Shallow Backtracking in Prolog Programs

Micha Meier

ECRC-ECRC-95-11

technical report ECRC-ECRC-95-11

Shallow Backtracking in Prolog Programs

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 : micha@ecrc.de

Abstract

The efficiency of Prolog compilers is increasing rapidly but the Prolog programs still cannot compete with traditional languages when executing simple conditionals. In this paper we present a possibility to increase Prolog performance by exploiting the shallow backtracking. Shallow backtracking is initiated when a call fails to unify with the head of a clause and it backtracks to another clause in the same procedure, as opposed to deep backtracking which requires going back in the search tree and trying an alternative of a previous goal. We introduce modified OR-level instructions and data management and discuss the impact on performance of Prolog programs.

1 Introduction

One of the outstanding features of Prolog is the ability to yield multiple solutions of a procedure call by backtracking to a previous OR-node and trying another, not yet explored alternative. One of Prolog's drawbacks is that backtracking is the **only** means to solve a failure of any kind, even a failure of a simple arithmetic test, e.g. inequality of two numbers. Backtracking is such a powerful and general tool that, when used inappropriately, it can significantly degrade the performance of even simple procedures. Although partial evaluation might cure some of the drawbacks, there still remain cases where backtracking must be used.

The concept of shallow backtracking has been introduced in [15] to make the distinction between two types of backtracking, one of which can be handled in a simpler way than the other one. *Shallow backtracking* occurs when the unification of a call with a clause head fails but there still are other untried alternative clauses for it. In terms of AND/OR tree traversal, which is what Prolog does during the execution, the shallow backtracking occurs while expanding the immediate AND successor of an OR node. The other type of backtracking is *deep backtracking* and it occurs when a call to a procedure fails to unify with a clause and there is no other alternative to try. In this case the system has to go back and try an alternative of a previous goal. In the later version of Prolog machine [16] the concept of shallow backtracking is no longer present, which we consider as a drawback, as it may improve the performance of many procedures significantly. Our approach to perform shallow backtracking differs from [15] where the system does not try to postpone the creation of a choice point, it only does not restore the information from the choice point when shallow backtracking occurs.

We suppose that the reader is familiar with the principles of WAM ([16], for more detailed explanation see [17] or [4]) we will nevertheless describe its stack and register management in order to express the impact of backtracking. The environment stack in WAM corresponds to the procedure invocation stack in traditional languages however its structure and management is more complicated to allow backtracking, i.e. restoring a previous execution state in order to try another alternative. The stack contains two types of frames, *environments* and *choice points*. Environments are the classical procedure invocation frames containing permanent (local) variables and control information whereas choice points contain arguments of the procedure call and the values of several state registers. In the AND/OR tree which is expanded during the execution of a Prolog program, the environments represent the AND nodes and choice points the OR nodes [10]. Whenever a procedure is called which has several clauses that potentially match the call, a choice point is pushed on the stack. The important registers used in the WAM are the environment pointer **E** which points to the current environment, the **B** pointer pointing to the last choice point, the **CP** pointer representing the return code address; the virtual machine which was used to incorporate shallow backtracking slightly differs in some registers but the suggested optimizations apply to the original concept as well. The main difference is that the machine maintains the register **TE** which points to the top of the environment stack rather than to compute the stack top dynamically. Apart from that, some registers have different (we hope more consistent) names : **TG** (top of the global stack), **GB** (global stack backtrack point) and **TT** (top of the trail).

Each choice point contains *argument registers*, which are used to pass arguments to the called procedure, and state registers : **TT**, **TG**, **B**, **E**, **CP** and a slot for the address of the alternative clause **BP(B)**. When backtracking occurs, the argument registers are refreshed from the choice point and the control registers are used to restore the state of the stacks. Yet the two actions may be redundant, e.g. if the argument registers still have the right value or if the stacks did not change since the creation of the choice point. We will show in an example what are the drawbacks of the Prolog machine that concern condition testing.

Example:

> Let us define a procedure *memberchk* that will be used to add an element to a set represented by an open-ended list. It checks whether the list contains the element and inserts it at the end if it is not found. Its Prolog code is
>
> **memberchk(Item, [Item|_]) :- !.**
>
> **memberchk(Item, [_|Rest]) :- memberchk(Item, Rest).**

When this procedure is called, the first argument is always a nonvariable and the second is either a variable representing the empty set, or a list ended by a free variable rather than by *nil*. It is clearly deterministic, only one clause will match the call. However, it is not possible to make any compile-time decisions as to which clause will match the call, particularly no *indexing* is possible.

During the execution of this procedure the following will be done :

- a choice point is created on the stack - two argument registers and 6 control registers are pushed on it
- the second argument register is unified with **[Item|_]**, **Item** being the first argument
- if the unification succeeds, the choice point is popped, some registers are restored (**GB**, **B**) and the procedure exits

- if the unification fails, the argument registers and control registers are restored from the choice point, the choice point is removed and the execution continues at the alternative clause whose address was stored in the choice point
- the second argument register is unified with [_|**Rest**] and the procedure calls itself with a new second argument, **Rest**.

The same procedure written in a C-like language would be

```
memberchk(item, list)
term item;
cons *list;
{
        while (*list != VAR)
                if (list->car == item)
                        return;
                else
                        list = list->cdr;
        *list = new_cons(item, VAR);
}
```

The difference between the two approaches is tremendous.

This is of course a special example where no structures are created on the global stack that would have to be erased on backtracking and no variables are bound that would have to be unbound. Still most of what the Prolog machine does when executing this procedure is redundant: the argument registers do not change during the unification of the first clause, most of the control registers are unchanged as well. Moreover, since memory accesses are costly, pushing a choice point on the stack and popping it right after causes a significant loss of performance.

The reason for the big discrepancy between the two examples is that Prolog's only way to execute *if-then-else* statements is using a choice point even if the condition to be evaluated is quite simple and deterministic. In the above program, the system only tests whether the second argument is unifiable with a *cons* cell whose head is the first argument; if not, it need not do much to restore the state before testing this condition.

Whenever *deep* backtracking occurs, the choice point is relevant and all the information stored there is useful; on the other hand, in *shallow* backtracking we could save a considerable amount of work. Instead of creating the choice point right at the procedure beginning, we could postpone this until calling the first user-defined Prolog subgoal in the body. This will in some cases lead to much more efficient code, especially when the choice point will not be created at all. Procedures which are deterministic in the sense that only one clause will match any admissible call do not in fact need a choice point in the stack, the necessary information can be kept in registers. This applies both to procedures where the head unification can succeed only in one clause and to procedures whose clauses have some additional tests in subgoals followed by a cut, especially e.g. guarded clauses.

Other procedures which are not deterministic, i.e. more than one clause will match the call, do not require the same treatment - when the choice point has to be created anyway, there does not seem to be any reason to try to postpone that. However, if the procedure performs some shallow backtracking, it could use the information saved in the registers rather than to access the choice point on the stack which can speed it up.

Example:

```
?- mode iTrans(+, -).
...
iTrans([[L, Nl], inc, Lo], [[Nl], std, Lo]).
iTrans([[L, Nl], inc, Lo], [[Nl], inc, Lo]).
iTrans([[L, Nl], inc, Lo], [[L, Nl], inc, Lo]).
iTrans([[L, Nl], dec, Lo], [[L], std, Lo]).
iTrans([[L, Nl], dec, Lo], [[L], dec, Lo]).
iTrans([[L, Nl], dec, Lo], [[L, Nl], dec, Lo]).
...
```

Since it is fairly uncommon to index on the second element of a list, in the call

?- iTrans([[a, c], dec, [a, b, c, d]], S)

the head unification will fail three times before yielding any solution; using shallow backtracking will improve the performance even if the procedure is not deterministic.

The rest of this paper is organized as follows : in Section 2 we present our strategy to handle shallow backtracking and the necessary changes in the abstract machine, in Section 3 the new data structure is introduced, Section 4 comments the abstract instructions and Section 5 discusses the impact on performance and implementing other primitives like the *cut*.

2 Handling Alternatives in Prolog

We will now present a method to handle choice points efficiently by fully exploiting the possibilities of shallow backtracking. For deterministic procedures, the choice point will not be created at all and for nondeterministic ones it will be created only when the head unification succeeds and a user-defined Prolog subgoal in the body is going to be executed. When shallow backtracking occurs, the choice point will be used only to access information that is not available anywhere else. As the suggested change is not quite straightforward, some problems may arize; let us now analyze them and present a solution.

- Since the **try**/**try_me** instructions will no longer be responsible for pushing a choice point, we need another instruction which would correspond to the clause neck and which would test whether it is necessary to create the choice point. This will be the **neck** instructions and/or its derivatives. It will be generated before the **proceed** instruction for facts and in the neck of rules.

- In the usual WAM, when a procedure with several clauses is called, a choice point is pushed on the stack and then the first alternative clause is executed. If this clause has more than one subgoal in the body, an environment is pushed on the stack which is therefore above the choice point. When the first alternative fails, the environment is popped and the next alternative is tried. This may cause problems when we postpone the choice point creation: we would like to push the choice point only when it is certain that the head unification succeeded, i.e. no shallow backtracking will occur. But then the clause environment would have to be pushed first and the choice point would be pushed after it - in the wrong position.

  Example :

  ```
  p(X, a) :- q, r.
  p(X, b) :- c, d.
  ```

  The right sequence of frames is

  If the head unification with `p(X, a)` succeeds and some of the predicates in the body of the first clause fail, all frames more recent than the choice point should be deallocated.

- A related problem concerns accessing permanent variables in the current environment during the head unification: if some permanent variables

```
|     . . .     |
|---------------|
|     ChP(p)    |           |
|---------------|           |
|     E(p)      |           |
|---------------|           |
|     E(q)      |           V
|---------------|

                        stack top
```

are bound, they have to be allocated somewhere to keep track of the binding and therefore we cannot postpone the environment allocation (except maybe for the control part) until it is known whether there will be a choice point on the stack. [1]

- Another problem could be that some of the control or argument registers may actually change during the head unification and we would not be able to restore them without the choice point if the unification fails, or to create the choice point if it succeeds.

- The compiler cannot decide generally whether the machine will be in a deterministic state or not and whether a failure will cause deep or shallow backtracking as this often depends on the value of goal arguments. Hence we must be able to recognize this situation dynamically by maintaining a new state register. Another register will be necessary to specify whether the current procedure already has pushed a choice point or not because some alternative clauses may be executed after both shallow and deep backtracking.

  Note that some authors use the term *nondeterministic state* and *shallow backtracking* for a situation where the top frame on the stack is a choice point. This does not suit our scheme where the optimizations are possible only when this choice point belongs to the current procedure.

  Example:

  p(X, Y) :- q(1, X), r.
  p(X, Y) :- . . .
  r :-

  A choice point is pushed for *p*. If *q* does not leave any objects on the stack, it is still at the top when *r* is called but at that

[1] It is also possible to keep the permanent variables in registers and to move them into the environment only when the unification succeeds. For software implementation, however, this would be a better solution only if the ratio of failed to successful unifications is sufficiently high. Experimental results show [1] that there are much more successful unifications than failed ones. Apart from that, the proposed design does not prevent this kind of optimization.

time the values of several registers, e.g. the argument registers or the trail pointer, have changed and the original values would be lost unless saved in the choice point.

3 Memory Organization

To be able to handle shallow backtracking efficiently we will introduce a new stack and several new registers and modify the usage of some old registers.

3.1 Stack Splitting

We use separate stacks for environments and for choice points. The first consequence is that we can push an environment before the choice point without violating the right order, and the second one is that the stack management is more unified (the idea of separating the two stacks was first suggested in [14] to improve the locality of stack references).

The only change which is due to stack splitting concerns local variable trailing and management of frames. If only one stack is used, the register **B** points to the most recent choice point and the register **E** points to the current environment. The top of stack **TE** is always set to the greater of **B** and **E**; variables that are allocated before the most recent choice point, i.e. whose addresses are less than **B** have to be *trailed* when they are bound so that they may be reset when backtracking occurs.

After splitting the stacks the register **B** does not point any longer to the environment stack and it cannot be used for its management. We therefore need a new register **EB** (environment stack backtrack point) to play this role. Its use is almost the same as for the **GB** register (backtrack point in the global stack) which points to the global stack and is used for trailing global variables and for resetting the top of the global stack. The only disadvantage of introducing this new register is that (similarly to **GB**) it has to be stored in each choice point and restored in the **cut** instruction.

On the other hand, the active choice point is now always at the top of the choice point stack and the previous one is directly below it. This means that the choice point stack is a real stack and we do not need to store the reference to the previous choice, i.e. the **PB** field and thus the size of a choice point is not increased.

3.2 Lifetime of Choice Point Fields

Several control registers may change during the head unification which would prevent us from storing the right values in the choice point or from restoring the right state after shallow backtracking. Here we list the items needed to create the choice point in the choice point stack together with their lifetime :

CE(B) the pointer to the current environment, obtained from the **E** register value at the procedure entry. The **E** register is changed when a new environment is created; its old value is stored in the environment so that it may be restored after the environment is deallocated.

CP(B) the continuation pointer which points to the code to be executed after the current goal is solved. It is set from the **CP** register that does not change until the first Prolog subgoal (which is not the last one) in the clause body is called.

B(BP) the address of the next alternative clause, it is compiled in the body of **try** and **retry** instructions [1]

TE(B) pointing to the top of the environment stack at the moment the choice point was created. It is set using the **TE** (top of the environment stack) register or the new **EB** (environment stack backtrack point) register. The register **EB** must be set before the head unification since it is used to test whether local variable bindings have to be trailed and it does not change during the unification.

TG(B) pointing to the top of the global stack at the moment the choice point was created. It is set using the **TG** (top of the global stack) register or the **GB** register. Similarly to the previous one, **GB** must be set before the head unification to be able to trail global variables.

TT(B) the top of trail at the moment of choice point creation, obtained from the **TT** (top of the trail) register. Since some variables may be trailed in the head unification, the value of **TT** may be changed when the unification is finished.

We can see that the registers **CP**, **EB** and **GB** do not change in the head unification and hence we can use their values for the corresponding choice point fields. On the other hand, the **BP(B)**, **CE(B)** and **TT(B)** items must be treated in a different way since the registers **E** and **TT** may change between the **try** instruction and the clause neck and **BP** is available only in the indexing instructions.

[1] Everything that is being said about the **try**, **retry** and **trust** instructions applies, of course, to **try_me**, **retry_me** and **trust_me** instructions.

The rest of the choice point contains the arguments of the call. Very often it is the case that the argument registers are not changed during the head unification. It is not always the case as the following, well known, example shows :

Example :

append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).

(the generated WAM code is)

get_list $\mathbf{A}_1$

unify_variable $\mathbf{X}_4$

unify_variable $\mathbf{A}_1$

get_list $\mathbf{A}_3$

unify_value $\mathbf{X}_4$

unify_variable $\mathbf{A}_3$

execute append/3

The **unify_variable** instructions change the argument registers directly, hence if the unification of the third argument fails, the first argument register is no longer valid and it has to be restored from the choice point (if there is any) even when the backtrack is shallow.

When some of the argument registers change during the head unification, it is due to an optimization of register allocation - it directly loads an argument of the first subgoal; the usual compiling strategy is to minimize the data movement between the head and the first goal [2]. The above instruction

unify_variable $\mathbf{A}_1$

is a shorthand for

unify_variable $\mathbf{X}_5$

...

put_value $\mathbf{X}_5$**,** $\mathbf{A}_1$

and it therefore saves one register-to-register transfer. With such optimizations it would not be possible to re-use argument registers on shallow backtracking and we would therefore have to execute two register-memory transfers for each argument, one to save it to the choice point and another one to restore it. As shallow backtracking occurs fairly often [18] it seems that a more efficient approach would be to sacrifice the unification optimizations that change argument registers in favor of shallow backtracking. Other possibilities are discussed in Section 5.

3.3 The Alternative Clause

In classical WAM, the address of the alternative clause is stored into the choice point by the **try/retry** instructions so that when the unification fails, the **fail** instruction performs a jump to this address. If no choice point were created, this address has to be stored somewhere else, unless it is compiled directly into the code for unification; this is not always possible as the alternative clause in an indexed procedure generally depends on the call arguments and can be specified only at runtime. We therefore introduce a new register **BP** to store the alternative address and to handle it similarly as the **CP** register. It will be set by the **try/retry** instructions and stored into the choice point when necessary. This will save at least one memory access on shallow backtracking since the alternative address will be available in a register.

3.4 Environment Pointer

If the clause whose head is being matched has no environment, the **E** register is not changed after the unification and we do not need to restore it after a failure or in order to save it into a choice point. On the other hand, when an environment is pushed on the local stack, the value of **E** changes. The old value of **E** is always stored in the environment and it can be accessed there if necessary. The disadvantage of this approach is that we would need two different **neck** instructions, one for clauses without environment which would store into the choice point the current value of **E** and another one for clauses with an environment which would use **CE(E)**.

Some systems postpone environment creation until a permanent variable is accessed hoping to save time when the unification fails before environment pushing [13]. However, this technique is not possible here since if the unification succeeds, the previous value of **E** is not available in a register anyway.

We suggest here another approach which is even more efficient: the permanent variables in clause head can be accessed through the stack top pointer rather than as an offset from the **E** pointer. If the unification succeeds, the environment is completed by saving the control information (continuation) and changing the **E** register after the choice point with the right value of **E** has been created. The instruction **allocate** thus follows the **neck** instruction. If the unification fails, the **E** register need not be reset and we have not lost the time by allocating the frame. This has an advantage even for deterministic (one-clause) procedures which thus will not create an environment if the head unification fails.

The permanent variables in **put** instructions, however, have to be accessed as usual through the **E** register since the clause environment might not be at the stack top when the body subgoals are executed. We therefore need two sets of **unify** instructions, one for the head that uses the stack top pointer to access the permanent variables, and another one for the clause body that uses the **E** register. This suits many of the current implementations as they use different **unify** instructions for the body anyway, since they are executed always in *write mode* (always creating a structure) and no explicit test for the mode bit is needed.

3.5 Trail Backtrack Point

The **TT** register may change during the unification by trailing variable bindings. The easiest method to keep its value is to introduce a new register **TB** (trail backtracking point) to store the value of **TT** before the head unification started. This register will have a significant value only during the head unification(s). Note that when backtracking occurs, bindings are untrailed up to **TT(B)** which is not accessed each time to test the end of untrailing loop but a temporary register is used to keep **TT(B)**. We just extend the usage of this temporary register from the **try** instruction up to successful unification.

3.6 New State Registers

The compiler cannot generally decide whether the machine will be in a deterministic state or not and whether a failure will cause deep or shallow backtracking as this often depends on the value of goal arguments. Therefore we introduce a new boolean state register **D** (D for **D**eterministic or **D**eep) which specifies whether there are some alternatives for the current call or not. If it is *true* it means that there are no alternatives, the state is deterministic and a failure will cause deep backtracking. This register will always have a significant value : the **try/retry** instructions set it to *false* whereas the instructions **neck** and **trust** set it to *true*. Another flag is necessary to remember whether the current procedure already has pushed a choice point or not: the boolean register **ChP** is *true* if the current procedure already has a choice point.

4 New Instruction Set

Here we describe the WAM instructions that are changed in our design. A substantial modification compared to the original WAM is contained in the indexing **try**/**retry**/**trust** as well as the **fail** instructions. The **try** instruction only sets the above mentioned registers and stores the value of the **BP** register. The **fail** instruction shrank to an unconditional jump to the address stored in the **BP** register. This is due to a change in the **retry** and **trust** instructions.

In the original WAM the **fail** instruction resets all registers from the choice point, untrail trailed variables and jumps to the address in **BP(B)** which is an address of a **retry** or **trust** instruction. The ECRC abstract machine, on the other hand, does not use a fixed choice point size nor stores the arity in a choice point; in case of deep backtracking the argument registers are refreshed in the **retry**/**trust** instructions that have the arity as an argument. If a part of restoring would be done in the **fail** instruction and the rest in the indexing instructions, it would be necessary to test the register **D** twice. Instead, all the job is pushed to the indexing instructions which enables us not to test the **D** register at all: We know that backtracking caused by a failure between **neck** and **try** instruction is deep while a failure between a **try** or **retry** instruction and a **neck** or **trust** instruction will cause shallow backtracking.

We have specified two entries to each of **retry** and **trust** instructions depending on the value of the register **D**. The first entry, which is the beginning of the instruction, corresponds to deep backtracking and it contains the code to restore some state registers as well as the argument registers. At a fixed offset from the beginning of the instruction there is the *indeterminate entry* for shallow backtracking. These entries are used in the following way: the **BP** register is initially set by **try** and **retry** instructions to point to the indeterminate entry so that when shallow backtracking occurs, no test on **D** has to be made. In the clause neck a fixed offset is subtracted from **BP** so that **BP** points to the deep backtrack entry.

The **neck** instruction mentioned above tests whether the choice point has to be created and if so, it is pushed using the information in registers.

This approach favors the shallow backtracking, since a failure in a nondeterminate state requires almost no action. Of course other configurations which favor the deep backtracking are possible. The idea to use two different entries for backtracking instructions was first expressed in a different context in [12].

try_me_else L
This is the first instruction in the sequence of clauses that may possibly perform deep backtracking. The registers **TE**, **TG** and **TT** are saved into their corresponding 'B' registers, the **BP** register is saved into the previous choice point and it is set to L (the shallow entry). Moreover, the flags **D** and **ChP** are reset.

```
D := false;
ChP := false;
EB := TE;
GB := TG;
TB := TT;
BP(B) := BP;
BP := L;
```

retry_me_else L, Arity
This instruction introduces the code for a clause in the middle of a chain of clauses that may perform shallow backtracking. It has two entries, depending on the state of the **D** flag. If **D** is true, the backtracking is deep and the registers **TB**, **E** and **CP** have to be restored from the choice point. Furthermore, the argument registers are refreshed from the choice point, the flags are set and the register **TE** is reset from **EB**. If **D** is false, no special action takes place. In both cases the register **TG** is reset from **GB** and variables on the trail are untrailed up to the value of **TB**.

```
true( D) entry:
        E := CE(B);
        CP := CP(B);
        TB := TT(B);
        TE := EB;
        Refresh_arguments_from_choice_point(B, Arity)
        D := false;
        ChP := true;
        /* goto lab; */
not( D) entry:
lab:
        Untrail TT to TB;
        TG := GB;
        BP := L;
```

trust_me Arity
This instruction is the last in the sequence of clauses that may perform shallow backtracking. Similarly to the **retry** instruction it has two entries depending on the value of **D** flag. If **D** is set, the same operations are performed as for the **retry** instruction, except that the flags are not set and the **B** register is reset. If the **D** flag is not set, this instruction sets it and the register **B** is set to point to the previous frame. Then, for both entries, trailed variables are untrailed, the **TG** register is set to **GB** and the registers **EB**, **GB** and **BP** are reset from the choice point. This sequence effectively erases the choice point from the stack.

```
true( D) entry:
        E := CE(B);
        CP := CP(B);
        TB = TT(B);
        TE := EB;
        Refresh_arguments_from_choice_point(B, Arity)
        B := B - 6 - Arity;
        goto lab;
not( D) entry:
        D := true;
        if ChP then
                B := B - 6 - Arity;
        endif;
lab:
Untrail TT to TB
TG := GB;
EB := TE(B);
GB := TG(B);
BP := BP(B);        /* true( D) entry ! */
```

neck Arity
this instruction is generated before the first subgoal that may backtrack or change some registers or before the clause end. In case the **D** flag is reset, the choice point is created depending on the flag **ChP**, the pointer **BP** is updated to point to the *deep* entry of the **retry/trust** instruction and the **D** flag is set.

```
if not(D) then
        if not(ChP) then
                Create_ChP(Arity, TB, GB, EB, _, CP, E)
        endif;
        BP := Deep(BP);
        D := true;
endif;
```

neckcut Arity
This instruction is generated for a *cut* that occurs before any subgoal that may backtrack or change some significant registers. If the flag **D** is not set, the registers **EB**, **GB**, **BP** and possibly **B** are reset from the choice point. At the end, the flag **D** is set.

```
if not(D) then
        if ChP then
                B := B - 6 - Arity;
        endif;
        EB := TE(B);
        GB := TG(B);
        BP := BP(B);
        D := true;
endif;
```

neck_bodycut Arity
This instruction is generated before the first subgoal that may backtrack or change some registers in a clause that contains a *cut* (not handled by the *neckcut* instruction). The appropriate **B** pointer is stored in the first permanent variable and the choice point is created if necessary.

```
if not(D) then
        if ChP then
                $Y_1$ := B - 6 - Arity;
        else
                $Y_1$ := B;
                Create_ChP(Arity, TB, GB, EB, _, CP, E)
        endif;
        D := true;
        BP := deep(BP);
else
        $Y_1$ := B;
endif;
```

fail
the execution continues at the address stored in **BP**

```
P := BP;
```

cut
the instruction for a cut in the clause body which is not a neckcut. The **B** register is reset from the appropriate permanent variable, **TE** points to the end of the current environment and the registers **EB**, **GB** and **BP** are reset from the choice point. Thus all frames above the current environment are popped from the environment stack.

```
        B = $Y_1$;
        TE = E;
        EB = EB(B);
        GB = GB(B);
        BP = BP(B);
```

5 Discussion

Our aim was to speed up the shallow backtracking process without slowing the rest down too much; let us now look at the efficiency of the proposed design.

5.1 Performance Results

We have compared the speed of a system without shallow backtracking, a system that uses shallow backtracking only as far as restoring argument registers is concerned, and a system that performs all the optimizations described in this paper. In all cases ECRC Prolog was used (originally treating only the second case) with some modifications to simulate the other two approaches. A performance test with the *memberchk* example brought the following results :

- When only the **D** register was introduced and on shallow backtracking only the argument registers were not refreshed, the gain in speed compared to the standard WAM was about 15%.
- With a full shallow backtracking as described here the gain was 65%.
- To vizualize the impact of register optimizations, the performance of naive reverse with the above changes was measured and it decreased about 3% resp. 4% compared to the standard WAM.

ECRC Prolog is a portable system using C as intermediate language, so that abstract machine registers are represented by C variables instead of using machine registers. Moreover, since the system supports coroutining, data management is slightly more complicated and the impact of any change might be less significant. We suppose that on other software systems all the figures could be higher. To specify the negative impact more precisely, we have measured the performance of a compact system [7] which showed a 6% decrease of speed of naive reverse with full shallow backtracking due to non-optimized unification and a test in the clause neck; this should be considered as the upper bound of the negative impact of the proposed approach. [1] The measurements of shallow backtracking impact with the CHAT-80 program and others are currently in preparation.

[1] *Naive reverse* is a standard benchmark example which is deterministic, contains no shallow backtracking and very few abstract instructions are generated for one inference step, 7 with register optimizations and 9 without them. This implies that the impact of register optimizations in this example is very near to its upper bound over all Prolog procedures.

If we look at the impact of the suggested change on different types of procedures we obtain the following picture (ordered in descending order of advantage):

- Procedures that perform a lot of shallow backtracking, e.g. database procedures which are not completely indexed, will be speeded up significantly.
- Procedures that perform some deep and some shallow backtracking will be speeded up by the extent of shallow backtracking.
- Procedures consisting of one clause or those which are completely indexed so that only one clause will always be tried will not be affected at all as the **neck** instruction is not generated for them.
- Procedures performing only deep backtracking, e.g. generators, will not be speeded up with the restriction that environment allocation is avoided whenever possible. However, if the compiler can deduce that a procedure belongs to this class, it can generate indexing instructions that handle only deep backtracking and do not manage the new flags, they only have to manage the **D** register if it is used for cut implementation.

5.2 Register Optimizations

Procedures that use optimized register allocation might not be speeded up by the same extent as those where register optimizations are not possible. However, there are often possibilities to overcome this drawback. If the compiler can recognize that the head unification of a clause cannot cause shallow backtracking and that the choice point will not be pushed in the neck of this clause, it can safely generate optimized unification instructions. This is the case when only one clause is a possible candidate for matching, hence either in one-clause procedures or clauses in indexed procedures which cannot be entered from a `try` or `retry` instruction, which is a very simple test. Note however, that this class of procedures is not identical with deterministic procedures, since clauses in which the creation of a choice point is prevented by a (neck)cut can still perform shallow backtracking.

If e.g. the concatenate/3 procedure is written with the *nil* case first, all the register optimizations can be performed without affecting a possible shallow backtracking. Although it might seem awkward to compile a clause only in the context of a procedure, for an efficient compiler it is necessary anyway since it is the only way to recognize that the unification of an indexed argument can be omitted or reduced.

Shallow backtracking might be interesting even in context of partially evaluated programs : the memberchk/2 procedure can be partially evaluated to yield (see e.g. [11]):

```
memberchk(Item, List) :-
        List = [First|Rest],
        (
                Item = First -> true
                ;
                memberchk(Item, Rest)
        ).
```

The Prolog system still has to create a choice point for the alternative in the body of *memberchk*, it only does not have to save and restore the argument registers; this would then be another possibility (probably only theoretical) to keep register optimizations.

The consequence is therefore as follows : the optimized register allocation is not compatible with shallow backtracking. Since each register optimization saves one register-to-register transfer whereas shallow backtracking saves several memory accesses, and shallow backtracking occurs fairly often, it should be given the preference. Procedures which are known not to perform shallow backtracking or those where the impact of register optimizations is very high may be compiled in the traditional way using the register optimizations if desired.

5.3 Built-in Predicates

As the reader might have noted, we have not identified the **neck** instruction that actually changes the state from *shallow* to *deep* with the position of the ':-' operator in the clause. The reason is the obvious difference between Prolog procedures and built-in predicates, more precisely those that are not written in Prolog and that do not backtrack. If the arguments to those predicates are passed in such a way that the arguments of the calling procedure are not destroyed, we can postpone the choice point creation and changing the state to *deep* until they are successfully executed.

Example :

```
?- mode partition(+, +, -, -).
partition([X|L], Y, [X|L1], L2) :-
        X =< Y,
        !,
        partition(L, Y, L1, L2).
partition([X|L], Y, L1, [X|L2]) :-
        partition(L, Y, L1, L2).
partition([], _, [], []).
```

This is a part of the quicksort program, the procedure partition/3 is deterministic. Indexing on the first argument will filter out the last clause; if we identify the clause neck with the end of the head, we will have to create the choice point when finishing the unification of the first clause (which always succeeds). However, the arithmetic test does not change any of the state registers, so the optimal action would be to perform a shallow backtrack if it fails which would prevent the creation of the choice point.

In this way, the clause neck will correspond to the position of a *guard* in a guarded clause.

5.4 Cut Implementation

The **D** register can be used to implement the *cut* in an elegant way. The problem of cut implementation in WAM is that when entering the clause that contains a cut, it is not possible to decide whether the preceding choice point belongs to the current procedure or to a former one. The first attempt to implement it in ECRC Prolog was using a register which would be set to the current choice point by each **call** and **execute** instruction ([3], similarly in [5]). This approach penalizes each procedure call no matter whether the called procedure contains some cuts or not.

The **D** register, on the other hand, gives the possibility to find out whether the current procedure has a choice point or not even after entering the clause. The compiler can therefore generate a special **neck** instruction for clauses containing a cut in the body. This instruction stores either **B** or the previous frame address into a permanent variable depending on the value of **D**. Clauses without an explicit cut are no longer affected (supporting the opinion that the cut should not act in a metacall [8]). When the cut occurs in the neck, which is a very common case, the execution is extremely efficient.

5.5 Coroutining

The implementation of coroutining controlled by *wait declarations* of MU-Prolog [9] requires undoing the unification in case a call delays [6]. The proposed design can fulfill this requirement in an elegant way without additional constraints.

6 Conclusion

We have presented a modified indexing instruction set and memory organization for WAM which takes full advantage of shallow backtracking. The suggested system favors shallow backtracking in Prolog procedures up to a 65% increase in speed while keeping the efficiency for other procedures.

The modifications include avoiding to create and access choice points if not necessary and postponing the allocation of an environment until the head unification is known to succeed. Moreover they yield a convincing reason to separate the environment stack and the choice point stack and allow an advantageous implementation of cut. If needed, the conservative indexing instructions slightly changed may coexist with the new ones to avoid unnecessary tests or allow register optimizations for procedures where it is desirable.

Acknowledgements

We would like to thank Hervé Gallaire, Jacques Noyé and Dominique de Villeneuve for reading and commenting earlier drafts of this paper, François Bry for the initial problem, Reinhard Enders for supplying examples and all members of ECRC for the stimulating environment.

Bibliography

[1] Dominique de Villeneuve. Some benchmarks on the abstract machine of ecrc-Prolog. Technical Report lpp/cag, ECRC, October 1986.

[2] Saumya K. Debray. Register allocation in a Prolog machine. In *Proceedings of the 3rd Symposium on Logic Programming*, pages 267–275, Salt Lake City, September 1986.

[3] K. Estenfeld, D. Henry de Villeneuve, M. Meier, and B. Poterie. Ecrc-Prolog version 0.1 implementation. Technical Report LP-4, ECRC, March 1985.

[4] John Gabriel, Tim Lindholm, E. L. Lusk, and R. A. Overbeek. A tutorial on the warren abstract machine for computational logic. Technical Report ANL-84-84, Argonne National Laboratory, 1984.

[5] Ilyas Cicekli Kenneth A. Bowen, Kevin A. Buettner and Andrew K. Turk. The design and implementation of a high-speed incremental portable Prolog compiler. In *Third International Conference on Logic Programming*, pages 650–656, London, July 1986.

[6] Micha Meier. Compilation of wait declarations. Internal Report IR-LP-1102, ECRC, June 1985.

[7] Micha Meier. Program to measure potential lips. *Arpanet PROLOG Digest*, 4(23), 1986.

[8] Chriss Moss. Cut & paste - defining the impure primitives of Prolog. In *Third International Conference on Logic Programming*, pages 686–694, London, July 1986.

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

[10] Jacques Noye. Icm the abstract machine. Technical Report CA-19, ECRC, April 1986.

[11] Richard O'Keefe. On the treatment of cuts in Prolog source-level tools. In *Symposium on Logic Programming*, pages 68–72, Boston, July 1985.

[12] Jean Rohmer. The alexander method. further steps towards efficient ai languages implementation. Technical report, BULL Corporate Research Center, March 1986.

[13] Peter Van Roy. A Prolog compiler for the plm. Technical Report UCB/CSD 84/203, Computer Science Division, University of California, November 1984.

[14] E. Tick and D. H. D. Warren. Towards a pipelined prolog processor. In *IEEE 1984 International Symposium on Logic Programming*, pages 29–40, Atlantic City, February 1984.

[15] David H. D. Warren. Implementing prolog - compiling predicate logic programs. D. A. I. Research Report 39, University of Edinburgh, May 1977.

[16] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI, October 1983.

[17] David Scott Warren. The runtime environment for a Prolog compiler using a copy algorithm. Technical Report 83/052, SUNY at Stony Brook, March 1984.

[18] Akira Yamamoto, Masaki Mitsui, Hiroyuki Toshida, Minoru Yokota, and Katsuto Nakajima. The program characteristics in logic programming language esp. Technical report, Systems Laboratory, Oki Electric Co., Ltd, Tokyo, Japan, 1986.