QE-Andorra: A Quiche–Eating Implementation of the Basic Andorra Model

Francisco Bueno
Saumya K. Debray
María José García de la Banda
Manuel Hermenegildo

TR Number CLIP05/96

QE-Andorra: A Quiche–Eating Implementation of the Basic Andorra Model

Technical Report Number: CLIP05/96
February 1996

Authors

F. Bueno
M. García de la Banda
M. Hermenegildo
Universidad Politécnica de Madrid (UPM), Facultad de Informática,
28660-Boadilla del Monte, Madrid, Spain.

S. Debray
University of Arizona, Computer Science Department,
85721 Tucson, Arizona, USA.

Keywords

Logic Programming, Concurrent Programming/Parallelism, Andorra Model, Program Transformation, Language Implementation

Acknowledgements

The work of F. Bueno, M. García de la Banda and M. Hermenegildo was funded in part by ESPRIT project 7195 "ACCLAIM" and by CICYT projects TIC93-0975-CE and TIC93-0737-C02-01 IPL-D. The work of S. Debray was funded in part by the National Science Foundation under grant number CCR-9123520.

We would like to thank D.H.D. Warren, V. Santos-Costa, R. Yang, and the rest of the Andorra-I team for making a version of the Andorra-I system available to us and for useful discussions.

Abstract

The characteristics of CC and CLP systems are in principle very different. However, a recent trend towards convergence in the implementation techniques for these systems can be observed. While CLP and Prolog systems have been incorporating capabilities to deal with user–defined suspension and coroutining, CC compilers have been trying to coalesce fine–grained tasks into coarser–grained sequential threads. This convergence of techniques opens up the possibility of having a general purpose kernel language and abstract machine to serve as a compilation target for a variety of user–level languages. We propose a transformation technique directed towards such an objective. In particular, we report on techniques to support the Andorra computational model, essentially emulating the Andorra-I system, via program transformation into a sequential language with delay primitives. The system is automatic, comprising an optional program analyzer and a basic transformer to the kernel language. It turns out that a simple (parallel) CLP (or Prolog) system with dynamic scheduling is sufficient as a kernel language for this purpose. The preliminary results are quite encouraging: performance of the resulting system is comparable to the current Andorra-I implementation.

Contents

1 Introduction

Many current proposals for parallel or concurrent logic programming languages and models are actually "bundled packages", in the sense that they offer a combined solution affecting a number of issues such as choice of computation rule, concurrency, exploitation of parallelism, etc. This is understandable since a practical model has to offer solutions for all the problems involved. However, the bundled nature of (the description of) these proposals has two significant disadvantages. First, performing comparisons among these systems becomes difficult due to the complexity of determining how different design decisions might affect performance. Second, reusing components of one system in another turns out to be a difficult task. As a result, implementors are typically forced to build their systems from scratch, by writing their own runtime systems and constructing compilers to compile programs into low-level languages such as C or assembler. The tremendous engineering and manpower overhead involved in such an enterprise means that, in many cases, implementors may be unable to take advantage of clever optimizations that other researchers have implemented, or to invest the time and effort necessary to turn their systems from research prototypes to mature and robust systems that can be shared with other researchers and users. Very often, researchers find the "non-research" engineering overhead of system implementation sufficiently daunting that their ideas and languages do not make it past the paper design stage.

This is an unfortunate state of affairs, and leads to a great deal of duplicated effort and wasted time. This situation can be improved by performing a "separation analysis" of the execution model underlying the language and isolating its fundamental principles [15]. Such un-bundling not only identifies fundamental principles but also shows that the applicability of such principles can be enhanced by allowing the transfer of good features of one model to another. This fact at the same time explains and is supported by the recent trend towards convergence in the analysis and implementation techniques of models that are in principle very different. In fact, the techniques used in various and-parallel implementations of Prolog (e.g., [14, 22]), in the implementations of various committed choice languages (e.g., [3, 11, 16, 26, 27]), and in the implementation of sequential Prolog systems using coroutining (e.g., [8, 2, 25, 9]), are often very similar.

This convergence of trends opens up the possibility of having a general purpose kernel language and abstract machine to serve as a compilation target for a variety of user-level languages. Given a sufficiently high level intermediate language of this kind, and carefully crafted compilers and runtime systems for this language, the implementation of other logic programming languages would be simplified considerably: rather than reinvent all aspects of an implementation from scratch, it would be possible to share the "back end" across different systems, and thus require only the construction of compilers to the intermediate representation. Encouraging initial results in this direction have been demonstrated in the sequential context by the QD-Janus system [7]. QD-Janus, which compiles down to SICStus Prolog and uses the delay primitives of the Prolog system to implement data-flow synchronization, offers performance that is competitive with some natively implemented concurrent logic programming systems.

In this paper we set a more ambitious target: supporting the Basic Andorra Model [22], which incorporates a rather smart, concurrent, determinacy-based selection rule. We show how a standard Prolog/CLP system with delay primitives can serve as a target language for compiling Andorra (and, in particular, Andorra-I) programs, and achieve good performance. We show how to translate such programs (Section 2), including the determinacy-based suspension conditions (Section 3) and support for built-ins (Section 4). We also propose optimizations that can be applied both during and after such translation (sections 5 and 6). The transformation is illustrated in Section 7 with a detailed example in which we follow all the different steps involved. We also show how using our approach it is possible to arbitrarily mix Prolog and Andorra execution (Section 8). Our transformation can then be seen as a straightforward way of adding Andorra-style search to a Prolog or CLP system, which can be used optionally in parts of a program. Finally, in Section 9 we provide some experimental results which are encouraging: performance of the resulting system is comparable to the direct Andorra-I implementation [23].

We do not mean to imply that the performance of a system implemented using our approach is optimal or that it will achieve better results than a native Andorra-I implementation, but rather that the technique is practical and allows the support of the Basic Andorra Model on a generic system with reasonable performance. Our main overall message is that using this approach, it is possible to construct implementations for logic programming languages—even those whose execution models depart quite significantly from Prolog's—in terms of simple and easy-to-verify source-level transformations, in a fairly straightforward way (e.g., without having to reimplement garbage collectors, stack shifters, complex low-level compiler optimizations such as register allocation and instruction scheduling, etc.), in a fairly short time, and nevertheless attain fairly good performance.

2 A General Transformation for the Andorra Model

In the Basic Andorra Model a goal—or, more precisely, a reduction—is delayed until either it becomes determinate, or it becomes the leftmost reduction and no determinate reduction is available. Implementing the first condition is conceptually simple (although the machinery required might be nontrivial); achieving the second condition is more complicated because it is necessary to keep track of the order in which the goals would have been reduced in a sequential system with a left-to-right computation rule. This operational behavior can be realized in a system with a left-to-right computation rule as follows by maintaining two sequences of goals: a list of *non-delayed* goals, and a list of *delayed* goals (see [6, 19] for more formal accounts of operational semantics for logic programming languages with delay primitives). The non-delayed goals are inspected from left to right, and reductions are carried out as follows:

1. If the leftmost non-delayed goal is determinate, it is reduced; otherwise, it is delayed, i.e., transferred to the list of delayed goals.
2. If the list of non-delayed goals is empty, i.e., no deterministic reductions are possible, the leftmost delayed goal is reduced.
3. Whenever a delayed goal becomes determinate (because of bindings created by other goals), it is awakened and moved to the list of non-delayed goals.

To achieve this effect, we propose to use program transformation into a language with left-to-right computational rule and delay. The main technical challenge here is to maintain, as efficiently as possible, enough information to allow us to easily determine (*a*) which of a collection of delayed goals is the leftmost, and (*b*) which of a collection of delayed goals have become determinate and should be awakened. There are many ways in which this can be done, mainly depending on the concept of determinacy chosen and the kind of structure used to identify leftmost delayed goals. The concept of determinacy will be specified later on in Section 3. To identify the leftmost of a list of delayed goals, we use a list of variables *Inf*, where each variable is associated with a delayed goal, and the relative order among the variables in this list reflects the relative left-to-right reduction order among the corresponding goals. A delayed goal becomes the leftmost when the associated variable becomes bound to a non-variable term.

Our program transformation constructs the list *Inf* using difference lists. In particular, for each program predicate `p/n`, we create two predicates: `p/n+2` and `p_susp/n+2`. The definition of `p/n+2` is the following:

```
p(X̄,L,L1):- L = [S|L0], det_susp( (det_cond(X̄);nonvar(S)),
p_susp(X̄,L0,L1) ).
```

where $\bar{X}$ is a sequence of n distinct variables, $det\_cond(\cdot)$ is a condition on $\bar{X}$ such that $det\_cond(\bar{X} \wedge c)$ is true if and only if $p(\bar{X})$ is determinate in a constraint store c, and $det\_susp/2$ is a suspension primitive of the language which delays the goal provided as second argument until the condition provided as first argument becomes true. Our definitions of these two functions are given in Section 3. Intuitively, `S` is the variable associated with `p(X̄)` in the list *Inf*, and `L0` and `L1` are pointers that delimit the sublist of *Inf* corresponding to the variables associated with the subgoals in the derivation tree of `p(X̄,L0,L1)`. The variables `L0` and `L1` can be thought of as links that connect the reductions needed to execute `p(X̄,L0,L1)` with those needed to execute the goals to the left and right of `p(X̄,L0,L1)`, respectively, in the chain *Inf*. Since the role of a variable in the list *Inf*, associated with a goal G, is to indicate when G has become the leftmost delayed goal (by becoming bound to a non-variable term), we refer to these variables as "leftmost-tokens."

The definition of `p_susp/n+2` is derived from the definition of `p/n` as follows:

- For every fact $\mathtt{p(\bar{X})}$, we add the fact $\mathtt{p\_susp(\bar{X},L,L)}$.
- For every clause $\mathtt{p(\bar{X})\text{:- }q_1(\bar{Y}_1)\ ,\ \cdots,\ q_n(\bar{Y}_n).}$, with $n > 0$, we add the clause $\mathtt{p\_susp(\bar{X},L_1,L_{n+1})\text{:- }q_1(\bar{Y}_1,L_1,L_2),\ \cdots,\ q_n(\bar{Y}_n,L_n,L_{n+1}).}$

When a fact is selected during the execution of `p/n+2`, no further reductions are needed, and therefore we must unify the pointers associated to the goal thus closing the associated list. Otherwise, we need to split up such list in as many sublists as goals appear in the body of the clause, always keeping the left-to-right order.

The instantiation of the leftmost tokens is achieved by means of a transformed query:

- For a given query $\mathtt{\text{:- }q_1(\bar{Y}_1),\ \cdots,\ q_n(\bar{Y}_n)}$, where $n > 0$, we create the new query
$\mathtt{\text{:- }q_1(\bar{Y}_1,L_1,L_2),\ \cdots,\ q_n(\bar{Y}n,L_n,L_{n+1}),\ wakeup(L_1,L_{n+1}).}$

where `wakeup/2` is defined as follows:

```
wakeup(L1,L2) :- L1==L2, !.
wakeup([L1|L2],L3) :- L1=up, wakeup(L2,L3).
```

Given a language with left-to-right computation rule and which awakes suspended goals as soon as possible (*eager awakening*), the sequential execution of the transformed program in the target language emulates the determinate phase of an Andorra-I execution: the program is executed left-to-right but only determinate goals are reduced, non-determinate goals being suspended. As the reduction of determinate goals progresses, suspended goals which have become determinate are woken and the determinate phase continues. When no more determinate goals are available, `wakeup/2` is reduced and instantiates the first token in the chain-list, thus awakening the leftmost suspended (non-determinate) goal.

Example 2.1 Consider the following Andorra-I program:

```
p:- p1,p2.      q(X).     p1(X).     p2.
p.              q(Y).     p1(Y).
```

Following the transformation mentioned above, we will obtain the transformed program:

```
p(L,L1):-    L=[S|L0], det_susp( nonvar(S), p_susp(L0,L1) ).
q(X,L,L1):-  L=[S|L0], det_susp( nonvar(S), q_susp(X,L0,L1) ).
p1(X,L,L1):- L=[S|L0], det_susp( nonvar(S), p1_susp(X,L0,L1) ).
p2(L,L1):-   L=[S|L0], det_susp( (true;nonvar(S)), p2_susp(L0,L1) ).

p_susp(L1,L3):- p1(L1,L2),p2(L2,L3).    q_susp(X,L,L).    p1_susp(X,L,L).
p_susp(L,L).                            q_susp(Y,L,L).    p1_susp(Y,L,L).

p2_susp(L,L).
```

Consider the query `:- p,q(X)`. The transformed query is: `:- p(L1,L2), q(X,L2,L3), wakeup(L1,L3).` The following trace represents the computation states in the execution of the transformed program. Note that some steps are summarized and the current store is omitted and already applied to the resolvent.

$\langle \mathtt{p(L1,L2)} : \mathtt{q(X,L2,L3)} : \mathtt{wakeup(L1,L3)}, \mathtt{nil}\rangle$
    `L1=[Sp|L11]`
$\langle \mathtt{q(X,L2,L3)} : \mathtt{wakeup([Sp|L11],L3)}, \mathtt{p\_susp(L11,L2)}\rangle$
    `L2=[Sq|L21]`
$\langle \mathtt{wakeup([Sp|L11],L3)}, \mathtt{q\_susp(X,L21,L3)} : \mathtt{p\_susp(L11,[Sq|L21])}\rangle$
    `Sp=up`
$\langle \mathtt{p\_susp(L11,[Sq|L21])} : \mathtt{wakeup(L11,L3)}, \mathtt{q\_susp(X,L21,L3)}\rangle$
$\langle \mathtt{p1(X,L11,L12)} : \mathtt{p2(L12,[Sq|L21])} : \mathtt{wakeup(L11,L3)}, \mathtt{q\_susp(X,L21,L3)}\rangle$
    `L11=[Sp1|L111]`
$\langle \mathtt{p2(L12,[Sq|L21])} : \mathtt{wakeup([Sp1|L11],L3)}, \mathtt{p1\_susp(A,L111,L12)} : \mathtt{q\_susp(X,L21,L3)}\rangle$
    `L12 = [Sq|L21]`
$\langle \mathtt{wakeup([Sp1|L111],L3)}, \mathtt{p1\_susp(A,L111,[Sq|L21])} : \mathtt{q\_susp(X,L21,L3)}\rangle$
    `Sp1=up`
$\langle \mathtt{p1\_susp(A,L111,[Sq|L21])} : \mathtt{wakeup(L111,L3)}, \mathtt{q\_susp(X,L21,L3)}\rangle$
    `L111 = [Sq|L21]`
$\langle \mathtt{wakeup([Sq|L21],L3)}, \mathtt{q\_susp(X,L21,L3)}\rangle$
    `Sq=up`
$\langle \mathtt{q\_susp(X,L21,L3)} : \mathtt{wakeup(L21,L3)}, \mathtt{nil}\rangle$
    `L21 = L3`
$\langle \mathtt{wakeup(L3,L3)}, \mathtt{nil}\rangle$
$\langle \mathtt{nil}, \mathtt{nil}\rangle$

There is, in fact, a simpler and more elegant variation of this transformation that was suggested to us by Lee Naish [20], and which we use in the remainder of this paper. It is similar to the short-circuit technique. Basically, a pair of variables is added to each predicate in the program and a chain of them formed in all clause body atoms. Once a goal succeeds, the chain is closed by unifying the two variables acting as pointers to the chain. The call to `wakeup/2` is replaced by instantiating the first variable in the chain to a token, which is then passed around by closing the chain. At arrival of the token to a predicate, it is leftmost in the execution. The transformation is as follows:

- For each program predicate `p/n`, the following clause is added:
  `p(X̄,I,O):-` *det_susp*`( (`*det_cond*`(X̄);nonvar(I)), (p_susp(X̄,I,NO),NO=O) ).`

- The definition of `p_susp/n+2` is derived from the definition of `p/n` as explained before, and substitutes it.

- For a given query `:-` $q_1(\bar{Y}_1)$, $\cdots$, $q_n(\bar{Y}_n)$, where $n > 0$, we create the following new query
  `:-` $q_1(\bar{Y}_1, T_1, T_2)$, $\cdots$, $q_n(\bar{Yn}, T_n, T_{n+1})$, T_1`=up.`
  so that the last "wakeup" goal starts the token-passing mechanism.

Because this avoids consing up a list structure on the heap, this transformation is usually more efficient than that described earlier. However, this is not always the case, since the goals delayed include now a structure. Preliminary experiments show that the tradeoff between the two transformations varies from a speedup of 3 to a slowdown of 1.5.

3 Determinacy Conditions and Suspension Declarations

The concept of determinacy (of goals, literals, or predicates) has been defined in a number of different ways in the logic programming community. In general, the idea is that, for a given goal to a predicate, at most one clause will succeed. Informally, a predicate is said to be determinate if every goal that complies with the intended use of that predicate is determinate. In a particular program, we will say that a literal is determinate if all goals arising from that literal are; and a predicate is determinate if all literals for that predicate in the program are determinate.

In the following we will apply the definition used in the Andorra-I compiler, namely, *flat determinacy* [5, 24]. This refers to determinacy that can be recognized by means of a simple analysis of head unification and built-ins. We will use this exact notion of determinacy for two very practical reasons: first, we would like to make a fair comparison between our results and those of Andorra-I; and, second, we would like to reuse the determinacy detection phase of the Andorra-I compiler in our implementation.

Another relevant issue in the context of determinacy detection is the nature and complexity of the determinacy conditions, which can vary from the simplest conditions such as `true` or `false`, to complex conjunctions and/or disjunctions of tests regarding the instantiation states, the type, or the unifiability of some variables. In fact, it is generally possible to express the *det_susp*/2 function introduced in the previous section using the suspension primitives that are available in most general purpose Prolog/CLP systems [2, 8, 9, 25]. SICStus Prolog, for example, provides coroutining facilities by means of `block` declarations and `when` meta-calls, among others. The block declaration takes the form:

`:- block` *Spec*`,` ...`,` *Spec*.

where each *Spec* is a mode specification of the goals for the predicate, and specifies a condition for blocking goals of the predicate referred to by it. When a goal for the predicate is to be executed, the mode specifications are interpreted as conditions for blocking the goal, and if at least one condition evaluates to *true*, the goal is blocked. A `block` condition evaluates to *true* iff all arguments specified as "`-`" are uninstantiated, in which case the goal is blocked until at least one of those variables is instantiated. The more general (and more expensive) `when` meta-call has the general form:

`when(`*Condition*`,`*Goal*`)`

and blocks *Goal* until *Condition* is true, where *Condition* is a Prolog goal given by the following restricted syntax:

Condition ::= `nonvar(X)` | `ground(X)` | `X ?= Y` | *Condition*, *Condition* | *Condition*; *Condition*

Whenever the determinacy condition *det_cond* only contains `nonvar` tests over the predicate arguments, then a `block`-like suspension declaration can be used. It will only be necessary to put *det_cond* in conjunctive normal form and define one mode spec for each conjunct, where each of these will have the corresponding argument replaced by a "`-`" flag. If this is not possible, but the condition fits into the above syntax, then `when`-like literals can be used. Again, the condition can be put in either conjunctive or disjunctive normal form, each item in this expression replaced by the corresponding checks, and an expression built from the latter. If neither of them can be used, user-defined predicates should be used, which will be the result of compiling the decision graph which corresponds to the condition into Prolog itself. This compilation process is based on the observation that conditions can always be reduced to conjunctions of tests from the above syntax, plus other Prolog tests. For each of the former, a predicate exists which suspends until the condition is satisfied, and then checks all of the latter. See Section 7 for an example, and [10] for details.

4 Handling Non-Pure Features

In this section we will discuss the transformation of built-ins. There are basically three kinds of built-ins. The first type is formed by those built-ins which can be considered as normal user goals whose low-level implementation is only due to efficiency reasons. In this case, there are only two differences between the general transformation defined in the previous section and that applied to these built-ins.[1] Firstly, the determinacy condition is obtained from an internal database rather than from the ex-

[1]To avoid having to include such transformed built-ins in each transformed program, they are part of a special module that is loaded with every transformed program.

amination of the definition of the predicate. Secondly, the predicate provided as the second argument of *det_susp*/2 is the original built-in.

Example 4.1 One possible transformation for the `>/2` built-in could be the following:

```
>(X,Y,I,O) :- when( (ground(X/Y);nonvar(S)), (X>Y,O=I) ).
```

The second class is formed by those built-ins whose early evaluation may affect the correctness of the execution. For example, side-effect built-ins such as `write/1`, meta-logical predicates such as `var/1`, or pruning operators such as cuts. In the Andorra-I system this problem is easily solved by just delaying such built-ins until they become leftmost [5]. In our approach, this simply corresponds to creating a *false* determinacy condition.

The third class is formed by built-ins which can be affected by the early execution of goals which are dependent on it. In particular, whenever:

- an early failure prevents the execution of a side-effect, or
- an early binding affect cuts, meta-logical predicates, and side-effects that assume their arguments to be unbound.

Many different approaches can be taken in order to eliminate this problem [5]. We will follow the same approach taken by the Andorra-I system, namely to detect these *sensitive* built-ins and prevent any execution of goals to the right of one such built-in until both all goals to its left and the built-in itself have been completely executed. In order to do this we will split the token-passing chain in two parts, and force the token to circulate the leftmost part before allowing it to reach the rightmost one. Since the method is simple and intuitive, we omit its formal definition. The technique is as follows.

The idea is to add a wakeup goal just after the transformed sensitive built-in, thus forcing all goals to the left to be executed before continuing with those to its right. This implies adding (at least) an extra argument to the predicates in order to provide the wakeup goal with the appropriate "initial" pointer. Note that for those built-ins which belong both to the second and third classes, allocating the wakeup goal just *before* the *original* (non transformed) built-in, would both prevent any execution of goals to its right and delay the built-in until it becomes leftmost, thus solving both problems.

Example 4.2 Consider the following fragment of a program:

```
p(X):- q(X), r(X), write(X), s(X).
```

An early execution of `write(X)` might affect the correctness of the execution. Thus we will delay it until leftmost. Also, it is a "sensitive" built-in and therefore we must disallow early execution of the goals to the right. The predicate can be transformed as follows:

```
p(X,I,O,I1):- det_susp( nonvar(S), (p_susp(X,I,NO,I1),NO=O) ).

p_susp(X,T1,T4,In):- q(X,T1,T2,In), r(X,T2,T3,In), In=up, write(X), s(X,T3,T4,T3).
```

The transformation is just changed in that we have to add an extra argument, which will be the first argument of the intermediate wakeup goal introduced just before the sensitive built-in. Then, all goals in the chain between `In` and `T3` will be executed before the computation proceeds to after the wakeup goal.

5 Optimizing the Transformation

Several sources of possible overheads arise in the above transformation. First, the amount of code generated: for each procedure definition, the (generic) transformation yields at least two other procedures. Second, the addition of some arguments — at least the pointers to the chain-list of tokens, which may happen to be unnecessary. Finally, the need for detecting the conditions for determinism and the leftmost goal, and the related suspension on these conditions. In this section we discuss some optimizations that can be performed regarding these three sources of possible overhead, based on information available prior to performing the transformation. We then define an improved transformation.

5.1 Simple enhancements

If a goal is determinate at the time it is first considered for execution, it should not be suspended to be immediately woken. Therefore, a translated program which checks the determinacy condition *det_cond*(X̄) first before blindly suspending with *det_susp*/2 can be more efficient at execution time:

```
p(X̄,I,O):-
       goal(det_cond(X̄)) -> p_det(X̄,I,NO), NO=O
                          ; det_susp( (det_cond(X̄);nonvar(I)), (p_susp(X̄,I,NO), NO=O) ).
```

In general, *det_cond*(X̄) may not be directly executable in the target language, and thus it must be mapped into suitable goals by *goal*(*det_cond*(X̄)). If this goal succeeds, a specialized version of `p_susp/n+2` which does not need to suspend, `p_det/n+2`, will be called. Because this predicate will only be called when goals are known to be determinate, it is possible to avoid the creation of choice-points when reducing the predicate with its clauses by appropriately adding cuts. This would closely simulate the commitment to certain clauses introduced by Andorra-I once a reduction is known to be determinate.

5.2 Unchaining calls to predicates

This optimization is based on the observation that certain clauses do not need the extra arguments of chain pointers for the token to be passed. Such clauses include facts, and others which only have literals in their body which, in turn, do not need such arguments, either. A clause is, therefore, said to be *unchained* if all literals in its body are either constraints (or unification equations), "always-executable" built-ins (such as `true`), or goals for an unchained predicate. A predicate is said to be unchained if all clauses in its definition are unchained and the predicate itself is determinate.

Note that an unchained clause does not need to have the extra arguments because, as soon as it is reduced, the left-to-right execution of its body corresponds to the Andorra model, due to the nature of the literals in it. Also, since the previous definitions are recursive, one might imagine that an analysis for "unchainedness" would require an iterative fixpoint computation, but it turns out that this is not necessary. The reason for this is that if all of the literals of a clause are unchained, except for a recursive call, then the clause will be unchained only if this recursive call does not need a token to be passed, and this is so only if it is determinate. For handling mutually recursive predicates in this manner, we regard them to be a single predicate in what follows. Therefore, a simpler algorithm in this style, which will allow marking clauses and predicates as unchained, can be defined as follows:

$$\begin{aligned}
\mathit{unchained}(\mathtt{p/n}) &\leftarrow \mathit{constraint}(\mathtt{p/n})\\
\mathit{unchained}(\mathtt{p/n}) &\leftarrow \mathit{ready\_builtin}(\mathtt{p/n})\\
\mathit{unchained}(\mathtt{p/n}) &\leftarrow \mathit{determinate}(\mathtt{p/n}) \wedge \forall C \in \mathit{defn}(\mathtt{p/n})\ \mathit{unchained}(C)
\end{aligned}$$

$$\begin{aligned}
\mathit{unchained}(C) &\leftarrow \mathit{body}(C) = \emptyset\\
\mathit{unchained}(C) &\leftarrow \forall \mathtt{p/n} \in \mathit{body}(C)\\
&\qquad (\neg \mathit{recursive}(\mathtt{p/n}) \rightarrow \ \mathit{unchained}(\mathtt{p/n}))\wedge\\
&\qquad (\mathit{recursive}(\mathtt{p/n}) \rightarrow \ \mathit{determinate}(\mathtt{p/n}))
\end{aligned}$$

where $\mathit{constraint}(\mathtt{p/n})$ holds if `p/n` is a constraint symbol, $\mathit{ready\_builtin}(\mathtt{p/n})$ if it is an "always-executable" built-in, $\mathit{determinate}(\mathtt{p/n})$ if predicate `p/n` is known to be determinate, $\mathit{recursive}(\mathtt{p/n})$ if it is recursive, $\mathit{defn}(\mathtt{p/n})$ gives the set of clauses defining it, and $\mathit{body}(C)$ gives the list of predicates of the body of a clause C. Given this definition, unchained goals and predicates can be identified in linear time by a simple depth-first traversal of the call graph of the program.

Given that some predicates and clauses are marked as unchained, the definition of `p_susp/n+2` presented in Section 2 can be modified to take this into account:

- If *unchained*(`p/n`), then we can avoid the two extra arguments, the definition of `p_susp/n` being the result of renaming the functor `p` by the functor `p_susp`.
- Otherwise, for every clause C in the set of clauses SC defining `p/n`:
  - If $C \equiv$ `p(`$\bar{X}$`).`, it is transformed into `p_susp(`$\bar{X}$`,T,T).`
  - If $C \equiv$ `p(`$\bar{X}$`):- q`$_1$`(`$\bar{Y}_1$`)` $\cdots$ `q`$_n$`(`$\bar{Y}_n$`).`, with $n > 0$, and $\exists i \in [1,n]$ *unchained*(`q`$_i$`/n`$_i$), C is transformed into the clause `p_susp(`$\bar{X}$`,T`$_1$`,T`$_{n+1}$`):- Q`$_1, \cdots,$`Q`$_n$`.` where

$$\mathtt{Q}_i = \begin{cases} \mathtt{q}_i\mathtt{(}\bar{\mathtt{Y}}_i\mathtt{),\ T}_i\mathtt{\ =\ T}_{i+1} & \textit{if unchained}(\mathtt{q}_i/\mathtt{n}_i) \\ \mathtt{q}_i\mathtt{(}\bar{\mathtt{Y}}_i\mathtt{,\ T}_i\mathtt{,\ T}_{i+1}\mathtt{)} & \textit{otherwise} \end{cases}$$

Obviously the unification equations can be solved during the transformation, substituting one variable for the other. In the following we will denote by *chain*(SC) the function which transforms the set of clauses SC following the above proposed method.

5.3 Determinacy condition is true

It is sometimes possible through program analysis to determine that a determinacy condition will always succeed. When the determinacy condition is reduced to true, the general transformation defined previously, which was based on the construction of an if-then-else, can obviously be reduced to its "then" part. Therefore the extra clause that the transformation adds amounts to a simple renaming, which, in fact, can be performed at transformation time. Alternatively, this can be achieved by partial evaluation with a simple one-step unfolding.

Global analysis can help in determining such situations. The conditions for determinacy of a predicate are often expressed as checks on the degree of instantiation of certain argument variables. A mode or moded type analysis can then guarantee that the required degree of instantiation is always reached at the time of executing a given goal. Let SC be the set of clauses in the definition of a predicate. Let us assume that we associate with each clause $C_i \in SC$ the subset M_i of the set of facts which define the meaning of such clause, so that the mutual exclusion of such subsets makes the predicate determinate. The condition to be checked is that for a given abstract constraint λ, and for every constraint c approximated by λ, it holds that for every two sets M_i and M_j, $i \neq j$, there are at least one $f_i \in M_i$ and one $f_j \in M_j$, such that either $f_i \wedge c$ or $f_j \wedge c$ is inconsistent, but not both. Note that, in our terms, a determinacy condition is one built up from the M_i sets which is sufficient for such inconsistency to hold. The main issue in performing such a global analysis is to take into account the different selection rule being used (see Section 6).

In some cases, even a simple local analysis of the program can allow optimizations. A simple case of determinate goals, for which nothing more than local inspection of the program text is needed, is that of goals of a predicate defined by a unique clause. In this case, a straightforward compile-time optimization can be achieved by a naive one-step unfolding of such goals. For some other predicates, determinacy can be inferred by simply examining the head of the clauses and/or simple built-ins appearing at the beginning of the body.

5.4 Determinacy condition is false

When the determinacy condition is reduced to false, it is clear that the general transformation based on an if-then-else defined previously can be reduced to its "else" part. The only thing needed is introducing a leftmost-token (i.e., to attach a particular variable), so that goals always suspend until leftmost. In this case the original definition of predicate `p` is not only renamed to `p_susp`, but additionally an extra argument is added on which to suspend.

In this case, the condition to be checked is that, for at least two sets M_i and M_j of facts which are true for clauses C_i and C_j of the predicate, it holds that for every $f_i \in M_i$ and $f_j \in M_j$, $i \neq j$, and for every constraint c (which could possibly happen upon execution of the program), $f_i \wedge f_j \wedge c$ is consistent. In some cases, it is possible to detect simply from the definitions of predicates that it is not possible for the clauses to be exclusive. This happens if for every $f_i \in M_i$ and $f_j \in M_j$, $i \neq j$, $f_i \wedge f_j$ is consistent.

5.5 An optimized algorithm

The optimizations presented, except that of unchaining calls, can be defined in terms of program specialization and code reduction. Having such a specializer, together with a simple partial evaluator, the transformation proposed can default to the most general one, plus unchaining. All other optimizations will then be performed by specialization and partial evaluation of the transformed program. However, because doing the whole process in one single step can be more efficient, and also because some unfoldings can be done which relate to predicates affected by delay declarations (something that a partial evaluator will normally not consider), we present a generic algorithm performing the translation from the original program and achieving all the optimizations.

Let SC be the set of clauses which define predicate `p/n`, with most general goal $\mathtt{p(\bar{X})}$, and $C(\bar{\mathtt{X}})$ be the determinacy condition w.r.t. $\mathtt{p(\bar{X})}$, already simplified making use of the information available. The transformation will substitute SC by a new set of clauses SC' as follows:

- If $C(\bar{\mathtt{X}}) = true$ and *unchained*(`p/n`) holds, the predicate will never suspend and no goal will be suspended during its execution. Therefore, we need neither suspension conditions, nor chain pointers. Thus, $SC' = SC$.
- If $C(\bar{\mathtt{X}}) = true$ but *unchained*(`p/n`) does not hold, the predicate will never suspend but goals might suspend during its execution. Thus we might need chain

pointers. Therefore, $SC' = chain(SC, \texttt{p})$.

- If $C(\bar{\texttt{X}}) = false$ and $unchained(\texttt{p/n})$ holds, the predicate will always be initially suspended but, once it has been woken, no goal will be suspended during its execution. Thus, we will just need to attach a variable for the leftmost token, and place a condition on the instantiation state of such variable. Therefore SC' is equal to SC plus the following clauses:

  ```
  :- block p(?̄,-,?).
  p(X̄,I,O):- p(X̄), O=I.
  ```

- If $C(\bar{\texttt{X}}) = false$ and $unchained(\texttt{p/n})$ does not hold, we may also have to add chain pointers. Therefore SC' is equal to $chain(SC, \texttt{p\_det})$ plus the following clauses:

  ```
  :- block p(?̄,-,?).
  p(X̄,I,O):- p_det(X̄,I,NO), O=NO.
  ```

- Otherwise, SC' is formed by the following clauses:

  $\texttt{p(}\bar{\texttt{X}}\texttt{,I,O):-}$ $goal(C(\bar{\texttt{X}}))$ `->` $work\_goal(\texttt{p(}\bar{\texttt{X}}\texttt{)},\texttt{I},\texttt{O})$ `;` $det\_susp(C(\bar{\texttt{X}}),\texttt{p(}\bar{\texttt{X}}\texttt{)},\texttt{I},\texttt{O})$
  $work\_def(SC, \texttt{p(}\bar{\texttt{X}}\texttt{)})$

  where $work\_def$, $work\_goal$, and det_susp are defined as follows:

$$work\_def(SC, \texttt{p(}\bar{\texttt{X}}\texttt{)}) = \begin{cases} SC & if\ unchained(\texttt{p/n}) \\ chain(SC, \texttt{p\_det}) & otherwise \end{cases}$$

$$work\_goal(\texttt{p(}\bar{\texttt{X}}\texttt{)},\texttt{I},\texttt{O}) = \begin{cases} \texttt{p(}\bar{\texttt{X}}\texttt{), O=I} & if\ unchained(\texttt{p/n}) \\ \texttt{p\_det(}\bar{\texttt{X}}\texttt{,I,NO), O=NO} & otherwise \end{cases}$$

$$det\_susp(C(\bar{\texttt{X}}),\texttt{p(}\bar{\texttt{X}}\texttt{)},\texttt{L0},\texttt{L1}) =$$
$$\begin{cases} if\ block(C(\bar{\texttt{X}})) & \begin{cases} \texttt{p\_susp(}\bar{\texttt{X}}\texttt{,I,O).} \\ \texttt{:- block p\_susp(}block(C(\bar{\texttt{X}}))\texttt{,-,?).} \\ \texttt{p\_susp(}\bar{\texttt{X}}\texttt{,I,O):- } work\_goal(\texttt{p(}\bar{\texttt{X}}\texttt{)},\texttt{I},\texttt{O})\texttt{.} \end{cases} \\ if\ when(C(\bar{\texttt{X}})) & \texttt{when((} when(C(\bar{\texttt{X}}))\texttt{ ; nonvar(I)), } work\_goal(\texttt{p(}\bar{\texttt{X}}\texttt{)},\texttt{I},\texttt{O})\texttt{)} \end{cases}$$

where $chain(SC, \texttt{f})$ is identical to the function $chain(SC)$ defined before but using the functor `f` instead of `p_susp`, $goal(C(\bar{\texttt{X}}))$ gives the Prolog goal corresponding to a determinacy condition $C(\bar{\texttt{X}})$, $block(C(\bar{\texttt{X}}))$ gives the sequence of `block` annotations corresponding to $C(\bar{\texttt{X}})$ or fails if it is not possible to do so, and $when(C(\bar{\texttt{X}}))$ does the same for `when` annotations. These functions, as well as the general transformation for when these two cannot be applied, have been informally defined in Section 3.

6 Applying Global Analysis

As mentioned during the description of the optimizations, global analysis can potentially greatly improve detection of when the optimizations can be applied. Unfortunately in this application Prolog semantics based analyses are not safe: such analyses will not take into account that goals may run "ahead of their turn." Therefore the inferred state of instantiation of the variables would not be guaranteed to always hold at execution time. Analyses designed for traditional concurrent logic languages cannot be used "as is" either, since most of these analyses do not take into account backtracking and are designed for a different delay rule. Even if backtracking were included in the semantics, the language assumed is generally concurrent by default, and all possible interleavings of the computation usually have to be considered. Since this is not our case, a loss of accuracy may be expected from such an analysis.

Analyses aimed at sequential programs with delay primitives, such as for example [19], are on the other hand closer to our purposes. Such an analysis can be applied directly to the transformed programs (we refer to this as an "a posteriori" analysis). The analysis keeps track of the possible suspensions occurring at each point in the program, while computing safe approximations of the instantiation states of the program variables. Such information can then be used to reduce the determinacy conditions, even if they cannot be detected to always hold, nor to always fail. Each test in the condition can be checked against the inferred information. Those which are found to be "abstractly executable" [12] (reducible to true, false, or some simple constraints) in their (abstract) context are replaced, and the applicable code reduction performed.

Alternatively, an analysis can be designed that directly models Andorra execution and can thus be applied before the transformation (we refer to this as an "a priori" analysis). Such an analysis can be based on an extension of classical Prolog analysis technology, following similar ideas to those of [6]. It keeps track of the relevant properties of the variables, while determining if the literals in the program will definitely not suspend. If it is not known that a literal will not suspend, then a safe approximation of the execution state is taken, and analysis proceeds. The non-suspension conditions on a predicate are identified as the *demand* of the predicate — in our context this demand is given by the determinacy conditions. The analysis identifies goals whose degree of instantiation satisfies the demand of the predicate.

Note that for both types of analysis, if definite success or failure of the determinacy conditions cannot be determined, the analysis can be unfolded for the two cases (as is already done in [19]). The cases in which determinacy of a predicate might condition the determinacy of other predicates can then be captured to a higher extent. To take advantage of this situation, and still be safe w.r.t. all cases at execution time, multiple program specialization [28, 21] of the resulting program should be done.

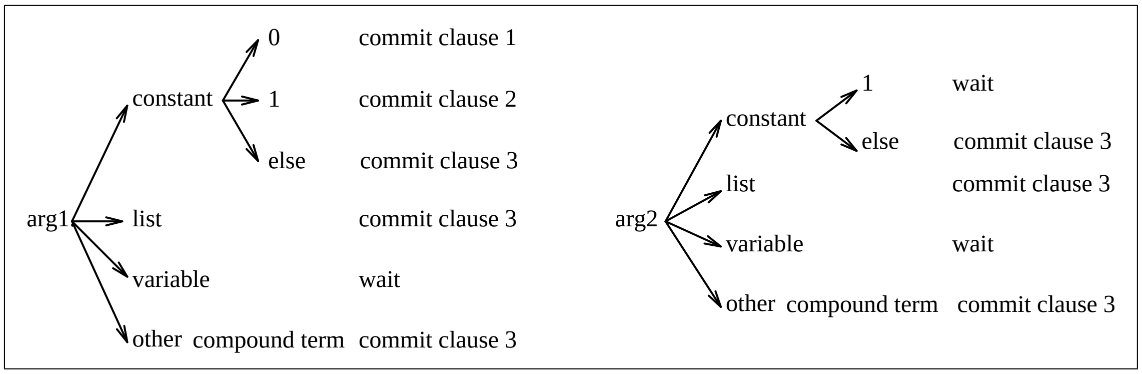

Figure 1: Andorra-I Decision Graph.

7 An Example

In this section we will illustrate the transformation and optimization procedure by means of a detailed, simple example, the well known program for computing the Fibonacci series.

```
fib(0,1).
fib(1,1).
fib(A,B) :- A>1, C is A-1, D is A-2, fib(C,E), fib(D,F), B is F+E.
```

The decision graph created by the Andorra-I preprocessor for this program is the one shown in Figure 1. Given that graph, it is easy to conclude that `fib(X,Y)` will be determinate as soon as (a) `X` becomes non-variable or (b) `Y` becomes a term not unifiable to 1. If only condition (a) had been necessary, a `block` suspension primitive would had been enough:

```
:- block fib(-,?,-,?).

fib(A,B,I,O) :- fib_det(A,B,I,NO), NO=O.

fib_det(0,1,T,T).
fib_det(1,1,T,T).
fib_det(A,B,T1,T7) :- >(A,1,T1,T2), is(C,A-1,T2,T3), is(D,A-2,T3,T4),
                      fib(C,E,T4,T5), fib(D,F,T5,T6), is(B,F+E,T6,T7).
```

However, given the need for a non-unifiability test, even a `when` declaration is not enough. Thus, the transformation needs to perform an explicit checking on the conditions the above graph represents:

```
fib(A,B,I,O):- nonvar(I), !, fib_det(A,B,I,NO), NO=O.
fib(A,B,I,O):- nonvar(A), !, fib_det(A,B,I,NO), NO=O.
fib(A,B,I,O):- nonvar(B), !,
        B\==1 -> fib_det(A,B,I,NO), NO=O
               ; when( ( nonvar(I) ; nonvar(A) ), fib(A,B,I,O) ).
fib(A,B,I,O):-
        when( ( nonvar(I) ; nonvar(A) ; nonvar(B) ), fib(A,B,I,O) ).
```

Given a(n) (abstract) query where the first argument is known to be ground, a relatively simple abstract interpretation-based analysis can usually determine that it will be ground recursively for all calls to `fib/2`, and thus that `fib/2` is determinate. Furthermore, it will also determine that the built-ins are directly reducible. In this case, the unchaining optimization discussed in Section 5.2 would be applicable, and would result in code identical to the original Prolog program.

An a posteriori analysis can also provide similar information. Furthermore, this kind of analysis may allow further optimizations. For example, given the transformed program, it is perfectly possible that queries occur with the first two arguments free and the third ground. In this case, since suspensions do occur, the a priori analysis would probably not infer the necessary information for optimizing the code. However, an a posteriori analysis can determine that for such queries no goal is ever determinate (i.e. the goals are only awakened on the attached variable `I`), and suspensions can be further reduced. The following program could be obtained:

```
:- block fib(?,?,-,?).

fib(X,Y,I,O) :- fib_det(X,Y,I,NO), NO=O.
```

8 Mixing Prolog and Andorra-I Code

As argued in the introduction, while the Basic Andorra Principle is certainly interesting for its pruning capabilities, in some cases a simple Prolog execution may be more desirable. This can be the case for programs known to be deterministic, or for which a fixed ordering of choice points is known to be best. In Prolog execution the determinacy checking overhead can be avoided (as well as the associated compilation time). One interesting possibility that the transformational approach brings is to mix execution in "Andorra mode" with normal (in this case, Prolog) execution. It is quite easy to call from straight Prolog code to "Andorra transformed" code and the other way around. We will assume that the source is marked in some way to distinguish those predicates that should be compiled as normal Prolog predicates from those that are to be compiled to support the Andorra model. The transformation is then done only on those predicates (files, modules,...) marked as meant to run under the Andorra model. Calls from Prolog to Andorra goals are done in the same way as shown previously for queries: a call to `wakeup/2` is introduced after the goal, so that the whole Andorra computation is completed before continuing the Prolog execution (if this is what is desired — we

Bench	Andorra-I	QE-Andorra+SICStus		$A/QE_{compiled}$	A/QE_{interp}	SICStus
	(A)	Compiled	Interpreted			Compiled
`crypt`	196,972	196,400	214,620	1.003	0.918	2,070,760
`dia_sums`	135,023	280,010	891,860	0.482	0.151	3,730
`fib`	112	89	480	1.258	0.233	30
`map`	470	19	39	24.737	12.051	10
`money`	751	980	1,010	0.766	0.744	1,606,370
`mqu`	136,196	205,660	1,045,369	0.662	0.130	> 2 days
`mutest`	223	69	409	3.232	0.545	10
`qu_evan`	8,956	9,550	14,200	0.938	0.631	49
`qu_vitor`	7,765	31,310	34,020	0.248	0.227	190
Geometric Mean:				1.214	0.523	–

Table 1: Execution times in milliseconds

assume the intended operational behavior is to isolate both executions).[2] Calls from Andorra-I to Prolog are simply not transformed (no leftmost-token passing), and will be then executed normally, outside the context of any Andorra goals (again, if this is what is desired). An interesting alternative, from the point of view of marking Andorra and Prolog execution parts would also be to simply mark certain calls as Andorra calls (by, for example, wrapping them in a `bam/1` goal). The compiler would then simply generate special, transformed versions of all the predicates called by that goal and its descendents (in addition to the normal ones). This avoids having to mark program parts instead.

9 Performance Figures

In this section we present results obtained from a preliminary implementation of the proposed approach. The system implements the automatic transformation described in the previous sections. Optimizations are applied as follows (no global analysis is used in the experiments presented). First, the necessary code to avoid the suspension of a goal that is determinate at the moment it is processed is added to the program, as explained in Section 5.1. Second, eligible predicates are unchained following the algorithm described in Section 5.2. Third, cases in which the determinacy conditions are `false` are taken into account and simplified. Finally, predicates which are determinate because their definition is a single clause are also taken into account and simplified. However, they are not unfolded into the calling clause since Andorra-I does not perform this optimization.

The benchmark programs have been previously used in benchmarking Andorra-I

[2]Note that this has some resemblance to a deep guard, as used in AKL [17].

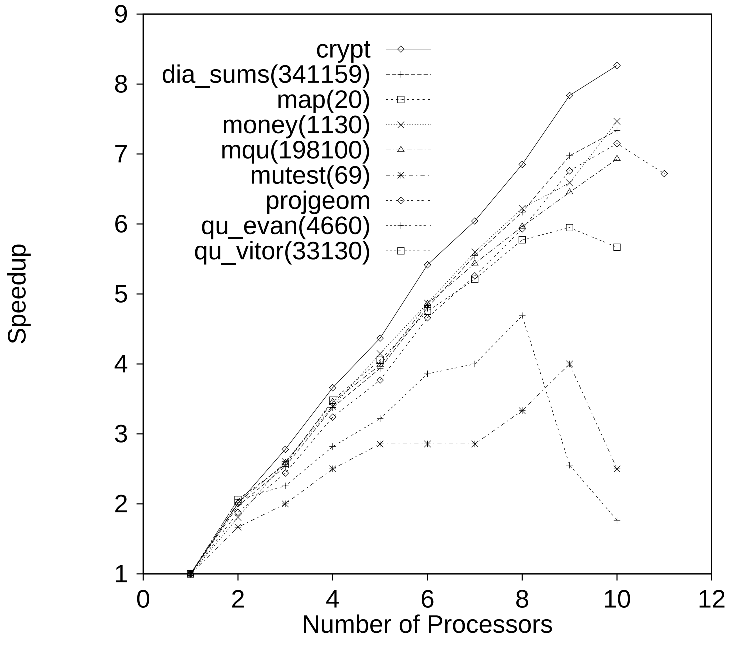

Figure 2: Speedup Curves: Sicstus 3.0/Muse

and have been taken directly from the Andorra-I distribution. They were first run directly with the Andorra-I system, on a 55MHz SPARC-10, with 64Mbytes of memory, SunOS 4.1.3. The same programs were transformed by our preprocessor and then run on SICStus 2.1(8), on the same machine. For the latter, two cases were considered: compiling to native code, and interpreting the source code. This was done because the implementation technology of the Andorra-I system available to us (which is not the latest, fully compiler-based version), is somewhere between a compiler and an optimized interpreted system. The bodies of the clauses are executed by an optimized interpreter written directly in C, which should be faster than the source-level meta-interpreter used in the SICStus interpreter. The indexing operations are compiled by the preprocessor into specialized instructions. These may be slower than the native code produced by the SICStus compiler, although on the other hand the indexing is more sophisticated, since it is done on all arguments, while SICStus only indexes on the first argument. Table 1 shows the resulting execution times in milliseconds (the times given for QE-Andorra include garbage collection time; we were unable to determine whether the Andorra-I system was doing garbage collection or not). For comparison, the result of running the original programs directly on SICStus Prolog, compiling to native code, is also shown.

The performance of the resulting system, even without any optimizations based on global dataflow analysis, is comparable to the native Andorra-I implementation: using compiled SICStus Prolog, our system is about 20% faster, on the average, than the Andorra-I system; if we disregard the two outliers in the benchmark suite—`map`, on which QE-Andorra significantly outperforms Andorra-I, and `qu_vitor`, where it is outperformed—the two systems have essentially identical performance on the average. By using global analysis, these results could be improved further. Preliminary experiments on some of the benchmarks show encouraging speedups (e.g. 1.45 for `fib`, 1.09 for `map`).

The comparison with direct Prolog execution shows that there is advantage to avoiding the transformation overheads, except for those cases where the Andorra selection rule is performing better search. This supports our idea of mixing Prolog/Andorra execution, and also that if the transformation can be optimized more, better results can be obtained.

One additional advantage of the transformation is that existing parallel implementations can be exploited with little additional effort. In particular, note that or-parallelism comes for free by simply running the transformed program on an or-parallel system, such as Muse [1] or Aurora [18]. In order to test this we ran a subset of the transformed programs (those that involve search) on SICStus V3.0, with the or-parallelism option turned on (which uses the Muse model), on a 10 processor SPARC Server 2000 (each processor running at 55MHz). The results are represented graphically in Figure 2. Quite reasonable speedups were obtained automatically on these programs.

10 Conclusions

We have reported on a transformation technique which allows supporting the Andorra computational model, essentially emulating the Andorra-I system, via program transformation into a sequential language with delay primitives. We have also proposed several optimizations to the transformation. The system is automatic, comprising a basic transformer to the kernel language, which can optionally be interfaced with a global analyzer. The preliminary results are quite encouraging: performance of the resulting system is comparable to the current Andorra-I implementation, even without global analysis.

Given the results obtained, we argue that the "quiche-eating" approach is practical, and allows the support of the Basic Andorra Model on a generic system with reasonable performance. We do not mean to suggest that the performance of a system implemented using our approach is optimal or that it will achieve in the end better results than a highly optimized, native Andorra-I implementation, but rather that the transformational approach we pursued is viable. This is specially useful in view of the proposed methods for combining traditional Prolog (or CLP) code and Andorra code.

We plan to further optimize and benchmark the system. Coupling the transformation with the global analyses we have outlined could drastically improve the overall performance. The global analyses sketched are actually under construction. We are also planning on testing performance on other parallel systems. In particular, we also plan on implementing and-parallelism (both determinate-dependent and also independent) by using the recently proposed notions of independence in systems with delay [4] and the associated compilation technology. And/Or parallelism can potentially be directly supported on a system like ACE [13].

Acknowledgments

We would like to thank D.H.D. Warren, V. Santos-Costa, R. Yang, and the rest of the Andorra-I team for making a version of the Andorra-I system available to us and for useful discussions. The work of F. Bueno, M. García de la Banda and M. Hermenegildo was funded in part by ESPRIT project #7195 "ACCLAIM" and by CICYT project TIC93-0975-CE. The work of S. Debray was funded in part by the National Science Foundation under grant number CCR-9123520.

References

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

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

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

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

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

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

7. S. K. Debray. QD-Janus : A Sequential Implementation of Janus in Prolog. *Software—Practice and Experience*, 23(12):1337–1360, December 1993.

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

9. European Computer Research Center. *Eclipse User's Guide*, 1993.

10. M. García de la Banda, F. Bueno, S. Debray, and M. Hermenegildo. QE-Andorra: A Quiche–Eating Implementation of the Basic Andorra Model. Technical Report CLIP13/94.0, T.U. of Madrid (UPM), September 1994.

11. I. Foster and S. Taylor. *Strand* : A practical parallel programming tool. In *1989 North American Conference on Logic Programming*, pages 497–512. MIT Press, October 1989.

12. F. Giannotti and M. Hermenegildo. A Technique for Recursive Invariance Detection and Selective Program Specialization. In *Proc. 3rd. Int'l Symposium on Programming Language Implementation and Logic Programming*, pages 323–335. Springer-Verlag, 1991.

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

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

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

16. A. Houri and E. Shapiro. A sequential abstract machine for flat concurrent prolog. Technical Report CS86-20, Dept. of Computer Science, The Weizmann Institute of Science, Rehovot 76100, Israel, July 1986.

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

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

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

20. L. Naish, personal communication, June 1995.

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

22. V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, April 1990.

23. V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In *1991 International Conference on Logic Programming*, pages 825–839. MIT Press, June 1991.

24. V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *1991 International Conference on Logic Programming*, pages 443–456. MIT Press, June 1991.

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

26. E. Tick and C. Bannerjee. Performance evaluation of monaco compiler and runtime kernel. In *1993 International Conference on Logic Programming*, pages 757–773. MIT Press, June 1993.

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

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