Effectiveness of Global Analysis in Strict Independence-Based Automatic Parallelization*

F. Bueno
M. García de la Banda
M. Hermenegildo

Facultad de Informática, UPM
28660-Boadilla del Monte, Madrid - Spain
{bueno,maria,herme}@fi.upm.es

Abstract

This paper presents a study of the effectiveness of global analysis in the parallelization of logic programs using *strict independence*. A number of well-known approximation domains are selected and their usefulness for the application in hand is explained. Also, methods for using the information provided by such domains to improve parallelization are proposed. Local and global analyses are built using these domains and such analyses are embedded in a complete parallelizing compiler. Then, the performance of the domains (and the system in general) is assessed for this application through a number of experiments. We argue that the results offer significant insight into the characteristics of these domains, the demands of the application, and the tradeoffs involved.

1 Introduction

Abstract Interpretation [8] aims at statically – i.e. at compile-time – inferring information about run-time properties of programs. The purpose of the process is generally to use such information to perform optimizations which improve some characteristics of the program or its execution. 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 [27, 24, 26]). This paper contributes to fill this gap, for a particular type of optimization: automatic parallelization of logic programs based on *strict* independence detection (see, for example, [12] and its references).

In a previous study [27, 13], we have reported on a first set of experiments in abstract interpretation-based program parallelization. However, being essentially a feasibility study, that work necessarily had several shortcomings: it included only one domain (a simple depth-K/sharing domain); it

used a relatively simple basic parallelizer and analyzer (for example, no multivariance was implemented); and it presented the results only in terms of program simplifications. Since then, several new parallelization algorithms [20] and abstract analyses (i.e. domains and the associated abstract functions) relevant to the application [25, 5, 16, 22, 21, 6] have been proposed. Furthermore, a complete parallel platform [11], a set of performance evaluation tools [10], and a second-generation analysis framework [22, 23] have become operational.

In this paper we report on the implementation of a wide collection of such analyses (including a simple local analysis), describe their integration into the parallelizing compiler and run-time system, and study their efficiency, accuracy, and effectiveness in program parallelization. We propose algorithms for the non-trivial task of exploiting the information provided by each of the analyses in the application. The information gathered by the analyzers is evaluated not only in terms of its accuracy, i.e. its ability to determine the actual dependencies among the program variables, but also of its effectiveness, measured in terms of code reduction and also in terms of the ultimate performance of the parallelized programs, i.e. the speedup obtained with respect to the sequential version. We argue that our work not only assesses the importance of abstract interpretation in the task of automatic parallelization, but also sheds new light on several subtle characteristics of the domains (which give them their respective power) and on their relationship to the parallelization process itself.

2 The &-Prolog System and Language

The analyses in the study have been integrated into the &–Prolog system [11], which 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. It is a complete Prolog system, based on the SICStus Prolog implementation, offering full compatibility with this system. Prolog code is parallelized automatically by the compiler, in a user-transparent way. Compiler switches determine whether or not code will be parallelized and through which type of analysis.

The &-Prolog language is essentially Prolog, with the addition of the parallel conjunction operator "`&`" (used in place of "," –comma– when goals are to be executed concurrently), a set of parallelism-related builtins, and a number of synchronization primitives which allow expressing both restricted and non-restricted parallelism. Combining these primitives with the Prolog constructs, such as "`->`" (if-then-else), parallel execution of goals can be conditionally triggered. 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. &-Prolog also allows programs to be parallelized manually.

3 Annotation Process

The automatic parallelization process is performed in the &-Prolog system as follows. Firstly, if indicated by the user, the program is analyzed using one or more global analyzers. Secondly, the *annotators* perform a source-to-source transformation (referred to as the *annotation*) of the program in which the body of each clause is annotated with parallel expressions. Such expressions may include run-time tests which encode the notion of independence used. Additionally the annotators can also invoke local (i.e. clause level) analyzers to infer further information regarding the components of a clause.

The annotation process itself is divided into two subtasks. The first aims at identifying dependencies between each two literals in a clause and generating the minimum number of tests which, when evaluated at run-time, ensure the independence of the goals corresponding to such literals. The second task aims at applying a particular strategy to obtain an optimal (under such a strategy) parallel expression among all the possibilities detected previously, hopefully further optimizing the number of tests. In the following we will briefly explain both steps in the particular context of strict independence.

Note that, in general, side–effects cannot be allowed to execute freely in parallel. In order to avoid their parallelization, the annotators use the information derived by the analyzer described in [19] which propagates the side–effect characteristic of builtins yielding side–effect procedures. None of them are parallelized by the current implementation. Also, some limited knowledge regarding the granularity of the goals, in particular the builtins, is used. As a result builtins are not parallelized in general.

3.1 Identifying Dependencies

The presence of dependencies among goals depends directly on the notion of independence used. For concreteness, in this study we use *strict independence*, which has been widely used in the literature (see e.g. [7, 9, 14, 4, 18, 15, 17]). We follow mainly [12]. Two goals g_1 and g_2 are said to be strictly independent for a given substitution θ iff $\text{vars}(g_1\theta) \cap \text{vars}(g_2\theta) = \emptyset$. The definition can be easily extended to a collection of goals and it can also be applied to terms and substitutions without any change.

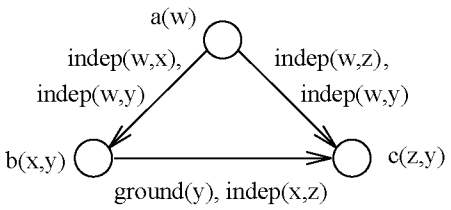
Given a collection of literals, $g_1, \ldots, g_n$, we would like to generate at compile-time a condition *i_cond* which, when evaluated at run-time, guarantees the strict independence of the goals which are instantiations of such literals. Consider conditions including "true", "false", or any set, interpreted as a conjunction, of one or more elements of the form $ground(x)$, 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. Given a set of literals (and no other information) a correct *i_cond* is $\{ground(x)|x \in SVG\} \cup \{indep(x, y)|(x, y) \in SVI\}$, where $SVG = \{v \mid \exists i, j(i \neq j, v \in vars(g_i) \cap vars(g_j))\}$ and $SVI = \{(v, w) \mid v, w \notin SVG, \exists i, j(i < j, v \in vars(g_i), w \in vars(g_j))\}$.

Example 3.1 *Consider the literals* `a(w), b(x,y), c(z,y)`. *Possible sequences of literals that can be considered for parallel execution and their associated i_conds are:*

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)`	{y}	$\{(\mathtt{x},\mathtt{z})\}$	$\{ground(\mathtt{y}), indep(\mathtt{x},\mathtt{z})\}$
`a(w), b(x,y), c(z,y)`	{y}	$\{(\mathtt{w},\mathtt{x}),(\mathtt{w},\mathtt{z}),(\mathtt{x},\mathtt{z})\}$	$\{ground(\mathtt{y}), indep(\mathtt{w},\mathtt{x}), indep(\mathtt{w},\mathtt{z}), indep(\mathtt{x},\mathtt{z})\}$

In general a groundness check is less expensive than an independence check. Thus, replacing independence checks with groundness checks is preferable. Also, note that, for efficiency reasons, we can improve the conditions further by grouping pairs in SVG and SVI.

The dependencies between literals in a clause can be represented as a dependency graph, i.e. a directed acyclic graph where each node represents a literal and each edge represents the dependency between the connected literals. Edges are added following the left-to-right precedence relation given by the clause body. A conditional dependency graph (CDG) is one in which the edges are adorned 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." In our case, we will associate with each edge which connects a pair of literals the tests for their strict independence. The following figure shows the CDG for example 3.1.



3.2 Simplifying Dependencies

The tests generated in the process described above imply the strict independence of the literals for all possible substitutions, thus ensuring that the goals resulting from the instantiations of such literals will also be strictly independent. However, independence only needs to be ensured for those substitutions that can appear in a given program. This observation is instrumental when exploiting the results from static analysis in simplifying dependencies.

The simplification process is based on identifying tests which are ensured to either fail or succeed w.r.t. some information. We propose a method for performing such simplification. 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* GI for groundness and independence:[†] a subset of the first order logical theory, such that each element κ of GI defined over the variables in C is a set of formulae (interpreted as their conjunction) containing only elements of the form $ground(x)$ or $indep(y,z)$, $\{x,y,z\} \subseteq vars(C)$, and

[†]Note that this domain can actually itself also be considered an abstract domain.

such that $\kappa \not\vdash false$, and $\forall \kappa \in GI$: $\kappa \supseteq \{ground(x) \rightarrow indep(x,y) | \{x,y\} \subseteq vars(C)\} \cup \{ground(x) \leftrightarrow indep(x,x) | x \in vars(C)\}$. For the sake of simplicity, in the rest of the paper this formula will be assumed to be part of any κ, although not explicitly written down.

For any program point i of a clause C where a test T_i on the groundness and independence of the clause variables is checked, the simplification of such test, based on an element $\kappa_i \in GI$ over the variables of C, is defined as the refinement of T_i to yield $T_i' = improve(T_i, \kappa_i)$, where:

$$improve(T_i, \kappa_i) = \begin{cases} true & if\ \kappa_i \vdash T_i \\ false & if\ \exists t \in T_i, \kappa_i \vdash \neg t \\ \{t\} \cup improve(T_i \setminus \{t\}, \kappa_i \cup \{t\}) & otherwise \end{cases}$$

Note that there is an implicit restriction on the selection of $t \in T_i$ in the above definition of *improve* since the order in which t is selected can influence its result. We will avoid such non deterministic behavior by first selecting groundness conditions (because of their lower cost at run–time), then those which do not appear as the consequent in any atomic formula of κ_i, and then the rest. This will be done following a left-to-right selection rule.

Building this formula includes translating information from the domain used in the analysis to the GI domain and may be non-trivial. In Section 4 we present algorithms for building this formula for each of the domains for global analysis used in our experiments. However, in order to illustrate the dependency simplification process, we now introduce the other type of analysis used in our experiments and the simplification algorithm for it, and apply it to an example.

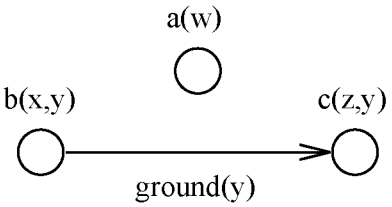
Local Analysis

Local analysis considers each clause in isolation. The information inferred is based on knowledge regarding the semantics of the builtins and the free nature of the first occurrences of variables. This derived information can be directly expressed in terms of elements of the GI domain. Consider a clause C and the set Fv_1 of variables not occurring in $head(C)$. Then $\kappa_1 = \{\neg ground(x) \mid x \in Fv_1\} \cup \{indep(x,y) \mid x \in Fv_1, x \neq y, y \in vars(C)\}$. The analysis proceeds left to right with the g_i goals in the body of C. 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 builtin $\kappa_{i+1} = \kappa_i \setminus (\{\neg ground(x) | x \in vars(g_i)\} \cup \{indep(x,y) | \{x,y\} \cap vars(g_i) \neq \emptyset \wedge \{x,y\} \setminus vars(g_i) \not\subseteq Fv_{i+1}\})$
- if g_i is a builtin, let κ_{g_i} be the representation for the semantics of g_i in GI. Then $\kappa_{i+1} = (\kappa_i \setminus Incons) \cup \kappa_{g_i}$ where $Incons$ is the minimum formula s.t. $\kappa_i \cup \kappa_{g_i} \vdash false$ and $(\kappa_i \setminus Incons) \cup \kappa_{g_i} \not\vdash false$

Example 3.2 *Let us illustrate the simplification of dependencies. Consider the sequence of literals in example 3.1, augmented with a builtin:*

`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 executing the builtin. Since this information is downwards closed, the local analysis will be able to derive that this holds not only just after the execution of the builtin, but also at every point in the clause to the right of it. Thus* $\kappa_i = \{ground(x), ground(w)\}$ *for all points* $i > 1$. *The CDG for the same literals of example 3.1 becomes, by applying the improve function with this information, the following one:*



3.3 Building Parallel Expressions

Given a clause, several possible annotations are possible. Different heuristic algorithms implement different strategies to select among all possible parallel expressions for a given clause. The study of such algorithms is beyond the scope of this paper (see [2] for such a study). Herein, and unless otherwise noted, we will use the **MEL** algorithm [20], which was also the one used in previous studies [27, 13]. This algorithm tries to find points in the body of a clause where it can be split into different parallel expressions (i.e. where edges labeled as "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 the right part of the sequence split, and the process continues with the left part. Once an expression has been built, it can be further simplified, unless it is unconditional. Based on the local or global information, the overall condition built by the annotation algorithm can possibly be reduced again.

Example 3.3 *Consider the sequence of literals in example 3.1, augmented with a different builtin:* `y = f(x,z), a(w), b(x,y), c(z,y).` *Now the analysis can derive* $\kappa_i = \{ground(x) \wedge ground(z) \leftrightarrow ground(y)\}$ *for all points* $i > 1$. *Although the CDG does not vary, the annotation of the literals is:* `y = f(x,z), ( ground(y) => a(w) & b(x,y) & c(z,y))`, *the test being simpler than the one in example 3.1 which corresponds to this annotation.*

4 Global Analysis-Based Test Simplification

The analyzers we have studied include the ASub domain of Søndergaard [25] with the abstract functions presented in [5]; the Sharing domain of Jacobs and Langen [16] with the abstract functions presented in [22]; the Sharing+Freeness abstract domain and abstract functions defined in [21], and also the combination of the domains ASub with Sharing and ASub with Sharing+Freeness presented by Codish et al in [6]. They have all been embedded in PLAI [22, 23], one of the components of the &-Prolog system compiler, which is a domain independent analysis framework implemented in Prolog, and based on the model of Bruynooghe [1] with the optimizations described in [22, 23].

4.1 ASub Domain

The domain ASub 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, $\gamma_{ASub} : \mathsf{ASub} \to 2^{Sub}$, 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 $x \not R\ x \Rightarrow linear(x\theta)$.

Consider an abstract substitution $\lambda_i \in \mathsf{ASub}$ for program point i of a clause C. The contents of the corresponding $\kappa_i \in GI$ are as follows:

- $ground(x)$ if $x \in G$
- $indep(x, y)$ if $x \not R\ y$

Note that in this case κ_i does not contain either $\neg ground(x)$ nor $\neg indep(x, y)$ for any $\{x, y\} \subseteq vars(C)$, thus no tests in the CDG can ever be reduced to false with only this information.

Example 4.1 *Consider a clause C such that $vars(C) = \{x, y, z, v, w\}$ and an abstract substitution $\lambda = (\{x\}, \{(z, w), (z, v)\})$. The corresponding κ will be: $\{ground(x), indep(y, z), indep(y, w), indep(y, v), indep(w, v)\}$.*

4.2 Sharing Domain

The Sharing domain 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: $occs(\theta, U) = \{x \in dom(\theta) | U \in vars(x\theta)\}$. If $occs(\theta, U) = V$ then θ maps the variables in V to terms which share the variable U. The concretization function $\gamma_{Sharing} : Sharing \to 2^{Sub}$ is defined as $\gamma_{Sharing}(\lambda) = \{\theta \in Sub \mid \forall U \in \mathsf{Var}.\ occs(\theta, U) \in \lambda\}$.

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.

Consider an abstract substitution $\lambda_i \in \mathsf{Sharing}$ for program point i of a clause C. The contents of the corresponding $\kappa_i \in GI$ is as follows:

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

- $ground(x_1) \wedge \ldots \wedge ground(x_n) \rightarrow indep(y,z)$ if $\forall S \in \lambda_i$: if $\{y,z\} \subseteq S$ then $\{x_1,\ldots,x_n\} \cap S \neq \emptyset$
- $indep(x_1,y_1) \wedge \ldots \wedge indep(x_n,y_n) \rightarrow ground(z)$ if $\forall S \in \lambda_i$: if $z \in S$ then $\exists j \in [1,n]$, $\{x_j,y_j\} \subseteq S$
- $indep(x_1,y_1) \wedge \ldots \wedge indep(x_n,y_n) \rightarrow indep(w,z)$ if $\forall S \in \lambda_i$: if $\{w,z\} \subseteq S$ then $\exists j \in [1,n]$, $\{x_j,y_j\} \subseteq S$

Each implication can be derived by eliminating the required sets in λ_i so that the antecedent of the implication holds, and then looking for new $ground(x)$ or $indep(x,y)$ which now become true. As in ASub, no tests in the CDG can ever be reduced to false with only this information.

Example 4.2 *Consider the clause C in which $vars(C) = \{x,y,z,v,w\}$ and the abstract substitution $\lambda = \{\{y\},\{z,w\},\{z,v\}\}$. The corresponding κ will be $\{ground(x), indep(y,z), indep(y,w), indep(y,v), indep(w,v), ground(z) \leftrightarrow ground(w) \wedge ground(v), indep(z,v) \wedge indep(z,w) \rightarrow ground(z), indep(z,v) \rightarrow ground(v), indep(z,w) \rightarrow ground(w)\}$. Note that κ contains all the information derived in example 4.1 plus that provided by the power of the set sharing information regarding groundness propagation.*

4.3 Sharing+Freeness Domain

The Sharing+Freeness domain aims at inferring *groundness*, *sharing*, and *freeness* information. It combines 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 $\gamma_{Fr}(\lambda_{fr}) = \{\theta \in Sub \mid \forall x \in \mathsf{PVar} : \text{if } x \in \lambda_{fr} \text{ then } free(x\theta)\}$.

Consider an abstract substitution $\lambda_i \in$ Sharing+Freeness for program point i of clause C. Then κ_i is formed by the following tests, in addition to those for $\lambda_{sh} \in$ Sharing presented in the previous section:

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

The intuition behind each implication is as before. The main difference is that now updating the abstraction λ_i for the antecedent to hold can create an "incoherent" abstraction. In this case κ_i allows the simplification of conditions which will always fail. This is an extra gained precision in addition to that which comes out of the synergistic interaction between the two components of Sharing+Freeness.

Example 4.3 *Consider the same clause C as in 4.2 and the same sharing component $\lambda_{sh} = \{\{\mathtt{y}\}, \{\mathtt{z}, \mathtt{w}\}, \{\mathtt{z}, \mathtt{v}\}\}$. Consider the freeness component $\lambda_{fr} = \{\mathtt{w}\}$. The corresponding κ will be the result of adding the following formulae to the one obtained in the example above: $\{\neg ground(\mathtt{w}), \neg indep(\mathtt{z}, \mathtt{w})\}$. This information, in addition to that derived by λ_{sh}, makes $\kappa \vdash \neg ground(\mathtt{z})$.*

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 non freeness for non ground variables.

4.4 Combined Domains

As mentioned before, we have also considered the evaluation the analyzers resulting from the combination of the ASub and Sharing and ASub and Sharing+Freeness domains. The information approximated by such domains can be used to simplify the CDG simply by translating the information inferred by each domain into the GI domain, conjoining the resulting κs, and applying the techniques described in previous sections.

5 Experimental Results

A relatively wide range of programs has been used as benchmarks. Due to lack of space, they are not discussed here (see `ftp://clip.dia.fi.upm.es`). Instead, we have selected a representative collection, for which the following table gives (in our view) more insight into the complexity of each of them, useful for the interpretation of the results. Av, 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; S, and M are respectively the percentage of simply and mutually recursive predicates; Gs is the total number of different goals solved in analyzing the program, i.e., the total number of syntactically different calls.

Bench.	Av	Mv	Ps	S	M	Gs
aiakl	4.58	9	7	57	0	9
ann	3.17	14	65	20	36	73
bid	2.20	7	19	31	0	27
boyer	2.36	7	26	3	23	29
browse	2.63	5	8	62	25	9
deriv	3.70	5	1	100	0	1
fib	2.00	6	1	100	0	1
hanoiapp	4.25	9	2	100	0	3
mmatrix	3.17	7	3	100	0	3
occur	3.12	6	4	75	0	4
peephole	3.15	7	26	7	46	28
qplan	3.18	16	46	32	28	51
qsortapp	3.29	7	3	100	0	4
read	4.20	13	24	12	33	47
serialize	4.18	7	5	80	0	7
tak	7.00	10	1	100	0	1
warplan	2.47	7	29	31	17	36
witt	4.57	18	77	35	22	96

5.1 Efficiency Results

The following table presents the efficiency results in terms of analysis times as well as the time needed to compile the benchmark under

SICStus Prolog (Prol. column) in seconds (SparcStation 10, one processor, SICStus 2.1, native code). It shows for each benchmark and analyzer the average times out of ten executions. In the following, S (**s**et sharing) denotes the analyzer based on the Sharing domain, P (**p**air sharing) denotes the analyzer based on the ASub domain, SF (**s**et sharing + **f**reeness) denotes the analyzer based on the Sharing+Freeness domain, and P.S and P.SF denote the analyzers based on the combined domains.

Bench.	Average					
	Prol.	S	P	SF	P.S	P.SF
aiakl	0.17	0.20	0.43	0.22	0.32	0.37
ann	1.76	19.40	5.54	10.50	16.37	17.68
bid	0.46	0.32	0.27	0.36	0.46	0.56
boyer	1.12	3.56	1.38	4.17	2.91	3.65
browse	0.38	0.13	0.17	0.15	0.21	0.24
deriv	0.21	0.06	0.05	0.07	0.09	0.11
fib	0.03	0.01	0.01	0.02	0.02	0.02
hanoiapp	0.11	0.03	0.03	0.04	0.06	0.07
mmatrix	0.07	0.03	0.03	0.03	0.04	0.05
occur	0.34	0.04	0.03	0.05	0.06	0.07
peephole	1.36	5.45	2.54	3.94	7.00	7.45
qplan	1.68	1.54	11.52	1.84	2.60	3.36
qsortapp	0.08	0.04	0.05	0.05	0.08	0.09
read	1.07	2.09	1.89	2.35	2.99	3.51
serialize	0.20	2.26	0.23	0.62	0.52	0.67
tak	0.04	0.02	0.02	0.02	0.02	0.04
warplan	0.80	15.71	5.02	8.71	15.74	17.68
witt	1.86	1.98	16.24	2.26	2.87	3.42

5.2 Effectiveness Results: Static Tests

One way to measure the accuracy and effectiveness of the information provided by abstract interpretation-based analyzers is to count the number of CGEs which actually result in parallelism, 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 benchmarks have been parallelized in the following different situations: without any kind of information (N in the table), with information from the local analysis (L), and with that provided by each of the global analyzers. The results are shown in tables 1 and 2. The results for the combined analyzers are in most cases the same as those for the best of the analyzers being combined. Only the exceptions are shown. Note that to obtain the results we inhibited local analysis so as to measure the power of the global analyzers by themselves.

5.3 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 to that of the sequential program. Since we are interested in the quality of the parallelization process, and not in the characteristics of a particular run-time system, this should ideally be done for an unbounded number of processors and in a controlled environment. Such ideal parallel execution time has been obtained using the simulation tool IDRA [10]. This tool takes as input a real execution trace file of a parallel program run on the &-Prolog system (i.e., an encoded description of the events that

Bench. Program	Total CGEs					Uncond. CGEs						
	N	L	S	P	SF	N	L	S	P	SF	P.S	P.SF
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		
hanoiapp	1	1	1	1	1	0	0	0	0	1		
mmatrix	2	2	2	2	2	0	0	0	2	2		
occur	3	3	2	2	2	0	1	1	2	2		
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		
serialize	2	1	2	2	1	0	0	0	0	0		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	

Table 1: Results for Effectiveness — Static Tests

Bench. Program	Conditions: ground/indep						
	N	L	S	P	SF	P.S	P.SF
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		
hanoiapp	7/0	2/1	3/0	3/0	0/0		
mmatrix	2/8	2/8	0/2	0/0	0/0		
occur	2/9	2/5	0/1	0/0	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		
serialize	4/7	0/4	4/5	4/0	0/1		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	

Table 2: Results for Effectiveness — Static Tests

occurred during such execution) and the time for its sequential execution, and computes the achievable speedup for any number of processors. For the benchmarks used, and up to 10 processors, the results obtained with IDRA and those of the actual implementation are within 5% [10], so only the IDRA results are presented. For larger numbers of processors the speedup of the actual system understandably gets gradually smaller than that computed by IDRA. Unfortunately, the lower speedup also tends to hide the differences

between the different parallelizations. Since this is not due to the parallelization itself, but rather to the characteristics of the particular version of the parallel system and scheduler, we feel that the IDRA results are more illustrative, and have chosen to show them instead. They are given, for a representative subset of the benchmarks, in Figure 1. 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 in one of the situations shown in the static tests. A curve has been labeled with more than one situation when either the resulting parallelized programs were identical or the differences among the speedups obtained were negligible (i.e., impossible to distinguish by looking at the diagram).

6 Discussion and Conclusions

The efficiency results in terms of time required by the analysis suggest that the analysis process is reasonably efficient (all the analysis code is written in Prolog). Typically, the analysis takes less than 2 or 3 seconds and is within the same order of magnitude as the SICStus compilation time (note that the SICStus compiler code is more mature and optimized than our analysis code). The longest execution (Sharing for ann) takes 19.40 seconds which is still reasonable, considering the complexity of the benchmark. When comparing the analysis times for each analyzer, the results appear inconclusive due to the high number of parameters involved which, for simplicity, are not shown: number of specializations, of iterations in each computation, etc.

For example, the above mentioned parameters have similar values in a number of cases represented by bid, deriv, fib, hanoiapp, mmatrix, occur, qsortapp, and tak. In these cases the relative complexity of the analyzers is then clearly reflected in the figures in the table: 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 tradeoffs are much more complex than implied by the complexity of the abstract operations. The important intervening factor is accuracy. An accurate analysis generally produces smaller abstract substitutions and also affects the fixpoint computation by reducing the number of iterations, specializations, etc. This effect can be observed in aiakl, qplan, and witt in which the lack of groundness propagation in the ASub analyzer affects the accuracy. 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, serialize and warplan) in which the Sharing or the Sharing+Freeness analyzers take much longer than ASub due to the lack of (accurate) linearity information. These shortcomings are alleviated in the corresponding combined domains, so that the time is less than the expected sum of the times of each original component.

Memory consumption has also been studied. Although lack of space

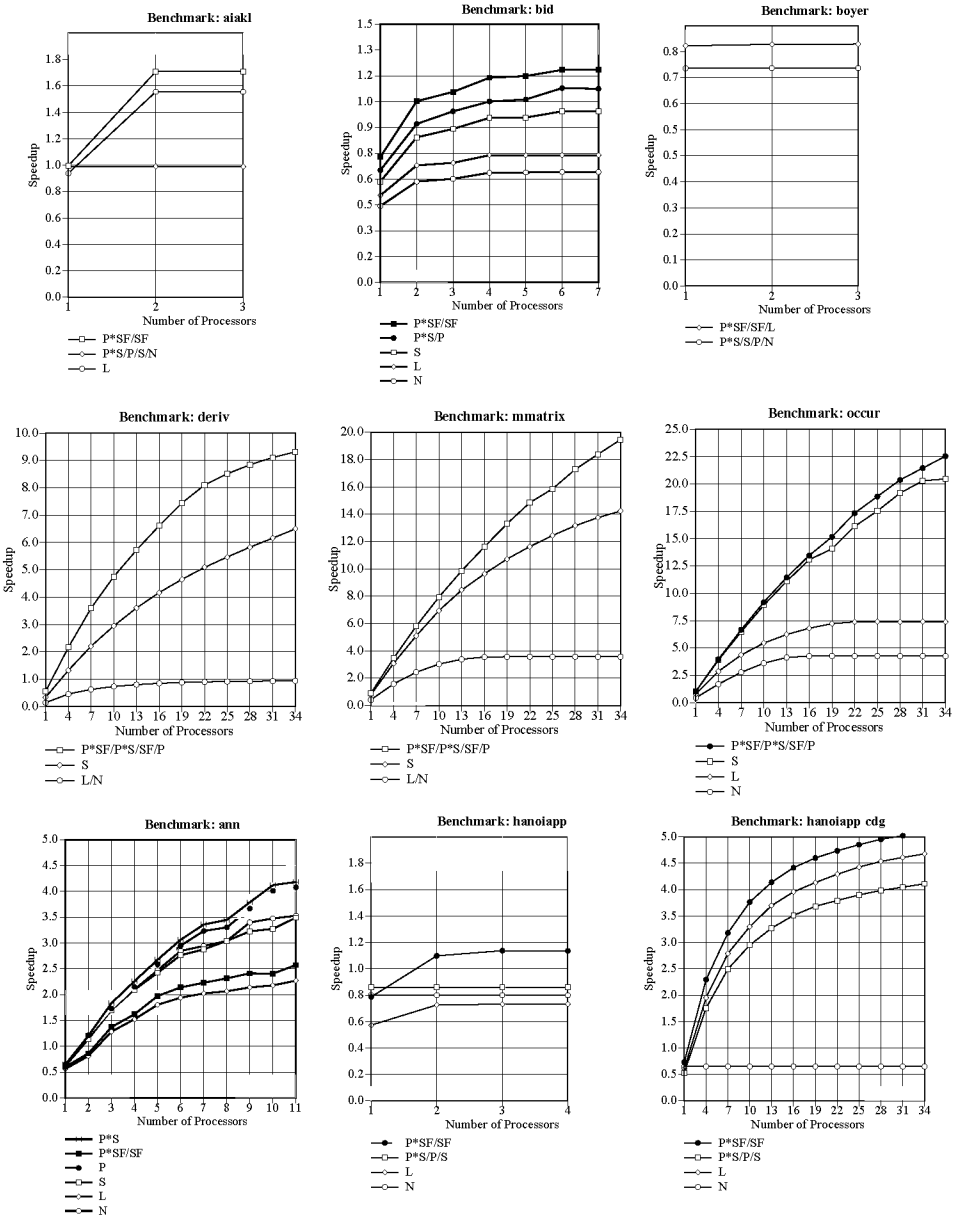


Figure 1: Results for Effectiveness — Dynamic Tests

forces us to omit the results, it is worth noting some points. First, high memory consumption often indicates a long execution time and vice versa. This is specially true when considering the global stack, where most of the memory consumption takes place. The fact that the analyzers do not consume much heap space has been a big surprise considering the heavy use of the database performed during the analysis. Second, local and choice-point stacks and trail consumption is negligible. Since global stack consumption is 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.

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. With respect to the first point, tables 1 and 2 show that definite non–groundness 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 of them in a fair number of cases: ann, bid, boyer, peephole, qplan, warplan, witt. Sharing+Freeness proves to be the most accurate, giving always the least number of CGEs. It is important to note 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 16 out of 23 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, it is clear that ASub is better than Sharing (at least for independence checks, though not for groundness!) and that Sharing+Freeness is the best in terms of accuracy, although often at a cost in analysis time. This can be surprising when noticing that the sharing information provided by ASub is usually more accurate than that of Sharing+Freeness as shown in [6]. This apparent contradiction is solved when considering the amount of 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 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 three cases: ann, qplan, and witt. This is not surprising since the last two are in the class of programs for which ASub loses information. In the case of ann, 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. There are two exceptions for Sharing+Freeness combined with ASub: serialize, and boyer. In both of them an independence check is eliminated, thanks to the more accurate linearity information provided by ASub.

An important conclusion from the study is the importance of non-groundness information in addition to that of sharing and groundness. The Sharing+Freeness domain turns out to be quite sufficient in this sense, offering acceptable results in most cases. However, 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.

Finally, we discuss the effectiveness of the analysis process in terms of actual speedups obtained from the parallelization of the program. Note that the results include the overhead of running the independence checks and thus slow-downs can be observed. The first observation is that speedups obtained for a given benchmark do reflect in some ways the accuracy results. Accordingly, the overall results favor the Sharing+Freeness analysis. This can also be observed in the number of unconditional CGEs. 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 CGE because its test 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 the independence test, which turns out to be much more complex, is performed. The other analyzers, being 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 also does. Thus, ASub also proves to be quite powerful.

Although, as mentioned before, studying the tradeoffs among the different annotators is beyond the scope of this paper, **MEL** and **CDG** offer generally similar results, with an edge for **MEL** when not much information from global analysis is available (this is why it was used in the present study), and some advantage for **CDG** in the converse case. The curves for hanoiapp (parallelized using **MEL**) and hanoiapp-cdg (parallelized using **CDG**) illustrate this point.

A final issue is the importance that a few checks may have. This was already mentioned for ann but is also a factor elsewhere. 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 only due to two groundness and four independence checks. And in mmatrix the significant difference between Sharing+Freeness and ASub is due to only two independence checks. On the other hand in aiakl and bid no significant difference in speedup is observed despite variations of ten independence and five groundness checks respectively.

In summary, the experiments confirm the importance of global data-flow analysis in the parallelization task. Inevitably, also small speedups (or even slow-downs) are obtained for some benchmarks, such as aiakl, bid and boyer due to their lack of parallelism based on strict independence. We hope that using more general independence conditions, such as, for example, non-strict independence [12, 3], will allow us to extract parallelism in even more cases.

References

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

[2] F. Bueno, M. García de la Banda, and M. Hermenegildo. A Comparative Study of Methods for Automatic Compile-time Parallelization of Logic Programs. In *PASCO'94*. World Scientific, September 1994.

[3] D. Cabeza and M. Hermenegildo. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In *Int. Static Analysis Symposium*, Namur, Belgium, September 1994. To appear.

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

[5] M. Codish, D. Dams, and E. Yardeni. Derivation and Safety of an Abstract Unification Algorithm for Groundness and Aliasing Analysis. In *ICLP'91*, pages 79–96, Paris, France, June 1991. MIT Press.

[6] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains. In *ACM PEPM'94*, pages 194–206. ACM, June 1993.

[7] J. S. Conery. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, 1987.

[8] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM POPL'94*, pages 238–252, 1977.

[9] D. DeGroot. Restricted AND-Parallelism. In *Int. Conf. on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

[10] M. J. Fernández, M. Carro, and M. Hermenegildo. IDeal Resource Allocation (IDRA): A Technique for Computing Accurate Ideal Speedups in Parallel Logic Languages. TR FIM26.3/AI/92,C.S. Dept., UPM. 1992.

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

[12] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 1994. To appear.

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