) = \{x \in dom(\theta) | U \in vars(x\theta)\}$, where $dom(\theta)$ is the domain of the function θ. If $occs(\theta,U) = V$ then θ maps the variables in V to terms which share the variable U. The concretization function $\gamma_{Sharing} : \mathsf{Sharing} \to 2^{Sub}$ is defined as follows:

$$\gamma_{Sharing}(\lambda) = \{\theta \in Sub \mid \forall U \in \mathsf{Var}.\ occs(\theta,U) \in \lambda\}.$$

Intuitively, each set in the abstract substitution containing variables $v_1,\ldots,v_n$ represents the fact that there may be one or more shared variables occurring in the terms to which $v_1,\ldots,v_n$ are bound. If a variable v does not occur in any set, then there is no variable that may occur in the terms to which v is bound, and thus those terms are definitely ground. If a variable v appears only in a singleton set, then the terms to which it is bound may contain only variables which do not appear in any other term.

Let us now present the relation between Sharing and the *GI* domain. Consider an abstract substitution $\lambda_i \in \mathsf{Sharing}$ for program point i of a clause C. The contents of κ_i follow from the following properties of λ_i over $vars(C)$:

—$ground(x)$ if $\forall S \in \lambda_i :\ x \notin S$

—$indep(x,y)$ if $\forall S \in \lambda_i :$ if $x \in S$ then $y \notin S$

—$ground(x_1) \wedge \ldots \wedge ground(x_n) \to ground(y)$ if $\forall S \in \lambda_i :$ if $y \in S$ then $\{x_1,\ldots,x_n\} \cap S \neq \emptyset$

—$ground(x_1) \wedge \ldots \wedge ground(x_n) \to indep(y,z)$ if $\forall S \in \lambda_i :$ if $\{y,z\} \subseteq S$ then $\{x_1,\ldots,x_n\} \cap S \neq \emptyset$

—$indep(x_1,y_1) \wedge \ldots \wedge indep(x_n,y_n) \to ground(z)$ if $\forall S \in \lambda_i :$ if $z \in S$ then $\exists j \in [1,n], \{x_j,y_j\} \subseteq S$

—$indep(x_1,y_1) \wedge \ldots \wedge indep(x_n,y_n) \to indep(w,z)$ if $\forall S \in \lambda_i :$ if $\{w,z\} \subseteq S$ then $\exists j \in [1,n], \{x_j,y_j\} \subseteq S$

The meaning of each implication in κ_i can be derived by eliminating the required sets in λ_i so that the antecedent of the implication holds, and looking for the new facts $ground(x)$ or $indep(x,y)$ in the updated abstract substitution, which now become true. For example, in the fifth rule, all sets in which both x_i and y_i appear have to be eliminated in order to satisfy $indep(x_i,y_i)$. If z does not appear in any other set, the independence of these variables will imply the groundness of z. As in ASub, no tests in the CDG can ever be reduced to false with only this information.

Example 5.2. Consider a clause C for which $vars(C) = \{x, y, z, w, v\}$ and the abstract substitution $\lambda = \{\{y\}, \{z, w\}, \{z, v\}\}$. The corresponding κ will be the set $\{ground(x)$, $indep(y, z)$, $indep(y, w)$, $indep(y, v)$, $indep(w, v)$, $ground(z) \leftrightarrow ground(w) \wedge ground(v)$, $indep(z, v) \wedge indep(z, w) \rightarrow ground(z)$, $indep(z, v) \rightarrow ground(v)$, $indep(z, w) \rightarrow ground(w)\}$.

Note that in the above example κ contains all the information derived in Example 5.1 plus the information provided by the power of the set sharing for groundness propagation, in contrast with that of the pair-sharing representation. On the other hand, the linearity information present in ASub usually provides more accurate sharing. This is because whenever a program variable is known to be linear, we can ensure that no sharing is created among the variables in terms that variable is unified to. For example, if the unification $x = f(y, z)$ appears in the program and x is known to be linear, we can ensure that no sharing between y and z is created due to this unification. Otherwise, sharing between y and z will have to be assumed.

5.3 Sharing+Freeness Domain

The Sharing+Freeness domain [Muthukumar and Hermenegildo 1991] aims at inferring *groundness*, *sharing*, and *freeness* information. The abstract domain approximates this information by combining two components: one, $Sh = 2^{2^{\mathrm{PVar}}}$, is the same as the Sharing domain; the other, $Fr = 2^{\mathrm{PVar}}$, encodes freeness information. The concretization function $\gamma_{Fr} : Fr \rightarrow 2^{Sub}$ is defined as follows: $\gamma_{Fr}(\lambda_{fr})$ approximates all concrete substitutions θ such that for every $x \in \mathsf{PVar}$: if $x \in \lambda_{fr}$ then $free(x\theta)$.

Let us now present the relation between this domain and the domain GI. Consider an abstract substitution $\lambda_i \in$ Sharing+Freeness for program point i of clause C. The contents of κ_i follow from the following simple albeit crucial property of $\lambda_i = \langle \lambda_{sh}, \lambda_{fr} \rangle$ over $vars(C)$:

—$\neg ground(x)$ if $x \in \lambda_{fr}$

In this case κ_i allows the simplification of conditions which will always fail. This provides additional precision to that which comes out of the synergistic interaction between the two components of Sharing+Freeness.

Furthermore, other information is obtained by combining the above with that obtained from the sharing component $\lambda_{sh} \in$ Sharing, as in the previous section. For example, $\neg indep(x, y)$ can be obtained from $\neg ground(x)$ and $indep(x, y) \rightarrow ground(x)$. This allows us to also establish the following assertions:

—$\neg indep(x, y)$ if $y \in \lambda_{fr}$ and $\forall S \in \lambda_{sh}$: if $y \in S$ then $x \in S$

—$ground(x_1) \wedge \ldots \wedge ground(x_n) \rightarrow \neg indep(y, z)$ if $z \in \lambda_{fr}$ and $\exists S \in \lambda_{sh}$ such that $\{y, z\} \subseteq S$ and $\forall S \in \lambda_{sh}$: if $\{y, z\} \cap S = \{z\}$ then $\{x_1, \ldots, x_n\} \cap S \neq \emptyset$

—$indep(x_1, y_1) \wedge \ldots \wedge indep(x_n, y_n) \rightarrow \neg indep(y, z)$ if $z \in \lambda_{fr}$ and $\exists S \in \lambda_{sh}$ such that $\{y, z\} \subseteq S$ and $\forall S \in \lambda_{sh}$: if $\{y, z\} \cap S = \{z\}$ then $\exists j \in [1, n]$, $\{x_j, y_j\} \subset S$

The intuition behind each implication is that by updating the abstraction λ_i so that the antecedent holds a new abstraction is obtained in which the consequent holds.

Example 5.3. Consider the same clause C as in Example 5.2 and the same sharing component $\lambda_{sh} = \{\{y\},\{z,w\},\{z,v\}\}$. Consider the freeness component $\lambda_{fr} = \{w\}$. The corresponding κ will be the result of adding $\{\neg ground(w)\}$ to the formula obtained in the previous example. This information, in addition to that derived by λ_{sh}, makes $\kappa \vdash \neg ground(z)$ and $\kappa \vdash \neg indep(z,w)$. Using this information, any test labeling an edge of a CDG including $ground(w)$ or $ground(z)$ or $indep(z,w)$ can be reduced to false.

Note that in the example above $\neg ground(z)$ was derived even though $z \notin \lambda_{fr}$. This is a subtle characteristic of the Sharing+Freeness domain which gives it a significant part of its power. Furthermore, although not directly related to strict independence, the Sharing+Freeness abstract domain is also able to infer definite nonfreeness for nonground variables: if in the example above, v were also an element of λ_{fr}, then z would be ensured to be bound to a term not only nonground, but also nonfree. This characteristic is of use in applications such as analysis of programs with dynamic scheduling [Marriott et al. 1994], nonstrict independence [Cabeza and Hermenegildo 1994], etc.

It is clear that the Sharing+Freeness domain subsumes the Sharing domain and that it is more powerful than ASub in describing the groundness and independence relationships among program variables. However, it would be an error to think that Sharing+Freeness is more accurate than the ASub domain. In fact, as shown in [Codish et al. 1995], the two domains are incomparable, their accuracy depending on characteristics of the analyzed program. Furthermore, the empirical evaluation performed there showed that ASub is usually more accurate.

However, as we will see in the evaluation results, this fact does not preclude the Sharing+Freeness domain from often being more useful for automatic parallelization than ASub. The reason for this comes directly from the combination of the power of the set-sharing abstraction and the freeness information. Set sharing does not only allow groundness propagation by approximating information similar to that abstracted by the Prop domain[10] [Marriott and Søndergaard 1989], but also allows establishing relationships among the groundness and independence characteristics of the variables. Freeness allows inferring nongroundness information; in combination with set sharing it allows propagating such nongroundness information and inferring definite sharing (i.e., $\neg indep(x,y)$ facts). Note that such power cannot be obtained by combining freeness, pair sharing, and Prop, since the relationship between groundness and independence would be lost in some cases.

Example 5.4. Consider the following clause:

```
p(x,y,z,v) :- x = 3, y = f(z,v), q(...), r(...), ...
```

Assuming it is called with the abstraction of the empty substitution, the abstract substitution of each domain at the point of calling q is as Table III shows. The information inferred in terms of GI is also given.

[10]Prop approximates definite groundness dependencies by means of positive Boolean functions closed under intersection.

Table III. Example Abstract Substitution Information

ASub	$([x],[(y,z),(y,v)])$ $ground(x),\ indep(z,v)$	implies
Sharing	$[[y,z],[y,v]]$ $ground(y) \leftrightarrow ground(z) \wedge ground(v)$ $indep(y,z) \rightarrow ground(z)$ $indep(y,v) \rightarrow ground(v)$ $indep(y,z) \wedge indep(y,v) \leftrightarrow ground(y)$	additionally implies
Sharing+Freeness	$([[y,z],[y,v]],[z,v])$ $\neg ground(z),\ \neg ground(v)$ and thus: $\neg ground(y),\ \neg indep(y,z),\ \neg indep(y,v)$	additionally implies

Note that with respect to the domain GI, the information provided by each abstract domain is comparatively more elaborate than the previous one.

5.4 Combined Domains

As mentioned before, we have also considered for evaluation the analyzers resulting from the combination of the domains ASub and Sharing, on one hand, and ASub and Sharing+Freeness, on the other. The domain combination, presented by Codish et al. [1995], is based on the *reduced product* approach of Cousot and Cousot [1979]. In this approach domains are combined simply by, at each step in the analysis, inferring the information for each domain and then removing redundancies. The advantage of this approach is that it allows us to infer more accurate information from the combination without redefining neither the abstract domains nor the basic abstract operations of the original domains. As a result not only is a proof of correctness of the new analyzer unnecessary, but also the gain in accuracy obtained by simply removing redundancies at each step is significant, as shown in [Codish et al. 1995].

The information approximated by the combined domains can be used in the simplification task by simply translating the information inferred by each domain into the GI domain, conjoining the resulting κs, and applying the techniques described in previous sections.

6. PROGRAM ANALYSIS

As mentioned in the introduction, abstract interpretation of logic programs allows the systematic design and verification of data flow analyses by formalizing the relation between analysis and semantics. Therefore, abstract interpretation is inherently semantics sensitive, different semantic definition styles yielding different approaches for program analysis. For logic programs we distinguish between two main approaches, namely *bottom-up* analysis and *top-down* analysis. While the top-down approach propagates the information in the same direction as SLD-resolution does, the bottom-up approach propagates the information as in the computation of the least fixpoint of the immediate consequences operator T_P. In addition, we distinguish between *goal-dependent* and *goal-independent* analyses. A goal-dependent analysis provides information about the possible behaviors of a specified (set of) initial goal(s) and a given logic program. In contrast, a goal-independent analysis considers only the program itself.

In the process of automatic parallelization we are interested in inferring accurate information regarding the substitutions affecting these goals in any proof which can be constructed with the given clause in the given program. It seems that a top-down analysis framework performing goal-dependent analysis is the most appropriate for this task. However, it is important to note that recently a number of studies have extended the area of applicability of both the bottom-up and top-down frameworks and their relations with goal-dependent and goal-independent analysis [Giacobazzi 1993; Codish et al. 1994; Codish et al. 1997]. In this study we have used a top-down, goal-dependent framework, namely the abstract interpretation system PLAI, mainly because of the more mature state of its implementation.

6.1 Global Analysis: The Abstract Interpretation Framework PLAI

The PLAI abstract interpretation system is a top-down framework based on the abstract interpretation framework of Bruynooghe [1991] with the fixpoint optimizations described in [Muthukumar and Hermenegildo 1992]. Although a detailed description of this system is outside the scope of this article, we will point out several features which are relevant to our study, as they either allow efficient analysis or a more effective parallelization.

The framework is based on an abstraction of the (SLD) AND-OR trees representing the execution of a program for a given set of entry points. An entry point is a literal and a description of the possible values of the arguments of external calls to the literal. Such descriptions can be given for example using the abstract domains supported. The abstract AND-OR graph allows the framework to provide information at each program point, a feature which is crucial for many applications (such as, for example, reordering, automatic parallelization or garbage collection). For each given goal and abstract call substitution, PLAI builds a node in the abstract AND-OR graph and computes its (possibly many) abstract success substitution(s). Note that, in doing this, PLAI computes the specialized versions (also referred to as multivariants) for each goal, thus allowing for a quite detailed analysis. The current implementation of the framework allows the user to choose between obtaining a transformed program showing all variants generated for each clause and the particular information inferred for each program point, or the original program in which the information for different variants is collapsed into one by means of the upper bound operation of the particular abstract domain. Note that, since the framework treats (and takes advantage of) programs in nonnormalized form, different call formats of the same literal (e.g., p(1), p(x)) yield different goals.

The analysis algorithm is as follows. For each pair of goal and abstract call substitution for this goal, and for each clause matching the goal, the corresponding entry abstract substitution for the clause is computed, which yields the call substitution for the first literal in the clause body. The body is then traversed by recursively applying the same algorithm: the success substitution of each goal is the call substitution of the next one. The success substitution of the last goal is the exit substitution of the clause. The definition considers each literal and call substitution (and clause[11]) distinctly, and therefore naturally captures multivariants.

[11]It is possible to produce an abstract success substitution for each applicable clause, or to collapse all resulting substitutions for all clauses of a predicate into a single one. This choice is essentially

The substitutions computed are stored in a so-called memo table. The memo table is also used to keep track of information necessary for the fixpoint computation inherent to the analysis.

PLAI implements a highly optimized fixpoint algorithm, defined in [Muthukumar and Hermenegildo 1989b; 1990a; 1992] (similar to the algorithm independently proposed in [Le Charlier et al. 1993]), based on a distinction between recursive and nonrecursive predicates. Briefly, the analysis proceeds as follows. First, a preprocessing of the program is performed to fold disjunctions and determine recursive predicates; then the core of the analysis starts. Nonrecursive predicates are analyzed in one pass. For the recursive predicates, nonrecursive clauses are analyzed first and once, and the result is taken as a first approximation of the answer. If no nonrecursive clause is found, the least domain element "bottom" is taken as first approximation. Then, a fixpoint computation for the recursive clauses starts. The number of iterations performed in this computation is reduced by keeping track of the dependencies among nodes and the state of the information being computed. In some cases the fixpoint algorithm is able to finish in a single iteration (i.e., in only one pass), even if there are recursive predicates, thanks to this information.

The whole computation is domain independent. This allows plugging in different abstract domains, provided suitable interfacing functions are defined. Let Sub^α be an abstract domain. The domain-dependent functions are as follows:

—$project : Sub^\alpha \times Literal \rightarrow Sub^\alpha$, such that $project(\lambda, g)$ is the projection of λ over $vars(g)$;

—$call\_to\_entry : Sub^\alpha \times Literal \times Literal \times Literal^* \rightarrow Sub^\alpha$, such that the output of $call\_to\_entry(\lambda, g, h, B)$ is the entry substitution for clause h:-B corresponding to call substitution λ after unifying $h = g$, projecting the result over the variables in h, and extending the result so that it approximates the values of the new variables in B;

—$exit\_to\_success : Sub^\alpha \times Literal \times Literal \rightarrow Sub^\alpha$, such that the output of $exit\_to\_success(\lambda, g, h)$ is the success substitution for goal g corresponding to exit substitution λ after unifying $h = g$, and projecting the result over the variables in g;

—$extend : Sub^\alpha \times Sub^\alpha \rightarrow Sub^\alpha$, such that $extend(\lambda, \lambda_c)$ is an extension of λ to $vars(\lambda_c)$ (where $vars(\lambda) \subseteq vars(\lambda_c)$) which is consistent with λ_c.

From the user point of view, it is sufficient to specify the particular abstract domain desired. This information is passed to the fixpoint algorithm, which in turn calls the appropriate abstract functions for the given abstract domain. The definitions of the abstract functions for the domains we have studied can be found in [Muthukumar and Hermenegildo 1991; 1992] for the Sharing and Sharing+Freeness domains, and are derived from the unification algorithm of [Codish et al. 1991] for the ASub Domain.

6.2 Local Analysis

A definition of local analysis more general than that of Example 3.6, can be given in terms of abstract interpretation. For this, we use the abstract functions over

related to the level of detail at which the analysis information is desired.

the abstract domains of global analysis. Let Sub^α be an abstract domain. The functions used are, in addition to those defined above, the following:

—$call\_to\_success\_builtin : Sub^\alpha \times Literal \rightarrow Sub^\alpha$, such that the output of $call\_to\_success\_builtin(\lambda, g)$ is the success substitution of built-in g with call substitution λ;

—$top\_exit\_value : Sub^\alpha \times Literal \rightarrow Sub^\alpha$, such that $top\_exit\_value(\lambda, g)$ (where $vars(\lambda) \subseteq vars(g)$) is the "topmost" substitution in the lattice $(Sub^\alpha, \sqsubseteq)$ such that $\lambda \sqsubseteq top\_exit\_value(\lambda, g)$, but it still preserves the information of λ which is downwards closed (i.e., which is preserved in forward computations), and $vars(top\_exit\_value(\lambda, g)) \subseteq vars(g)$.

These functions are domain dependent and are part of the interface for abstract domains to the domain-independent fixpoint computation framework of PLAI. Essentially, local analysis proceeds by starting from a "safe" abstract substitution at the entry of the clause and progressing left-to-right through the clause body literals (but without going into the procedures that are being called). The entry substitution used is the "topmost" substitution for the variables in the clause head plus the abstraction of the empty substitution for the free variables of the body. For each literal, if it is a built-in, its analysis is left to the abstract domain function for built-ins. Otherwise, a "topmost" abstract substitution for the literal, which is coherent with the call abstract substitution of this literal, is taken. This is also left to a domain-dependent function.

More formally, given a clause $C \equiv h\text{:-}B$, where $B = \langle g_1, \ldots, g_n \rangle$, local analysis is the result of a function $local\_analysis(h, B) = \{\lambda_1, \ldots, \lambda_{n+1}\}$, where each λ_i is the abstract call substitution of each g_i, and λ_{n+1} is the success substitution of g_n. This function is defined as follows:

$$local\_analysis(h, B) = \{\lambda_1\} \cup local\_entry\_to\_exit(\lambda_1, B)$$
$$\text{where } \lambda_1 = call\_to\_entry(top\_exit\_value(bottom, h), h, h, B)$$

$$local\_entry\_to\_exit(\lambda, B) = \begin{cases} \emptyset & \text{if } B = \epsilon \\ \{\lambda_2\} \cup local\_entry\_to\_exit(\lambda_2, \langle g_2, \ldots, g_n \rangle) & \\ & \text{if } B = \langle g_1, \ldots, g_n \rangle \end{cases}$$
$$\text{where } \lambda_2 = local\_body\_goal(\lambda, g_1)$$

$$local\_body\_goal(\lambda, g) = extend(local\_call\_to\_success(\lambda', g), \lambda')$$
$$\text{where } \lambda' = project(\lambda, g)$$

$$local\_call\_to\_success(\lambda, g) = \begin{cases} call\_to\_success\_builtin(\lambda, g) & \text{if } g \text{ is a built-in} \\ top\_exit\_value(\lambda, g) & \text{otherwise} \end{cases}$$

Although the above definition is itself domain independent, the results achievable heavily rely on the abstract domain used. A suitable domain which yields similar results to the first definition given in Example 3.6 for strict independence is the Sharing+Freeness domain. In fact, with the Sharing+Freeness domain better results are obtained, as it is able to handle propagation of properties (see Section 5), which is not the case in the first presentation of local analysis.

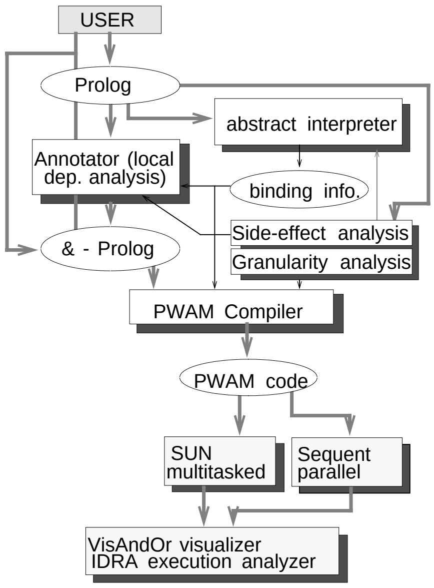

Fig. 7. Evaluation system.

7. EVALUATION ENVIRONMENT

In this section we will briefly describe the &-Prolog system [Hermenegildo and Greene 1991] in which our evaluation has been performed. The run-time system of &-Prolog can generate traces of the parallel execution of programs, which can then be used to visualize or analyze such executions. A tool for analyzing the parallelism available in a program from such traces, IDRA, is also described.

7.1 The &-Prolog System

This system comprises a parallelizing compiler aimed at uncovering goal-level, restricted (i.e., fork and join) independent and-parallelism and an execution model/ run-time system aimed at exploiting such parallelism. The run-time system is based on the Parallel WAM (PWAM) model, an extension of RAP-WAM [Hermenegildo 1986a; 1986b], itself an extension of the Warren Abstract Machine (WAM) [Warren 1983]. It is a complete Prolog system, based on the SICStus Prolog implementation, offering full compatibility with the DECsystem-20/Quintus Prolog ("Edinburgh") standard. In addition, the &-Prolog language extensions provide basic facilities for expressing parallelism at the source level. Prolog code is parallelized automatically by the compiler, in a user-transparent way (except for the increase in performance). Compiler switches determine whether or not code will be parallelized and through which type of analysis. Alternatively, parallel code can be written by the user, the compiler then checking such code for correctness.

As shown in Figure 7, the &-Prolog parallelizing compiler is composed of several basic modules which correspond directly to the steps of our parallelization methodology: global (and local) analyzers inferring information that is useful for the detection of independence, side-effect and granularity analyzers inferring information which can yield the sequentialization of independent goals for reasons of efficiency or maintenance of observable behavior, annotators which parallelize the Prolog programs using the information provided by the analyzers, etc. The parallelized programs can be executed within the run-time system using one or more processors.

7.2 IDRA: Ideal Resource Allocation

As mentioned before, one of the main aims of this work is to evaluate the usefulness of the information provided by the analyzers using the speedup obtained with respect to the sequential program as the ultimate performance measure. This can be done quite simply by running the parallelized programs in parallel and measuring the speedup obtained. However, this speedup is limited by the number of processors in the system and the quality of the scheduler and thus *does not necessarily provide directly useful information regarding the quality of the annotation per se.* In order to concentrate on the available parallelism itself, it is more desirable to determine the speedups for an ideal scheduler and an unbounded number of processors while still taking into account real execution times for the sequential parts and scheduling overheads. IDRA [Fernández et al. 1996] is an evaluation environment which has been designed for this purpose.

The &-Prolog system can optionally generate a trace file during an execution. This file is an encoded description of the events that occurred during the execution of a parallelized program. Examples of such events are parallel fork, start goal, finish goal, join, etc. Events are labeled with a precise timestamp. Since &-Prolog generates all possible parallel tasks during execution of a parallel program, even if there are only a few (or even one) processor(s) in the system, all possible parallel program graphs, including a fairly good estimate of their exact execution times, can be constructed from this data. IDRA takes as input (a) a real execution trace file of a parallel program run on the &-Prolog system over one processor and (b) the time for the (sequential) execution of the original sequential version of the same program, which is used as the unit of measure (1) in the speedup graphs. With these two inputs, it computes the curve of achievable speedup with respect to an increasing number of processors, and using ideal scheduling.

Note that although such "ideal" parallel execution is essentially a simulation, it uses as data a real trace execution file. Real execution times of sequential segments and all delay times are taken into account (including not only the time spent in creating the agents, distributing the work, etc., but also the interruptions of the operating system, etc.), and therefore it is possible to consider the results as a very good approximation to the *best possible* parallel execution. The approach is similar in spirit to that of AndOrSim [Shen and Hermenegildo 1991], which was shown to produce speedups which closely matched those of the real &-Prolog implementation for the numbers of processors available on the systems in which &-Prolog was run, in that speedups are constructed from data from a real execution. Arguably, the method used in IDRA is potentially more accurate, since the unit of measure in AndOrSim was number of resolutions while IDRA uses actual execution time. In fact, the speedups provided by IDRA do correlate well with those of the actual execution.

Table IV. Performance of IDRA

Benchmark	Query		Number of Processors					
			1	2	3	4	5	6
`ann`	pa(3)	speedup	1	1.81	2.65	3.22	3.81	4.26
		IDRA	1	1.80	2.68	3.28	3.95	4.53
`fib`	pmain(10)	speedup	1	1.81	2.52	3.13	3.87	4.37
		IDRA	1	1.99	2.97	3.95	4.91	5.85
`gfib`	pmain(10)	speedup	1	1.82	2.70	3.38	4.23	4.65
		IDRA	1	1.96	2.89	3.81	4.40	5.04
`mmatrix`	pm(13)	speedup	1	1.93	2.82	3.80	4.61	5.56
		IDRA	1	1.98	2.94	3.90	4.80	5.69
`qsortapp`	pqs(5)	speedup	1	1.76	2.24	2.55	2.77	2.81
		IDRA	1	1.76	2.27	2.58	2.81	3.02

Table IV illustrates this point by comparing the speedups obtained with the &-Prolog system on a Sequent Symmetry multiprocessor for a number of programs[12] with the speedup figures computed by IDRA from traces of a parallelized program which has been run on a single processor. Logically, IDRA speedups should be an upper bound for the real system, and it is the case that the speedups predicted by IDRA are always larger than those achieved by the actual system. It can be observed, however, that in some cases IDRA estimates are quite close to the observed speedups, while in others a slight divergence with the number of processors is observed. This is the case mostly for programs that have very fine granularity, which makes the nonoptimal scheduling of the real system have a more significant impact. This can be observed by comparing the results for the two versions of "fibonacci," one of which has been annotated to perform run-time granularity control and thus creates large grain parallelism. The results for this case are much closer to those of IDRA.

8. EXPERIMENTAL RESULTS

In this section we present the results of the comparison among the five analyzers (abstract domains *and associated abstract functions*) currently embedded in PLAI: ASub, Sharing, Sharing+Freeness, and the combinations of ASub with Sharing and ASub with Sharing+Freeness. The aim is to determine the accuracy and effectiveness of the information provided by the analyzers in their application to automatic program parallelization, as well as the efficiency of the analysis process itself. In the experiments, no global task granularity control is performed, i.e., the compiler parallelizes as many tasks as possible which are identified as (potentially) independent. However, as mentioned before, some limited knowledge about the granularity of the literals, in particular the built-ins, is applied at the local (clause) level. Essentially, the only kind of built-ins allowed to be run in parallel are metacalls in which the called (user) literal can be determined statically. Also, side-effect built-ins and procedures are not parallelized.

It could be argued that the experiments are somewhat weak due to the lack of a "best" parallelization with which to establish the comparisons. However, there are many problems when trying to determine what the optimal parallelization might be. One could think that a parallelization "by hand" performed by the programmer

[12]Benchmarks used in the evaluation will be further described in Section 8.1.

Table V. Benchmark Profiles

Bench.	AgV	MV	Ps	Non	Sim	Mut	Gs
aiakl	4.58	9	7	42	57	0	9
ann	3.17	14	65	43	20	36	73
bid	2.20	7	19	68	31	0	27
boyer	2.36	7	26	73	3	23	29
browse	2.63	5	8	12	62	25	9
deriv	3.70	5	1	0	100	0	1
fib	2.00	6	1	0	100	0	1
graduat	5.00	13	75	83	16	0	85
grammar	2.13	6	6	100	0	0	7
hanoiapp	4.25	9	2	0	100	0	3
mmatrix	3.17	7	3	0	100	0	3
msbib	4.19	30	110	65	35	0	330
occur	3.12	6	4	25	75	0	4
palin	4.18	7	5	20	80	0	7
peephole	3.15	7	26	46	7	46	28
progeom	3.59	9	9	33	66	0	13
qplan	3.18	16	46	39	32	28	51
qsortapp	3.29	7	3	0	100	0	4
query	0.19	6	4	100	0	0	4
rdtok	3.07	7	22	31	27	40	30
read	4.20	13	24	54	12	33	47
tak	7.00	10	1	0	100	0	1
warplan	2.47	7	29	51	31	17	36
witt	4.57	18	77	42	35	22	96
zebra	2.06	25	6	66	33	0	7

would be the best. However, for complex programs (as many of those used in our experiments), it turns out that the automatic parallelization often does better than what we have been able to do by hand in a reasonable amount of time. Furthermore, it is not always easy to prove that a parallelization performed by the programmer is correct. Additionally, such parallelization might not take into account all possible information, as, for example, information regarding the granularity of the goals, since, in general, such information depends on the size of the particular arguments given as input. In fact, the best possible parallelization depends on the input and requires using a run-time simulator which runs the program in all possible parallelization schemes, selecting the best from the results obtained in those real executions. However, for most programs this is not practical. Thus we will not attempt to compare parallelizations against an "ideal," but rather we will perform relative comparisons.

8.1 Benchmark Programs

A wide range of programs has been used as benchmarks.[13] The benchmarks range from very simple (toy) programs to real application programs. Among the former, `bid` computes an opening bid for a bridge hand; `boyer` is the classical theorem

[13]Both system and benchmarks are available either by ftp at `clip.dia.fi.upm.es`, or from `http://www.clip.dia.fi.upm.es`, or by contacting the authors.

prover in Gabriel's benchmarks; `browse` is a parser, also from Gabriel's benchmarks; `deriv` performs symbolic differentiation; `fib` computes the Fibonacci numbers; `grammar` generates/recognizes a small set of English; `hanoiapp` solves the Towers of Hanoi problem (with append); `mmatrix` multiplies two matrices; `occur` checks occurrences of sublists within lists of lists; `palin`, from D.H.D. Warren, recognizes palindrome sentences; `qsortapp` is the quick-sort algorithm (with append); `query` performs simple database queries (from D.H.D. Warren); `tak` computes the Takeuchi function, and `zebra` is the classical puzzle. More elaborate programs are: `aiakl`, which is part of an abstract interpreter developed at SICS (the Swedish Institute for Computer Science); `ann` is the &-Prolog **MEL** annotator; `peephole` is the SB-Prolog peephole optimizer; `progeom` builds a perfect difference set of some given order; `qplan` is the query scheduler of CHAT80; `rdtok` is O'Keefe's public domain Prolog tokeniser; `read` is Warren and O'Keefe's public domain Prolog parser; `warplan`, also from D.H.D. Warren, computes plans for a robot to perform actions in a defined world, and `witt` is a conceptual clustering algorithm developed as an example application by students at UPM. Finally, we have also studied two programs which are applications, currently in use: `graduat` was developed at New Mexico State University and is used to check whether the credits earned by a student are enough for obtaining the degree; `msbib` was developed at UPM and is used to merge and translate into several formats bibliography files.

Table V attempts to provide good insight into the complexity of the benchmarks which should be useful for the interpretation of the results:

—AgV, MV are respectively the average and maximum number of variables in each clause analyzed (dead code is not considered);

—Ps is the total number of predicates analyzed;

—Non, Sim, and Mut are respectively the percentage of predicates which are nonrecursive, simply recursive, and mutually recursive;[14]

—Gs is the total number of different goals solved when analyzing the program, i.e., the total number of syntactically different calls.

The number of variables in a clause affects the complexity of the analyses because the abstract functions greatly depend on the number of variables involved. Note that when abstract unification is performed, the variables of both the subgoal and the head of the clause to be unified have to be considered. Therefore, the number of variables involved in an abstract unification can be greater than the maximum number of variables shown in the table. The number of recursive predicates affects the complexity of the fixpoint algorithm, possibly increasing the number of iterations needed.

8.2 Efficiency Results

Table VI presents the efficiency results in terms of analysis times in seconds (SparcStation 10, 55MHz, HyperSPARC processors, SICStus 2.1, native code). It shows for each benchmark and analyzer the average times out of 10 executions. The compilation times for SICStus 2.1 are also shown for reference. In the following, S (Set

[14]Simply recursive refers to predicates whose recursive cycles only contain that predicate. Mutually recursive refers to predicates whose recursive cycles contain two or more predicates.

Table VI. Analysis Times

Benchmark program	Average of 10 runs					
	SICStus C.	S	P	SF	P×S	P×SF
`aiakl`	0.13	0.20	0.43	0.22	0.32	0.37
`ann`	1.26	19.40	5.54	10.50	16.37	17.68
`bid`	0.32	0.32	0.27	0.36	0.46	0.56
`boyer`	0.79	3.56	1.38	4.17	2.91	3.65
`browse`	0.30	0.13	0.17	0.15	0.21	0.24
`deriv`	0.14	0.06	0.05	0.07	0.09	0.11
`fib`	0.03	0.01	0.01	0.02	0.02	0.02
`graduat`	3.95	47.42	0.97	1.79	17.02	3.72
`grammar`	0.19	0.08	0.05	0.11	0.13	0.18
`hanoiapp`	0.11	0.03	0.03	0.04	0.06	0.07
`mmatrix`	0.05	0.03	0.03	0.03	0.04	0.05
`msbib`	2.01	0.70	2.16	0.90	1.11	1.38
`occur`	0.24	0.04	0.03	0.05	0.06	0.07
`palin`	0.15	2.26	0.23	0.62	0.52	0.67
`peephole`	0.97	5.45	2.54	3.94	7.00	7.45
`progeom`	0.17	0.14	0.13	0.17	0.22	0.27
`qplan`	1.21	1.54	11.52	1.84	2.60	3.36
`qsortapp`	0.06	0.04	0.05	0.05	0.08	0.09
`query`	0.19	0.03	0.02	0.05	0.07	0.12
`rdtok`	0.49	1.93	1.44	2.26	2.14	3.88
`read`	0.81	2.09	1.89	2.35	2.99	3.51
`tak`	0.05	0.02	0.02	0.02	0.02	0.04
`warplan`	0.58	15.71	5.02	8.71	15.74	17.68
`witt`	1.35	1.98	16.24	2.26	2.87	3.42
`zebra`	0.17	0.14	0.10	0.19	0.29	0.42
Arith. mean	0.63	4.13	2.01	1.63	2.93	2.76
Ratio	1.00	6.57	3.20	2.60	4.67	4.39
Geom. mean	0.30	0.38	0.30	0.36	0.49	0.58
Ratio	1.00	1.27	1.00	1.18	1.63	1.92

sharing) denotes the analyzer based on the Sharing domain; P (Pair sharing) denotes the analyzer based on the ASub domain; SF (Set sharing+Freeness) denotes the analyzer based on the Sharing+Freeness domain; and P×S and P×SF denote the analyzers based on the combination of P with S, and P with SF, respectively.

The results in Table VI suggest that the analysis process is reasonably efficient (recall that all the analysis code is written in Prolog). Typically, the analysis takes less than 3 seconds. The longest execution (Sharing for `graduat`) takes 47.42 seconds, which is still not unreasonable, considering the complexity of the benchmark. The analyzers take from only 5% more time than SICStus 2.1 compilation to emulated code, to around 25 times for the most costly. On the average, global analysis takes from the same to 6.57 times more than a straightforward clause to clause compilation of the same code. Some analyses (particularly when using the Sharing domain) are specially costly and skew the arithmetic means. We have observed that this occurs precisely in the cases in which the analysis is not being able to derive accurate information, which generally means that the abstract values being manipulated are very large. This suggests that in actual use of the compiler a bound should be kept on the size of the abstract values, and performance traded for precision (which is probably being lost anyway) in those cases in which such sizes are above a given threshold (this is technically referred to as a "widening" in abstract interpretation).

Regarding the comparison among the different analyzers, it seems difficult to derive a clear pattern of behavior from the results shown in the table. The reason is the high number of parameters involved which, for simplicity, are not included in the table: number of specializations, number of recursive and mutually recursive predicates, number of iterations in each computation, number of variables involved in each abstract unification, etc. However, when the above-mentioned parameters are taken into account, quite interesting conclusions can be derived from Table VI. For example, in the cases in which such parameters have similar values (`bid`, `deriv`, `fib`, `hanoiapp`, `mmatrix`, `occur`, `qsortapp`, and `tak`), the analysis time reflects the relative complexity of the analyzers: the abstract operations of the Sharing+Freeness analysis are more complex than those of Sharing (since it has an additional component), and these in turn are much more complex than those of ASub.

However, in general the trade-offs are much more complex than implied by the complexity of the abstract operations. The important intervening factor is accuracy. An accurate analysis generally computes smaller abstract substitutions, thus reducing the time needed for the abstract operations. Accuracy also greatly affects the fixpoint computation: its absence usually results in more iterations, specializations, etc. This effect can be observed in a number of cases in which the lack of groundness propagation in the ASub analyzer greatly affects efficiency: `aiakl`, `qplan`, `msbib`, and `witt`. In these benchmarks, the total number of iterations within fixpoint computations for ASub is approximately 6.5 times that of the other analyzers, and in the last two benchmarks the number of specializations increases by 2.5 times. Conversely, there are other cases (e.g., `ann`, `boyer`, `graduat`, `palin`, and `warplan`) in which the Sharing or the Sharing+Freeness analyzers take much longer than ASub due to the lack of (accurate) linearity information. In all the cases these shortcomings are alleviated in the corresponding combined domains [Codish et al. 1995], so that the time is less than the expected sum of the times of each original component.

Table VII presents efficiency results in terms of memory consumption. For each benchmark and analyzer it shows the number of kilobytes for the global stack segment (showing the size and number of terms created) and the dynamic database (showing the amount of data asserted, including, e.g., the memo table) created in the process. All measurements have been made disallowing garbage collection during the analysis. Corresponding results for the local stack, choice-point stack, and trail have been obtained but are not included. The reason is that for the local stack the amount of memory used is negligible compared to those of global stack and database (never more than 69 Kbytes, and usually less than 10). Also, the number of choice-points created and backtrackings performed is negligible.

Comparing the time spent and the memory consumed in the analysis process it is clear that one accurately follows the other, i.e., high memory consumption often indicates a long execution time and vice versa. This is especially true when considering the global stack, where most of the memory consumption takes place. It

Table VII. Analysis Memory Usage

Bench. Program	Global Stack					Database				
	S	P	SF	P×S	P×SF	S	P	SF	P×S	P×SF
`aiakl`	139	313	168	240	306	12	17	24	17	29
`ann`	11417	3228	6283	10872	12387	305	238	402	407	545
`bid`	154	133	201	261	366	22	25	39	29	46
`boyer`	2221	752	2774	1810	2432	88	54	131	80	127
`browse`	72	85	86	132	160	12	15	20	17	25
`deriv`	37	33	48	65	85	5	6	9	6	10
`fib`	4	4	8	8	15	2	2	4	3	4
`graduat`	38164	757	1839	17015	5218	325	136	211	164	259
`grammar`	49	31	81	89	130	5	4	8	6	9
`hanoiapp`	19	18	28	35	52	3	3	7	4	8
`mmatrix`	17	16	24	31	43	3	3	5	4	6
`msbib`	703	2288	956	1167	1572	220	895	390	319	494
`occur`	25	22	33	43	59	4	5	8	6	9
`palin`	1551	163	438	369	511	37	10	25	14	23
`peephole`	3341	1599	2524	4957	5570	108	97	152	141	195
`progeom`	88	80	120	155	212	9	11	19	13	22
`qplan`	714	8872	1039	1478	2258	88	233	172	127	224
`qsortapp`	30	26	39	52	69	4	4	7	5	8
`query`	18	13	34	40	93	4	3	6	4	7
`rdtok`	754	513	1039	991	2094	99	71	148	92	172
`read`	1103	958	1410	1899	2520	68	72	120	86	138
`tak`	7	7	15	14	28	5	6	9	6	10
`warplan`	9605	3364	5552	11382	13301	112	89	137	145	193
`witt`	973	12484	1331	1766	2470	108	293	236	148	274
`zebra`	74	43	131	183	335	18	15	39	23	44

is interesting to observe that memory consumed in the database (where the memo table is stored) is almost negligible compared to that of the global stack, and it is heavily related to the number of specializations which occur in each analysis. The fact that the analyzers do not consume much database space has been a big surprise considering the heavy use of the memo table performed during the analysis. Since global stack consumption is quite related to the size of the substitutions each analyzer handles, it can be concluded that the size of the (representations of the) abstract substitutions dominates the consumption of memory (and time) by the analyzers. Nonetheless, the more specializations and fixpoint iterations, the more substitutions the analyzer has to handle, and this, as already mentioned, depends on the accuracy inherent to each domain.

8.3 Effectiveness Results: Static Tests

One way to measure the accuracy and effectiveness of the information provided by analyzers is to count the total number of CGEs obtained, the number of these which are unconditional, and the number of groundness and independence tests in the remaining CGEs, which provides an idea of the overhead introduced in the program. The results are shown in Table VIII and Table IX. Benchmarks have been parallelized using the **MEL** annotator in the following different situations: without any kind of information ("N" in the table), with information from the local analysis

Table VIII. Results for Effectiveness—Static Tests

Bench.	Total CGEs					Uncond. CGEs						
Program	N	L	S	P	SF	N	L	S	P	SF	PxS	PxSF
aiakl	2	2	2	2	2	0	0	0	0	2		
ann	28	14	26	26	12	0	0	0	0	0		
bid	8	6	8	8	5	0	0	3	5	5		
boyer	3	2	3	3	2	0	0	0	0	0		
browse	9	5	5	5	4	0	0	0	0	0		
deriv	5	4	4	4	4	0	0	0	4	4		
fib	1	1	1	1	1	0	1	1	1	1		
graduat	23	7	22	22	6	0	0	2	3	6		
hanoiapp	1	1	1	1	1	0	0	0	0	1		
mmatrix	2	2	2	2	2	0	0	0	2	2		
msbib	30	18	24	24	17	0	1	2	2	17	4	
occur	3	3	2	2	2	0	1	1	2	2		
palin	2	1	2	2	1	0	0	0	0	0		1
peephole	11	2	11	11	2	0	0	1	1	1		
qplan	31	20	31	31	18	0	0	3	3	16	6	
qsortapp	1	1	1	1	1	0	0	0	0	1		
read	2	1	2	2	1	0	0	1	1	1		
tak	1	1	1	1	1	0	1	0	0	1		
warplan	16	11	14	14	9	0	1	0	0	1		
witt	39	24	39	39	24	0	2	5	5	22	11	
zebra	4	3	3	3	2	0	1	1	1	1		

("L"), and with that provided by each of the global analyzers. The results for the combined analyzers are in all but five cases as good as those for the best of the analyzers combined. Only the exceptions are shown in the tables. Note that to obtain the results we *inhibited local analysis completely when using global analysis in order to measure the power of the global analyzers alone* (in practice either only local or both types of analysis would be enabled).

To have an overall idea of the effect of each of the analyzers, Table X shows the arithmetic means of the number of (conditional and unconditional) CGEs (C), the number of unconditional CGEs (U), the fraction of unconditional CGEs (U/C), the number of ground and independence checks (G and I), and the fraction of ground and independence checks with respect to conditional CGEs (G/(C-U) and I/(C-U), where C-U gives the number of conditional CGEs).

Regarding the effectiveness of the information inferred by each analyzer, there are two key issues to be studied: whether the results of the analysis are effective in eliminating CGEs which have a test that will always fail, and whether they are effective in eliminating tests that will always succeed, in other words, whether the analysis information helps in reducing the overhead introduced to detect both the absence and presence of parallelism. With respect to the first point, Tables VIII and IX show that definite nongroundness and definite sharing, achieved in the case of the Sharing+Freeness analysis due to the combination of sharing and freeness, is quite effective. While Sharing and ASub can only help in eliminating CGEs by identifying dead code (which is not parallelized) the local analysis (unable to detect dead code) is able to eliminate more CGEs than either Sharing or ASub

Table IX. Results for Effectiveness—Static Tests

Bench.	Conditions: ground/indep						
Program	N	L	S	P	SF	PxS	PxSF
`aiakl`	7/5	0/10	5/0	5/0	0/0		
`ann`	76/129	14/36	60/38	60/19	6/14	60/18	
`bid`	9/22	7/12	5/7	5/0	0/0		
`boyer`	5/4	4/2	5/1	5/0	4/1		4/0
`browse`	9/25	3/9	4/3	4/3	2/2		
`deriv`	5/16	4/16	0/4	0/0	0/0		
`fib`	0/4	0/0	0/0	0/0	0/0		
`graduat`	47/122	7/51	35/45	35/0	0/0		
`hanoiapp`	7/0	2/1	3/0	3/0	0/0		
`mmatrix`	2/8	2/8	0/2	0/0	0/0		
`msbib`	90/160	4/35	64/7	67/8	0/0	/0	
`occur`	2/9	2/5	0/1	0/0	0/0		
`palin`	4/7	0/4	4/5	4/0	0/1		0/0
`peephole`	23/13	3/4	14/10	14/6	1/2		
`qplan`	62/196	13/57	53/7	61/42	2/1	53/1	
`qsortapp`	5/1	0/1	4/0	4/0	0/0		
`read`	2/7	1/6	1/0	1/0	0/0		
`tak`	6/6	0/0	3/0	3/0	0/0		
`warplan`	28/22	14/11	25/15	25/11	11/7		
`witt`	107/287	20/135	64/24	98/43	0/2	64/4	
`zebra`	8/221	5/251	2/7	2/6	0/6		

Table X. Results for Effectiveness—Static Tests—Arithmetic Mean

	Arithmetic Means						
Analysis	C	U	U/C	G	G/(C-U)	I	I/(C-U)
N	10.57	0.00	0.00	24.00	2.27	60.19	5.69
L	6.14	0.38	0.06	5.00	0.87	31.14	5.40
S	9.71	0.95	0.10	16.71	1.91	8.38	0.96
P	9.71	1.43	0.15	18.86	2.28	6.57	0.79
SF	5.57	4.00	0.72	1.24	0.79	1.71	1.09
PxS	5.44	2.88	0.53	9.29	3.61	2.00	0.78
PxSF	5.44	4.33	0.80	1.24	1.11	1.62	1.46

in a fair number of cases: `ann`, `bid`, `boyer`, `graduat`, `msbib`, `peephole`, `qplan`, `warplan`, `witt`. Sharing+Freeness proves to be the most accurate at this task, giving always the least number of CGEs. We would like to point out that although some elimination of CGEs was expected at the beginning of the study, the actual impact of the results of this type of analysis is quite surprising: the Sharing+Freeness analysis can reduce the number of CGEs in 18 out of 25 benchmarks (the complete set used), and the reduction is often of half or more of the CGEs created without analysis.

However, it is when considering the simplification of the conditions in the CGEs that global analysis shows its power: even in the cases where Sharing or ASub have to deal with more CGEs than the local analysis, the total number of tests is usually less. Regarding the comparison among the different global analyzers, the

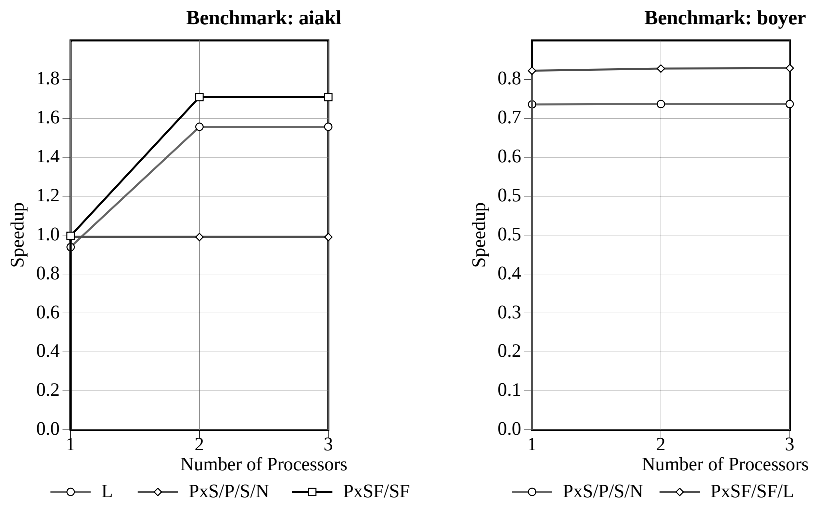

Fig. 8. Effectiveness of global analysis. Dynamic tests: No/low speedups.

results show that ASub often performs better than Sharing (at least for independence checks, though not for groundness) and that Sharing+Freeness is the best by far in terms of effectiveness. This can sometimes come at a cost in analysis time. However, in larger programs, even analysis time tends to favor Sharing+Freeness because, despite the more complex abstract operations, the added precision tends to reduce the size of the abstract values and the number of fixpoint iterations. The effectiveness of Sharing+Freeness can be surprising when contrasted with the fact that the sharing information provided by ASub is usually more accurate than that of Sharing+Freeness, as shown in [Codish et al. 1995]. This apparent contradiction is clarified when considering the link between the groundness and independence information provided by the *set*-sharing information, already pointed out when translating this information into the *GI* domain, which allows the annotators to significantly simplify the tests for parallelization.

The combined analyzers always obtain at most the same number of tests as those of the best of the analyzers combined, and, in a few cases, slightly better results are obtained. The number of tests obtained by ASub is reduced when combined with Sharing in four cases: `ann`, `msbib`, `qplan`, and `witt`. This is not surprising, since the last two are in the class of programs for which ASub loses information. In the cases of `ann` and `msbib`, the advantage is due to the ability of the Sharing domain to infer independence of two variables from the independence of others, an ability which ASub lacks. The number of tests obtained by ASub is reduced when combined with Sharing+Freeness in two cases: `palin` and `boyer`. In both of them an independence check is eliminated, thanks to the more accurate linearity information provided by ASub.

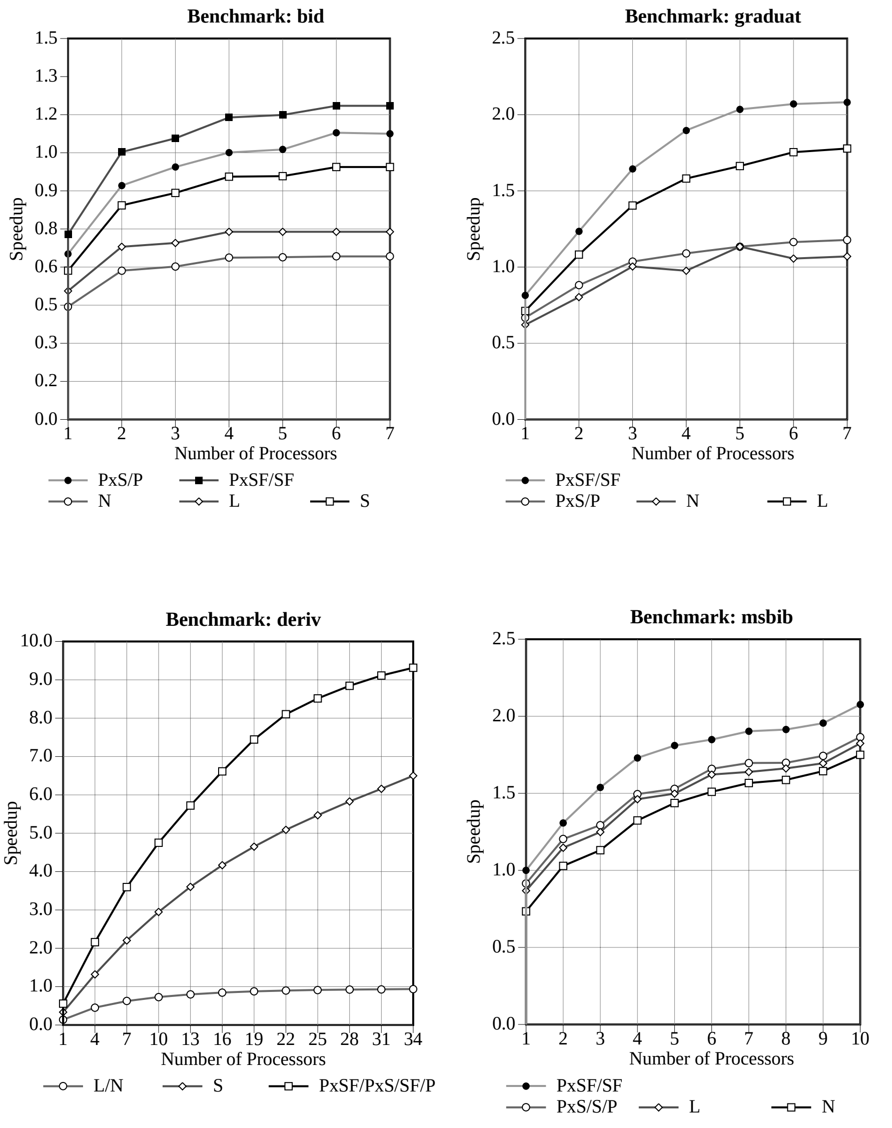

Fig. 9. Effectiveness of global analysis. Dynamic tests: Low/medium speedups.

8.4 Effectiveness Results: Dynamic Tests

An arguably better way of measuring the effectiveness of the annotators is to measure the speedup achieved: the ratio of the parallel execution time of the program (ideally for an unbounded number of processors) to that of the sequential program. This ideal parallel execution time has been obtained using the simulation tools described in Section 7. Because of the computational cost of the measurement

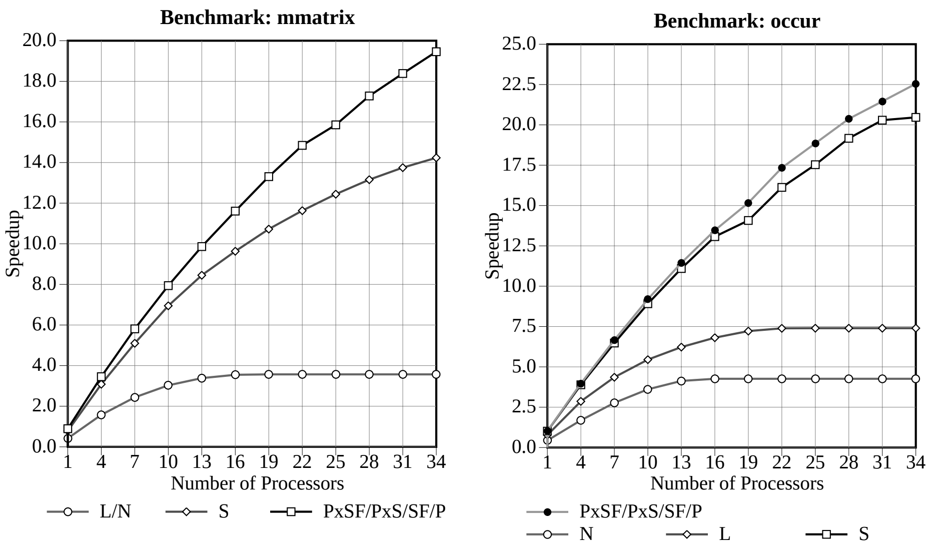

Fig. 10. Effectiveness of global analysis. Dynamic tests: Good speedups.

environment used, the programs have been run on quite small data, and thus the speedups obtained will be, expectedly, low: speedups are highly dependent on input data size in most of the considered benchmarks, and they can often be increased or decreased almost arbitrarily by making the data set larger or smaller. Note, however, that we are not interested really in the maximum speedups obtained. Instead, our real interest lies in the ratios between the resulting speedups with and without abstract interpretation, and, within the latter, the relative performance of the different domains. Furthermore, we are also interested in observing how effective global analysis is in reducing the overhead due to run-time independence checking when using conditional parallelization algorithms (such as **MEL** or **CDG**). One of the disadvantages of conditional parallelization is that the run-time overhead can actually lead to slowdowns specially when running on one processor. We would like to observe to what extent such slowdowns can be reduced by applying information obtained from global analysis. Note that slowdowns can be easily avoided by simply using an annotator, such as **UDG**, which only generates unconditional CGEs—in fact, **UDG** has been shown to achieve quite reasonable speedups in practice, while guaranteeing no slowdowns [Bueno et al. 1994; Bueno Carrillo 1994]. However, for the reasons mentioned above, we will present mainly results for conditional parallelization using, in most cases, the **MEL** annotator.

Dynamic results for a representative subset of the benchmarks used are presented in Figures 8–12. For each benchmark a diagram with speedup curves obtained with IDRA is shown. Each curve represents the speedup achievable for the parallelized version of the program obtained with the **MEL** annotator (unless otherwise stated), in each one of the analysis scenarios studied in the static tests. A curve has been labeled with more than one situation when either the resulting parallelized programs

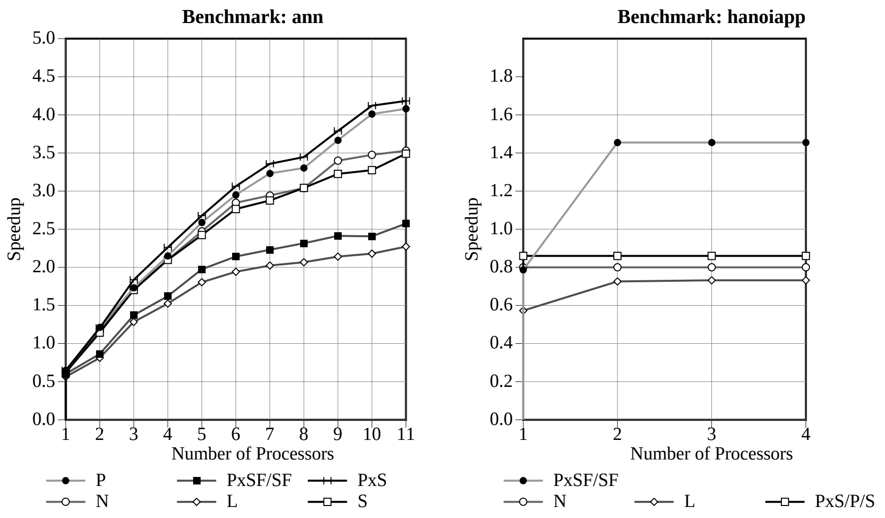

Fig. 11. Effectiveness of global analysis. Dynamic tests: `ann` / `hanoiapp`.

where identical or the differences among the speedups obtained were negligible (i.e., impossible to distinguish by looking at the graph).

The main conclusion from the dynamic tests is that data flow analysis is quite effective in automatic parallelization. This is true not only in terms of reducing the cost of conditional parallelism and allowing for more unconditional parallelism, but also in preserving the sequential performance for nonparallelizable programs.

Regarding the relationship between the accuracy results and the effectiveness results, we can conclude that speedups obtained for a given benchmark generally reflect the accuracy results. Accordingly, the overall results favor the Sharing+Freeness analysis. However, there are exceptions to this. In particular, in the case of `ann`, better results can be observed for all other analyzers except local! The reason for this is interesting: it is due to a particular clause being annotated in two different ways. With most of the analyzers, a CGE with a groundness test is built. The better information obtained by the Sharing+Freeness analysis allows eliminating this groundness test because it will always fail, and a new CGE with a number of independence checks is then built. It turns out that all tests will ultimately fail at run-time. However, with Sharing+Freeness an independence test, which turns out to be much more expensive, is performed. The other analyzers, which are less accurate, do not eliminate the groundness test, which turns out to fail early and thus give better performance. Both ASub and the local analysis perform as well as Sharing+Freeness in some cases. Sharing also behaves sometimes as well as Sharing+Freeness, but in those cases ASub performs well too. Thus, ASub also proves to be quite powerful.

We have observed many other instances in which the results can depend critically on only a few checks out of the large collection that may appear in a par-

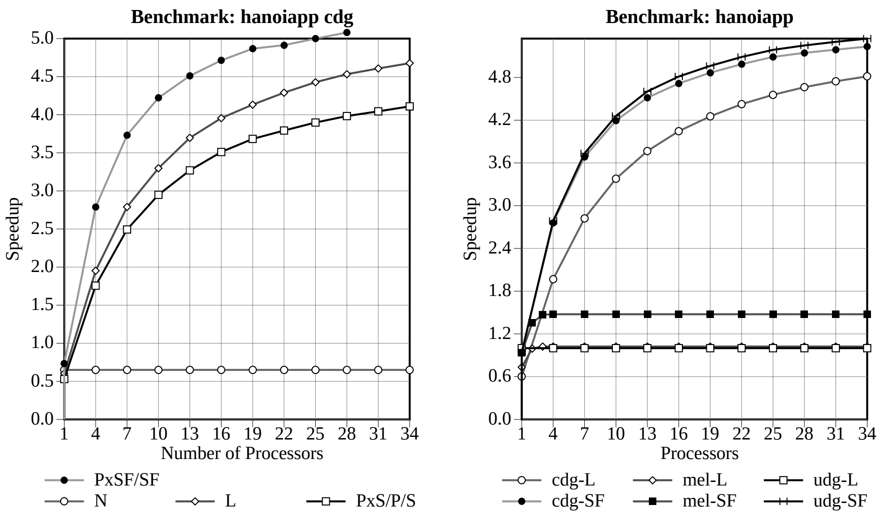

Fig. 12. Effectiveness of global analysis. Dynamic tests: `hanoiapp CDG` and other annotators.

allelized program. In `deriv`, the important differences in speedups are due to only four independence checks. In `occur` a significant difference can be observed between Sharing+Freeness and `no analysis` that is due to only two groundness and four independence checks. And in `mmatrix` the significant difference between Sharing+Freeness and ASub is due to only two independence checks. No significant difference in speedup is observed in `aiakl` and `bid` despite variations of ten independence and five groundness checks respectively.

As mentioned before, studying the trade-offs among the different annotators is beyond the scope of this study, and details can be found in [Bueno et al. 1994; Bueno Carrillo 1994]. However, for completeness, we summarize herein some of the main conclusions of that study: as expected, using **UDG** avoids slowdown situations, which makes it an obvious choice for completely automatic parallelization. For example, using **UDG** the sequential performance is preserved for `boyer` while excellent speedups are obtained for `hanoiapp` (as shown in Figure 12). The obvious drawback is that sometimes unconditional parallelization results in smaller speedups or even no speedup at all in some programs for which conditional parallelization achieves good parallel performance. This was observed, for example, for `ann` in which all CGEs obtained contain at least one test, whose cost is small. As a result, **UDG** obtains no speedups (and, of course, no slowdowns) while the other annotators do produce useful speedups. Interesting differences were also observed between **MEL** and **CDG**, which are illustrated for example in the curves for `hanoiapp` (parallelized using **MEL**) and `hanoiapp-cdg` (parallelized using **CDG**) shown in Figure 12. **MEL** correctly but inefficiently parallelizes a call to hanoi and a call to append, while **CDG** parallelizes a call to hanoi with a sequence composed of the other call to hanoi and a call to append. The latter results in much higher speedups.

9. CONCLUSIONS

We have studied the effectiveness of global analysis in the parallelization of logic programs using *strict independence*. To this end, we have proposed and proved correct a methodology for the application in the parallelization task of the information inferred by abstract interpretation, using a parametric domain. We have then selected a number of well-known approximation domains, explained their translation into the parametric domain, built analyses based on them, embedded such analyses in a complete parallelizing compiler, and studied the performance of the resulting system.

Although we have observed that reasonable speedups can be obtained in some cases using only local analysis, our overall conclusion is that global data flow analysis based on abstract interpretation is indeed a very powerful tool in this application. It allows not only to significantly reduce the number of run-time independence checks and/or increase the amount of unconditionally parallel code, but it is also quite successful at detecting sequential code statically, thus saving any wasteful detection of sequentiality at run-time, for the cases in which conditional parallelization is selected. Global analysis results in greatly increased speedups in most cases if conditional parallelism is selected, and in all cases if unconditional parallelism is selected instead. Global analysis also results in reduced slowdowns when running conditionally parallelized programs on one processor (using the &-Prolog run-time system unconditionally parallel programs typically show essentially the same performance as sequential programs on one processor).

We have also concluded that the cost of global analysis can be reasonable even for quite sophisticated abstract domains. We have observed that the increase in the precision of the inferred information given often has the beneficial effect of reducing the analysis time below that of simpler analyses, specially for larger and more complex programs. We have also observed that large analysis times are related to large abstract values (and, thus, memory consumption), which are in turn related to loss of precision. Thus, it also follows from our results that analyzers should implement a "widening" provision for trading execution time for precision, to be used in case certain abstract value sizes go over a given threshold. This is particularly relevant for the Sharing domain.

With respect to the abstract domains studied, we note the importance in our application of nongroundness and definite sharing information, in addition to possible sharing and groundness. The Sharing+Freeness domain turns out to be quite useful in this sense, offering quite good results in most cases. However, we have also observed that in some cases the results from Sharing+Freeness can be improved by coupling it with the ASub domain, a combination which gives the absolute best results for the domains considered. ASub and Sharing gave reasonable and similar overall results, with a relatively large advantage for one or the other in some cases. Therefore, we can conclude that the "ideal" abstract domain (from the accuracy point of view, and taking into account the domains studied) for detecting strict independence would be one combining at least the following information: set sharing, freeness (or some other form of nongroundness), and any information (such as linearity or a depth-K abstraction) which could improve the sharing information.

It is quite satisfactory to observe that the system built in the context of this study can parallelize automatically programs ranging from small benchmarks to applications. However, we feel that there is still potential for further improvement. In our experience, larger programs tend to make more use of side-effects and sometimes of obscure features of the source language or operating system. A parallelizing compiler, and, especially, its global analysis phase, has to be able to deal correctly and as accurately as possible with these uses. We have addressed previously this problem for the case of logic programs [Bueno et al. 1996] (and many of the solutions proposed are present in the parallelizer used in this study), but this is an area that still requires additional work. Another important avenue for improvement is the exploitation of more advanced notions of independence. One such notion is "nonstrict independence" [Hermenegildo and Rossi 1995] for which we have recently developed automatic parallelization technology based on global analysis [Cabeza and Hermenegildo 1994]. Intuitively, this type of parallelism allows parallelizing procedures that share variables (pointers) by observing that the uses of such shared variables (pointers) do not "interfere." Another avenue for improving results is controlling the sizes of the tasks to be parallelized, which is vital if executing in a distributed environment (see, e.g., [López García et al. 1996] and its references). Another important potential avenue for improvement may be to detect parallelism at other levels of granularity than the goal level used in our study, as suggested in [Hermenegildo and The CLIP Group 1994; Bueno Carrillo 1994; Pontelli et al. 1997; Bueno et al. 1998]. Increased performance may also be obtained by exploiting additionally other types of and-parallelism [Santos-Costa et al. 1990; Gupta et al. 1991; Shen 1996] and or-parallelism [Ali and Karlsson 1990; Lusk et al. 1990]. Finally, we are also working on extending our results to the automatic parallelization of CLP programs, using as a starting point the generalized notions presented in [García de la Banda et al. 1993; García de la Banda 1994; García de la Banda et al. 1996].

ACKNOWLEDGMENTS

Thanks are due to Kalyan Muthukumar, whose original implementation of a fixpoint algorithm for global analysis has evolved throughout the years into the PLAI system, also to John Gallagher, whose implementation of the Tarjan algorithm for computing recursive predicates is the one that PLAI incorporates, and to German Puebla, for his numerous enhancements to the code of PLAI. Finally, we would like to thank the anonymous referees for numerous comments which contributed to enhance the presentation.

REFERENCES

Ali, K. A. M. and Karlsson, R. 1990. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming* (October 1990), pp. 757–776. MIT Press.

Bacon, D., Graham, S., and Sharp, O. 1994. Compiler Transformations for High-Performance Computing. *Computing Surveys 26*, 4 (December), 345–420.

Biswas, P., Su, S., and Yun, D. 1988. A Scalable Abstract Machine Model to Support Limited-OR/Restricted AND Parallelism in Logic Programs. In *Fifth International Conference and Symposium on Logic Programming* (Seattle,Washington, 1988), pp. 1160–1179.

Bruynooghe, M. 1991. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming 10*, 91–124.

Bruynooghe, M. and Janssens, G. 1988. An Instance of Abstract Interpretation Integrating Type and Mode Inference. In *Fifth International Conference and Symposium on Logic Programming* (Seattle, Washington, August 1988), pp. 669–683. MIT Press.

Bueno, F., Cabeza, D., Hermenegildo, M., and Puebla, G. 1996. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, Number 1058 in LNCS (Sweden, April 1996), pp. 108–124. Springer-Verlag.

Bueno, F., García de la Banda, M., and Hermenegildo, M. 1994. A Comparative Study of Methods for Automatic Compile-time Parallelization of Logic Programs. In *First International Symposium on Parallel Symbolic Computation* (September 1994), pp. 63–73. World Scientific Publishing Company.

Bueno, F., Hermenegildo, M., Montanari, U., and Rossi, F. 1998. Partial Order and Contextual Net Semantics for Atomic and Locally Atomic CC Programs. *Science of Computer Programming 30*, 51–82. Special CCP95 Workshop issue.

Bueno Carrillo, F. 1994. *Automatic Optimisation and Parallelisation of Logic Programs through Program Transformation.* Ph. D. thesis, Universidad Politécnica de Madrid (UPM).

Cabeza, D. and Hermenegildo, M. 1994. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In *1994 International Static Analysis Symposium*, Number 864 in LNCS (Namur, Belgium, September 1994), pp. 297–313. Springer-Verlag.

Chang, J.-H., Despain, A. M., and Degroot, D. 1985. And-Parallelism of Logic Programs Based on Static Data Dependency Analysis. In *Compcon Spring '85* (February 1985), pp. 218–225.

Chang, S.-E. and Chiang, Y. P. 1989. Restricted AND-Parallelism Execution Model with Side-Effects. In E. L. Lusk and R. A. Overbeek Eds., *Proceedings of the North American Conference on Logic Programming* (1989), pp. 350–368. MIT Press, Cambridge, MA.

Chassin, J. and Codognet, P. 1994. Parallel Logic Programming Systems. *Computing Surveys 26*, 3 (September), 295–336.

Clark, K. and Gregory, S. 1986. Parlog: Parallel Programming in Logic. *Journal of the ACM 8*, 1–49.

Codish, M., Bruynooghe, M., García de la Banda, M., and Hermenegildo, M. 1997. Exploiting Goal Independence in the Analysis of Logic Programs. *Journal of Logic Programming 32*, 3 (September), 247–261.

Codish, M., Dams, D., and Yardeni, E. 1991. Derivation and Safety of an Abstract Unification Algorithm for Groundness and Aliasing Analysis. In *Eighth International Conference on Logic Programming* (Paris, France, June 1991), pp. 79–96. MIT Press.

Codish, M., Dams, D., and Yardeni, E. 1994. Bottom-up Abstract Interpretation of Logic Programs. *Theoretical Computer Science 124*, 93–125.

Codish, M., Mulkers, A., Bruynooghe, M., García de la Banda, M., and Hermenegildo, M. 1995. Improving Abstract Interpretations by Combining Domains. *ACM Transactions on Programming Languages and Systems 17*, 1 (January), 28–44.

Conery, J. S. 1983. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* Ph. D. thesis, The University of California At Irvine. Technical Report 204.

Cousot, P. and Cousot, R. 1977. 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* (1977), pp. 238–252.

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

Debray, S. Ed. 1992. *Journal of Logic Programming, Special Issue: Abstract Interpretation*, Volume 13(1–2). North-Holland.

Debray, S. K. 1989. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems 11*, 3, 418–450.

DeGroot, D. 1984. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems* (November 1984), pp. 471–478. Tokyo.

DeGroot, D. 1987. Restricted AND-Parallelism and Side-Effects. In *International Symposium on Logic Programming* (August 1987), pp. 80–89. San Francisco: IEEE Computer Society.

Fernández, M., Carro, M., and Hermenegildo, M. 1996. IDRA (IDeal Resource Allocation): Computing Ideal Speedups in Parallel Logic Programming. In *Proceedings of EuroPar'96*, Number 1124 in LNCS (August 1996), pp. 724–734. Springer-Verlag.

García de la Banda, M. 1994. *Independence, Global Analysis, and Parallelism in Dynamically Scheduled Constraint Logic Programming.* Ph. D. thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain.

García de la Banda, M., Bueno, F., and Hermenegildo, M. 1996. Towards Independent And-Parallelism in CLP. In *Programming Languages: Implementation, Logics, and Programs*, Number 1140 in LNCS (Aachen, Germany, September 1996), pp. 77–91. Springer-Verlag.

García de la Banda, M., Hermenegildo, M., and Marriott, K. 1993. Independence in Constraint Logic Programs. In *1993 International Logic Programming Symposium* (October 1993), pp. 130–146. MIT Press, Cambridge, MA.

Getzinger, T. W. 1993. *Abstract Interpretation for the Compile-Time Optimization of Logic Programs.* Ph. D. thesis, University of Southern California, Dept. of Computer Science.

Giacobazzi, R. 1993. *Semantic Aspects of Logic Program Analysis.* Ph. D. thesis, Università degli Studi di Pisa, Dipartimento di Informatica.

Gupta, G. and Costa, V. S. 1992. Cuts and Side-Effects in And-Or Parallel Prolog. *Journal of Logic Programming 27*, 1 (April), 45–71.

Gupta, G. and Jayaraman, B. 1989. Compiled And-Or Parallelism on Shared Memory Multiprocessors. In *1989 North American Conference on Logic Programming* (October 1989), pp. 332–349. MIT Press.

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

Hermenegildo, M. 1986a. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* Ph. D. thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712.

Hermenegildo, M. 1986b. 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 (July 1986), pp. 25–40. Imperial College: Springer-Verlag.

Hermenegildo, M. 1997. Automatic Parallelization of Irregular and Pointer-Based Computations: Perspectives from Logic and Constraint Programming. In *Proceedings of EUROPAR'97*, LNCS (August 1997), pp. 31–46. Springer-Verlag. (invited).

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

Hermenegildo, M. and Rossi, F. 1995. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming 22*, 1, 1–45.

Hermenegildo, M. and The CLIP Group. 1994. Some Methodological Issues in the Design of CIAO - A Generic, Parallel, Concurrent Constraint System. In *Principles and Practice of Constraint Programming*, Number 874 in LNCS (May 1994), pp. 123–133. Springer-Verlag.

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

Jacobs, D. and Langen, A. 1988. Compilation of Logic Programs for Restricted And-Parallelism. In *European Symposium on Programming* (1988), pp. 284–297.

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

JACOBS, D. AND LANGEN, A. 1992. Static Analysis of Logic Programs for Independent And-Parallelism. *Journal of Logic Programming 13*, 2 and 3 (July), 291–314.

KALÉ, L. V. 1987. Completeness and Full Parallelism of Parallel Logic Programming Schemes. In *Fourth IEEE Symposium on Logic Programming* (1987), pp. 125–133. IEEE.

LE CHARLIER, B., DAGIMBE, O., MICHEL, L., AND VAN HENTENRYCK, P. 1993. Optimization Techniques for General Purpose Fixpoint Algorithms: Practical Efficiency for the Abstract Interpretation of Prolog. In *Workshop on Static Analysis* (September 1993), pp. 15–26. Springer-Verlag.

LIN, Y.-J. 1988. *A Parallel Implementation of Logic Programs*. Ph. D. thesis, Dept. of Computer Science, University of Texas at Austin, Austin, Texas 78712.

LIN, Y. J. AND KUMAR, V. 1988. AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results. In *Fifth International Conference and Symposium on Logic Programming* (August 1988), pp. 1123–1141. MIT Press.

LÓPEZ GARCÍA, P., HERMENEGILDO, M., AND DEBRAY, S. K. 1996. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation 22*, 715–734.

LUSK ET AL., E. 1988. The Aurora Or-Parallel Prolog System. In *International Conference on Fifth Generation Computer Systems* (November 1988). Tokyo.

LUSK ET AL., E. 1990. The Aurora Or-Parallel Prolog System. *New Generation Computing 7*, 2,3.

MARIEN, A., JANSSENS, G., MULKERS, A., AND BRUYNOOGHE, M. 1989. The Impact of Abstract Interpretation: an Experiment in Code Generation. In *Sixth International Conference on Logic Programming* (June 1989), pp. 33–47. MIT Press.

MARRIOTT, K., GARCÍA DE LA BANDA, M., AND HERMENEGILDO, M. 1994. Analyzing Logic Programs with Dynamic Scheduling. In *20th. Annual ACM Conf. on Principles of Programming Languages* (January 1994), pp. 240–254. ACM.

MARRIOTT, K. AND SØNDERGAARD, H. 1989. Semantics-based dataflow analysis of logic programs. *Information Processing*, 601–606.

MCALOON, K. AND TRETKOFF, C. 1989. 2LP: A logic programming and linear programming system. Technical Report 1989-21, Brooklyn College of CUNY, CIS department.

MELLISH, C. 1986. Abstract Interpretation of Prolog Programs. In *Third International Conference on Logic Programming*, Number 225 in LNCS (July 1986), pp. 463–475. Springer-Verlag.

MUTHUKUMAR, K., BUENO, F., DE LA BANDA, M. G., AND HERMENEGILDO, M. 1999. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *Journal of Logic Programming 38*, 2 (February), 165–218.

MUTHUKUMAR, K. AND HERMENEGILDO, M. 1989a. Complete and Efficient Methods for Supporting Side Effects in Independent/Restricted And-parallelism. In *1989 International Conference on Logic Programming* (June 1989), pp. 80–101. MIT Press.

MUTHUKUMAR, K. AND HERMENEGILDO, M. 1989b. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming* (October 1989), pp. 166–189. MIT Press.

MUTHUKUMAR, K. AND HERMENEGILDO, M. 1990a. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90 (April), Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759. Available from `http://www.clip.dia.fi.upm.es`.

MUTHUKUMAR, K. AND HERMENEGILDO, M. 1990b. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *Int'l. Conference on Logic Programming* (June 1990), pp. 221–237. MIT Press.

MUTHUKUMAR, K. AND HERMENEGILDO, M. 1991. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming* (June 1991), pp. 49–63. MIT Press.

MUTHUKUMAR, K. AND HERMENEGILDO, M. 1992. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming 13*, 2/3 (July), 315–347.

Originally published as Technical Report FIM 59.1/IA/90, Computer Science Dept, Universidad Politecnica de Madrid, Spain, August 1990.

Pontelli, E., Gupta, G., Pulvirenti, F., and Ferro, A. 1997. Automatic Compile-time Parallelization of Prolog Programs for Dependent And-Parallelism. In *Proc. of the Fourteenth International Conference on Logic Programming* (July 1997), pp. 108–122. MIT Press.

Ramkumar, B. and Kale, L. V. 1989. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *1989 North American Conference on Logic Programming* (October 1989), pp. 313–331. MIT Press.

Santos-Costa, V., Warren, D., and Yang, R. 1990. 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* (April 1990). ACM.

Santos Costa, V. M. 1993. *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I.* Ph. D. thesis, University of Bristol.

Saraswat, V. 1989. *Concurrent Constraint Programming Languages.* Ph. D. thesis, Carnegie Mellon, Pittsburgh. School of Computer Science.

Sarkar, V. (1989). *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Pitman, London.

Sarkar, V. 1990. Instruction Reordering for Fork-Join Parallelism. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Volume 25 (June 1990), pp. 322–336.

Shapiro, E. 1989. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys 21*, 3 (September), 412–510.

Shen, K. 1996. Overview of DASWAM: Exploitation of Dependent And-parallelism. *Journal of Logic Programming 29*, 1–3 (November), 245–293.

Shen, K. and Hermenegildo, M. 1991. A Simulation Study of Or- and Independent And-parallelism. In *International Logic Programming Symposium* (October 1991), pp. 135–151. MIT Press.

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

Szeredi, P. 1989. Performance Analysis of the Aurora Or-Parallel Prolog System. In *1989 North American Conference on Logic Programming* (October 1989). MIT Press.

Taylor, A. 1990. LIPS on a MIPS: Results from a prolog compiler for a RISC. In *1990 International Conference on Logic Programming* (June 1990), pp. 174–189. MIT Press.

Taylor, S., Safra, S., and Shapiro, E. 1987. A parallel implementation of flat concurrent prolog. In E. Shapiro Ed., *Concurrent Prolog: Collected Papers* (Cambridge MA, 1987), pp. 575–604. MIT Press.

Ueda, K. 1987. Guarded Horn Clauses. In E. Shapiro Ed., *Concurrent Prolog: Collected Papers*, pp. 140–156. Cambridge MA: MIT Press.

Van Hentenryck, P. 1989. Parallel Constraint Satisfaction in Logic Programming. In *Sixth International Conference on Logic Programming* (Lisbon, Portugal, June 1989), pp. 165–180. MIT Press.

Van Roy, P. and Despain, A. M. 1990. The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler. In *North American Conference on Logic Programming* (October 1990), pp. 501–515. MIT Press.

Warren, D. 1983. An Abstract Prolog Instruction Set. Technical Report 309, SRI International.

Warren, D. 1987a. OR-Parallel Execution Models of Prolog. In *Proceedings of TAPSOFT '87*, Lecture Notes in Computer Science (March 1987). Springer-Verlag.

Warren, D. 1987b. The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In *International Symposium on Logic Programming* (August 1987), pp. 92–102. San Francisco: IEEE Computer Society.

Warren, R., Hermenegildo, M., and Debray, S. K. 1988. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming* (August 1988), pp. 684–699. MIT Press.

Westphal, H. and Robert, P. 1987. The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism. In *Symp. of Logic Prog.* (August 1987), pp. 436–448.

Received June 1996; revised November 1997; accepted January 1998