Automatic Compile-Time Parallelization of Prolog Programs for Dependent And-Parallelism

Enrico Pontelli, Gopal Gupta
Laboratory for Logic and DBs,
New Mexico State University
Box 30001,Dept. CS
Las Cruces, NM 88003
{gupta,epontell}@cs.nmsu.edu

Francesco Pulvirenti, Alfredo Ferro
Dipartimento di Matematica
Università di Catania
Città Universitaria
Catania, Italy 95125
{fpulvi,ferro}@alpha.dipmat.unict.it

Abstract

The problem of automatically detecting dependent and-parallelism in Prolog programs has not been studied so far. In this paper we discuss major issues involved in using static analysis to detect dependent and-parallelism in Prolog programs. We present a static analysis technique based on abstract interpretation that detects (fruitful) dependent and-parallelism. Our method makes use of several types of information about the program—sharing and freeness, granularity of subgoals, and binding-times of program variables—computed using abstract interpretation. The information collected from different sources is used to produce a suitable annotation of the original program. This annotation identifies: (i) the most fruitful sources of parallelism; and, (ii) the dependencies that must be respected during execution. A prototype compiler that incorporates these ideas has been implemented and tested on the ACE and-or parallel Prolog system. The (very encouraging) results obtained are reported here.

Keywords: Prolog, Dependent And-Parallelism, Compile-time Analysis

1 Introduction

It is well known that logic programming is an ideal paradigm for programming today's parallel computers. The referential transparency of the logic programming languages, together with the implicit nondeterminism present in their operational semantics, makes them amenable to implicit exploitation of parallelism [13]. By 'implicit' we mean that parallelism can be automatically exploited from programs written for sequential computers without requiring any user intervention. This automatic exploitation of parallelism is done either at runtime or through static analysis at compile-time.

Two major forms of parallelism are generally considered in logic programs. *Or-parallelism (ORP)* is exploited by exploring in parallel the paths created in the SLD tree by multiple clauses matching the same subgoal. This results in multiple parallel computations each of which produces a distinct solution to the original query.

And-parallelism is exploited by concurrently solving subgoals belonging to the same resolvent. It is common to classify and-parallelism further:

- *independent parallelism (IAP):* parallelism is exploited only between subgoals that cannot "affect" (see later) each others' execution [16]. Thus, such subgoals cannot have any common unbound variables that they will compete to bind.
- *dependent parallelism (DAP):* parallelism is allowed between subgoals that can "influence" each other's execution [26].

Logic programs contain substantial parallelism [29]. The main aim behind parallel execution is to achieve better performance, i.e., reduced execution time. However, exploiting the last ounce of parallelism available may have the opposite effect: parallel execution time may actually be more than the sequential execution time, due to the dominance of overhead for parallelism over the actual execution time. To avoid this, the grain size of each parallel task has to be taken into account. There has been a lot of work on estimating grain size of parallel tasks in various contexts (IAP [31, 9], functional programming [17], and imperative languages [20]).

While granularity control appears to be sufficient to tackle the efficiency issues in those forms of parallelism where parallel computations are independent (i.e., IAP and ORP), it is not sufficient to guarantee good execution efficiency of parallelism in the case of DAP. Consider two subgoals p, q. Assume that $\tau(p)$ and $\tau(q)$ are the execution times of p and q, respectively. In case of IAP, if the two subgoals are executed at the same time, we are guaranteed an improvement in execution speed (since the total parallel execution time will be $max(\tau(p), \tau(q))$ which is less than $\tau(p) + \tau(q)$). This is not necessarily true (in general) for DAP. If p and q access (and bind) a common variable, then additional steps will be required in order to guarantee that the whole computation is consistent w.r.t. the common variable. This may require either suspension of computations [26] or undoing of computations in presence of inconsistencies. Thus, in the worst case, a parallel computation of p, q may take more time than the corresponding sequential execution. For example, in an execution model like DDAS [26], which uses suspensions to guarantee the same order of variable bindings as in sequential execution, the worst case will occur when p produces a binding for a common variable at the end of its execution, while q requires the value of the variable immediately at the beginning; this will lead to a parallel execution time $\approx \tau(p) + \tau(q)$, which is the same as the sequential execution time—this clearly comes from the fact that the two computations are inherently sequential, and do not overlap. The actual situation may be worse than this, since we did not take into account the overhead of managing parallel executions (e.g., suspensions/resumptions) and the problems that may come from poor scheduling.

The goal of this work is to design and test static analysis techniques aimed at annotating standard Prolog programs for DAP execution. This involves automatically detecting the best sources of parallelism in the program and trying to rule out negative situations like those described above. The general idea adopted is that of combining sharing+freeness analysis and cost analysis with binding-time analysis, and use the results to verify whether the parallel subgoals have sufficiently large grain and an adequate level of overlapping to justify their execution in parallel. The ideas have been tested in a simple prototypical implementation and tested on a set of benchmarks. By keeping the annotation for parallelism at the level of the source language [15, 26, 24] we also make it possible to verify that a given annotation (e.g., hand-coded by the programmer) is valid and/or is likely to produce efficient execution (and speedups).

The static analysis tool we have developed for annotating Prolog programs for DAP execution rely heavily on existing general purpose tools for static analysis developed by groups at Madrid [14, 22] and Arizona [10]. Thus, our work shows the utility of developing general purpose, modular analysis tools on top of which one can build customized static analysis tools for computing other interesting properties of logic programs.

Our work is by no means a definitive answer to the problem of static detection of dependent and-parallelism in programs, rather it is a first step in this direction (as an example, automatic detection of IAP at compile-time, after many years of study, is still an object of research and refinements). Our work shows that the problem of statically annotating DAP in Prolog programs is solvable, and, in our opinion, cost and binding-time analysis are the basic elements of to solve this problem.

2 ACE Source Annotation Language

The target of the DAP analysis process is to automatically generate an annotated version of the original program that contains appropriate information for *fruitfully* (efficiently) exploiting DAP. The target execution model chosen for DAP is DDAS (Dynamic Dependent And-parallel System) [26], though our ideas will work for most of the execution models for DAP. The DDAS model, in our opinion, represents the most natural generalization of IAP to DAP and the model that is more likely to produce efficient executions (and good support for Prolog semantics). A DDAS program is essentially a Prolog program containing parallel annotations. The parallel annotations are expressed using *Extended Conditional Graph Expressions (ECGE)* [26], which represent a generalization of the concept of CGE, originally designed by DeGroot [11] and refined and improved by Hermenegildo [15]. The purpose of an ECGE is twofold:

1. identify a sequence of subgoals that are meant to be executed (conditionally or unconditionally) in parallel;
2. identify the variables that are common to different parallel subgoals (*shared* or *dependent* variables [24, 26]), as they require special treatment.

Given the goals $\ldots G_1, \ldots, G_n \ldots$, in which the subgoals $G_1, \ldots, G_n$ are to be executed in DAP, the general structure of an ECGE is the following:

$$\ldots, \$mark([X_1, \ldots, X_n]),$$
$$(\langle Cond \rangle \Longrightarrow \$and\_goal(\theta_1, G_1^{\theta_1})\ \&\ \ldots \&\ \$and\_goal(\theta_n, G_n^{\theta_n})), \ldots$$

where:

- $X_1, \ldots, X_n$ are the shared variables for subgoals $G_1, \ldots, G_n$, i.e. all those variables for which different subgoals may attempt conflicting bindings;
- if $X_1^j, \ldots, X_{k_j}^j \subseteq \{X_1, \ldots, X_n\}$ are the shared variables present in the subgoal G_j, then θ_j is a renaming substitution for the variables $X_i^j (1 \leq i \leq k_j)$—i.e., a substitution which replace each X_i^j with a brand new variable.

  In our framework this mapping is described as a sequence of pairs $[X_i^j, X_i^{new(j)}]$, where $X_i^{new(j)}$ is the new variable introduced to replace variable X_i^j.
- $Cond$ is a condition, that will be evaluated at runtime (e.g., for checking groundness, independence, or comparing dynamically computed grain-sizes to thresholds, etc.). The $Cond$ is set to be true and not used in our current analyzer.

For example, a DAP annotated version of the recursive clause in the program for *naive reverse* will look as follows:

```
nrev([X|Xs],  Y) :- $mark([Z]),
        ( $and_goal([[Z,Z1]],nrev(Xs, Z1)) &
          $and_goal([[Z,Z2]],append(Z2, [X], Y)) ).
```

The **$mark/1** is just a directive to the compiler identifying shared variables. The shared variables are given different names in each of the goals, the correspondence between new and the previous names is encoded in $\$and\_goal$.

This format is slightly different (but equivalent) to the ECGE used in the DDAS/DASWAM system. The only additional feature is the variables renaming; this feature has been added to support the novel DAP implementation scheme introduced in the ACE system [24], which requires that each parallel subgoal owns a private access path to each shared variable—the new variables introduced via the renaming pairs are meant to supply a fresh access path.

DDAS' execution model is based on identifying *producer/consumer(s)* of each shared variable. For each shared variable X, at each point of execution one and only one subgoal (the *producer*) is entitled to create a binding for X, while all the other subgoals (*consumers*) are allowed read-only accesses. Whenever a consumer attempts to generate a binding for an unbound shared variable, it is promptly suspended, and its execution delayed until a proper binding for the variable has been created by the producer (or the subgoal itself becomes the producer for that variable, see below).

In DDAS the exact producer goal of a variable's binding is detected dynamically at runtime (hence the name Dyamic Dependent And-parallel System). Since the problem of detecting the exact producer of a given variable V is undecidable, we typically approximate it by considering the leftmost subgoal that has access to V as its current producer. This approximation may need to be refined during the execution (e.g., once it becomes known that the current "producer" is actually not going to bind the variable). The refinement is done by transferring during execution the "role" of producer from one subgoal to another: the next leftmost active subgoal that can access the same variable. This flow of the "right-to-bind" from the left to the right of the resolvent is a clear mimicking of the left-to-right execution model of Prolog, and seems to be the only practical method to obtain efficient DAP execution that respects (sequential) Prolog semantics.

Execution models different from DDAS have been proposed (see [26] for a survey of the various models proposed for DAP), as well as different implementation schemes have been described for DDAS (DASWAM, Attributed-variables based implementations, and ACE). In the rest of this work we will focus exclusively on implementation schemes based on DDAS, as it has been proved, both practically [27, 24] and theoretically [24], that such schemes are superior to other schemes in terms of efficiency and practicality.

3 Dependent And-Parallelism

3.1 DAP: Definition

Formally, we can denote the process of performing one step of resolution (assuming Prolog semantics, i.e., left-to-right exploration of the derivation tree) to reduce a subgoal q in a goal $(q, q_1, \ldots, q_n)$, using a clause $p :- B$ and a unifier μ, as follows:

$$\langle (q, q_1, \ldots, q_n), \theta \rangle \vdash_P \langle (B, q_1, \ldots, q_n)^\mu, \theta \circ \mu \rangle$$

where the notation $\langle R, \theta \rangle$ represents the current resolvent (R) and the current computed substitution (θ) w.r.t. program P. A sequence of n consecutive resolution steps is indicated by $\vdash^n_P$. The computation terminates when no more subgoals are left, i.e. the *terminal* configuration $\langle \Box, \sigma \rangle$ is reached (where $\Box$ denotes an empty resolvent), or when the leftmost subgoal cannot be solved with any of the available clauses (failing derivation). In the rest of this presentation we will be mainly concerned with forward execution.

Two subgoals are considered *independent* if the bindings produced by one of them do not "affect" the computation of the other[1]. More formally, two subgoals p, q are said to be *independent* iff

$$\forall R, \sigma, n\, (\langle p, \emptyset \rangle \vdash^n_P \langle R, \sigma \rangle \rightarrow$$
$$\forall R', \theta, m\, (\langle q, \emptyset \rangle \vdash^m_P \langle R', \theta \rangle \leftrightarrow \langle q^\sigma, \sigma \rangle \vdash^m_P \langle R'^\sigma, \sigma \circ \theta \rangle))$$

and vice versa (i.e., with p and q interchanged). Intuitively, the formula states that if p and q are independent, then the derivation steps taken by q are independent from the bindings which are produced during the derivation of p. Note that $\emptyset$ is used in this paper to denote the empty substitution.

The "classical" example of DAP is represented by a conjunctive goal of the form: $p(X)$, $q(X)$ where the two subgoals "compete" (in parallel) to create the binding for the common (or *shared* or *dependent*) variable X [26, 16]. Dependent subgoals have at least one variable in common, and the binding produced by one of them for such a variable influences the structure of the computation of the other subgoals. Given two goals p, q with a *shared variable* X, then the following holds:

- $X \in vars(p) \cap vars(q)$;
- $\exists\, n, \theta, R$ s.t. $\langle p, \emptyset \rangle \vdash^n_P \langle R, \theta \rangle$ and:
  - $X^\theta \neq X$—If $X^\theta = X$ then the goal p does not produce a binding for X. In such a case p and q can be regarded as independent and executed in parallel without any restriction. This case is known as non-strict independence and-parallelism (NSIAP) [16];
  - the binding for X affects the computation of q, that is

    $$\exists\, m\, (\langle q, \emptyset \rangle \vdash^m_P \langle R', \eta \rangle \wedge \langle q^\theta, \theta \rangle \not\vdash^m_P \langle R'^\theta, \theta \circ \eta \rangle)$$

    that is there is a step in the execution of q where the presence of the binding for X leads to a different action (e.g., selection of a different clause, or failure). This definition of dependence can be proved to be dual of the definition of independence given in [7].

3.2 Implementation of DAP

While it is clear that parallel execution of independent goals is going to give some advantages (modulo parallel overhead), the same cannot be said for DAP. In fact, in a query such as above, p will "produce" a binding for X while q will process (or "consume") it. If this order between production of binding and its consumption is to be preserved, q will be "suspended" until execution of p is over. However, this is not always the case, and execution of p and q can be *overlapped* in certain situations:

1. q may first perform significant amount of computation before it needs the binding of X; this computation can be overlapped with computation of p, because it doesn't depend on X;

[1] For the sake of simplicity, we will focus on data dependencies only. The discussion can be extended in a straightforward manner to include control dependencies due to side-effects. The reader is referred to [16] for a more detailed discussion of the various degrees of independence in logic and constraint programming.

2. p may first partially instantiate X. In such a case q can start working with the partially instantiated value, while p is busy computing the rest of the binding for X.

Thus a DAP computation will be fruitful whenever there is a good amount of "overlapping" between producer and consumers, either because of case 1 or case 2 above.

The aim of a parallel system is not to obtain the maximum possible speedup, but rather achieving the best absolute speed. Parallelism may not always result in improved speed as parallel execution introduces high overheads. This can be rectified by making sure that parallel tasks have large enough granularity. With DAP an additional consideration, as discussed above, is that there should be an overlap between the producer and consumer goals for parallel execution to be fruitful. Taking granularity issues into account, one can say that this overlap should be substantial, otherwise we may not see any improvement in speed. Thus, a DAP system should be able to detect "good" sources of parallelism and identify those shared variables on which fruitful dependent and-parallel computations are centered. Given that computing the set of such shared variables is undecidable in general [24], our task should be to find an approximation to this set. This is indeed our goal in this paper. We use abstract interpretation techniques to compute an approximation to the set of shared variables on which fruitful DAP computations are centered. We call such shared variables *parallelizable shared variables*.

4 Analysis and Annotation for DAP

In this section we will present the general idea and structure behind a static analysis system to produce DAP annotation for an arbitrary Prolog program. Some of the ideas that will be encountered are similar to those that have been adopted in parallelizing systems for IAP [22, 4]. This should come to no surprise: as observed by Hermenegildo [14] and others [34], there is really no such thing as "dependent" parallelism. DAP is nothing else than a finer grain instance of the general principle of independence (where the concept of independence is applied to the level of variable bindings instead of the level of subgoals). This has the great advantage of reusing some of the ideas, and even some of the tools already built for IAP. Nevertheless, transferring these concepts to a finer level of granularity opens a variety of new and very challenging problems, which require the development of new analysis schemes (e.g., analysis of overlapping of goals discussed in Section 3.2).

The problem of analyzing Prolog programs for identifying DAP is relatively unexplored. Substantial effort has been invested in detecting and analyzing IAP [22, 18, 9, 31], however, relatively little work is present in the literature for detecting and analyzing DAP. The only work in this area, to our knowledge, is that by Giacobazzi and Ricci [12], which attempts a bottom-up abstract interpretation to identify pipelined computations. The idea is very interesting and probably can be integrated in our scheme, but taken by itself is very unlikely to produce effective results in a scheme like DDAS. Some similarities are also shared with the various studies on *partitioning techniques* for declarative concurrent languages [32, 19], aimed at identifying partitioning of the program components into sequential threads.

Abstract interpretation techniques are also used in another parallel logic programming system, the Reform Prolog system [1]; nevertheless, in Reform the issue is not that of identifying parallelism (user annotations are used for this purpose), but to guarantee safeness of the execution (deterministic bindings to shared variables). Thus the scope and the problems of static analysis in the Reform system are completely different from what we are considering here.

The approach we have taken can also be seen as an application of the theoretical framework developed by Bueno et al. [21, 3].

A different problem, which shares some similarities to what we are considering in this context, is represented by the management of parallel tasks in committed-choice languages [33].

4.1 Overview

The process of DAP-annotating a program can be intuitively seen as the process of designing a function, $\mathcal{R}$, which given a clause, returns a list of parallelizable shared variables and a partitioning of the body goals into sets of subgoals that can be executed in DAP. Formally,

$$\mathcal{R} : P \longrightarrow \bigcup_{i=1}^{n} \mathcal{P}(\mathcal{P}(Body_i) \times \mathcal{P}(Var_i))$$

where $P = \{Cl_1, Cl_2, \ldots, Cl_n\}$ is the program (seen as a set of clauses), $Cl_i = (H_i\!: -Body_i)$ where $Body_i$ is treated as the set of goals in the body of clause Cl_i, and Var_i denotes the set of variables in the body of the clause Cl_i. Additionally, $\mathcal{R}$ is restricted to compute only partitions of the body $Body_i$ in consecutive groups of subgoals. As mentioned earlier, the function $\mathcal{R}$ is used to split the body of a clause into groups of subgoals which can be profitably run in DAP, identifying at the same time the variables that are shared between the subgoals present in each element of the partition.

The way in which the analysis process is currently organized is based on a successive refinement of the function $\mathcal{R}$. The basic steps of this refinement are as follows:

1. At the beginning, the function $\mathcal{R}_0$ is defined in the most naive way. The body of each clause is simply subdivided into sequences of subgoals of maximal length satisfying the condition that no sequence of length greater than one should contain any subgoal that is a Prolog builtins[2]. Regarding the set of parallelizable shared variables associated with each sequence, the initial approximation is very conservative and consists of all the variables in the clause. At this point we don't take groundness of variables and possible sharings into account.

[2] Only exception to this case is represented by subgoals based on aggregation predicates, like `setof` and `findall`.

2. The first refinement, leading to the function $\mathcal{R}_1$, is an attempt to improve the knowledge about variables shared between subgoals belonging to the same element of the partition. To this end we can rely on the information that can be extracted through an abstract interpretation process. Knowledge about *sharing* and *freeness* has been shown to be instrumental in the process of computing independence of subgoals [22, 4, 23]. Thus, there is no surprise that sharing and freeness information can be directly applied to identify a good approximation to the actual set of variables shared between a set of subgoals. Knowledge about freeness and groundness allows to shrink the set of potentially shared variables, while knowledge about sharing (together with observations on which variables appear in which subgoals) allows to establish variables that definitely need to be included.

3. the second level of refinement, leading to a function $\mathcal{R}_2$ is aimed at improving the partitioning of the clause bodies. The fact that we have a certain number of subgoals which can be potentially executed in DAP does not imply that we really want to execute them in parallel. There are mainly two issues involved:

   (a) We need to guarantee that only those partitions that have dependent goals that *overlap* will be executed in DAP. For this we need to know the instance when a producer binds a shared variable, and the instance when a consumer consumes it. A producer and consumer should be executed in parallel only if this overlap is substantial. We need to perform *binding-time analysis* in order to estimate these instances.

   (b) Given a goal in a parallel conjunction, we need to estimate the size of its execution tree. If this size is too small, then it may be futile to execute such a goal in parallel; the improvement in performance obtained, in fact, is likely to be overcome by the cost of starting the parallel task. In such a case, it should be executed sequentially. We need to perform *cost analysis* to estimate the size of a goal.

   The global effect of the above mentioned analysis is to refine the partition function $\mathcal{R}$. As a result of the granularity analysis various sequences may be split around the lowest cost subgoals (thus a conjunction `(p & q & r & s & t)` may be broken up into `(p & q), r, (s & t)` if it is discovered that `r` is over very small granularity; and a conjunction `(p & q & r)` may be transformed to `(p & (q, r))`. As a result of the binding-time analysis, sequences may be split in order to keep subgoals that do not have a sufficient degree of overlapping in different subsequences. The net result of both these analyses is that the lengths of parallel conjunctions as well as the sets of parallelizable shared variables becomes smaller and smaller (even though the parallel tasks may grow in size—as mentioned before).

4. the existence of sharing and freeness information allows us to infer *modes*, which can be used to perform a slightly modified form of annotation. This modified annotation is accepted by the ACE system. The modified annotation allows us to indicate whether a goal is a producer or a consumer. For example, in the `tp` benchmark, one of the annotated clauses has the form

   ```
   ....
   opposite(Conclusion,Denial),
   add_conjunction(Premise,Denial,fs([],[],[],[])).
   ```

   The sharing and freeness information indicates that: (i) before entering this sequence of subgoals only the variable `Denial` is not ground; and, (ii) the variable `Denial` is ground after the first subgoal (`opposite`).

   Thus, we can easily infer that the goal `opposite` will definitely bind the shared variable to a ground term; consequently `add_conjunction`, in case of suspension, will not have to verify the additional condition which may turn it into a producer, but it will just need to wait for the binding to be produced. The annotation in this case has the following format:

   ```
   $mark([Denial]),
   ( $and_goal([[Denial,Denial1,producer]],
         opposite(Conclusion,Denial1)) &
     $and_goal([[Denial,Denial2,consumer]],
         add_conjunction(Premise,Denial2,fs([],[],[],[])))).
   ```

5. further improvements of $\mathcal{R}$ can be realized by applying some additional steps of analysis. We are currently experimenting with a simple determinacy analyzer to improve detection of DAP. Determinate subgoals are preferable for DAP and, in absence of side-effects, they can be treated always as producers (no need of suspensions).

In the next sections we describe in detail how the analysis for each of the steps above is done in our compiler.

4.2 Local Analysis

As briefly mentioned above, the first step is to perform a local analysis on each clause of the program to identify the clauses that need to be considered for the rest of the analysis. Facts and clauses whose body is composed only by tests/builtins are not considered further for parallelization. For each remaining clause the first approximation of the partition $\mathcal{R}$ is created ($\mathcal{R}_1$). This initial partitioning follows the same general schema suggested by Hermenegildo et al. [22], which uses occurrences of predicates based on builtins as separators between different subsequences of subgoals.

At the same time, for each identified sequence, the analyzer detects the initial set of variables for which annotation may be required (i.e., variables shared between subgoals). Without the help of global analysis, this process will be necessarily very conservative, since we do not yet have any clue about possible sharings between variables. Thus, given a clause

$$H :- B_1, B_2, \ldots, B_n$$

and assuming that the sequence of subgoals is $B_i, B_{i+i}, \ldots, B_j$, the set of variables to be annotated contains (conservatively):

- $T_i = vars(B_i) \cup \ldots \cup vars(B_j)$;
- for $1 \leq k \leq i-1$ we have
$$T_k = T_{k+1} \cup \begin{cases} vars(B_k) & \text{if } T_k \cap vars(B_k) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$
- $T_0 = T_1 \cup vars(H)$.

The intuition is that, lacking global analysis, if two variables X and Y appear in the same subgoal B_k, then we shall assume that they may get aliased.

For example, consider the following clause extracted from a peephole optimizer program[3]:

```
popt11(movreg(T,R), [getstr(S,R)|PilRest], OptList) :-
    (peep_chk(PilRest,R) ->
       OptList = [getstr(S,T)|OptRest];
       not(peep_chk(PilRest,R)),
       popt1(PilRest, OptRest)),
    OptList = [movreg(T,R),getstr(S,R)|OptRest].
```

The annotator will select only one sequence of subgoals of cardinality greater than one, the one composed of: `not(peep_chk(PilRest,R)),popt1(PilRest, OptPilRest)` The others subgoals are followed/preceded by builtin predicates. The set of potentially shared variables is computed as follows (note that $i = 3$ and $1 \leq k \leq 2$ here):

- $T_3 = \{PilRest, R, OptPilRest\}$;
- $T_2 = T_3 \cup \{OptInstList\}$;
- $T_1 = T_2$;
- $T_0 = T_1 \cup \{T, S\}$.

We will see later how this set of variables can be considerably reduced by using global analysis.

4.3 Sharing and Freeness

The system is based, as shown in figure 5, on the presence of an abstract interpreter capable of identifying at each point of the program the sharing and freeness abstraction [23, 6]. A large variety of approaches have been studied to perform detection of sharing and freeness information. Our work is relatively independent from how this abstract interpreter works, as long as the interpreter is capable of computing the value of sharing and freeness at each point in the program (i.e., before and after each subgoal in each clause). Sharing and freeness information is used for many purposes: (i) for computing modes, (ii) for refining the set of parallelizable shared variables, and (iii) for binding-time analysis.

In the current implementation, we have interfaced the PLAI [22] abstract interpreter (part of the CIAO [14] system) developed by the CLIP group in Madrid. This system is designed to produce, among other things, the value of the sharing+freeness abstract substitutions in each point in the program. A simple interface has been designed to enclose this information in an annotated version of the program, in a format suitable to the successive analysis phases.

[3]Part of the Aquarius benchmarks suite.

4.4 Granularity Control

The problem of estimating grain size has been already addressed by various researcher in the context of and-parallelism [9, 10, 31, 30]. Most of these schemes, in order to gain precision, delay the actual granularity test to run-time; compile-time analysis is used to build cost functions [9, 31]. The current format of our DAP annotation can accommodate evaluation of cost functions and comparison to thresholds in the condition part.

Nevertheless, for simplicity in our current system we have selected a simpler approach (that, we later realized, has been briefly mentioned in a previous work by Tick [30]), in which the evaluation of costs is completely achieved during compile-time. Though considerably less accurate (especially in presence of complex recursions), this method has worked satisfactorily in most of the tests performed.

The scheme is based on the following idea: during the first scan of the program (the one during which the previously described local analysis is performed) a dependency graph is created. The graph $G_P = \langle V, E, \mathcal{F} \rangle$ for the program P has a node for each procedure used in the program, and an edge from p/n to q/m whenever a subgoal using q/m appears in the body of a clause whose head uses p/n. $\mathcal{F}$ is a function $\mathcal{F} : E \rightarrow P$ which associates to each edge in the program a label, represented by a clause from P (the clause which generated such edge).

This graph is processed by:

- first identifying the strongly connected components (SCC)s of the graph[4];
- the graph is processed bottom-up, one SCC at the time, and it is translated into a set of inequalities (expressing, intuitively, the fact that the cost of the head of each clause should be $\geq$ than the sum of the costs of the procedures in the body). Given a node $n \in V$, assuming that $Succ_n(c) = \{m \in V | (n,m) \in E \wedge \mathcal{F}(n,m) = c\}$, then then corresponding disequations generated are:
$$cost(n_c) \geq \sum_{q \in Succ_n(c)} cost(q)$$
$$cost(n^{min}) \geq min_{c \in P(n)}\{cost(n_c)\}$$
$$cost(n^{max}) \geq \sum_{c \in P(n)} cost(n_c)$$
$$cost(n^{avg}) \geq \frac{cost(n^{min}) + cost(n^{max})}{2}$$
where $P(n)$ denotes the set of clauses in P whose head is built using the predicate p associated to n. The disequations allow estimation of 3 separate costs for each predicate, a minimum cost, a

[4]The need to work at the level of SCCs has been suggested by Debray [8].

maximum one, and an average. The different estimations are used in the rest of the analysis to deal with borderline cases.

- the disequations are solved (a minimal solution is searched for). The solving of the system is realized by converting the disequations into a $CLP(FD)$ program and feeding it to a finite-domains constraint solver.

The cost estimates are used for deciding for each goal that is put into a parallel conjunction by the local analysis whether or not it should stay there. If the size of a goal is too small, the parallel conjunction is either split at that point, or that goal is absorbed in the next goal. The computed cost estimates are quite approximate, but turn out to be sufficient for dealing with most of the benchmarks considered. Rarely (see section 5) the resulting annotation has been imprecise due to the lack of accuracy of the cost analysis. It is important also to observe that we tried to make the use of the costs (both as granularity control and during the successive binding-time analysis) as conservative as possible—thus, if the estimate lacks accuracy, this will most of the time translate into fewer DAP annotations in the code.

We are currently investigating the usage of different cost analysis techniques, in particular the one suggested by Debray et al. [9, 10]. These techniques are dynamic, in the sense that they produce at compile-time only cost-functions, which are meant to be evaluated at run-time, using term sizes information.

4.5 Binding-time Analysis

In general, it is not obvious, given a set of DAP subgoals $\{G_1, \ldots, G_n\}$, how much parallelism will be actually exploited during the execution. A worst-case situation can arise from subgoals whose clauses have the following format:

```
p(X) :- <Large Computation> X = a.

q(X) :- X = a, <Large Computation>.
```

This produces a computation shown in figure 1. Clearly the amount of parallelism exploited is minimal if not zero.

This problem is specific to DAP and is related to having enough overlapping between producer and consumer to justify their DAP execution, as discussed earlier. Figure 2 illustrates the general case of two subgoals with a common shared variable executing in DAP. Speedups will be possible only if there is some overlapping between the two computations. Also, this overlapping should be of large enough grain size for parallelism to be fruitful, also mentioned earlier. The degree of overlap can be measure by designing an approximation of the time-line of the different parallel subgoals w.r.t. each shared variable (as in figure 2) and use the information obtained from cost analysis to estimate the degree of overlapping between the computations. In the rest of this work we will refer (maybe a bit inappropriately) to this part of the analysis as *binding-time analysis*.

Binding time analysis is based on estimating the number of steps that a subgoal will take before attempting to bind a variable.

4.5.1 The Concrete Semantics

Intuitively the concrete domain should be capable of identifying, at each point of the execution, the binding times for the various variables used in the program.

We need to identify two times for each variable, the time at which the variable has been introduced and that at which the variable is bound. Thus we can define a function that for each variable gives the current lifetime. Let V be the set of all variables in the language. Let B be a subgoal, and let σ be a substitution. The Concrete domain for a single variable is represented by the set $D_c = \mathcal{P}(\boldsymbol{N} \cup \{\infty\})$. We define a function

$$\psi : B_P \times \Sigma \times V \to D_c$$

which, intuitively, assigns to each variable a measure of its binding time inside the computation of a certain subgoal. Let $var(B)$ be the variables in B.

Suppose we have a derivation which starts at a certain goal G^σ. Suppose X is a variable that appears in G^σ. The number of steps performed to bind the variable X in the computation for G is k, computed as follows: if

$$\langle G, \sigma \rangle \vdash_P^k \langle B, \theta \rangle$$

with $\theta(X) \neq X$, and, additionally, $\forall j < k$ we also have that

$$\langle G, \sigma \rangle \vdash_P^j \langle B, \theta' \rangle$$

such that $\theta'(X) = X$ (with respect to the same choice of steps in the derivation). Thus we have that, for a given derivation π, $\psi_\pi(G, \sigma, X) = k$. Regarding the absolute value of $\psi(G, \sigma, X)$, it can be defined as the set of binding times over all possible derivations for $\langle G, \sigma \rangle$: $\psi(G, \sigma, X) = \{\psi_\pi(G, \sigma, X) | \pi \in \Pi\}$ where Π is the set of all derivations for $\langle G, \sigma \rangle$.

Of course, a computation may diverge. This is clearly undecidable, but in that case the concrete semantics assign ∞ to such a derivation.

This is the general concrete semantics that gives the binding-lifetimes of each variable X in a goal G^σ.

4.5.2 The Abstract Domain

We need to select an adequate abstract domain to perform the estimation of the binding time. Again the choice of the domain is dependent on whether we want to give a lower or upper bound to such estimation.

In general, let us consider as abstract domain $D_\alpha = \boldsymbol{N} \cup \{\infty, \perp\}$. This is a lattice (with the assumption $\perp \leq x$ and $x \leq \infty$ for every $x \in D_\alpha$, and given $n_1, n_2 \in \boldsymbol{N}$ then $n_1 \leq n_2$ if the number n_1 is numerically smaller than n_2). This lattice is dangerous for abstract interpretation since it admits infinitely ascending chains, and so we should be careful. To avoid problems, we introduce a cut off point, n_{cut}, in the abstract domain. As we will see later it is possible to come up with a good value for n_{cut} based on the abstract interpretation procedure adopted. Intuitively, binding-times of all variables whose actual binding-times are more than n_{cut} will be approximated by ∞.

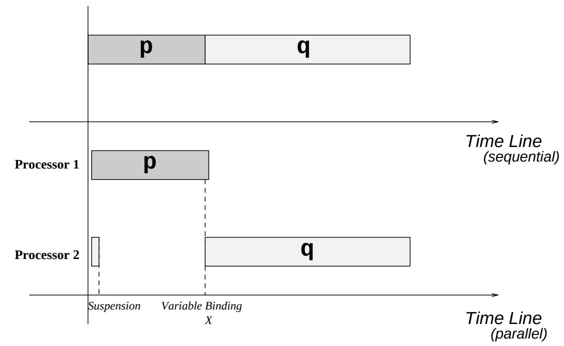

Figure 1: Irregular DAP computation

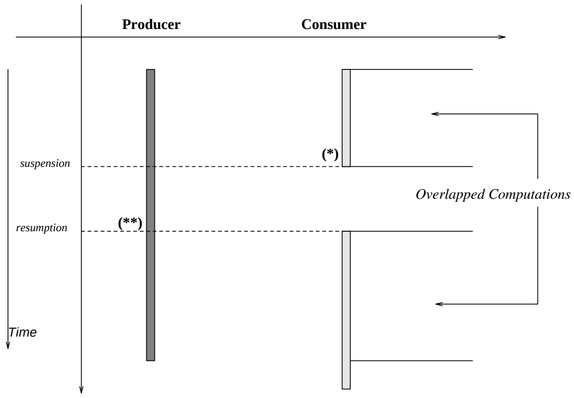

Figure 2: General DAP computation

One observation: in the rest of the discussion we will consider the use of worst-case binding-time for the estimations. The same discussion can be developed with respect to best-case or using averages.

The functions to perform abstraction and concretization are defined as follows:

- the abstraction function $\alpha : D_c \to D_\alpha$

$$\alpha(N) = \begin{cases} m & \text{if } m \leq n_{cut} \text{ where m = max N} \\ \infty & \text{otherwise} \end{cases}$$

  In particular, we have that $\alpha(\emptyset) = \perp$.

- the concretization function $\gamma : D_\alpha \to D_c$

$$\gamma(n) = \begin{cases} \{m | m \leq n\} & \text{if } n \text{ is a successor ordinal} \\ \{m | m < n\} & \text{if } n \text{ ordinal limit} \end{cases}$$

  In particular we have that $\gamma(\perp) = \emptyset$ (empty set).

The relation between abstract and concrete domain can be understood as follows:

- if we take an element in D_α, it is either n, $\perp$, or ∞. Its concretization can lead to one of the following cases:
  - for ∞, the concretization gives the whole set $\boldsymbol{N} \cup \{\infty\}$. Its abstraction (since it definitely contains element over the n_{cut}) will be again ∞.
  - for $\perp$, it gives $\emptyset$. Its abstraction will give basically $\perp$.
  - for n, this must be under the n_{cut}, thus when concretized it will give all elements under n. This abstracted will give n again.
- if we take an element in D_c. It is a set of numbers and/or ∞. Now when we abstract we can obtain:
  - if all elements are under n_{cut}, and the bigger is m, then the abstraction will give m. From here, if we concretize we get all elements under m, which includes all the elements we started from.
  - if there are elements over n_{cut}, then the abstraction gives ∞ which concretized gives the whole domain back.

Thus we proved that this is indeed a Galois insertion. Note that

- α is monotonic (in fact, if we take $S_1 \subseteq S_2$, then we must have $\alpha(S_1) \leq \alpha(S_2)$);
- γ is monotonic (if we take $\xi_1 \leq \xi_2$ it is easy to show that they produce monotonic concretization).

4.5.3 Operational Behaviour

The operational behaviour of our abstract interpretation algorithm uses Bruynooghe's framework. At each point of the execution we want to to compute an abstract substitution which abstracts the binding time at that point.

As illustrated in figure 3 (taken from [2]), various components need to be identified to perform a top-down abstract interpretation:

1. given the substitution δ_X^{init}, which is the initial substitution (on all the variables of the goal), we need to project it on the variables of the call `B` (restrict X to Y), producing δ_Y^r.
2. various clauses can be used to solve the call. The jth clause has head `H`. δ_h^j is the substitution obtained after head unification, projected on the variables of the clause.
3. δ_1, initial substitution for the first subgoal in the body, `B1`, is the same as δ_h^j, except that it may cover the additional variables present in the body of the clause.
4. again, we project δ_1 on the variables of `B1` obtaining the entry call δ_1^r. At the exit we will obtain a substitution δ_1^s, which can be projected back on all the variables in the clause, producing δ_2.
5. at the end we obtain the final substitution δ_{n+1}, which can be restricted to the variables in the head obtaining δ_{succ}^j.

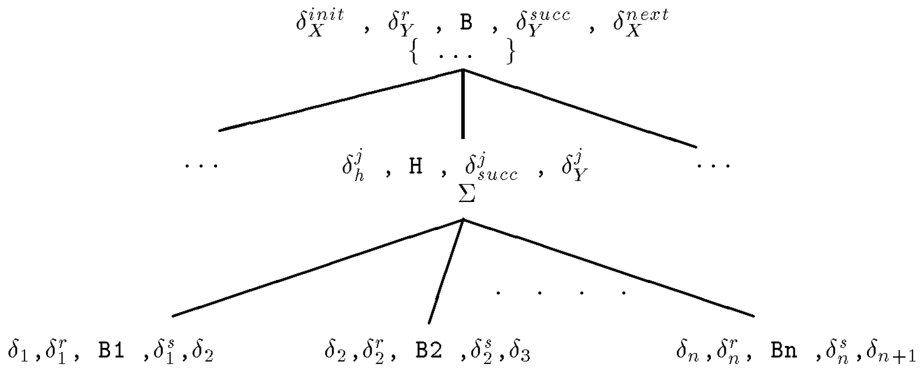

Figure 3: Fragment of abstract interpretation of a call

6. we can perform a back-unification to further restrict the solution to the variables Y in the call, obtaining δ_Y^j.
7. all the solutions coming from the different clauses can be combined to produce the exit substitution δ_Y^{succ}.
8. finally, the complete exit substitution is obtained by extending δ_Y^{succ} back to all the variables X in the goal, producing δ_X^{next}.

In this framework the above described operations are performed as follows. Each abstract substitution maps each variable to an element of D_α in order to describe its binding-time. At the beginning, the initial substitution sees all the variables unbound, and this is described as $\delta_0 = \{X \mapsto \perp | X \in vars(\text{Query})\}$.

- *restriction* of substitutions to specific sets of variables does not require any special action;
- execution of a = built-in (which is going to be present if we assume as in [2] that all calls and clauses' heads have been normalized), requires an immediate modification of the current substitution. The principle is simple: if δ is the substitution from the previous unification, and δ' is the one immediately after, then, assuming that σ is the substitution describing the effect of the unification ($\forall X \in dom(\delta)$):
  - if $\delta(X) \neq \perp$, then $\delta'(X) = \delta(X)$;
  - if $\delta(X) = \perp$ and $\sigma(X) = X$, then $\delta'(X) = \perp$;
  - if $\delta(X) = \perp$ and $\sigma(X) \neq X$, then $\delta'(X) = 1$;
- in analogous way, given an entry substitution δ for a call B and assuming that from the call we can synthesize the abstract substitution ξ, then we can build the next substitution δ' as follows:
  - if $\delta(X) \neq \perp$ then $\delta'(X) = \delta(X)$;
  - if $\delta(X) = \perp$ and $\xi(X) = \perp$ (or X is not in its domain), then $\delta'(X) = \perp$;
  - if $\delta(X) = \perp$ and $\xi(X) \neq \perp$, then $\delta'(X) = \xi(X) + 1$.
- from the solutions $\delta_1, \delta_2, \ldots, \delta_n$ in the body of the clause, we can synthesize the output substitution δ_{succ}^j for the whole clause as follows:
  - it is defined only for the variables that are in the head of the (normalized) clause;
  - if X is one such variables, we compute $i = min\{j | 1 \leq j \leq n \wedge \delta_i(X) \neq \perp\}$ if there is such i, then we assign

$$\delta_{succ}^j(X) = \delta_i(X) + \sum_{j=1}^{i-1} cost(B_j)$$

  Otherwise we simply leave $\delta_{sum}^j(X) = \perp$.
- given a set of substitutions produced for the various clauses matching a call, the global substitution is given by the usual lub.

Regarding the handling of recursion, in traditional abstract interpretation systems memoization and tabling are used for this purpose. In our context we would like to rely on a different condition. The idea is that the deepening of the expansion is stopped if the accumulated cost for a relevant variable reaches the n_{cut} limit.

A similar scheme can be formulated to obtain a lower bound estimation. In presence of more sophisticate cost analysis mechanisms (which produce cost functions), then the analysis becomes more complex, requiring comparison of functions with the n_{cut} value and generation of new cost functions to express the binding-time of the variables. This framework is currently still under development.

Consider the following example, extracted from the `Takeuchi` benchmark. The original clause is

```
tak(X,Y,Z,A) :-
       X1 is X - 1, Y1 is Y - 1, Z1 is Z - 1,
       tak(Z1,X,Y,A3),
       tak(Y1,Z,X,A2),
       tak(X1,Y,Z,A1),
       tak(A1,A2,A3,A).
```

The initial partitioning of the clause, followed by the sharing and freeness analysis, will produce the first level of annotation:

```
tak(X,Y,Z,A) :-
        X1 is X - 1, Y1 is Y - 1, Z1 is Z - 1,
        $mark([A1,A2,A3]),
        ($and_goal([[A3,A31]],tak(Z1,X,Y,A31)) &
          $and_goal([[A2,A21]],tak(Y1,Z,X,A21)) &
          $and_goal([[A1,A11]],tak(X1,Y,Z,A11)) &
          $and_goal([[A1,A12],[A2,A22],[A3,A32]],
               tak(A12,A22,A32,A))).
```

As we can see the analyzer has annotated 4 subgoals. Intuitively, this is not the ideal annotation; the last subgoal cannot start without having all the three input arguments available, which are going to be produced as last step in the execution of each of the previous three subgoals. Binding-time analysis is capable of detecting this situation. In fact the final result produced by the analyzer is the following:

```
tak(X,Y,Z,A) :-
        X1 is X - 1, Y1 is Y - 1, Z1 is Z - 1,
        ( tak(Z1,X,Y,A3) &
           tak(Y1,Z,X,A2) &
           tak(X1,Y,Z,A1)),
        tak(A1,A2,A3,A).
```

The important point to note is that our analysis revealed that while `Takeuchi` has DAP, it is futile to exploit it.

4.6 Annotation

The previous analysis steps produced information which is collected in various data structures. In particular the following information is available:

- a partition of each clause into sequences of subgoals;
- identification of the shared variables for each sequence of subgoals;
- an estimation of the cost of each procedure and of each clause in the program;
- an estimation of the binding-times for each variable shared in at least one sequence.

Clearly only those clauses should be annotated with DAP that contain at least one proper sequence of subgoals. Each of these sequences can be potentially annotated for DAP (i.e., the subgoals in the sequence may be executed in DAP). Our granularity and binding-time analysis is then applied to identify fruitless ones—i.e., those that are inherently sequential or that have too small a grainsize. These fruitless sequences are either eliminated (reduced to sequential execution) or further subdivided in order to remove the source of inefficiency (by moving out the the small sized goal out of the parallel conjunction). Details of how these operations are performed are given next:

The information available to us from our analysis is the following:

- an estimation of the cost ($cost(p/n)$) of each predicate symbol p/n used in the program;
- an estimation $\tau(B_i, X)$ of the binding-time for the variable X within the computation originating from the subgoal B_i.

A *cost graph*j is built for each sequence using this information. Given a sequence of subgoals $\Sigma = \langle B_1, \ldots, B_n \rangle$, the cost graph for this sequence $G^C_\Sigma = \langle N, E \rangle$ is defined as follows:

- $N = \{B_1, \ldots, B_n\}$
- E: edges in the cost graph are colored. *Blue* edges enforces the Prolog execution order in the graph; thus, $\forall i (1 \leq i \leq n-1 \wedge (B_i, B_{i+1}, blue) \in E$.

  The second category of edges is composed by *red* edges. Red edges are used to represent critical points of the computation—either too fine grained tasks or pairs of subgoals that are not suitable for DAP execution (lack of adequate overlapping). Red edges are currently identified as follows:

  – a threshold ξ_{cost} is selected at the beginning of the compilation process. This is used to state the minimum size of a task acceptable for parallel execution. Two red edges, (B_{i-1}, B_i, red) and (B_i, B_{i+1}, red) are added to E for each subgoal B_i such that $cost(pred(B_i)) \leq \xi_{cost}$[5].

  – another threshold ξ_{bind} is selected at the beginning of the compilation. This is used to state the minimum amount of overlap required to allow parallel execution of two subgoals having a shared variable in common. Thus, if the two subgoals B_i and B_j have a variable X in common, and B_i is suspected of being the producer of X (easily detectable through the sharing and freeness information previously computed), and B_j is a consumer of X, i.e. $i < j$, and additionally $overlap(\langle B_i, \tau(B_i, X)\rangle \langle B_j, \tau(B_j, X)\rangle) \leq \xi_{bind}$ then a red edge (B_i, B_j, red) will be added to E.

The function *overlap* can be defined in various ways. The idea is that it should return a quantitative measurement of the overlapping existing between a computation α which binds a common variable at time t_α and a computation β which attempts the first reading of the shared variable at time t_β. At present *overlap* has been realized by a simple proportion between cost of the computations (obtained from the cost analysis) and the binding times.

Once the cost graph has been created, it needs to be processed in order to produce an annotation satisfying the requirements imposed by the presence of red edges. This is realized by applying a modified algorithm for detection of *bridges* and *articulation points* in graphs [5].

The choice of the right threshold for the amount of overlapping is a very delicate issue, especially for certain programs. Requiring a very high parameter may rule out various source of parallelism (also due to the rather inaccurate cost analysis currently used). On the other hand, admitting very low threshold may allow computations that have an excessively low degree of overlapping. Figure 4 indicates the variation of speedups for a

[5] Clearly, if $i = 1$ or $i = n$ only one edge will be added to E.

version of the 8-queens benchmark (called KKqueens) when the threshold selected during compilation is varied. The KKqueen benchmark has 3 possible points where DAP annotation can be inserted; best results are achieved with an average overlapping (producing only one annotation in the code). Lowering the threshold results in more goals getting annotated, causing an increase in overhead and a lower quality scheduling. On the other hand imposing a higher threshold removes all the annotations, leading to a sequential execution.

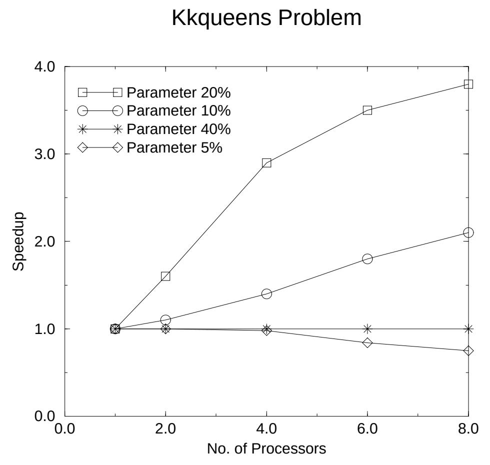

Figure 4: KKqueens Benchmark

5 Experimental Results

The ideas described thus far have been implemented in a preliminary prototypical implementation of an analyzer and annotator. The main aim of this implementation was to test the feasibility of these ideas—there is considerable scope for improvement in efficiency, accuracy, and effectiveness.

The main aim behind this project was to design a *modular* tool for generating DAP annotations—with emphasis on the modularity and interchangeability of the required components. Figure 5 illustrates the overall structure of our system. The different modules communicate through well-defined interfaces (implemented as set of easily adaptable DCG grammars to convert inputs and outputs into the desired format). This allows us to transparently replace any component of the system (e.g., the Cost Analyzer) and replace it with a better analogous component (when one becomes available), and obtain faster and more accurate results without any extra effort.

We selected a pool of benchmarks that allowed us to explore a sufficiently wide variety of programming styles and structures. We refer the interested reader to [25] for a complete description of the various benchmarks. Following the presentation style of [22], here we report (in the table below) only the main characteristics of these benchmarks. For each benchmark, *Cl* indicates the number of clauses, *Avg* the average number of subgoals per clause, *Seq* the number of uninterrupted sequences of at least two user-defined subgoals

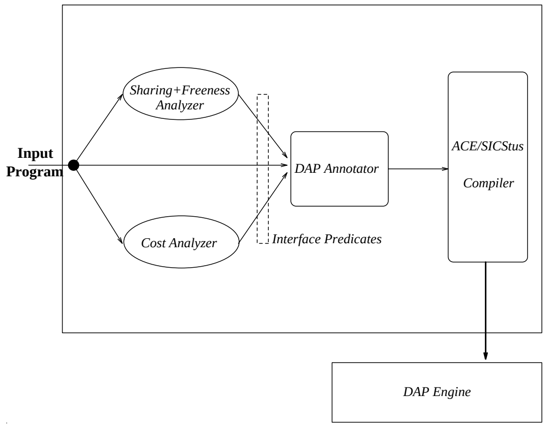

Figure 5: System Organization

presents in the clauses, *AvSeq* the average length of these sequences. Intuitively, each sequence corresponds to a sequence of goals that are potentially executable in DAP—using a "blind" annotator each of these sequences would be marked as source of DAP. Additionally, the field *AvVars* indicates the average number of variables that potentially need to be marked for this sequences (in absence of any analysis). As we can see, the example ranges from programs having very few sequences with very few variables in common, to cases of long sequences of goals and/or sequences with many variables which are potentially shared.

Bench.	Cl	Avg	Seq	AvSeq	AvVars
mmatrix	23	3.2	6	2.5	4
deriv	21	3.0	4	2.0	5
hanoi	6	4.0	2	2.5	5
queens	24	5.55	4	2.25	4
serialize	16	3.29	3	3.33	4.3
pascal	28	4.16	6	2.17	4.2
dsearch	41	2.88	5	2.4	3.4
match	29	3.44	8	2.25	7.9
tp	40	3.79	9	2.56	4.8
ga	178	3.9	13	3.15	8.2
mandel	29	2.62	9	2.22	8.3

5.1 Compilation Efficiency

An important issue to be considered is the efficiency of the compilation process. Table 5.1 shows the time required to analyze various benchmarks. The execution time (SICStus 2.1, times in ms., average of 4 executions) has been subdivided into the basic components of the analysis. *ShFr* indicates the time spent by the abstract interpreter for Sharing+Freeness (the PLAI interpreter) to compute the abstract substitutions; *Cost* indicates the time spent during cost analysis; and *DAP* describes the time invested in the actual DAP annotation. Additionally, the % column indicates the percentage of total analysis time which is spent in DAP annotation, and *IAP/DAP* indicates how much slower is the DAP analysis of the program with respect to the IAP analysis (made using the &-Prolog/CIAO compiler for IAP) of the same program.

The compilation times are quite reasonable. The complexity of detecting (good) DAP is considerably higher than that of identifying IAP. Nevertheless, the difference in execution time between the two analysis is minimal ($\leq 20.0\%$). Execution times for DAP annotation can degenerate only if the parameter specifying the depth of unfolding during the binding time analysis is increased (the cost of unfolding has a worst-case complexity of $O(2^r)$, where r is the depth of the unfolding). Nevertheless, in our experience a parameter of 2 or 3 has been sufficient to tackle virtually all the benchmarks (the times indicated above have been taken using a depth of unfolding equal to 2, i.e., $\xi_{bind} = 2)^6$.

Bench.	ShFr	Cost	DAP	Perc.	IAP/DAP
mmatrix	1259	280	1039	37.9%	19.5%
deriv	1297	289	620	30%	2.2%
hanoi	728	59	190	19.5%	0.5%
queens	1348	339	1149	38.1%	17.3%
serialize	4518	189	1440	23.4%	15.3%
pascal	4378	650	2019	29.4%	10.0%
dsearch	1069	239	830	34.19%	3.9%
match	3678	519	2810	35.09%	13.8%
tp	2709	579	2660	39.6%	20.5%
ga	11118	2390	15570	34.17%	23.8%
mandel	1215	216	1134	38.77%	20.3%

Same considerations apply to the impact of the DAP annotation on the whole analysis process—which represents on average 30% of the whole analysis time. Our results suggest that inclusion of analysis of DAP in a compiler is indeed feasible.

5.2 Effectiveness Results

There are two basic parameters that can be used to estimate the effectiveness of the DAP annotator:

1. its ability to uncover maximal parallelism avoiding fruitless cases.
2. the ability of the compiled code to generate speedups during its parallel execution.

These two features are referred to as *static tests* and *dynamic tests* by others [22].

The estimation of the effectiveness of the DAP analyzer in detecting and annotating sources of parallelism can be estimated by studying the code produced. We consider the following issues:

- Identification of parallelism: from our experiments we can easily observe that the analyzer has been quite successful in identifying parallelism. Table 1 indicates the variation in number of parallel conjunctions created for each benchmark with successive refinement in the analysis. The column *Initial* reports figures for the case when all consecutive (user-defined) subgoals are naively annotated for DAP in the local analysis phase.

  The results are quite positive. In all the benchmarks we succeeded in detecting parallelism (and this was expected, as many of these benchmarks are commonly used to evaluate IAP and DAP systems). Furthermore, the succession of binding-time analysis and granularity control managed to trim down both the number of sequences annotated and their size (number of subgoals present in the conjunctions). The variations in the binding-time analysis is not very evident, since the main effect of this analysis in the selected benchmarks was more aimed at removing from the sequences those subgoals that were not able to guarantee an adequate level of overlapping during an ideal parallel execution. Thus the number of sequences remains stable but the number of subgoals per sequence tended to decrease.

- another aspect to consider, in order to guarantee the effectiveness of the parallelization, is the number of variables that gets annotated for each parallel conjunction. Given a conjunction that is to be run in parallel, we would like to make sure that only the variables that are strictly necessary get annotated. Table 2 shows some statistics regarding the number of annotated variables, obtained during the different steps of analysis (initial grouping of goals, granularity/sharing analysis, binding-time analysis, granularity control).

- from the dynamic point of view, clearly we would like to observe what happens when we run the annotated programs in parallel. The following table reports the speedups achieved on some of the benchmarks. These results have been obtained on a Sequent Symmetry using the preliminary DAP prototype inserted in the ACE system [24]. Annotation has been made by estimating semi-automatically the threshold values on the base of the results from cost-analysis.

Bench.	*ACE Agents*				
	1	2	4	6	8
pascal	1.0	1.95	3.78	5.2	6.0
deriv	1.0	1.95	3.90	5.84	7.82
ga	1.0	2.0	3.83	5.66	7.1
quicksort	1.0	2.0	3.5	4.88	6.28
$\text{nrev}^{nogran.}$	1.0	1.77	3.05	4.02	4.85
$\text{nrev}^{gran.}$	1.0	1.0	1.0	1.0	1.0
hanoi	1.0	2.0	3.98	5.96	7.4
match	1.0	1.97	3.92	5.6	7.59
ClServ	1.0	2.0	3.97	5.82	7.59
mandel	1.0	1.71	2.71	4.67	5.16

The results are quite encouraging, since on most of the programs we have achieve good speedups (especially considering the relative inaccuracy of some of the analysis steps in the current prototype). The *nrev* benchmark is worthy of note. If granularity analysis is performed, the analyzer rejects all the annotations and produces a sequential program; disallowing granularity control a moderate speedups can be achieved (one annotation is produced). This choice can be arguable; nrev tends to generate very fine grained computations and leads to frequent suspensions of subgoals (as the consumer goals first started are the last ones to be reactivated), thus producing quite high overhead in the computation. On the other hand disallowing the annotation will lead to a loss of some speedup. Nevertheless, *nrev* is a quite peculiar program which seem to be troublesome also for other systems [27].

[6] This should be taken as a testimony to the efficiency of the DAP engine implemented in ACE [24]. The parallel overhead incurred is so low that even if producer binds after 2 Prolog calls, and the consumer consumes after 2 Prolog calls, the ACE engine is able to obtain speedups.

Bench.	Initial			No Cost Anal.			W/ Cost Anal.		
	#Seq.	Avg	Max	#Seq.	Avg	Max	#Seq.	Avg	Max
mmatrix	6	2.5	5	6	2.33	4	5	2.0	2
deriv	4	2.0	2	4	2.0	2	4	2.0	2
hanoi	2	2.5	3	2	2.5	3	1	2.0	2
queens	4	2.25	3	3	2.3	3	3	2.0	2
serialize	3	3.33	4	3	3.33	4	2	3.0	3
pascal	6	2.17	3	6	2.17	3	5	2.0	2
dsearch	5	2.4	3	4	2.5	3	2	2.0	2
match	8	2.25	3	3	2.33	3	3	2.0	2
tp	9	2.56	4	9	2.44	3	8	2.25	3
ga	13	3.16	8	14	2.64	6	6	2.17	3
mandel	9	3.67	4	7	2.25	3	3	2.67	3

Table 1: Number of Parallel Conjunctions

Bench.	Init.Group		Ground		Bind.Time		%	Cost		%
	AvgVar	MaxVar	AvgVar	MaxVar	AvgVar	MaxVar		AvgVar	MaxVar	
mmatrix	1.17	2	0.83	2	0.67	2	57.14%	0.67	2	42.86%
deriv	1.0	1	0	0	0	0	0%	0	0	0%
hanoi	2.0	4	0	0	0	0	0%	0	0	0%
queens	1.75	2	0.75	1	0.67	1	28.57	0.33	1	14.29%
serialize	1.67	2	1.67	2	1.67	2	100%	2.0	2	80%
pascal	1.67	3	1.33	3	1.0	2	60%	0.8	1	40%
dsearch	1.0	2	0.8	1	0.75	1	60%	1.0	1	40%
match	1.875	3	1.125	2	0.67	1	13.33%	0.67	1	13.33%
tp	1.22	5	0.78	2	0.56	2	45.45%	0.625	2	45.45%
ga	3.61	11	2.92	11	1.43	5	42.55 %	0.33	1	4.25%
mandel	3.78	13	1.67	7	0.89	4	23.53%	0.67	4	17.65%

Table 2: Number of variables annotated during the analysis

- the final question we may want to ask ourselves is whether the annotator managed to get close enough to the kind of annotation that a smart programmer would produce manually (we used the results previously reported [24] for the ACE system using hand annotated programs). The results are sufficiently good. Figure 6 compares speedup curves for some of the benchmarks (automatic annotation vs. manual annotation). In many programs the two annotations match perfectly. We have observed one case (`Genetic`) in which the programmer missed one small source of independent parallelism. In some examples, like `Match`, hand-coded versions behave slightly better, due to the presence of less annotations (some of the automatic annotations create unbalanced work distribution and some additional overhead). Also, the results achieved on certain benchmarks were highly unsatisfactory. This was the case for `Nrev` (as mentioned before, in the case of usage of granularity control) and `Boyer` (see figure 7), for which the system produces excessive annotations.

5.3 Conclusions and Future Work

In this work we focused on the issues of automatic detection of dependent and-parallelism in Prolog programs. The goal of this work was to develop an automatic parallelization tool capable of performing automatic annotation of Prolog programs, in order to allow their parallel execution on a dedicated DAP engine, like DASWAM or ACE.

We discussed the properties that a *good* DAP execution should satisfy (large granularity, good overlapping) and we translated them into properties for generating DAP annotations. Various levels of abstract interpretation (sharing and freeness, cost analysis, binding-time analysis) have been used to collect the information required for these annotations.

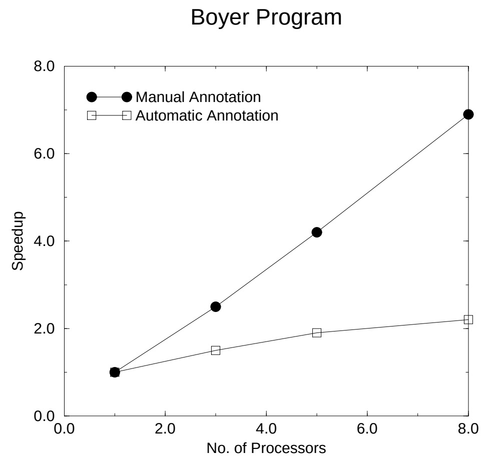

Figure 7: Boyer: Automatic vs. Manual Annotation

A preliminary prototype of an automatic analyzer and annotator was developed to test the feasibility of our ideas. This tool, which takes advantage of some state-of-the-art tools developed by other researchers, has been tested on a variety of benchmarks, with very satisfactory results.

There is still a lot of work that needs to be done. Granularity and overlapping are only two aspects that

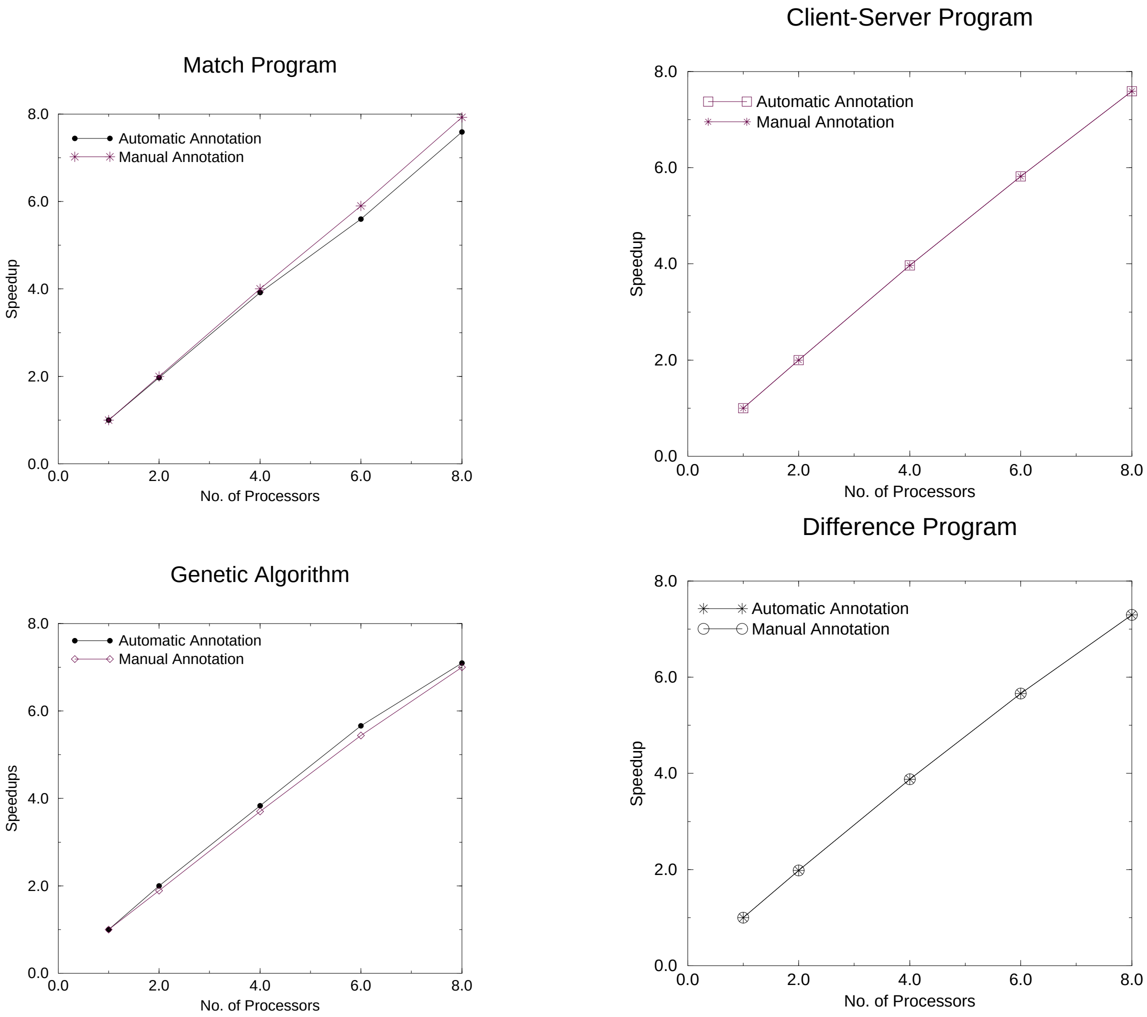

Figure 6: Speedups Comparison

characterize a good DAP computation. Other aspects need to be considered, such as:

- *determinacy:* determinate computations, especially in absence of side-effects, can produce extremely good results.
- *non-failure:* subgoals that are likely to fail should be kept outside of parallel annotations—failure in a DAP execution requires very complex and time consuming activities. This can be a key issues in automatic parallelization for DAP. We are currently developing an abstract interpretation scheme to infer likelihood of failure of a subgoal that is present in an annotated sequence.

Furthermore, information about costs, binding-times, sharing+freeness, and presence of side-effects within subgoals, can be used to perform rearrangements of the subgoals in the body of clause, in order to:

1. facilitate the construction of sequences of subgoals which satisfy the requirements of DAP;
2. improve exploitation of parallelism in presence of possibly failing subgoals;

As already mentioned before, in the current prototype we adopted quite a conservative and simple cost estimation scheme (based on static cost analysis). We are currently working on modifying the *Caslog* system [10] in order to supply cost information usable by our system. This implies introducing in the annotated program some simple tests (e.g., evaluation of cost functions), making the DAP annotation conditional.

To conclude, our attempt to perform automatic annotation of DAP seems to be quite successful. To the best of our knowledge this is the first ever effort to automatically identify DAP in Prolog programs. Our work is a first step towards a building a comprehensive tool, as there is plenty of scope for improvement. For some benchmarks the results of the analysis were not completely satisfactory, due to the lack of accuracy of some of the information collected. Improvements in this direction (e.g. cost analysis) will improve the general annotation capabilities of the system. In this sense we slightly disagree with the opinion put forward by other authors regarding the infeasibility of automatic annotation and the level of complexity of annotating programs for DAP [28]. In large applications detecting *good* sources of DAP manually is *not* easy, since it requires a deep understanding of the structure of the execution and interaction between the subgoals, clearly a task from which a user should be unburdened. Our DAP analysis and annotation tool shows that indeed this burden can be lifted from the user.

Acknowledgments

Thanks are due to Kish Shen of University of Manchester and Manuel Hermenegildo and his CLIP group at UPM, Spain, for various discussions and ideas. This work has been partially supported by NSF grants HRD 93-53271, INT 95-15256, and CCR 96-25358, and by NATO Grant CRG 921318.

References

[1] J. Bevemyr et al. Reform Prolog: the Language and its Implementation. In *Int'l Conf. on Logic Progr.* MIT Press, 1993.

[2] M. Bruynooghe. A Framework for the Abstract Interpretation of Logic Programs. Technical Report, Dept. of Computer Science, Katholieke Universiteit Leuven, 1987.

[3] F. Bueno. *Automatic Optimisation and Parallelisation of Logic Programs through Program Transformation.* PhD thesis, Univ. Politecnica de Madrid, 1994.

[4] D. Cabeza and M. Hermenegildo. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In *Int. Static Analysis Symposium*, Springer Verlag, 1994.

[5] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, 1992.

[6] A. Cortesi and G. Filè. Abstract Interpretation of Logic Programs. In *Symp. on Partial Evaluation and Semantics Based Program Manipulation*, ACM, 1991.

[7] M. García de la Banda, M. Hermenegildo, and K. Marriot. Independence in Constraint Logic Programs. In *Int. Symposium on Logic Programming*, MIT Press, 1993.

[8] S. Debray. Observations on weight estimation, 1995. (personal communication).

[9] S. K. Debray, N.-W. Lin, and M. Hermenegilo. Task Granularity Analysis in Logic Programs. In *Conf. on Progr. Language Design and Implem.* ACM Press, 1990.

[10] S.K. Debray and N.W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5), 1993.

[11] D. DeGroot. Restricted AND-Parallelism. In *Int'l Conf. on 5th Generation Computer Systems*, Tokyo, Nov. 1984.

[12] R. Giacobazzi and L. Ricci. Pipeline Optimizations in And-parallelism by Abstract Interpretation. In *Int. Conf. on Logic Programming.* MIT Press, 1990.

[13] G. Gupta. *Multiprocessor Execution of Logic Programs.* Kluwer Academic Press, 1994.

[14] M. Hermenegildo. CIAO, a Generic Parallel Concurrent Constraint Logic Programming System. In *Principle and Practice of Constraint Programming.* Springer Verlag, 1994.

[15] M. Hermenegildo and K. Greene. &-Prolog and its Performance. In *Int'l Conf. on Logic Prog.*, MIT Press, 1990.

[16] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs. *Journal of Logic Programming*, 22(1), 1995.

[17] L. Huelsbergen, J. Larus, and A. Aiken. Using Run-time List Sizes to Guide Parallel Thread Creation. In *Prof. Conf. on Lisp and Functional Programming.* ACM Press, 1994.

[18] D. Jacobs and A. Langen. Compilation of Logic Programs for Restricted And-Parallelism. In *European Symposium on Programming*, pages 284–297, 1988.

[19] A. King and P. Soper. Schedule Analysis of Concurrent Logic Programs. In *Int. Conf. and Symp. on Logic Programming.* MIT Press, 1992.

[20] B. Kruatrachue and T. Lewis. Grain Size Determination for Parallel Processing. *IEEE Software*, January 1988.

[21] U. Montanari, F. Rossi, F. Bueno, M. García de la Banda, and M. Hermenegildo. Towards a Concurrent Semantics based Analysis of CC and CLP. In *Princ. and Practice of Constraint Programming*, Springer-Verlag, May 1994.

[22] K. Muthukumar, F. Bueno, M. Garcia de la Banda, and M. Hermenegildo. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *Journal of Logic Programming*, 1997. (to appear).

[23] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *Int. Conf. on Logic Programming.* MIT Press, 1991.

[24] E. Pontelli and G. Gupta. Non-determinate Dependent And-Parallelism Revisited. Technical Report NMSU-CSTR-9601, Dept. of Computer Science, New Mexico State University, 1996.

[25] F. Pulvirenti, E. Pontelli, G. Gupta, and A. Ferro. Use of Sharing, Freeness, and Granularity Analysis for Compile-time Annotation of Dependent And-parallelism. Technical report, NMSU, Dept. of Computer Science, 1996. (to appear).

[26] K. Shen. *Studies in And/Or Parallelism in Prolog.* PhD thesis, U. of Cambridge, 1992.

[27] K. Shen. Initial Results from the Parallel Implementation of DASWAM. In *Int. Conf. and Symp. on Logic Programming.* MIT Press, 1996.

[28] K. Shen. Overview of daswam: Exploitation of dependent and-parallelism. *Journal of Logic Programming*, 1996. (to appear).

[29] K. Shen and M. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. In *Int. Logic Programming Symp.* MIT Press, 1991.

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

[31] E. Tick and X. Zhong. A Compile-time Granularity Analysis Algorithm and its Performance Evaluation. *New Generation Computing*, 11, 1993.

[32] K.R. Traub. Compilation as Partitioning: a New Approach to Compiling Non-strict Functional Languages. In *Conf. on Functional Progr. Languages and Architecture.* ACM Press, 1989.

[33] K. Ueda and M. Morita. Moded Flat GHC and its Message-oriented Implementation Technique. *New Generation Computing*, 1993.

[34] D. H. D. Warren. The Extended Andorra Model with Implicit Control. In *Parallel Logic Programming Workshop*, June 1990. SICS.