Better Late Than Never

Micha Meier

ECRC-ECRC-95-08

technical report ECRC-ECRC-95-08

Better Late Than Never

Micha Meier

European Computer-Industry Research Centre GmbH (Forschungszentrum)
Arabellastrasse 17
D-81925 Munich
Germany
Tel. +49 89 9 26 99-0
Fax. +49 89 9 26 99-170
Tlx. 52 69 10

For more information please contact : michaecrc.de

Abstract

Some Prolog systems are able to delay the execution of Prolog goals and resume it later. We were involved in the design, implementation and evaluation of several such systems and we summarise here our experiences with them. First we describe a general structure of such 'coroutining' systems and then we concentrate on the particular features of and choices made in three ECRC's logic programming systems: ECRC-Prolog, SEPIA and ECL^{i}PS^e .

1 Introduction

There are quite a few Prolog systems that were built in ECRC since its creation in 1984. Most of these systems had the ability to extend the default Prolog control, at least by suspending some goals and waking them later. With the appearance of CLP-like languages this type of control became very important, because it allows an easy implementation of constraint solvers in Prolog. We were involved in the design and implementation of some of these systems and we would like to share some of our good and bad experiences with these systems. We first describe the general structure of any coroutining system and mention various ways to design its particular features. In the following sections we concentrate on the particular systems: ECRC-Prolog, which was built in 1984–1986, SEPIA, its successor from 1987–1992, and ECL^{i}PS^e , created in 1992, which is intended to unify all LP systems currently being used in ECRC.

2 General Structure of a Coroutining System

Although there are relatively many systems able to delay and resume Prolog goals, most of the design and implementation details are part of Prolog folklore and they have been described only in few papers. Woken goals sometimes behave like coroutines and this is why delaying and waking goals is often referred to as *coroutining*, although real coroutines require a much more complex support. The first Prolog system with coroutining was Prolog-II with its **geler/2** primitive [3], which in other systems has the name **freeze/2**. **freeze(Var, Goal)** calls **Goal** if **Var** is instantiated, otherwise **Goal** is suspended and woken only after **Var** is instantiated. Similar mechanisms were available in other systems: MU-Prolog [9], IC-Prolog [4], ESP [2], SICStus Prolog [1], NU-Prolog [10] or in committed-choice languages [12, 13, 5, 14].

During our designs we have identified the main features which determine particular approaches and answer the important questions. Below we list the main questions that have to be answered by every design.

2.1 When to Delay a Call

Which condition has to be satisfied or violated for a call to be suspended? The condition can be expressed as a property of the caller (as e.g. in **freeze/2**), or as a property of the whole procedure (various declarations - **wait, block, when**, ...). The condition can specify when the call has to be suspended (e.g. *delay clauses* in SEPIA) or the other way round, when it is allowed to continue (e.g. *when declarations* of NU-Prolog); it can even pretend to do the former and do the latter, like the *wait declarations* of MU-Prolog.

Next question is, what flexibility is allowed for the condition itself, is it a simple variable test, a boolean condition, or even a Prolog goal? For many applications, e.g. to catch infinite loops, a simple variable test is sufficient, but for more elaborate control more flexibility is required.

2.2 How to Represent Delayed Calls and Variables

When a call is delayed, the system creates a *suspension* which contains enough information to wake the goal later. The suspension must contain at least the procedure identification and the arguments, for realistic applications it must contain a little more than that.

The suspensions have to be stored in a memory area that is more permanent than the environment (local) stack. This is necessary because the environment of a clause can be popped even if one of its subgoals was delayed. Woken suspensions which are no longer needed should be popped or garbage-collected.

Variables that may cause the goal to wake (we call them *suspending variables*) must be marked somehow and it must be possible to reach the suspension from these variables. This is usually achieved by binding these variables to a structure that contains a suspension pointer. As one variable may have several suspensions attached to it, and also one suspension may have several variables which can wake it, an appropriate data structure has to be used.

The suspending variables must be carefully chosen to avoid waking the goal when it is known that the condition for continuation is still not satisfied. For instance, if a goal delays until a set of variables becomes ground, it suffices to mark only one variable from the set.

Some built-in predicates are usually written in the implementation language. For them the suspending mechanism may be different, in particular the delaying condition is hardcoded.

2.3 When to Wake a Delayed Call

A suspended goal is usually woken when one of its suspending variables is instantiated. Some systems also allow waking when a suspending variable is bound to another one or updated in a different way, e.g. by adding a new suspended goal to it. There may be other events that might trigger the waking of a suspended goal, however in our systems we restricted ourselves to events caused by such variable updates.

Further question is, whether a woken goal is triggered immediately after the binding of the suspending variable (and thus possibly interrupting the unification), or in every clause's neck, or only at certain specified places.

2.4 How to Schedule Woken Goals

Programs that heavily use coroutining create a chaos of goals woken at different levels intermixed with normal goals and backtracks. Each coroutining system must decide how the woken goals will be scheduled: oldest suspended goals first or last, built-in suspended goals first, etc. It must be also specified if the execution of a woken goal can be interrupted by another woken goal or if new woken goals are put into a queue which is executed only after the first woken goal is completely solved.

In a WAM-based machine it is also necessary to include the woken goals somehow in the execution process although there is no **call** instruction that calls it; the continuation handling must be also specified.

2.5 How to Integrate Woken Goals in the Normal Execution

A woken goal behaves as if it were textually written in the body of the clause whose head unification has touched the suspending variable. This means that we have to interrupt somehow the execution of compiled goals, save enough data to be able to resume it later, start the execution of the woken goal(s), and finally resume the interrupted execution. This process has some important impacts on the WAM, on register optimisations, shallow backtracking, etc.

2.6 How to Re-delay a Woken Call

When the delaying condition is not a trivial one, it can happen that the goal cannot be allowed to continue even after one or more of its suspending variables have been instantiated. The goal is then said to **re-delay**. The re-delaying can be handled as a usual delay, which may cause some actions to be unnecessarily repeated (e.g. creating a suspension), or the system may use the fact that the goal was already delayed and re-use some of the already existing data. This, however, may again make the scheduling less transparent. An important detail is that when a call re-delays, new suspending variables might have to be taken, i.e. it is not generally possible to rely on the fact that the suspension is already connected with the right suspending variables. For example, if $p(X)$ delayed until X becomes ground, X is the suspending variable. Later, when X is bound to e.g. $f(Y)$, the goal is woken but it re-delays and a completely new suspending variable, namely Y has to be taken.

2.7 Memory Management and the Value Trail

Although it is possible to introduce coroutining into a Prolog system with no or little special-purpose data structures and mechanisms (an example is SICStus Prolog [1]), we have always based our design on a special architecture. It included special structures for the suspensions, for the variables themselves, special-purpose WAM-instructions and the value trail, i.e. a trail stack which records both the trailed address and the value previously stored in it.

The issue of garbage collection of the unused data has also to be addressed. For instance it is possible that some coroutining structures become garbage even if they are still accessible from some places.

All ECRC Prolog architectures used tagged words with at least 8 bits in the tag. This means that we never had problems defining new data types and indeed we did so in an abundant way.

2.8 What Can Be Done with the Delayed Goals

During the Prolog execution the user might want to ask some meta-queries about the suspended goals. He may ask what are the goals suspended by a particular variable, what are the goals suspended since the program start or another point in the execution. He may also want to debug the program and ask these questions to the Prolog debugger and expect further debugger support concerning the suspended goals. The least that must be provided is to collect floundering goals [1] at the end of the query execution and print them together with the usual answer substitution.

[1] i.e. goals that are still suspended

3 ECRC-Prolog

ECRC-Prolog was the first real Prolog system built at ECRC in 1984–1986. It was in fact a WAM-based compiler for an enhanced MU-Prolog [9], which generated a C program and this program was then normally compiled with the C compiler to yield a stand-alone binary program. This was a somewhat strange combination especially because the Prolog compiler was rather slow and the generated C program was usually very big. This resulted in such long compilation times of both the Prolog and C compilers, that the system (which was not even incremental) had only little chance to survive (and it did not).

The choice of MU-Prolog with its *wait declarations* was deliberate, at that time it seemed to be the Prolog system with the most advanced and flexible control features. We did not want to restrict ourselves to a pure **freeze/2** implementation; it seemed more logical to view the control as a property of the whole predicate and not only of a call to it. We are by the way still convinced that the declaration-based suspensions are more appropriate than annotations in the caller, because most [1] of the time the suspension is due to the arguments of the call and not to the place it is called from.

3.1 Delaying

Wait declarations were quite unique among similar concepts; first because few if any managed to use them correctly, we were all the time mixing and's and or's and 0's and 1's. Second, because the question whether a call delays or not could be answered only after the head unification. On one hand this gives the wait declarations more power than can be found in the static annotations or declarations, on the other hand the implementation was awkward.

A wait declaration specifies which of the goal arguments may be 'constructed' (this means something like 'instantiated', but not quite) and which not. There could be several wait declarations for one predicate and if a head unification violated all of them, the call was delayed. For example, the **append/3** predicate could be declared as

```
:- wait append(1, 1, 0), append(0, 1, 1).
```

which means that a call to **append/3** delays if both the first and the third

[1] Later (see 4.1) we have in fact found out that sometimes it actually *is* the caller who specifies if the call should delay or not.

argument need to be constructed. The fact that first the unification is finished and only then the suspension is tested has several implications:

- The delaying depends on the clause head. Clauses without nonvariables in the head could actually not delay and it was indeed sometimes necessary to use dummy clauses at the beginning of the procedure which would instantiate the necessary arguments.
- It is possible that some clauses of the same predicate delay while others do not. Apart from being an interesting idea, this has complicated the implementation.
- If the predicate has several matching clauses, a choice point must be pushed before the unification. However, if the call delays, this choice point has to be removed, otherwise we might backtrack through all clauses without actually executing them; this might be logically correct, but it is rather inefficient.
- If a call delays, its unification with the clause head must be undone (otherwise the call could not be woken, because there would be no variables whose instantiation would trigger the waking). It is in fact an interesting idea whether the instantiations of variables that do not influence waking could be kept or not.

If the delay condition is tested statically, at the beginning of the predicate code, the execution must always start at this point, and it is difficult or impossible to make optimisations that skip some code parts. The fact that the delaying in ECRC-Prolog was tested only after the head unification has the advantage that the predicate does not have to be entered at one precise point and this was used by the compiler for indexing: if it was known that after waking a certain argument must be instantiated, the resuming address was in the code that indexed (also) on this argument.

The unification of a call with a predicate with wait declarations was executed by special unification instructions which have created a bit mask of arguments that were constructed. When the unification failed, nothing happened. If it succeeded, there was a **neck** WAM instruction in the clause neck which has compared this mask with masks derived from the wait declarations. If there was a mask that allowed to continue, the execution continued normally (that is why the name *wait* is not quite appropriate). Otherwise, the system had to undo the unification, collect the suspending variables, make the suspension and link them together. Undoing the unification was a problem for calls without a choice point, because the binding of some variables might not have been trailed. One can of course change the rule and trail everything, but this seemed to be too high a price because the majority of goals do not delay. We have therefore introduced the *auxiliary trail* that recorded changes in the deterministic state. If the call delayed, it was used to undo the unification and

to pick up the variables that were bound in the unification, which then became the suspending variables. If the call did not delay, the auxiliary trail was simply cleared. The auxiliary trail was used only in predicates with delay declarations and thus only in the special unification instructions that also had to construct the bit mask.

3.2 Data Structures

The suspension was represented by a *delayed environment*, which contained the following items:

- the call arguments and arity,
- the resuming entry point address,
- flag specifying if a choice point has to be created when the goal is woken,
- the woken goal continuation.

The suspending variables were bound to a word with a special `suspend` tag, whose value was a list of suspensions.

3.3 Waking

Whenever a suspending variable was instantiated or bound to another one, the **neck** instruction took care of waking the suspensions in its associated list. As the suspending variables were always trailed on the value trail, it sufficed to check whether the unification has modified the value trail. If a suspending variable was bound to another one the two lists were merged together and nothing was woken, otherwise the goals on the list were woken.

The clause whose head unification had woken some delayed goals was then itself suspended, a *resuming environment* was created which contained the WAM argument registers with meaningful values, and the **CP** and **PP** registers.

The handling of continuation was quite interesting: the WAM **CP** register pointed either normally to the code of the next goal to execute, but it could also point to a delayed environment or to a resuming environment. When some goals were woken, their suspensions were linked together using their continuation field, the last one pointed to the resuming environment of the clause that had woken them. The **proceed** instruction tested where **CP** pointed to and performed the appropriate actions. The **neck** instructions did not actually call the woken goals, it only linked them together and invoked the first one. In case one of the woken goals failed, we had thus done unnecessary

work with linking the following ones. There were suggestions to call woken goals directly in each **neck** instructions and then to continue with the clause body. They were rejected because the goals had to be scanned anyway because of sorting (see below) and then it was simpler to use the direct continuations.

3.4 Scheduling

Since one goal can be suspended by more than one variable, it is necessary to mark the suspensions that were already woken and executed to prevent waking them again when another suspending variable is instantiated. In ECRC-Prolog this was done using the continuation field in the suspension; if it was set it meant that the goal was already woken and thus it was ignored for all the subsequent instantiations of other suspending variables.

Since we wanted to wake the goals strictly according to their age, the lists of all instantiated suspending variables were merged together and sorted so that oldest goals were guaranteed to be woken first. The woken goals were then inserted at the beginning of the current continuation chain. This means that the execution of a woken goal was itself interrupted when a new suspending variable was bound, and new woken goals were triggered immediately. Since the lists of woken goals were explicitly accessible, it would have been possible to insert the newly woken goals according to their age in this list so that waking would completely correspond to the age of the delayed goals, but this was rejected as exaggeration.

3.5 Re-delaying

Re-delaying a call was simpler than delaying - since the suspension already existed, it was simply added to the delay lists of all suspending variables in the call and its continuation was reset back to zero to mark that it was not woken. One could not be sure that the suspending variables for the re-delay are the same as those for the first delay. The consequence was that the same suspension might have occurred several times in the same list, but we did not find any efficient and general method to identify which variables already have this suspension in their list and which do not.

3.6 Memory Management

ECRC Prolog had three trails: the usual trail, the auxiliary trail needed to undo the unification when a deterministic call delayed, and the value trail. The main purpose of the value trail was for inserting new goals into the delay lists. This

is in fact not necessary – the delay lists may be ended up by a variable and new goals can be simply appended to the end and this link is trailed as usual. This approach, however, has the disadvantage that the time needed to build the whole list becomes quadratic in the list length. Although it is possible to reduce this overhead by variable shunting [11], the quadratic complexity remains. We have therefore inserted new suspensions at the beginning of the delay lists and the link from the variable to the list was value-trailed.

One fact deserves mentioning: when a plain variable delays a goal and becomes a suspending variable, it seemed that it could be trailed with the normal trail because it had no value before. Later, when new goals delay on it, the value trail would be used. However, the fact that we had two *separate* trail stacks was the cause of the most obscure bug I've seen so far and searched for almost a week: there were cases when the variable was not untrailed properly, no matter if we first untrailed the normal or the value trail.

3.7 Support for the Delayed Goals

There was not much that could be done with suspended goals apart from waking them. Floundering goals were discovered by scanning the value trail at the end of the query. Value-trailed items that were lists of suspensions were searched for a suspension which was not woken and if so, the system printed the message that some goals are still delayed, but it did not say which ones. There was no debugger support nor any built-in predicates to access the delayed goals.

3.8 Conclusion

Coroutining in ECRC-Prolog was actually very efficient, even if there was much extra work to do, e.g. sorting the delay lists. It was an order of magnitude faster than MU-Prolog and, especially with disjunctive delays, than SICStus 0.3. The idea to unify first and then check if the call delays is quite interesting and close to concurrent Prologs and it has several advantages, however the form of the wait declaration was not quite appropriate.

One of the objectives was to wake the delayed goals exactly in the order they were delayed. However, it turned out that after several delays and wakings no-one really knows what is happening. No matter what waking order was taken, it was almost impossible to tell which woken goal comes from where and who is who's parent. Preserving the order was thus not really helpful.

Several users have had problems with cuts in their programs. We have tried to find a scheme that would be safe with respect to cuts and still not too expensive, but didn't find any. Since then we were convinced that one should

not mix cuts and coroutining, and if possible get rid of the cut completely (keep only **once/1** and **->/2**). This does not solve the problem, but it makes a safer ground to build on.

4 SEPIA

In the SEPIA [7] design we have tried to learn from the problems and to do everything better (the "second system syndrome"). Since we felt that the main problems of the previous implementation were slow compilation, non-incrementality and bad performance, the main changes were made there. SEPIA is based on a WAM-emulator, its compiler is written in C to make the compilation as fast as possible, and there is a special compilation and execution mode for the execution with delayed goals. The aim is that in the non-coroutining mode the machine is as fast as possible, without any overhead caused by the special features of the system. There is, however, one major difference between SEPIA and other Prolog systems, namely the word size. In SEPIA the tag is 32 bits long and thus the size of every Prolog item is 2 words. There were several reasons for this, the main one was that we wanted both more space for the tag than 2 or 3 bits, and at the same time 32 bits for the value part so that pointers could be stored directly.

In SEPIA there are also two types of predicates: simple and regular, the former are an extension of in-line expanded predicates, any deterministic predicate written in C is simple, all others are regular. This distinction helps to identify sequences of predicates whose execution does not change any important WAM register.

4.1 Delaying

The delaying in SEPIA is controlled by *delay clauses*. A delay clause looks like

```
delay p(X, Y) if var(X), var(Y).
```

and it specifies explicitly under which condition the call to this procedure should delay. We have thus moved from dynamic delaying conditions in ECRC-Prolog to static ones. The use of delay clauses follows quite naturally from the requirements that the delaying must be flexible enough to allow specification of complex conditions, which are necessary to implement various constraints propagation schemes. The use of a delay condition is more natural than a 'continue' condition, because a predicate without any condition should never delay and thus an implicit continuation condition must be always assumed. A delay condition, on the other hand, expresses directly what should, or should not happen.

A delay clause in this form can be very easily compiled, by transforming it to

```
p(X, Y) :- var(X), var(Y), delay(p(X, Y)).
```

and this is more or less what the SEPIA compiler does. The head of a delay clause uses one-way pattern matching rather than the full unification, and also the body of the delay clause is not allowed to bind any variable in the call. This is important because delay clauses are in fact meta-clauses and they must not bind the object variables, they can only test the call arguments. The predicates allowed in the body of a delay clause are **var/1**, **nonground/1**, **\==/2** and user-defined external predicates. We planned initially to allow any subgoal in a delay clause, but it turned out that the combination of the above predicate was sufficient for almost all coroutining programs. For instance, the **and(In1, In2, Out)** predicate that implements the logical conjunction would have a delay clause

```
delay and(X, Y, Z) if var(X), var(Y), X \== Y, Z \== 1.
```

to delay exactly when it is not possible to solve it deterministically.

One special condition for the delay clauses was used very frequently, in particular in programs that implemented various sorts of constraint propagation. If we impose a constraint on a set of variables and we want this constraint to propagate as soon as possible, we have to put the suspended constraint on the delay list of every variable from this set and as soon as any of these variable is changed, we want to wake the constraint, do the propagation, and suspend again, unless the constraint is already solved. This cannot be done with the above mentioned predicates, unless we make specialised and awkward versions of the constraint that work with 2, 3, ..., n free variables in it. To allow this kind of processing, Joachim Schimpf devised the built-in condition **initial/1** that does the following: if the predicate is called directly, it succeeds and marks all variables in its argument as suspending variables. When the call is woken, this condition fails and the predicate will thus be executed.

This is in fact one of the rare occasions where the predicate should delay no matter what are its arguments, and it is thus the caller who decides about delaying. As a matter of fact, the solution with **initial/1** is just a trick to achieve caller-defined suspension, and it is often necessary to define auxiliary predicates to make it work. For example, when we define a constraint **</2** which takes as arguments two arithmetic terms with variables ranging over finite integer domains, in SEPIA it has to be implemented as follows:

```
A < B :-
    normal_form(A, B, A1, B1),
    propagate_lt(A1, B1).

propagate_lt(A, B) :-
    <update the domains of all variables to make
```

```
     them consistent with the constraint>
    ground((A, B)) ->
        true
    ;
        delay_lt(A, B).

delay delay_lt(A, B) if initial((A, B)).
delay_lt(A, B) :-
    propagate_lt(A, B).
```

In this way, after updating the domains the constraint is called recursively. This call delays and it waits for any variable occurring it to be updated, e.g. by modifying its domain.

4.2 Data Structures

The basic structure is similar to that of ECRC Prolog. A suspended call creates a suspension which contains its arguments, code address and the woken flag which specifies if this suspension has been already resumed or not. A suspending variable is represented by a sequence of at least three words, the first one is a variable with tag `suspending`, the second is a list of suspended goals that have to be woken when the variable is instantiated and the third one is a list of suspended goals to be woken even if the variable is bound to another suspending variable (this is necessary to implement $\backslash==/2$ in delay clauses).

There are also two additional stacks, one for the variables that are responsible for suspending the current goal. They are pushed on it by the body of delay clauses together with a flag which says which of the two delay lists should be used to hold this goal. The other stack is for suspending variables bound in the head unification; at the end of the unification they are collected from it and their suspensions are resumed.

4.3 Waking

The question of the interference of waking and the cut was analyzed thoroughly. The core of the problems is that the cut is a sequential operator whereas coroutining destroys the sequential execution. One problem concerns cutting over suspended goals:

```
max(X, Y, X) :- X >= Y, !.
max(_, Y, Y).
```

If X or Y is not instantiated, the test delays and the cut is executed, even if later Y is bound to a number greater than X. It would be too costly to implement this properly and therefore we decided to check this situation only in the debugger and let the debugger print a warning.

The other problem concerns the waking. If the head unification instantiates some suspending variables and there is a cut after the clause neck, should we wake the suspended goals before or after the cut? If we wake before the cut, we might cut away a choice point of a woken goal:

```
delay d(X, _) if var(X).
d(a, 1).
d(a, 2).
d(c, 2).

a(a) :- ..., !.
a(c).

p(X) :- d(X, Y), a(X), Y = 2.
```

When we call *p(X)*, the call to **d/2** will be woken inside **a/1**, its choice point will be cut and the unification $Y = 2$ fails. On the other hand, calling *p(a)* or *p(c)* succeeds.

If we wake after the cut, we might again commit to the wrong clause:

```
b(1) :- !.
b(2).

?- X > 1, b(X).
```

The call to $X > 1$ initially delays and if we first execute the the cut and only then wake the suspended goal, the query fails.

Since none of the alternatives is superior to the other and a sophisticated implementation would be too costly, we decided to take the pragmatic approach and wake when it suits best to the abstract machine. Therefore, SEPIA wakes only immediately before a regular goal or at clause end; sequences of simple (e.g. in-line expanded) predicates do not cause any waking. In this way, the user can still force waking before the cut, namely by inserting a regular goal (e.g. **true/0**) before the cut.

Waking is done as follows: in coroutining mode the compiler inserts **resume** instructions at places where a suspending variable might have been instantiated. The instruction checks the waking stack and if there are some suspending variables pushed on it, it calls the routine to wake these goals. To

have more efficiency, the waking routine was hardcoded using special WAM instructions and it was a source of numerous bugs. It might even be that writing it straight in Prolog would have made it almost as fast and much simpler, especially if we take into account that we needed two copies of it, one for the optimised case and one for the debugger (SEPIA has no interpreter, the debugger uses the compiled code enhanced by some debug instructions). This scheme is quite efficient because very often the compiler can recognise that no variables will be bound (mode declarations help, too). On the other hand, it is not possible to mix code compiled in coroutining and non-coroutining mode and so it is allowed to switch on the coroutining mode only before any user predicate is compiled. Sometimes this is quite inconvenient.

There is an interesting situation when an iterative clause (a non-unit clause without an environment) has woken some goals. We have to call the waking routine, but the clause has no environment, and so we have no place to store the continuation. This problem was solved by allocating the environment in the **resume** instruction and deallocating it by a **deallocate** instruction that follows it. If there are no goals to resume, no environment is allocated and the **deallocate** instruction is skipped.

Before a suspension is invoked, its `woken` bit is set and this change is value-trailed if necessary.

4.4 Scheduling

There is no particular waking order in SEPIA, the new suspensions were inserted at the beginning of the delay list and after the unification all delay lists of bound suspending variables were linked together and woken in this order. To make the linking in constant time, a pointer to the last element in the list was stored in the suspending variable. Later this became a circular list so that it was possible to insert new suspensions both at the beginning and at the end of the list, but this feature was never really used, because the users wanted mostly predicates of some type to be woken first or last and this could not be guaranteed if the unification instantiates more than one suspending variable.

4.5 Re-delaying

After a goal was woken, but one of the delay clauses succeeded again, it is re-delayed. At this time the pointer to the suspension is still available and so only its `woken` bit is reset to zero (unlike in ECRC-Prolog this change does not have to be trailed) and the suspension is placed in delay lists of all suspending variables of the call. This is still slightly inefficient because the suspension might already be inside some or all of them. It could have been optimised - goals suspended by delay clauses with only **var/1** conditions are guaranteed

to be in all appropriate delay lists on re-delay.

4.6 Memory Management

SEPIA has only one trail which can store entries of various types, each entry is tagged to recognise its type. All coroutining data structures are located on the global stack.

4.7 Support for the Delayed Goals

SEPIA, whose goal was to be a system that could be easily extended, needs various facilities to process suspended goals. We provided a predicate that, given a variable, converted its lists of suspensions to a list of goals and returned this to the user, so that it could be processed in Prolog. This turned out to be not quite sufficient, because the list does not contain the definition modules of the delayed procedures, so the goals cannot be called. In order to obtain a list of all currently suspended goals (needed also in the top level loop to print floundered goals), all suspensions are linked together, whenever a new goal is suspended, its suspension is prepended to this list. This is not too constly and it allows a very fast access to suspended goals; on the other hand, it complicates the garbage collection because all suspensions are accessible, even if they are garbage.

The debugger was enhanced to take into account the coroutining and the above mentioned cut warnings. It has delay and resume ports and commands to display suspended goals, to skip to the place where a goal is woken, etc.

4.8 Conclusion

The SEPIA coroutining turned out to be very useful and very efficient, we have not found any system with faster coroutining execution. The reason for this was of course that most of the primitives were hardcoded and impossible to change.

After some time, we implemented metaterms (attributed variables) [8] on top of the coroutining primitives, because it was very easy, but it was conceptually strange because normally one would expect coroutining to be built on top of attributed variables and not vice versa.

We have had many user requests to change the way coroutining works and it turned out that flexibility is more needed than performance. The users mostly wanted to use more than two delay lists in a suspending variable, or to change the waking order. With various constraint propagation systems being built on

top of SEPIA it became clear that everybody would benefit from lifting the coroutining implementation to a level higher where it could be more easily changed.

5 ECL^{i}PS^e

With metaterms already available in the language it was clear that implementing coroutining on top of them would be easy and flexible enough. In ECL^{i}PS^e , which was created by merging SEPIA with another ECRC LP system, MegaLog, we have, together with Joachim Schimpf, made metaterms into first class Prolog objects. They have their own syntax, compilation, etc., and we have built coroutining strictly on top of the metaterms. The code changes were in fact not very big, we have mostly replaced large portions of C code by several Prolog lines. This means more or less that there is no particular support for coroutining any longer, suspending variables are just metaterms whose attributes store the suspended lists and whatever else is needed. We have of course provided macro transformations for backward compatibility so that delay clauses still work normally, however the user has the possibility to inspect and modify any part of the coroutining scheme.

We have imposed only some restrictions on the data structures used: the suspended lists are difference lists so that one can add new suspensions at the beginning or at the end (using value-trailed destructive assignments), the suspension is a special opaque data type and we provided a predicate to convert a goal to a suspension and vice versa, another one to insert a suspension to a specified delay list in the metaterm's attribute and finally one that invokes the suspension and sets its `woken` flag. A goal can be delayed explicitly by creating a suspension and inserting it into a delay list. Since unification of metaterms raises an event, the waking is completely taken care of by the metaterm event handler. The handler usually takes the attributes of suspending variables, finds the suspensions and calls them.

The default scheduling strategy is like the one of SEPIA, however the users have now the possibility to change the way suspensions are handled; they can define new suspended lists and the order in which they are woken, e.g. to wake simple deterministic goals first, etc. They can even define one global delay list where all suspensions are stored and woken strictly in the order of their suspension, or divide predicates into several classes and state that while a predicate of a certain class is executing, it must not be interrupted by waking a suspended goal of a lower class, etc.

The performance of the ECL^{i}PS^e coroutining is of course below that of SEPIA, the primitive actions (suspending and waking) are in average 50% to 100% slower, however in large programs the proportion of these primitive actions seems to be not significant. The main point is nevertheless the increased flexibility of the whole design which will allow us to experiment with new systems and schemes and which could in fact bring much higher gains in

efficiency.

The experiences we have had so far with ECL^{i}PS^e are mostly positive. For instance, the library that implements arithmetic constraints over finite integer domains was adopted to the new scheme. It was previously written in SEPIA using only suspended goals. With metaterms it could be simplified and cleaned. While keeping the whole control and scheduling in Prolog, we have eliminated some of its bottlenecks due to slow Prolog processing of arithmetic expressions by rewriting them in C, and the performance of the resulting system is not too far away from CHIP, with the difference that in CHIP all the constraint processing is hardcoded in C while ECL^{i}PS^e can very easily define new constraints or change the control strategy for some or all of them.

Currently we are evaluating the new design and trying to identify features that may not fit together well. One of them is the question how to organise the use of metaterms. If one program assumes a certain structure of the metaterm attribute, it is not compatible with other programs that assume a different structure. It seems that making the metaterms module-dependent could be a simple way to solve it. Another problem is how to handle suspending variables in built-in predicates. When e.g. a suspending variable occurs in an asserted term, should all its suspensions be compiled with it or ignored? Similarly, in the instance test, we sometimes want to take the attributes into account and sometimes don't.

Since the delay clauses are no longer treated as special primitives, the compiler cannot take them into account and so it does not know whether a predicate might delay or not. It does not recognise re-delays, either. On the other hand, the user has the full control and can optimise re-delays explicitly, or even make iterative re-use of suspensions as suggested in [6]. This is a good example of the rule that the more a user can do, the less can be done by the compiler and vice versa.

Acknowledgements

We thank to Joachim Schimpf for valuable comments on a previous version of this paper and for his contribution in SEPIA and ECL^{i}PS^e development.

Bibliography

[1] Mats Carlsson. Freeze, indexing and other implementation issues in the WAM. In *Proceedings of the 4th ICLP*, pages 40–58, Melbourne, May 1987.

[2] Takashi Chikayama. Esp reference manual. Technical Report TR-044, ICOT, February 1984.

[3] Alain Colmerauer. Prolog II manuel de reference et modele theorique. Technical Report ERA CNRS 363, Groupe Intelligence Artificielle, Faculte des Sciences de Luminy, March 1982.

[4] S. Gregory K. L. Clark, F. G. McCabe. Ic-Prolog language features. In *Logic Programming, ed. Clark and Tarnlund*, pages 253–266. Academic Press, London, Departmemt of Computing, Imperial College, London, 1982.

[5] Yasunori Kimura and Takashi Chikayama. An abstract kl1 machine and its instruction set. In *Proceedings 1987 Symposium on Logic Programming*, pages 468–477, San Francisco, September 1987.

[6] Micha Meier. Recursion vs. iteration in Prolog. In *Proceedings of the ICLP'91*, pages 157–169, Paris, June 1991.

[7] Micha Meier, Abderrahmane Aggoun, David Chan, Pierre Dufresne, Reinhard Enders, Dominique Henry de Villeneuve, Alexander Herold, Philip Kay, Bruno Perez, Emmanuel van Rossum, and Joachim Schimpf. SEPIA - an extendible Prolog system. In *Proceedings of the 11th World Computer Congress IFIP'89*, pages 1127–1132, San Francisco, August 1989.

[8] Micha Meier and Joachim Schimpf. An architecture for prolog extensions. In *Proceedings of the 3rd International Workshop on Extensions of Logic Programming*, pages 319–338, Bologna, 1992.

[9] Lee Naish. An introduction to MU-PROLOG. Technical Report 82/2, University of Melbourne, 1982.

[10] Lee Naish. Negation and quantifiers in NU-Prolog. In *Third International Conference on Logic Programming*, pages 624–634, London, July 1986.

[11] Dan Sahlin and Mats Carlsson. Variable shunting for the WAM. In *Proceedings of the NACLP'90 Workshop on Prolog Architectures and Sequential Implementation Techniques*, Austin, October 1990.

[12] Ehud Shapiro. A subset of concurrent Prolog and its interpreter. Technical Report TR-003, ICOT, Tokyo, Japan, January 1983.

[13] K. Ueda. Guarded horn clauses. Technical Report TR 103, ICOT, 1985.

[14] Kazunori Ueda and Masao Morita. A new implementation technique for flat GHC. In *Proceedings of the 7th ICLP*, pages 3–17, Jerusalem, June 1990.