facultad de informática

universidad politécnica de madrid

Specialization and Optimization of Constraint Programs with Dynamic Scheduling

Germán Puebla
Manuel Hermenegildo

Specialization and Optimization of Constraint Programs with Dynamic Scheduling

Authors

Germán Puebla
Universidad Politécnica de Madrid (UPM), Facultad de Informática, 28660-Boadilla del Monte, Madrid - Spain, german@fi.upm.es

M. Hermenegildo
Universidad Politécnica de Madrid (UPM), Facultad de Informática, 28660-Boadilla del Monte, Madrid - Spain, herme@fi.upm.es

Keywords

Abstract Interpretation, Logic Programming, Constraint Logic Programming, Compile-time Analysis, Multiple Program Specialization, Optimization, Program Specialization.

Abstract

In this report we discuss some of the issues involved in the specialization and optimization of constraint logic programs with dynamic scheduling. Dynamic scheduling, as any other form of concurrency, increases the expressive power of constraint logic programs, but also introduces run-time overhead. The objective of the specialization and optimization is to reduce as much as possible such overhead automatically, while preserving the semantics of the original programs. This is done by program transformation based on global analysis. We present implementation techniques for this purpose and report on experimental results obtained from an implementation of the techniques in the context of the CIAO compiler.

1 Introduction

Most recent logic programming languages provide more flexible scheduling than Prolog traditional left-to-right computation rule. Some calls are dynamically "delayed" until their arguments are sufficiently instantiated to allow the call to run (efficiently). Such languages include constraint logic programming languages in which constraints which are "too hard" are delayed. Also, most implementations of concurrent constraint logic programming languages essentially follow a left to right fixed scheduling rule where certain goals suspend as determined by their `ask` guards. This scheduling, often referred to as *dynamic scheduling* increases the expressive power of constraint logic programs, but also introduces significant run-time overhead. The main purpose of this report is to reduce as much as possible this additional overhead introduced by dynamic scheduling by means of global analysis and program transformation while preserving the semantics of the original programs. The run-time overhead introduced by dynamic scheduling can be divided in two types:

Delaying cost: Execution of goals affected by delay declarations involves checking certain conditions to decide whether the goal should be delayed or not. Then, if the goal must be delayed some additional run-time overhead will be introduced to delay the goal, i.e. put the goal in the list of delayed goals, etc.

Waking cost: Goals that are delayed must be checked upon variable binding to see if they can be woken. In that case they must be executed before the following goal.

As well as computation time overhead, dynamic scheduling also introduces some memory consumption overhead, as delayed goals must be stored until they are woken and because dynamic scheduling precludes some low-level optimizations such as register allocation.

The potential benefits of the optimization of delay declarations were already illustrated in [9]. In that paper, a new global analysis framework for programs with dynamic scheduling was introduced and the information obtained was then used to optimize programs by hand. The optimized programs were used to obtain some preliminary empirical evaluations that showed very promising results. In this report we perform a detailed description of several optimization methods for reducing the run-time overhead introduced by dynamic scheduling while preserving the semantics of the original program. This study should be useful as a basis for automatic optimization of many kind of programs for execution models which implement dynamic scheduling.

1.1 Syntax and Semantics of the Delay Declarations

The delay declarations we will consider in this report are the following:

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

`Goal` is blocked until `Condition` is *true* . `Condition` is a Prolog goal given by the following restricted syntax:

Condition ::=
`nonvar(X)` | `ground(X)` | $?=(X, Y)$ | *Condition*, *Condition* | *Condition*; *Condition*
where `?=(X,Y)` is essentially a restricted type of `ask` unification. It succeeds if the terms `X` and `Y` are identical or they cannot unify.

```
freeze(?X,Goal).
```

`Goal` is blocked until `nonvar(X)` holds.

`:- 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.

We assume that other delay primitives such as `ask` agents are transformed into these primitives as shown in [4].

1.2 Equivalences Among Delay Declarations

`When/2` is the most expressive delay declaration of the three of them. In fact, both `freeze/2` and `block/1` can be expressed in terms of `when/2`. `Freeze/2` meta-calls can be defined as:

```
freeze(X,Goal):-
        when(nonvar(X),Goal).
```

As a result, in the remainder of the report we will not consider optimization of `freeze/2` at all, as they are a special case of `when/2`.

`Block` declarations can also be expressed in terms of `when/2` meta-calls. An important difference between them is that `block` declarations express conditions under which the literal is delayed and `when` meta-call express conditions under which the literal is *not* delayed. In general, `block` declarations could be replaced by as many `when/2` meta-calls as literals appear in the program for that predicate.

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

..., p(X,Y,Z),...
```

can be transformed into

```
..., when(((nonvar(X);nonvar(Z)),(nonvar(X);nonvar(Y))),p(X,Y,Z)), ...
```

In principle, and as we have seen in this section, all kinds of delay declarations can be transformed into `when` meta-calls. As a result, optimization of delay declarations can be reduced to optimization of `when` meta-calls. This is what we study in the following section.

2 Optimization of Delay Declarations

2.1 Optimization of `when` meta-calls

We will consider several cases in the optimization of `when` meta-calls. They correspond to different situations in which the information available allows reducing part of the condition in the `when` meta-call (referred to as `Condition`) or `Condition` as a whole to either the value *true* or the value *false* .

2.1.1 A check is true

By a check we mean one of the following simple tests: `ground(X)`, `nonvar(X)`, and `?=(X,Y)`. These three tests are *downwards closed*. This means that if they hold in a certain program point, they will continue to hold in forwards execution. Literals in `when` meta-calls are executed where they appear in the program code or later in forwards execution. `When` meta-calls are backtrackable in the sense that if they were delayed they are eliminated from the list of delayed literals when backtracking reaches the `when` meta-call. If they were not delayed backtracking proceeds for them as for any other literal. As backtracking cannot bind any variable, it is not possible that any `when` is woken during backwards execution. As checks are downwards closed and `when` meta-calls are only woken in forwards execution we can simplify `Condition` following the usual rules we now enumerate:

$$\begin{aligned}(true, Cond) &\equiv Cond\\ (Cond, true) &\equiv Cond\\ (true; Cond) &\equiv true\\ (Cond; true) &\equiv true\end{aligned}$$

2.1.2 A check is false

Unlike the tests `ground(X)`, `nonvar(X)`, and `?=(X,Y)` their negations are not downwards closed. This means that we cannot directly replace checks by *false* and simplify the resulting expression. For example, the fact that `ground(X)` fails at a certain program point does not guarantee that it will continue to hold in forwards execution. For example,

```
p:- when(ground(X),write(X)), q(X).
q(1).
q(2).
```

Thus, checks that are false must be maintained in `when` meta-calls as they may become true afterwards and wake the literal.

2.1.3 `Condition` *is* true

If we are able to simplify a whole `Condition` to the value *true* , we can be sure that the literal will not delay and the `when` meta-call can be replaced by the plain literal. For example,

```
p:- q(X), when(ground(X),write(X)).
```

```
q(1).
q(2).
```

Can be transformed into

```
p:- q(X), write(X).
q(1).
q(2).
```

2.1.4 `Condition` *is* false

The optimization of a `when` meta-call in which `Condition` is *false* is not as simple as when it is *true* . That `Condition` is false implies that this literal cannot be executed at the point it appears in the program. Instead it will be delayed. As said before, `Condition` cannot be simplified. One possible optimization is to reorder literals in the clause moving the `when` meta-call towards the right as long as we are sure that the literal cannot be executed. The potential benefits are two-fold. On the one hand we reduce delaying cost by not having to check `Condition` (we are sure it will fail), on the other hand we reduce waking cost, if the literal is delayed later we will save checking if it can be woken (these checks will also fail if the reordering is correct). The main problem in reordering goals is that to be able to safely move a `when` meta-call one or more positions to the right, it is not enough to prove that the meta-call will delay. We also have to prove the following two conditions, where *Lits* represents a sequence of one or more literals just to the right of the `when` meta-call being moved:

1. `Condition` is *false* after the execution of *Lits*.
2. The execution of *Lits* cannot leave additional delayed literals on the same variables (modulo variable aliasing) as `Condition` .

The second condition is needed to guarantee that the reordering will not modify the order in which goals delayed upon the same variable are woken. Unfortunately, this second condition is not easy to prove. We can instead use stronger sufficient conditions which are simpler to prove. Another possibility would be to ignore this second condition if we believe that the order in which goals delayed on the same variable are woken is not important in our application.

We now propose an algorithm to reorder `when` meta-calls in a clause that ensures that the two previous conditions hold. First we group as many contiguous literals as possible that will surely delay, i.e., they are `when` meta-calls (or other delay declarations) whose condition is *false*. Then we also group contiguous literals that will surely not wake any of the delayed literals and will not leave additional delayed goals. Finally, we reverse their order in the clause.

1. Let *Delayed* be a `when` meta-call which is guaranteed to suspend. Let *delayed_group* be $\{Delayed\}$ and *advanced_group* be $\{\}$. Let *Lit* be the next literal in the clause.
2. If there are more literals in the clause let *Lit* be the following one. If *Lit* surely delays, we add this literal to the delayed group. Repeat step 2.
3. If there are more literals in the clause let *Lit* be the following one. If *Lit* does not delay and cannot leave additional delayed literals then add *Lit* to *advanced_group*. If all the conditions for the meta-calls in *delayed_group* are sure to fail after executing *Lit* then Repeat step 3. Else goto step 4.

4. The new clause is obtained by reversing *delayed_group* and *advanced_group*.

2.2 Optimization of `Block` declarations by Specialization

As we have seen in Section 1.2, one possibility to optimize block declarations is to transform them into `when` meta-calls and then apply the optimizations described in Section 2.1. However, it is usually the case that `block` declarations are implemented in a much more efficient way than `when` meta-calls. Even if we are able to optimize the `when` meta-calls, the final program after translating `blocks` into `whens` and optimizing it may be slower than the original one. One possibility would be to transform `block` declarations into `whens` only if `Condition` can be simplified to *true* in all the literals that call the predicate affected by the `block` declaration. This means that the `when` meta-calls are not needed and is equivalent to eliminating the `block` declaration. In many practical situations `block` conditions could be simplified for some of the literals that call that predicate but not all. Note that `block` declarations affect all the literals that call the corresponding predicate. Thus, in principle, `block` declarations can only be simplified if the simplification is allowed in all the literals that call such predicate. This optimization scheme is too restrictive in that it precludes any kind of optimization for particular literals unless it is possible for all calls to the predicate. One way to overcome this problem is by means of multiple specialization of programs [13]. Multiple specialization involves the generation of several versions of a procedure for different uses. This technique has successfully been implemented to optimize logic programs [12], but, to our knowledge, has never been applied to optimize constraint logic programs with dynamic scheduling.

The main idea here is that whenever it is possible to simplify the `block` declaration for a predicate at a given literal (not necessarily all of them), we will create a special version for the predicate with a new name. In order not to increase the size of the multiply specialized program unnecessarily, all the literals for which the simplifications in the `block` declaration are the same should share the same version. Then, we will generate a simplified block declaration for the specialized version along with the code for the specialized version. Finally, we must replace calls to the general version by calls to the specialized versions whenever possible. For example, the following program:

```
:- block p(-,?,-),p(-,-,?).
p(X,Y,Z):-
        code_for_p/3.

..., p(A,B,C), p(Arg1,Arg2,[1,2,3]), ...
```

can be transformed into

```
:- block p(-,?,-),p(-,-,?).
p(X,Y,Z):-
        code_for_p/3.
:- block p_sp_vers(-,-,?).
p_sp_vers(X,Y,Z):-
        code_for_p_sp_vers/3.

..., p(A,B,C),p_sp_vers(Arg1,Arg2,[1,2,3]), ...
```

In this kind of multiple specialization the different optimizations from one version to another take place in the `block` declaration, but the code for all versions of a predicate is the same. Thus, one further program transformation we could use in order to avoid the increase in program size is to make the different versions for a predicate share as much code as possible. This can be achieved by creating a new predicate with the code of the original one but without any `block` declaration (`p_no_block`). The different specialized versions will have their `block` declaration and they will immediately call `p_no_block`. For example, the previous program can be transformed into

```
:- block p(-,?,-),p(-,-,?).
p(X,Y,Z):-
        p_no_block(X,Y,Z).

:- block p_sp_vers(-,-,?).
p_sp_vers(X,Y,Z):-
        p_no_block(X,Y,Z).


..., p(A,B,C),p_sp_vers(Arg1,Arg2,[1,2,3]), ...

p_no_block(X,Y,Z):-
       code_for_p/3.
```

2.2.1 A check is true

As said in Section 1.2, `block` declarations state conditions under which the literal must delay. For example, the declaration

```
:- block p(-,?,-),p(-,-,?).
..., p(Arg1,Arg2,Arg3), ...
```

can be interpreted as

```
...,if((var(Arg1),var(Arg3));(var(Arg1),var(Arg2)))
    then delay p(Arg1,Arg2,Arg3)
    else p(Arg1,Arg2,Arg3), ...
```

Following a reasoning similar to that of Section 2.1.1, the test `var/1` is not downwards closed, so no simplification can be done if it is sure to succeed.

However, if the whole condition can be simplified to *true* in a particular literal, that literal can be moved to the right in the clause, as we explained in Section 2.1.4, provided the two following conditions hold, where *Lits* represents a sequence of one or more literals just to the right of the literal being moved.

1. `Condition` is true after the execution of *Lits*.
2. The execution of *Lits* cannot leave additional delayed literals on the same variables (modulo variable aliasing) as `Condition` .

We can easily adapt algorithm in Section 2.1.4 to the case of literals affected by block declarations.

2.2.2 *A check is* false

`Block` declarations can only contain tests of the kind `var/1`. The negation of this check, `nonvar/1` is downwards closed and thus they can be simplified. It must be noted that due to the restrictive syntax of `block` declarations the suspension condition cannot be arbitrarily simplified. We want the simplified condition to be in turn expressible as a new block declaration, i.e., in disjunctive normal form. This can be achieved by only allowing simplifications that eliminate conjunctions of `var` tests. The result is obviously again in disjunctive normal form. The kinds of optimizations we can use are:

$$\begin{aligned}(fail, Cond) &\equiv fail\\ (Cond, fail) &\equiv fail\\ (fail; Cond) &\equiv Cond\\ (Cond; fail) &\equiv Cond\end{aligned}$$

2.2.3 *Generation of Simplified Block Declarations*

Once the suspension condition has been simplified, we generate a new block declaration for it. If the simplified condition is the value fail, no `block` declaration is needed. This means that the literal will never delay. If $condition = (C_1, \ldots, C_n)$ then we generate a `block` declaration of the form : $-block\ Spec_1, ..., Spec_n.$ such that $Spec_i = p\_new\_name(Arg1_i, Arg2_i, Arg3_i)$ where $Argk_i = '-'$ iff $C_i \Rightarrow var(Argk)$ and $Argk_i = '?'$ otherwise.

3 Analysis Information Required

In this section we will discuss the analysis information needed to perform the optimizations introduced in Section 2. The different optimizations introduced require different kinds of analysis information.

In order to simplify `when` conditions, we need analysis information to determine whether the tests `nonvar/1`, `ground/1`, and `?=(X,Y)` can be reduced to either *true* or *false*. As seen in Section 2.1.1, it is easy to use analysis information for simplifying checks to *true*. Information to reduce a check to *false* is only useful if it is possible to determine condition 2 in Section 2.1.4.

To simplify `block` declarations we are only interested in information that allows determining whether `var/1` is true or false. Here we are specially interested in information to reduce `var` tests to fail, that is equivalent to reducing `nonvar/1` tests to true. Again, information to reduce `var` tests to true is only useful if it is possible to determine condition 2 in Section 2.1.4.

Global analysis, in terms of abstract interpretation [5], seems to be a good candidate for inferring the information required for optimization of both `when` and `block` conditions. Unfortunately, global data-flow analyses used in the compilation of traditional programs [2, 11, 7], are not correct in the context of dynamic scheduling. In addition, it is not simple to extend analyses for traditional Prolog and constraint logic programming languages to languages with dynamic scheduling, as in existing analyses the fixed scheduling is crucial to ensure correctness

and termination. As a result, special analysis frameworks for dynamic scheduling, such as [9] have to be used. The framework presented in [6] presents little overhead for programs without delay and the performance on programs with delay is reasonable and considerably better than the only other comparable approach [9].

In addition to an analysis framework capable of dealing with dynamically scheduled programs, the choice of a suitable abstract domain is crucial. It should provide the kind of information we are interested in with enough accuracy to allow important optimizations and in a reasonable amount of time.

4 Implementation of Dynamic Scheduling Specialization and Optimization

4.1 Program Analysis

The current implementation of the specializer and optimizer for constraint programs with dynamic scheduling uses the analysis framework described in [6]. This framework is generic in the sense that it has a parametric domain and various parametric functions. The parametric domain is the descriptions chosen to approximate constraints. Different choices of descriptions and associated parametric functions provide different information and give different accuracy. The experimental results have been obtained using two different parametric (or abstract) domains. One of them, namely *def* (definite Boolean functions [1]) can be used with both logic programs and constraint logic programs. The other domain used, *sharing+freeness* [10] can only be used with logic programs.

4.2 Program Transformations

We present the current implementation of the optimization techniques introduced in Sections 2.1 and 2.2. We discuss the capabilities and limitations of the implementation.

Dom	`nonvar`	`ground`	`?=`	`var`	`not_g`	`not ?=`
def	X	X				
shfr	X	X		X	X	

Table 1: Information provided by abstract domains

4.2.1 Optimization of `Whens`

Table 1 shows for each type of test that can appear in a `when` meta-call whether it can be inferred with the information provided by each abstract domain implemented. A cross means that it is possible to infer that the test will succeed. The last three columns correspond to the negation of the simple tests (equivalent to showing the failure of the positive tests).

The current implementation is able to simplify conditions in `when` meta-calls when checks are shown to be true. Also, the optimizations allowed when `Condition` is simplified to true (Section 2.1.3) are implemented. Regarding the optimizations described in Sections 2.1.2 and 2.1.4, it is important to note that these optimizations are only possible if the analysis is able

to infer some kind of negative information regarding the checks allowed in `when` conditions. The Table 1 shows that it is not possible to infer any negative information with the abstract domain *def* and thus it is not possible to perform any of these optimizations. With the second abstract domain currently implemented in the CIAO compiler [3], *sharing+freeness*, we can derive information about `var/1`, `not_ground/1` but not about the third type of check. It would require a `depth(K)` analysis. However, the optimizations introduced in Sections 2.1.2 and 2.1.4 are not currently implemented in the system.

4.2.2 Optimization of `Blocks`

Regarding the optimization of `block` declarations, the program transformations described in Sections 2.2.2 and 2.2.3 are implemented. As previously stated, the restricted syntax allowed in `block` declarations forces new `block` conditions to be in disjunctive normal form. This is trivially achieved by eliminating from a `block` declaration of the form `:- block` $Spec_1$`,` ..., $Spec_n$`.` as many *Spec* terms as we can prove to be false. The main drawback of the optimization of `block` declaration is that it involves generating new versions of predicates, thus increasing program size. In order to minimize the size of the new program, before generating a new version of a predicate associated to a simplified block declaration we check if a version associated to the same simplified block declaration already exist. We have also implemented the program transformation introduced in Section 2.2 in which all versions for the same procedure share code. This makes the increase in program size minimal. In order to make each literal use the corresponding specialized version we rename calls to the generic predicate with calls to the specialized versions of the predicate.

The program obtained by specializing block declarations is finite, as the number of specialized versions we can generate for each predicate with a block declaration is finite and is bounded by $\sum_{i=1}^{n}\binom{n}{i}$, where n is the number of *Spec* terms in its `block` declaration.

5 Experimental Results

In order to assess the practicality of the optimizations proposed in this report, we have performed a series of experiments in the CIAO Compiler. The benchmarks used for the evaluation were: `permute`, which succeeds if the first argument is a permutation of the second, `qsort`, the classical quick sort program using append; `app3` which concatenates three lists by performing two consecutive calls to append; `nrev` which naively reverses a list; and the final two are the well-known CLP programs `fib` and `mortgage`, modified so that arithmetic delays until it can be computed by local propagation. All these benchmarks have been implemented in a reversible way, so that they can be used in two obvious modes of operation, forwards and backwards, through the use of suspension declarations. Note that though the declarative meaning of these programs explains both modes of operation, the fixed left-to-right scheduling rule does not allow running them in all modes in all cases.

5.1 Optimization of `When` meta-calls

The first experiment we performed was to measure the time required for the optimizations in our implementation. The results are shown in Table 2. Times are in milliseconds on a Sparc 10. For each benchmark program we give: **FAn** analysis time for forwards use of the program, **FOpt** optimization time using the information obtained in the previous analysis,

Benchmark	FAn	FOpt	BAn	FOpt
append3	33	16	116	30
nrev	56	23	112	33
permute	37	30	403	73
qsort	93	56	1472	210
mortgage	56	60	320	143
fib	82	60	257	83

Table 2: Optimization times

BAn analysis time for backwards use of the program, and **FOpt** optimization time using the information obtained for analysis backwards execution. The optimization times are comparable to those of analysis. Except for `mortgage`, they are always below analysis times for forwards execution. In the case of backwards execution, optimization times are always below analysis times.

It is important to note that, in principle, the optimization time can be considered to be the sum of the analysis and "optimization" times, as we need the analysis information to perform the optimizations. However, analysis information can be used for many other code optimizations apart from the optimization of dynamic scheduling, such as dead-code elimination, recognizing determinate code and thus allowing unnecessary choice-points to be deleted, improving the code generated for unification, recognizing calls that are independent and thus allowing the program to be run in parallel, etc. In other words, different types of program transformations and optimizations may share the analysis information. This is possible because the optimizations presented are semantics-preserving, and thus the analysis is still correct [8].

Benchmark	F	OF	B	OB	FS	BS
append3	32100	30	1750	1750	1070	1
nrev	20715	165	26200	26200	125.55	1
permute	2030	125	750	750	16.24	1
qsort	5320	135	5310	4550	39.41	1.17
fib	2150	110	5845	3340	19.55	1.75
mortgage	3380	150	4630	2750	22.53	1.68

Table 3: Simplification of `when` meta-calls

Table 3 shows the execution times of some constraint logic programs with dynamic scheduling expressed in terms of `when` meta-calls. This dynamic scheduling provides a more flexible use of such programs than the traditional left-to-right scheduling. The execution times are in milliseconds and have been obtained on a Sparc station IPC.

F is forwards execution time of the original program (which includes `when` meta-calls), **B** the original program but in backwards execution, **OF** is the program optimized for forwards execution, **OB** is the program optimized for backwards execution, **FS** is the forwards speed-up, i.e., **F/OF**, and **BS** is **B/OB**.

In all the benchmark programs our implementation of the optimization techniques has been

able to eliminate all delay declarations, obtaining a program which is equivalent to the original constraint program designed to work forwards. Thus, the automatic optimization of such programs allows having programs that are reversible and that have no run-time overhead when executed forwards. The benefits in forward execution are clearly stated by **FS** which shows significant speed-ups.

When these benchmarks are executed backwards, most of the `when` meta-calls are needed and thus are not eliminated by the optimizer. However, even in this case some speed-up is obtained due to the optimizer.

5.2 Specialization of `Block` declarations

We have used the same benchmark programs as in Section 5.1 but coded using `block` declaration instead of `when` meta-calls. The analysis information obtained has been useful to generate special versions of predicates with simplified `block` declarations. However, due to the very efficient implementation of `block` declaration in the CIAO system (based on SICStus Prolog), the run-time performance improvements are not as significant as in the simplification of `when` meta-calls. We believe, however, that optimization of `block` declarations may also be interesting, specially in systems where these declarations are not so efficiently implemented. The improvements obtained through specialization of `block` declarations are shown in Table 4. We have taken as a measure of optimality the total number of *Spec* terms needed in the `block` declarations. **B** is the number of terms in the original program, **OF** the number of them in the program optimized for forwards execution, and **OB** in the program optimized for backwards execution.

Benchmark	**B**	**OF**	**OB**
append3	2	0	1
nrev	3	0	3
permute	2	0	1
qsort	10	0	6
fib	5	0	2
mortgage	8	0	2

Table 4: Specialization of `block` declarations

We can clearly see that the optimizer has been able to eliminate all `block` declarations when the program was optimized for forwards execution. When the program was optimized to run backwards some improvements were still possible.

6 Conclusions and Future Work

We have presented a study of several optimization techniques for constraint programs with dynamic scheduling. We have also presented the conditions required to perform these optimizations and the information static analysis should provide to allow such optimizations. Part of these techniques have been implemented in the CIAO compiler and we have presented some experimental results that clearly show the potential benefits of the proposed optimizations.

References

1. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In Springer-Verlag, editor, *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 266–280, Namur, Belgium, September 1994.

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

3. F. Bueno. The CIAO Multiparadigm Compiler: A User's Manual. Technical Report CLIP8/95.0, ACCLAIM Deliverable D3.2/3-A4, Facultad de Informática, UPM, June 1995.

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

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

6. María García de la Banda, Kim Marriott, and Peter Stuckey. Efficient Analysis of Constraint Logic Programs with Dynamic Scheduling. Technical Report CLIP9/95.0, ACCLAIM Deliverable D3.2/3-A1, Facultad de Informática, UPM, March 1995.

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

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

9. 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.

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

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

12. 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.

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