1997 International Conference on Logic Programming, MIT Press.

Optimization of Logic Programs with Dynamic Scheduling

Germán Puebla
Facultad de Informática, Universidad Politécnica de Madrid
german@fi.upm.es

María García de la Banda, Kim Marriott
Dept. of Computer Science, Monash University
{mbanda,marriott}@cs.monash.edu.au

Peter J. Stuckey
Dept. of Computer Science, University of Melbourne
pjs@cs.mu.oz.au

Abstract

Dynamic scheduling increases the expressive power of logic programming languages, but also introduces some overhead. In this paper we present two classes of program transformations designed to reduce this additional overhead, while preserving the operational semantics of the original programs, modulo ordering of literals woken at the same time. The first class of transformations simplifies the delay conditions while the second class moves delayed literals later in the rule body. Application of the program transformations can be automated using information provided by compile-time analysis. We provide experimental results obtained from an implementation of the proposed techniques using the CIAO prototype compiler. Our results show that the techniques can lead to substantial performance improvement.

1 Introduction

Most "second-generation" logic programming languages provide a flexible scheduling in which computation generally proceeds left-to-right, but some calls are dynamically "delayed" until their arguments are sufficiently instantiated. This general form of scheduling, often referred to as *dynamic scheduling*, increases the expressive power of (constraint) logic programs. Unfortunately, it also has a significant time and space overhead.

The main objective of this paper is to develop and evaluate high-level optimization techniques for reducing this additional overhead, while preserving the semantics of the original program. We introduce two different classes of transformations. The first class simplifies the delay conditions associated with a particular literal. The second class of transformations reorders a delayed literal closer to the point where it wakes up. Both classes of transformations essentially preserve the search space and hence the operational behavior of the original program. The only caveat is that reordering may change the execution order of delayed literals that are woken at exactly the same time. Note that this order is system dependent and it is rare for programmers to rely on a particular ordering.

Using the CIAO prototype compiler we have built a tool which automatically optimizes logic programs with delay using the above transformations. Initial experiments suggest that simplification of delay conditions is widely applicable and can significantly speed up execution, while reordering is less applicable but can also lead to substantial performance improvements.

The promise of optimization of delay conditions using high-level program transformation was already illustrated in [7]. However, optimization was performed by hand and the particular transformation rules used were not detailed. Other related work has concentrated on detecting non-suspension (e.g., [6]) or is restricted to the case of some particular delayed conditions (e.g., [1]) usually found in functional languages, and the transformations applied do not guarantee that there will be no performance loss. In [4] program segments in which no suspension occurs are identified in order to perform low-level compiler optimizations. However, no suspension behaviour optimization or reordering is performed.

2 Programs with Delay

A *constraint* is essentially a conjunction of predefined predicates, such as term equations or inequalities over the reals, whose arguments are constructed using predefined functions, such as real addition. We let $\overline{\exists}_W \theta$ be constraint θ restricted to the variables W.

In dynamically scheduled languages the execution of some literal can be delayed until a particular delay condition holds. A *delay condition*, $Cond$, takes a constraint and returns $true$ or $false$ indicating if evaluation can proceed or should be delayed. Typical primitive delay conditions are `ground(X)` which holds iff `X` is constrained to a unique value, and `nonvar(X)` which holds iff `X` is constrained to be a non-variable term. Delay conditions can be combined to allow more complex delay behaviour. They can be conjoined, written $(Cond_1, Cond_2)$, or disjoined, written $(Cond_1; Cond_2)$.

We require a delay condition $Cond$ to satisfy three properties. First, it must be *downwards closed*: for any two constraints θ, θ' s.t. $\theta' \rightarrow \theta$, if $Cond$ holds for θ, then it also holds for θ'. Second, it should not take variable names into account: for any variable renaming ρ and any constraint θ, if $Cond$ holds for θ then $\rho(Cond)$ holds for $\rho(\theta)$. Third, it should only take into account variables in the condition: for any constraint θ, $Cond$ holds for θ iff $Cond$ holds for $\overline{\exists}_{vars(Cond)} \theta$ where $vars$ returns the set of variables occurring in a syntactic object.

A *delaying literal* is of the form $delay\_until(Cond, L)$, where $Cond$ is a delay condition and L is a literal. Evaluation of L will be delayed until $Cond$ holds for the current constraint store. Delay information can be *predicate-based* and *literal-based*. In the former, the delaying literal appears as a declaration before the definition of the predicate, each instance of the predicate inheriting the delay condition. In the latter, the delaying literal appears in the body of some clause only affecting the literal L. It is straightforward to use predicate-based declarations to imitate literal-based delay, and vice versa. For simplicity, we will restrict ourselves to literal-based delay.

An *atom* has the form $p(t_1, ..., t_n)$ where p is a predicate symbol and the t_i are terms. A *literal* is either an atom, a delaying literal or a primitive constraint. A *goal* is a finite, non-empty sequence of literals. A *rule* is of the form H:-B where H, the *head*, is an atom with distinct variables as arguments and B, the *body*, is a possibly empty finite sequence of literals. A *constraint logic program*, or *program*, is a finite set of rules. The *definition* of an atom A in program P, $defn_P(A)$, is the set of variable renamings of rules in P such that each renaming has A as a head and has distinct new local variables.

When formalizing applicability conditions for our transformations we will be interested in *annotated programs*, in which information about run-time behaviour is collected at *program points* in the initial query and program. Program points occur between literals and at the start and end of all bodies of all rules of the program. For instance, the rule A:-$L_1, ..., L_n$ has the program points A:-⓪$L_1$①$, ...,$ (n-1)L_nⓝ.

We are assuming that all rule heads are normalized, since this simplifies the examples and corresponds to what is done in the analyzer. This is not restrictive since programs can always be normalized. However, so as to preserve the behaviour of the original program under dynamic scheduling, the normalization process must ensure that head unifications are performed simultaneously, that is, grouped together in one primitive constraint. See for instance, the definition of `edge` in the `path` program of Example 2.1.

The operational semantics of a program is in terms of its "derivations" which are sequences of reductions between "states". A state $\langle A \mid \theta \mid D \rangle$ consists of the current sequence of active literals A, the current constraint θ, and the current sequence of delayed literals D. Our definition makes use of the parametric function $awoken(D, \theta)$, which returns a sequence of the delayed literals (stripped of their delaying condition) in D that are awoken by constraint θ. The order of the literals returned by *awoken* is system dependent[1]. A state $\langle L :: A \mid \theta \mid D \rangle$ can be *reduced* as follows:

1. If L is a primitive constraint and $\theta \wedge L$ is satisfiable, it is reduced to $\langle D' :: A \mid \theta \wedge L \mid D \setminus D' \rangle$ where $D' = awoken(D, \theta \wedge L)$.
2. If L is an atom, it is reduced to $\langle B :: A \mid \theta \mid D \rangle$ for some rule $(L$:-$B)$ in the definition of L.

[1]However, it is a brave programmer indeed who makes use of such a system dependent feature when programming.

3. If L is the delaying literal $delay\_until(C_L, L_L)$:
   - If C_L holds for θ, it is reduced to $\langle L_L :: A \mid \theta \mid D\rangle$.
   - Otherwise, it is reduced to $\langle A \mid \theta \mid D :: L\rangle$.

where :: denotes concatenation of sequences and we assume for simplicity that the underlying constraint solver is complete. A *derivation* from state S for program P is a sequence of states $S_0 \Rightarrow S_1 \Rightarrow ... \Rightarrow S_n$ where S_0 is S and there is a reduction from each S_i to S_{i+1}. A *derivation* from a query Q for program P is a derivation from the state $\langle Q \mid true \mid nil\rangle$ for P, where nil is the empty sequence.

The observational behavior of a program is given by its "answers" to queries. A finite derivation from a state S for program P is *finished* if the last state in the derivation cannot be reduced. A finished derivation from a state S is *successful* if the last state has form $\langle nil \mid \theta \mid D\rangle$. The constraint $D \wedge \bar{\exists}_{vars(S) \cup vars(D)} \theta$ is an *answer to S*.

Example 2.1 *The following program finds a path between two nodes in a directed graph.*

```
path(X,Y):- X=Y.
path(X,Y):-
      delay_until(ground(Z),edge(X,Z)),
      delay_until(ground(Y),path(Z,Y)).
edge(X,Y):- head(X,Y)=head(a,b).
edge(X,Y):- head(X,Y)=head(b,c).
```

3 Simplification of Delay Conditions

Delay conditions may be evaluated each time a variable is touched. Simplifying such conditions can then lead to significant performance improvement. Essentially the behaviour of a delay condition is only relevant during the lifetime of the delaying literal. Hence, we can replace one delay condition by another (more efficient) condition if they are equivalent for all constraint stores that occur during the lifetime of the delaying literal.

The lifetime of a delaying literal can be broken into three stages: *initial* states when it is first selected, *waking* states when it is woken, and *delaying* states when it sits in the collection of delayed literals. Consider the delaying literal $DL \equiv delay\_until(Cond, L)$. The *initial context* for DL, written $I(DL)$, is the set of constraints θ occurring in states of the form $\langle DL :: A \mid \theta \mid D\rangle$. The *delaying context* for DL, denoted $D(DL)$, is the set of constraints θ occurring in states of the form $\langle A \mid \theta \mid D\rangle$, where $DL \in D$. Finally, the *waking context* for DL, $W(DL)$, is the set of constraints θ such that either there is a derivation of the form $\cdots \Rightarrow \langle DL :: A \mid \theta' \mid D\rangle \Rightarrow \langle L :: A \mid \theta \mid D\rangle \Rightarrow \cdots$, or there is a derivation of the form $\cdots \Rightarrow \langle A' \mid \theta' \mid D'\rangle \Rightarrow \langle A \mid \theta \mid D\rangle \Rightarrow \cdots$ where $DL \in D' \setminus D$. We can restrict the constraints in the initial, delaying and waking contexts to the variables in DL since this does not affect the behaviour of the delay condition.

Example 3.1 *Consider the successful derivation for query* `?- Y = b, delay_until(ground(Y),path(X,Y))` *and the program of Example 2.1. The initial, waking and delaying contexts for each of the delaying literals are:*

DL	$I(DL)$	$W(DL)$	$D(DL)$
(a) `delay_until(ground(Y), path(X,Y))`	$\{Y = b\}$	$\{Y = b\}$	$\{\}$
(b) `delay_until(ground(Z),edge(X,Z))`	$\{true\}$	$\{Z = b\}$	$\{true\}$
(c) `delay_until(ground(Y),path(Z,Y))`	$\{Y = b\}$	$\{Y = b\}$	$\{\}$

Given the contexts for a delaying literal, simplification can be then performed by applying the following general rule:

SIMP-EQUIV: Replace a condition C, by a more efficient one C', when they are equivalent in all contexts. If $\forall\, \theta \in (I(DL) \cup W(DL) \cup D(DL))$, C holds for θ iff C' holds for θ, then we can rewrite C with C', denoted by $C \Longrightarrow C'$.

The following are special cases of this general rule which are particularly amenable to automatic application.

CONTEXT-INDEP: The following rewriting rules of Boolean algebra can always be exhaustively applied to obtain simpler delay conditions:

1.	$(Cond, true)$	$\Longrightarrow$	$Cond$	**3.**	$(Cond; true)$	$\Longrightarrow$	$true$
2.	$(true, Cond)$	$\Longrightarrow$	$Cond$	**4.**	$(true; Cond)$	$\Longrightarrow$	$true$
5.	$(Cond, false)$	$\Longrightarrow$	$false$	**7.**	$(Cond; false)$	$\Longrightarrow$	$Cond$
6.	$(false, Cond)$	$\Longrightarrow$	$false$	**8.**	$(false; Cond)$	$\Longrightarrow$	$Cond$

Their application will often be enabled by rules 9 and 10 below.

SIMP-TRUE: From downwards closure, delay conditions satisfied in all the initial contexts, are also satisfied in all delaying and waking contexts. Thus:

9. If $\forall\theta \in I(DL)$ $Cond$ holds for θ : $Cond \Longrightarrow true.$

Finally, we can replace `delay_until(`$true$`, `L`)` by L. Delaying literals (a) and (c) in Example 3.1 can be simplified in this way.

SIMP-FALSE: From downwards closure, if a delay condition is false in all waking contexts, it has been false throughout the life of a delaying literal:

10. If $\forall\theta \in W(DL)$ $Cond$ does not hold for θ : $Cond \Longrightarrow false.$

Example 3.2 *Consider the following program,* `append3`*, which appends three lists together and the query* `?- append3(X,Y,Z,[a,b,c]).`

```
append3(X,Y,Z,T):- delay_until((ground(X);ground(U)), append(X,Y,U)),
                   delay_until((ground(U);ground(T)), append(U,Z,T)).
append(X,Y,Z):-head(X,Y,Z) = head([],V,V).
append(X,Y,Z):-head(X,Y,Z) = head([A|X1],Y1,[A|Z1]),
               delay_until((ground(X1);ground(Z1)),append(X1,Y1,Z1)).
```

All calls to `append` *wake up with the first two arguments free and the last one ground. Hence, we can use rule 10 followed by rule 8 to remove the* *first primitive delay condition in all delaying literals. Also, the second delaying literal in* `append3` *as well as the delaying literal in the recursive rule of* `append`*, never delay since their third argument is ground in all initial contexts. Thus, using rule 9 the delaying condition can be removed. The resulting program (with program point annotations to be used later) is:*

```
append3(X,Y,Z,T):- ⓐ delay_until(ground(U), append(X,Y,U)),
                   ⓑ append(U,Z,T). ⓒ
append(X,Y,Z):- ⓓ head(X,Y,Z)=head([],V,V).ⓔ
append(X,Y,Z):- ⓕ head(X,Y,Z)=head([A|X1],Y1,[A|Z1]),
                ⓖ append(X1,Y1,Z1).ⓗ
```

SIMP-CHOOSE: Sometimes, when the delay condition contains disjunctions it is possible to use just part of the condition, discarding the rest:

11.	if $\forall\theta \in W(DL)$ $Cond$ holds for θ	:	$(Cond; Cond') \implies Cond$
12.	if $\forall\theta \in W(DL)$ $Cond$ holds for θ	:	$(Cond'; Cond) \implies Cond$

If both rule 11 and 12 can be applied to a disjunction $(Cond; Cond')$, efficiency considerations should be used to choose the best simplification.

SIMP-PRIM: Replace a primitive condition C_p by a more efficient one C'_p, if they are equivalent in all contexts:

13. If $\forall\ \theta \in (W(DL) \cup D(DL))$ C_p holds for θ iff C'_p holds for $\theta : C_p \Longrightarrow C'_p$.

For example, consider the `path` program. In each of the delaying and waking contexts for the delaying literal (b) the variable Z is either free or ground. Hence we could replace the primitive wakeup condition `ground(Z)` by `nonvar(Z)`, which is cheaper, obtaining the same behaviour.

Theorem 3.3 *Let* $DL \equiv$ *delay_until*$(Cond, L)$ *be a delaying literal and* $Cond'$ *be a delay condition obtained from* $Cond$ *by the application of the rewriting rules 1,...,13. Then:*

$$\forall\theta \in (I(DL) \cup D(DL) \cup W(DL))\ Cond \textit{ holds for } \theta \textit{ iff } Cond' \textit{ holds for } \theta$$

Thus, application of the rewriting rules will not change the operational behaviour of a program.

4 Reordering Delaying Literals

If a delaying literal is known to always delay at some point, it seems worthwhile to try to move it to a later point. In particular, we would like to move the delaying literal to a point where it must wake, thus removing the delay conditions. For this paper we restrict ourselves to the (seemingly simple) case of moving delaying literals in the query or rule body in which they appear.

Example 4.1 *Unfortunately, one has to be careful when moving delaying literals to later points, since this does not always preserve the search space of the program. Consider the following example program and the query* `?- delay_until(ground(Y),p(Y)), q(Y,Z).`

```
q(Y,Z) :- Y=2, long_computation(Z).
q(Y,Z) :- Y=3, Z=5.
p(Y) :- Y=3.
```

Since `Y` *is initially free,* `delay_until(ground(Y),p(Y))` *delays. Hence, we might consider moving it after the call to* `q`*. If we do, we can remove the delaying condition obtaining the reordered query* `?- q(Y,Z), p(Y).` *In the original query* `p(Y)` *is awoken before the* `long_computation` *occurs, it immediately fails and the second rule for* `q` *is tried. This succeeds waking* `p(Y)`*, which also succeeds. In the reordered query* `long_computation` *will be executed before* `p(Y)` *wakes up. In the extreme case, it may not terminate.*

Intuitively, reordering can only be performed if in the original program the execution of `q(Y,Z)` is guaranteed to have finished by the time `p(Y)` is executed.

Example 4.2 *Consider the simplified program of Example 3.2 and the query* `?- append3(X,Y,Z,[a,b,c]).`

The literal $DL \equiv$ `delay_until(ground(U), append(X,Y,U))` *can be reordered after* `append(U,Z,T)`*, even though DL does not delay until the execution of* `append(U,Z,T)` *is finished. This is because DL only wakes up at program point* ⓔ*, i.e. at the end of the execution of* `append(U,Z,T)`*. Hence, DL cannot affect the execution of* `append(U,Z,T)`*.*

We now formalize the transformation used in the above example. To reorder correctly we need to know at which points in the program a delaying literal can wake. We now define how to attach "wakeups" to program points. Consider the derivation:

$$\langle L :: A \mid \theta \mid D \rangle \Rightarrow \langle D' :: A \mid \theta \wedge L \mid D \setminus D' \rangle \Rightarrow^* \langle A \mid \theta' \mid D'' \rangle$$

where L is a primitive constraint, $\theta \wedge L$ is satisfiable and $D' = awoken(D, \theta \wedge L)$. In this derivation the delaying literals in D' have awoken at the program point immediately after the constraint L. However, the set of delaying literals that wakeup in between L and the execution of A is $D \setminus D''$. This is, in general, a superset of D', since the execution of D' may generate new constraints which may in turn wake up other delaying literals in $D \setminus D'$. For the above derivation, we then consider the set $D \setminus D''$ as waking up at the program point after L. We define the annotation of a program P for query Q as the mapping from the program points of P to the union, for all possible derivations of Q, of the sets of waking up literals at that program point.

Example 4.3 *In the program and query of Example 3.2, the wakeups attached to program point* ⓔ *are* $\{delay\_until(ground(U), append(X,Y,U))\}$. *The rest of program points have no associated wakeups.*

Example 4.4 *To see why we have to add all the delaying literals that wake up in between* L *and* A *consider the following program with query* `?- g(X,Y)`.

```
g(X,Y) :-    ⓐ delay_until(ground(X), p(X,Y,Z)),ⓑ
             delay_until(ground(Y), q(Y)), ⓒ r(X).ⓓ
r(X) :-      ⓔ X = 1.ⓕ
p(X,Y,Z) :- ⓖ Y = 1,ⓗ long_computation(Z).ⓘ
q(Y) :-      ⓙ Y = 2.ⓚ
```

The annotated program points with associated wakeups are:

ⓕ $\{delay\_until(ground(X), p(X,Y,Z)), delay\_until(ground(Y), q(Y))\}$
ⓗ $\{delay\_until(ground(Y), q(Y))\}$.

If we had annotated ⓕ *with the set of literals immediately awoken by* `X=1`, *that is* D', *we would only obtain the first delaying literal,* `delay_until(ground(X), p(X,Y,Z))`. *Thus, it would be hard to see that the second delaying literal wakes up within* `r(X)`.

Information about the program points at which a delaying literal can be awoken leads to a simple methodology for reordering a delaying literal. Before we detail the transformation we need to define at which program points a delayed literal can be awoken for reordering to be allowed. We first define the set of program points for a particular goal at which delayed literals may be awoken during the evaluation of the goal. The set of *instantiating program points* for a goal ⓐGⓑ, denoted $IPP(G)$, are:

$$IPP(G) = \begin{cases} \{ⓑ\} & \text{if } G \text{ is a constraint} \\ IPP(L) & \text{if } G = delay\_until(Cond, L) \\ \bigcup_{G:\texttt{-}B_i \in defn_p(G)} IPP(B_i) & \text{if } G \text{ is an atom} \\ IPP(L') \bigcup IPP(G') & \text{if } G = L', G' \end{cases}$$

Now we define the subset $NIPP(G)$ of instantiating program points which are *non-final* for a goal ⓐGⓑ. A delayed literal will not be allowed to move across a goal if it wakes up at a non-final point:

$$NIPP(G) = \begin{cases} \emptyset & \text{if } G \text{ is a constraint} \\ NIPP(L) & \text{if } G = delay\_until(Cond, L) \\ \bigcup_{G:\texttt{-}B_i \in defn_p(G)} NIPP(B_i) & \text{if } G \text{ is an atom} \\ IPP(L') \bigcup NIPP(G') & \text{if } G = L', G' \end{cases}$$

The set of *final instantiating program points* for goal ⓐGⓑ, denoted $FIPP(G)$, is simply $IPP(G) - NIPP(G)$. For example the instantiating program points for ⓑ$append(U, Z, T)$.ⓒ in Example 4.2 are {ⓔ, ⓖ }. The final instantiating program points for ⓑ$append(U, Z, T)$.ⓒ are { ⓔ }.

We can now define two transformation rules which provide sufficient conditions for reordering a delaying literal across the body in which it appears, while preserving the semantics of the program. Consider a rule of the form $H : -L_1, \ldots, L_i, DL, L_{i+2}, \ldots, L_j, L_{j+1}, \ldots L_n$ where DL is a delaying literal.

DOESNT-WAKE: We can reorder DL until immediately before L_{j+1} if DL is definitely delayed before L_{i+2} and it does not wake at any instantiating program point in the conjunction of literals $L_{i+2}, \ldots, L_j$.

FINAL-WAKE: We can reorder DL until immediately after L_j if DL is definitely delayed before L_{i+2} and it does not wake at any non-final (instantiating) program point in the conjunction of literals $L_{i+2}, \ldots, L_j$. In addition if DL wakes up at a final program point together with another delaying literal DL', then DL wakes only at final program points of the literal DL'.

Example 4.5 *Consider the program and query of Example 4.4. The literal* `delay_until(ground(Y), q(Y))` *only wakes up at* ⓕ *which is a final program point for* `r(X)`*. However, reordering such literals would result in* `long_computation(Z)` *being performed. This is why an additional condition is introduced in the* FINAL-WAKE *rule. As* `delay_until(ground(X), p(X,Y,Z))` *also wakes at program point* ⓕ *and* `delay_until(ground(Y), q(Y))` *wakes up at* ⓗ *which is a non-final program point within* `p(X,Y,Z)`*,* FINAL-WAKE *is not applicable. We can however move* `delay_until(ground(X), p(X,Y,Z))` *until after the conjunction* `delay_until(ground(Y), q(Y)), r(X)` *using the* FINAL-WAKE *rule, since it only wakes at* ⓕ*, a final program point of this conjunction. At this point it is guaranteed to wake, the delay condition can be removed, and the optimized rule is* `g(X,Y):- delay_until(ground(Y), q(Y)), r(X), p(X,Y,Z).`

If we now annotate this program for the query `?- g(X,Y)`*, the new annotations would show that* `delay_until(ground(Y), q(Y))` *could be moved after* `r(X)`*, using the* DOESNT-WAKE *rule.*

The reason why DOESNT-WAKE is correct is that since DL is not awoken during evaluation of $L_{i+2}, ..., L_j$ it cannot affect the evaluation, and so can be added later. The reason why FINAL-WAKE is correct is that since DL is the last literal evaluated before returning from L_j, we can equivalently evaluate it as the first literal after returning from L_j.

Unfortunately, there is a subtle problem with this reasoning. The problem is that both reordering rules may change the order in which literals are delayed, and so may affect the system dependent order in which literal are returned by *awoken*. This is only a problem in the case when more than one literal is awoken at the same time.

Example 4.6 *Consider the following program and query:* `?- g(T)`

```
g(T):- delay_until(ground(T),p(T)),
       delay_until(ground(T),q(T)), T = 1.  ⓐ
p(T):- T = 2.
q(T):- long_computation(Z).
```

At ⓐ both delaying literals wake. If awoken returns `p(T)::q(T)`*, the query quickly fails. There is no annotation in the body of* `q(T)` *which includes the delaying literal for* `p(T)`*, hence* FINAL-WAKE *is applicable for the literal. Hence,* `g(T) :- delay_until(ground(T), q(T)), T =1, p(T).` *is a correct reordering. But for this program the* `long_computation` *is executed.*

However, note that behaviour of the transformed program is equivalent to that of the original program if awoken had returned `q(T)::p(T)` *instead.*

Therefore we have the somewhat weaker correctness result for the reordering rules, that the transformed program behaves equivalently to the original program for some choice of the *awoken* function. However, as noted earlier it is rare for programmers to rely on the system dependent ordering of *awoken* to prune the program search space.

5 Automating the Optimization

We have built a prototype automatic transformation tool which works as follows. First, the original program is analyzed, and the program annotated with the inferred information is given to the optimizer. Using this information, the optimizer first simplifies delay conditions as much as possible and then reorders those literals which are sure to delay. To reduce problems of the kind presented in Example 4.6, whenever more than one literal is reordered to the same program point and no information about waking order is available, the optimizer keeps the relative order in which the reordered literals appeared in the original program. In addition, reordering may enable further optimizations, as the initial contexts in the new positions will in general be more instantiated than in the original ones. Hence, another analysis–optimization iteration could be performed. In some cases the current implementation can perform further optimizations without re-analysis. Other optimizations traditionally used with fixed-scheduling constraint logic programs can also be performed after transformation. Currently we perform dead code elimination and simplification of built-ins.

Different analysis frameworks have been recently developed for logic programs with dynamic scheduling (e.g., [7, 4, 3]). In our prototype we use the approach of [3]. However, simplification can be performed with any analysis framework which, for a given analysis domain, approximates the initial, delaying and waking contexts for each delaying literal. For reordering, the analyzer needs to provide a description of the set of waking up literals at each program point. For the traditional optimizations, the analyzer needs to also provide a description of the constraints at each program point.

The experimental evaluation uses the information provided by three different abstract domains. The Def domain[2] [2] approximates *groundness* information. Thus, it can be used to infer the satisfiability of *ground* and *nonvar* tests. The ShFr domain [10] approximates not only groundness but also *sharing* and *freeness* information. Freeness information allows us to prove the unsatisfiability of *ground* and *nonvar* tests. The Aeq domain[3] complements ShFr with more complex modes like non-freeness, non-groundness and linearity. Non-freeness and non-groundness allow more accurate information about the behaviour of *nonvar* and *ground* tests. Linearity improves sharing and therefore the propagation of the other properties.

6 Experimental Results

Four different sets of benchmarks have been used in our experiments. The first set corresponds to those used in [7, 3]. They are essentially new, reversible versions of some standard symbolic programs. The original programs used static scheduling and could only be run in one mode. In the new versions dynamic scheduling has been added to allow them to run both forwards and backwards. This first set includes `append3` (concatenates 3 lists), `nrev` (reverses a list in a naive way), `permute` (computes all permutations of a list), and `qsort` (the quick-sort algorithm). The second set corresponds to standard mathematical benchmarks in which dynamic scheduling has been added to arithmetic constraints so as to allow them to run both forwards and backwards. This set includes `fac` (factorial), `fib` (Fibonacci), and `mortgage`. The programs in the third set are programs with dynamic scheduling resulting from the automatic translation of concurrent logic programs by the Qd-Janus system [5]. Dynamic scheduling is used to emulate the concurrency present in the original programs. This set includes `nand` (a nand-gate circuit designer, written by E. Tick) and `transp` (matrix transposer, written by V. Saraswat). The Qd-Janus compiler already performs analysis and optimization of its input programs and aims to produce code with little redundant concurrency. The Prolog code it produces is competitive in performance with compilers specifically designed for concurrent logic programs. The last set are NU-Prolog programs written by L. Naish which exploit rather complex dynamic scheduling for different purposes, and which have been translated into SICStus. This set includes `nqueen` (coroutining n-queens), `slowsort` (a generate and test algorithm), `interpl` (simple interpreter for coroutining programs), and `termcompare` (term comparison).

The following table provides information regarding the complexity

[2] This domain is a variant of the *Prop* domain [8].

[3] This domain is a modification of that of Mulkers et al [9]. We have added more complex modes but do not make full use of the equation modelling component.

of the benchmarks used in our experiments. Cl is the number of clauses analyzed, Lit is the number of literals, and DL is the number of delaying literals. Since programs have been normalized the (usually high) number of term equations is not counted in Lit. DL includes all calls to predicates affected by a delay declaration. For the first two sets of benchmarks we will consider two different versions of each program: in the first one ground conditions are used in the delaying literals (`_gr` suffix), while in the second one nonvar conditions are used (`_nv` suffix). Note, however, that in `nrev` and `qsort` nonvar conditions do not always guarantee termination. Thus a mix of ground and nonvar conditions is used in the "`_nv`" version of these benchmarks. Different rows associated to the same benchmarks indicate different queries. For the first two sets of benchmarks they perform forward and backward execution.

Benchmark	Cl	Lit	DL
append3	3	3	3
nrev	4	3	3
permute	4	3	3
qsort	7	9	9
fac	8	27	3
fib	6	17	4
mortgage	8	29	5
nand	90	157	13
transp	112	180	20
nqueen	11	15	11
slowsort	9	8	8
interpl	11	10	3
termcompare	27	37	26

The programs have been implemented using `block` (SICStus predicate-based delay) declarations whenever possible, i.e., when only `nonvar` tests were involved. This is because they are the most efficient delay declarations in SICStus. Otherwise, `when/2` (SICStus literal-based delay) declarations were used. The only exceptions are the programs in the third class where the compiler produces literal-based `freeze` declarations.

Our first set of experiments evaluates the cost of the automatic transformation using our prototype compiler described in the previous section. The following table shows the analysis times in seconds for each of the abstract domains described in the previous section as well as the time in milliseconds required to optimize the programs using the information inferred. The times are for code run under SICStus Prolog version 3.0 on a 55MHz SPARCstation 10 with 64 MBytes of memory. An ∞ indicates that the analyzer ran out of memory because too many calling patterns were produced in the analysis.

Analysis times are generally acceptable, except for three programs: `qsort_nv`, `transp`, and `termcompare`. Their times are slow because of their complex dynamic behaviour. However, it should be remembered that the analysis of logic programs with dynamic scheduling is still in its infancy and that we are using a prototype analyzer. As this technology improves, analysis time should markedly decrease. Transformation times are very low – only when the amount of analysis information is enormous does the time reach more than one second.

Our second experiment evaluates the effectiveness of the optimizations. The following table shows the execution time in milliseconds for the original programs, and the speed-up obtained by the automatically transformed programs using simplification and then both simplification and reordering.

Benchmark	Analysis (sec)			Transformation (msec)		
	Def	ShFr	Aeq	Def	ShFr	Aeq
append3_gr	0.0	0.0	0.0	7	7	10
	0.1	0.1	0.2	13	20	10
append3_nv	0.0	0.0	0.0	7	10	7
	0.1	0.2	0.4	20	20	20
nrev_gr	0.0	0.0	0.0	10	10	10
	0.1	0.1	0.2	17	17	20
nrev_nv	0.0	0.0	0.0	10	10	10
	0.4	0.1	0.2	30	10	17
permute_gr	0.0	0.0	0.0	10	10	7
	0.3	0.5	0.2	17	23	13
permute_nv	0.0	0.0	0.0	7	10	10
	0.4	2.8	5.8	20	57	50
qsort_gr	0.0	0.1	0.1	13	13	10
	3.2	2.8	12.3	47	63	93
qsort_nv	0.0	0.1	0.1	10	20	13
	23.9	1517.3	∞	243	2730	∞
fac_gr	0.0	0.0	0.0	20	30	23
	0.3	0.2	0.4	40	37	57
fac_nv	0.0	0.0	0.0	23	23	27
	0.3	0.2	0.4	37	40	50
fib_gr	0.0	0.0	0.1	23	23	27
	0.8	0.6	0.9	43	40	70
fib_nv	0.0	0.0	0.1	13	20	30
	0.6	0.6	0.9	43	40	67
mortgage_gr	0.0	0.0	0.1	27	33	40
	0.7	0.3	0.5	77	50	67
mortgage_nv	0.0	0.0	0.1	33	30	40
	0.5	0.3	0.5	57	50	70
nand	0.5	0.7	1.6	297	303	373
transp	168.5	1621.5	∞	1087	1620	∞
nqueen	0.0	0.1	0.1	30	40	40
	2.6	3.7	7.5	77	113	140
slowsort	0.0	0.0	0.1	23	33	23
	0.3	0.5	1.2	33	43	43
interpl	0.9	3.0	2.0	37	50	37
termcompare	0.2	0.3	0.3	70	90	83
	7.8	19.1	158.3	233	380	1393

We do this for each abstract domain. Since the information provided by Def never allows reordering, its column has been eliminated from Simp. + Reord. A blank entry in the Simplification column indicates that no delay condition was optimized, and a blank entry in the Simp. + Reord column indicates no reordering was performed and hence the speedup is the same as for simplification alone. A † indicates that no delaying literals remain in the transformed program.

Our results demonstrate that both simplification and reordering can lead to an order of magnitude performance improvement, and that they give reasonable speedups in most benchmarks. The benchmarks `nand`, `transp`, `interpl` and `termcompare` which did not exhibit any measurable speedup

Benchmark	Orig	Simplification			Simp. + Reord	
		Def	ShFr	Aeq	ShFr	Aeq
append3_gr	339430	439.68 †	439.68 †	439.68 †		
	7438	1.49	1.49	1.49		2.10 †
append3_nv	816	1.06 †	1.06 †	1.06 †		
	3682	1.03	1.03	1.03		
nrev_gr	342220	1368.88 †	1368.88 †	1368.88 †		
	5864	5.55	5.55	5.55		68.19 †
nrev_nv	3682	1.03	1.03	1.03		
	1086		1.03	1.03		12.63 †
permute_gr	28982	11.13 †	11.13 †	11.13 †		
	1452	2.66	2.66	2.66		5.46 †
permute_nv	2574	1.00 †	1.00 †	1.00 †		
	836	1.02	1.02	1.02		
qsort_gr	5908	173.76 †	173.76 †	173.76 †		
	2138	2.31	2.31	2.31		
qsort_nv	818	27.27 †	27.27 †	27.27 †		
	1320		1.00	—		—
fac_gr	3268	1.54 †	1.54 †	1.54 †		
	15322	2.62	2.62	2.62		
fac_nv	2100	1.00 †	1.00 †	1.00 †		
	1830	1.07	1.07	1.07		
fib_gr	35784	61.06 †	61.06 †	61.06 †		
	37848	1.65	1.65	1.65		
fib_nv	668	1.11 †	1.11 †	1.11 †		
	722	1.06	1.06	1.06		
mortgage_gr	4202	6.55 †	6.59 †	6.59 †		
	5138	1.66	2.52	2.52	53.52 †	53.52 †
mortgage_nv	646	1.03 †	1.03 †	1.03 †		
	330	1.25	1.57	1.57	3.44 †	3.44 †
nand	464	1.00	1.00	1.00		
transp	5609	1.00	1.00	—		—
nqueen	27218	8.14	8.14	8.14		
	4684	1.39	1.39	1.39		
slowsort	1466	8.52	8.52	8.52		
	3388	1.00	1.00	1.00		
interpl	3160	1.00	1.00	1.00		
termcompare	4418	1.14	1.14	1.14		1.33
	4456	1.00	1.00	1.00		1.08

belong to the last two sets of benchmarks. It is perhaps not surprising that our optimizer found programs in these classes difficult to improve since they were either produced by a rather clever transformer which tries to avoid introducing delay where it is not needed or hand-crafted by an expert in dynamic scheduling. Unsurprisingly, the more sophisticated the analysis domain, the better the speed up. In particular the extra precision of Aeq is required to gain the most benefit from reordering.

Our results are promising. They show that the transformation techniques introduced in this paper can be automated and lead to significant performance improvement. This is important because dynamic scheduling looks set to become increasingly prevalent in (constraint) logic programming languages because of its importance in implementing constraint solvers and controlling search as well as for implementing concurrency. We noted that the effect of the transformation greatly depends on the implementation of the delay declarations, and therefore on the target language. In particular, since groundness is an expensive test its simplification gives great benefits. Lesser, although still significant benefits can be obtained for other delay conditions. However this work is only a first step. Many other techniques for the automatic transformation of programs with dynamic scheduling remain to be investigated.

Acknowledgements

We would like to thank Manuel Hermenegildo for his significant contribution to the ideas presented in this paper, Saumya Debray for his Qd-Janus benchmarks, Lee Naish for his benchmarks and helpful discussions, and Anne Mulkers for providing the implementation of the Aeq analysis domain.

References

[1] J. Boye. Avoiding dynamic delays in functional logic programs. In *Programming Language Implementation and Logic Programming*, number 714 in LNCS, pages 12–27, Estonia, August 1993. Springer-Verlag.

[2] M. García de la Banda and M. Hermenegildo. A Practical Approach to the Global Analysis of Constraint Logic Programs. In *1993 International Logic Programming Symposium*. MIT Press, Cambridge, MA, October 1993.

[3] M. García de la Banda, K. Marriott, and P. Stuckey. Efficient Analysis of Constraint Logic Programs with Dynamic Scheduling. In *1995 International Logic Programming Symposium*, Portland, Oregon, December 1995. MIT Press.

[4] S. Debray, D. Gudeman, and P. Bigot. Detection and optimization of suspension-free logic programs. *Journal of Logic Programming*, 29(1–3):171–195, October–December 1996.

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

[6] M. Hanus. Analysis of Nonlinear Constraints in CLP(R). In *Tenth International Conference on Logic Programming*, pages 83–99. MIT Press, June 1993.

[7] 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.

[8] K. Marriott, H. Søndergaard, and N.D. Jones. Denotational Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, 1994.

[9] A. Mulkers, W. Simoens, G. Janssens, and M. Bruynooghe. On the Practicality of Abstract Equation Systems. In *International Conference on Logic Programming*. MIT Press, June 1995.

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