SICS/R-90/9003

# On the Efficiency of Optimising Shallow Backtracking in Compiled Prolog

by

Mats Carlsson



SICS research report
R90003
ISSN 0283-3638

# On the Efficiency of Optimising Shallow Backtracking in Compiled Prolog‡

*Mats Carlsson*
*SICS, Swedish Institute of Computer Science*
*PO Box 1263*
*S-164 28 KISTA, Sweden*

## Abstract

The cost of backtracking has been identified as one of the bottlenecks in achieving peak performance in compiled Prolog programs. Much of the backtracking in Prolog programs is shallow, i.e. is caused by unification failures in the head of a clause when there are more alternatives for the same procedure, and so special treatment of this form of backtracking has been proposed as a significant optimisation. This paper describes a modified WAM which optimises shallow backtracking. Four different implementation approaches are compared. A number of benchmark results are presented, measuring the relative tradeoffs between compilation time, code size, and run time. The results show that the speedup gained by this optimisation can be significant.

---

‡ The paper also appears in *Logic Programming: Proceedings of the Sixth International Conference*, MIT Press, pp. 3–16, 1989.

# 1 Introduction

The distinction between two types of backtracking, a simple one which is amenable to optimisation, and a general one, was introduced in [8]. The simple type was called *shallow backtracking* and occurs when a head unification fails with more alternatives for the same procedure to try. The general case was called *deep backtracking*; it occurs when there are no more clauses of the current procedure to try when a head unification fails. If the machine state changes between a procedure call and a shallow failure can be minimised, then shallow backtracking can be implemented much more efficiently than deep backtracking.

We feel that there are many justifications to exploit an optimised mechanism for shallow backtracking in Prolog implementations. Firstly, the implementation of *if-then-else* where the *if* part consists of simple tests becomes more efficient, without complicating the compiler by introducing conditional jumps and the like, as in

```
Head :-
        ...
        (X<Y -> p(X, Y); q(X, Y)),
        ...
```

Secondly, in or-parallel Prolog implementations like Aurora [10], the relative cost of deep backtracking is higher than in sequential implementations. We expect that optimising shallow backtracking in Aurora will have a significant effect on performance. Finally, Tick showed [6] that shallow backtracking is the predominant form of nondeterministic Prolog execution.

The distinction between the two kinds of backtracking was built into PLM, the Prolog engine of Prolog-10. However, no such distinction is made in the later WAM, or "New Engine", for Prolog [9]. We assume herein that the reader is familiar with the WAM. To avoid ambiguities, we describe our WAM terminology in Section 2.

Proposals for incorporating shallow backtracking into the WAM have appeared in the literature, first in [6], for a detailed description see [3]. The key idea is that the *try* and *try_me_else* instructions only save a small part of the machine state, postponing completion of the choicepoint until a *neck* instruction is reached. This new instruction is inserted into the compiled code for each clause and is responsible for completing or updating a choicepoint where appropriate. It is placed at the earliest possible point in a clause where it can be determined that the head unification and any simple tests have succeeded. Consider, for example, the recursive clause of *append/3*:

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

| standard WAM code | WAM code with neck instruction |
|---|---|
| `get_list A1` | `get_list A1` |
| `unify_variable A4` | `unify_variable A4` |
| `unify_variable A1` | `unify_variable A5` |
| `get_list A3` | `get_list A3` |
| `unify_value A4` | `unify_value A4` |
| `unify_variable A3` | `neck` |
| `execute append/3` | `unify_variable A3` |
| | `put_value A5,A1` |
| | `execute append/3` |

Thus before the *neck* instruction is reached, the arguments for the body goal cannot be set up. As this example shows, inserting the *neck* instruction impacted the register allocation so that an extra *put_value* instruction was needed. Thus the proposed method may incur a performance overhead which may outweigh a faster backtracking mechanism. This fact motivated Van Roy et.al. [7] to introduce multiple entrypoints into compiled clauses. In their scheme, a compiled clause *C* can have four entrypoints depending on

(i) whether or not there are alternative clauses for *C*;
(ii) whether or not a previous clause has created a choicepoint for this procedure.

The four entrypoints correspond to different compiled versions of the head of *C*, specialised for the combinations of conditions (i–ii). These code streams then merge into a single compiled version of the body of *C*. Obviously, if condition (i) is true, unification failure triggers shallow backtracking, otherwise it triggers deep backtracking. Multiple entrypoints have obvious drawbacks: increased code size and compilation time.

With the *neck* instruction, the cost of creating choicepoints is slightly higher than in the standard WAM due to the negative effects on register allocation and to the extra instruction decode. However, we observe that creating and restoring choicepoints are rather expensive operations anyway.

This report tries to compare the relative merits of four different approaches to implementing backtracking (DB, SB1, SB2, and SB3) with respect to compilation time, code size, and run time. The unit of compilation is a clause in all approaches:

DB. This approach does not optimise shallow backtracking at all.
SB1. This approach inserts a *neck* instruction into each clause. There is one code stream per clause.
SB2. This approach generates two code streams for each clause: one with a *neck* instruction, used when there are alternatives for the current goal, and one without a *neck* instruction, used for determinate goals and for the last alternative in a list.

SB3. This approach is a hybrid between SB1 and SB2: Approach SB1 is taken when the body is empty or consists of simple tests only; approach SB2 is taken for other clauses.

In the above approaches, the register allocation is only impaired in code streams containing *neck*. For approaches SB2 and SB3 it is arranged at compile time so that the indexing code refers to the correct code stream, to avoid a runtime decision.

## 2 WAM Terminology

The implementation strategies described above have all been implemented as modifications SICStus Prolog [2], a compiler-based system with a WAM emulator written in C. Strategy SB3 is the one normally used in SICStus Prolog. Its abstract machine is close to the standard WAM but differs in some respects. For example, the local stack is split into an environment stack and a choicepoint stack. This modification was first proposed in [5] to improve the locality of memory references. Other changes were made as prerequisites for optimising shallowing backtracking, as described in Section 4.

We use the following abbreviations for the principal WAM registers:

| | |
|---|---|
| ***P*** | program pointer |
| ***CP*** | continuation program pointer |
| ***E*** | current environment |
| ***B*** | current choicepoint |
| ***TR*** | top of trail |
| ***H*** | top of heap |
| ***A1***, ***A2***, .. | argument registers |

A *choicepoint* is a stack frame with the following fields. Note that the "previous choicepoint" field is replaced by the "top of environment stack" field, since the local stack is split:

| | |
|---|---|
| ***P(B)*** | alternative program pointer |
| ***CP(B)*** | continuation program pointer |
| ***E(B)*** | current environment |
| ***A(B)*** | top of environment stack (shadowed by the ***AB*** register) |
| ***TR(B)*** | top of trail |
| ***H(B)*** | top of heap (shadowed by the ***HB*** register) |
| ***A1(B)***, ***A2(B)***, .. | argument registers |

An *environment* is a stack frame with the following fields:

| | |
|---|---|
| ***CE(E)*** | continuation environment |
| ***CP(E)*** | contnuation program pointer |
| ***Y1(E)***, ***Y2(E)***, .. | permanent variables |

## 3. Shallow backtracking in the WAM

In this section we describe the changes in the compiler and abstract machine necessary for optimising shallow backtracking. The compiler is a pure clause compiler, i.e. it translates one Prolog clause at a time to WAM instructions. The only way in which the indexing code is affected by the changes is by the introduction of two entrypoints per clause, as described in Section 3.2.

The key idea of the optimisation is that the *try* and *try_me_else* instructions allocate space for a choicepoint but only fill in a couple of fields, postponing completion of the choicepoint until the *neck* instruction is reached. This new instruction is inserted into the compiled code for each clause and is responsible for completing or updating the choicepoint after a successful head unification with more alternatives to try.

### 3.1 Basic scheme

We now describe the simplest approach, SB1. To implement it, a new WAM register, ***PB***, is introduced. ***PB*** holds a pointer to the next alternative clause for the current goal, or 0, if none exists. When head unification failure occurs, shallow backtracking is used if ***PB*** $\neq$ 0, otherwise deep backtracking must be used. In the shallow case, the arguments and most control registers are guaranteed to be valid: all that needs to be done is to reset all trail entries between ***TR(B)*** and ***TR***, and to restore the top of heap from ***HB***. In the deep case, all arguments and control registers must be restored from the choicepoint.

The *try* and *try_me_else* instructions are modified to create a *partial* choicepoint. The only valid fields of a partial choicepoint are ***TR(B)***, initialised to the current top of trail, and ***P(B),*** initialised to zero. The WAM registers ***AB***, ***HB***, and ***PB*** are initialised to respectively the current top of environment stack, the current top of heap, and the alternative clause. The *retry* and *retry_me_else* instructions are modified to update ***PB*** rather than ***P(B)***. The *trust* and *trust_me_else* instructions set ***PB*** to zero.

In the basic scheme, a *neck* instruction is inserted into all clauses. Its semantics is summarised by the following table:

| PB=0 | PB≠0, P(B)=0 | PB≠0, P(B)≠0 |
|---|---|---|
| noop | fill in choicepoint | set ***P(B)*** to ***PB*** |
| set ***PB*** to zero | set ***PB*** to zero | |

Thus, the *neck* instruction does nothing if ***PB*** is zero. Executing a *neck* with ***PB*** nonzero and ***P(B)*** zero corresponds to the first successful head unification for the current goal with more alternatives to try. In this case, all remaining fields of the choicepoint are filled in, including ***P(B)***, and ***PB*** is zeroed. Executing a *neck* with both ***PB*** and ***P(B)*** nonzero corresponds to a successful head unification after having backtracked to this choicepoint. ***P(B)*** is assigned the value of ***PB***, and ***PB*** is zeroed. A *neck* + *cut* combination is recognised as a special case: the current choicepoint is flushed, ***AB*** and ***HB*** are reset to their previous values, and ***PB*** is zeroed.

The compiler is modified to insert a *neck* instruction before the first *cut* or body goal which the compiler cannot compile inline. The *neck* instruction is inserted into all clauses including the last clause of a procedure, as the clause compiler has no way of knowing whether yet another clause for the same procedure will appear later. The *neck* instruction affects the compiler's register allocation as the instruction copies the procedure's argument registers when it completes a partial choicepoint. As a result, the register allocation is often poorer than in the standard WAM, requiring more temporary variables and more moves. The extra moves occur if arguments for the first body goal cannot be placed in the correct argument register until after the *neck* instruction. Notice that for clauses where the body is empty or consists of simple tests only, no extra moves are ever needed, as simple tests may take their arguments in any registers.

### 3.2 Optimisations

The drawback of the simple approach is for deterministic cases having to execute a no-op *neck* instruction which also impairs the register allocation. This can be avoided by emitting two instruction streams per clause: one with a *neck* instruction and one without. We call this approach SB2. It is arranged at load time so that *try(_me_else)* and *retry(_me_else)* instructions use the code streams with *neck* and all other ways of entering a clause use the code streams without the *neck*. The implications for the abstract machine is that ***PB*** is always nonzero when the *neck* instruction is reached, and for the compiler and loader some extra complications to manage the two code streams per clause.

To conserve space, the code for the body is shared between the two streams. The instruction streams are actually laid out as a single sequence with two entrypoints, *NonDet* and *Det*, and a jump. *NonDet* is entered when there are more alternatives to try; *Det* is used for the last alternative and for determinate calls. The *NonDet* instruction stream contains a branch to the body code whereas the *Det* stream just continues into the body code, minimising overhead for deterministic cases. The general instruction stream outline is depicted below:

```
NonDet:  < head unification instructions >
         ...
         < simple tests >
         neck
         < moves >
         branch S
Det:     < head unification instructions >
         ...
         < simple tests >
         < moves >
S:       < body code >
```

The drawback of this approach is the code size overhead which is particularly serious for unit clause databases. To reduce the overhead, we introduce approach SB3 as using SB1 when there is very little to gain from duplicating code, and SB2 otherwise.

In approach SB3, we emit a single code stream, with a *neck* instruction, for clauses where the body consists of simple tests only, in particular for unit clauses. For all other clauses two code streams are emitted. This strategy is based on the observation that for clauses where there are no general body goals constraining the register allocator, no extra register transfers are ever introduced by the *neck* instruction. The main implication for the abstract machine is that the *neck* instruction again needs to check whether ***PB*** is zero.

# 4 Nonstandard WAM features

The SICS abstract machine differs in some respects, not only by optimising shallow backtracking, from the standard WAM. Some of the other changes are however crucial for the shallow backtracking mechanism:

## 4.1 One-level indexing

In the SICS abstract machine, clause indexing is done in one step when a compiled predicate is entered, even in situations with a mixture of clauses with variable and nonvariable first arguments. This indexing step singles out an applicable subset of clauses, and at most one choicepoint is created. Although not essential for the shallow backtracking mechanism, this modification increases its effectiveness as the number of alternatives per choicepoints increases. This technique is further discussed in [1].

## 4.2 Split environment pointer

Since an *allocate* instruction can occur before a *neck* instruction, it would seem that the ***E*** register has to be saved by *try(_me_else)* and restored when a shallow failure occurs. This is avoided in the SICS abstract machine by introducing the ***E2*** register which is used instead of the ***E*** register whenever a permanent variable is accessed. The motivation for this is both to improve shallow backtracking and to delay as long as possible filling in the control part of environments and updating ***E***. In particular, ***E*** is never updated between the beginning of a clause and the *neck* instruction.

This modification is done by modifying the *proceed* instruction and splitting the *allocate* instruction into two parts: *preallocate*, placed before the first occurrence of a permanent variable, and *postallocate*, placed before the first *call* instruction:

```
preallocate:
      E2 = max(E+env_size(CP),A(B));

postallocate:
      CE(E2) = E;
      CP(E2) = CP;
      E = E2;

proceed:
      E2 = E;
      P = CP;
```

## 4.3 Preserving argument registers

With the shallow backtracking optimisation, the contents of the argument registers must not be altered prior to the *neck* instruction, as *neck* may copy these registers into a choicepoint after a number of unification steps. In the standard WAM, certain instructions (e.g. *get_value*, *unify_value*) are allowed to store the dereferenced result of the operation back into an argument register. However, this must be disallowed for shallow backtracking to work.

## 5 Performance evaluation

To study the effectiveness of optimising shallow backtracking, three performance aspects were studied: compilation time, code size, and execution time.

Strategy SB3 is currently implemented in SICStus Prolog. The other strategies were implemented by modifying parts of SICStus Prolog, carefully trying not to introduce any new overheads that would blur a comparison.

### 5.1 Benchmark programs

Four Prolog programs were studied. None of them can be considered a toy program. They were written by different people and likely represent different coding styles. The programs include two compilers which were expected to have very little nondeterminism. The other two programs were expected to contain a lot of nondeterminism. The programs were:

CHAT: an English language parser by F.C.N. Pereira and D.H.D. Warren, running a query that took 15.2 seconds. The size of CHAT is 2812 Prolog clauses.

PLM_COMPILER: the Berkeley Prolog compiler by Peter Van Roy, running a query that took 2.5 seconds. The program consists of 738 Prolog clauses.

PLWAM: the SICStus compiler by myself, compiling itself which took 250 seconds. The program contains 1147 Prolog clauses.

TP: a propositional theorem prover by Ross Overbeek, running a problem that took 47.7 seconds. The size of TP is 155 Prolog clauses.

In the size figures above, a disjunction (*P*; *Q*; *R*) counts as three clauses. The timings were made using approach DB (not optimising shallow backtracking).

In order to measure the amount of potential shallow backtracking, we computed for each program the following dynamic properties:

$P_f$ the fraction of deep failures to total failures, and
$P_c$ the fraction of completed choicepoints to total number of *try*:s and *try_me_else*:s.

These figures for the benchmarks are summarised in the following table.

**Table 1: Available shallow nondeterminism.**

| | $P_f$ | $P_c$ | $(1/P_f)+(1/P_c)$ |
|---|---|---|---|
| PLWAM: | .20 | .26 | 8.85 |
| CHAT: | .47 | .28 | 5.70 |
| TP: | .33 | .27 | 6.70 |

| | | | |
|---|---|---|---|
| PLM_COMPILER: | .23 | .34 | 7.29 |

These figures confirm Tick's observation that shallow backtracking is the predominant form of nondeterministic Prolog execution. The $(1/P_f)+(1/P_c)$ figure indicates the availability of shallow nondeterminism in each program. The higher the figure, the better the expected runtime speedup yielded by the shallow backtracking optimisation.

### 5.2 Performance data

The timing and code size data are presented in three tables below. All data have been normalised with respect to strategy DB:

**Table 2: Compilation time.**

| | SB1 | SB2 | SB3 |
|---|---|---|---|
| PLWAM: | .97 | 1.16 | 1.12 |
| CHAT: | .98 | 1.18 | 1.05 |
| TP: | .97 | 1.07 | 1.02 |
| PLM_COMPILER: | .97 | 1.15 | 1.10 |

**Table 3: Code size.**

| | SB1 | SB2 | SB3 |
|---|---|---|---|
| PLWAM: | 1.01 | 1.15 | 1.10 |
| CHAT: | 1.01 | 1.20 | 1.05 |
| TP: | 1.01 | 1.09 | 1.05 |
| PLM_COMPILER: | 1.01 | 1.16 | 1.10 |

**Table 4: Execution time.**

| | SB1 | SB2 | SB3 |
|---|---|---|---|
| PLWAM: | .92 | .89 | .90 |
| CHAT: | .96 | .93 | .92 |
| TP: | .85 | .85 | .85 |
| PLM_COMPILER: | .93 | .89 | .90 |

### 5.3 Discussion

Several observations can be made about the performance data presented above. Firstly, there is surprisingly little variation between the four sample programs. This shows that there is a fair amount of exploitable "shallow nondeterminism" over a wide range of applications. Secondly, the results do not correlate perfectly with the dynamic properties computed in Section 5.1, but we do note that optimising shallow backtracking yielded the least speedup for CHAT, which was expected. Lastly, we found that much of the deep backtracking in TP could easily be made shallow by rearranging code, unfolding tests, etc. By applying these changes we increased the scope of the shallow backtracking optimisation, yielding

a speedup factor of as much as 1.5 compared to the unchanged program under approach SB3.

The performance data is summarised in the following table of relative figures:

**Table 5: Performance summary.**

| | DB | SB1 | SB2 | SB3 |
|---|---|---|---|---|
| Compilation time: | 1.0 | ≈.97 | 1.07–1.18 | 1.02–1.12 |
| Code size: | 1.0 | ≈1.01 | 1.09–1.20 | 1.05–1.10 |
| Execution time: | 1.0 | .85–.96 | .85–.93 | .85–.92 |

As we can see from this table, strategies SB2 and SB3 offered the best runtime speedups (7% to 15%) for the sample programs. Strategy SB2 was not significantly faster for any of the programs, and as strategy SB3 is the more space economic of the two, it seems a sensible choice.

Strategy SB1 is attractive as it has very little compilation time and code size overhead. Although it did yield significant speedups for the sample programs, it can have a big run time overhead on examples like *append*, because register allocation is impaired in the deterministic case. In fact, strategy SB1 slowed down the well-known *naive reverse* benchmark, where *append* is the inner loop, by a factor of 1.29. This large factor is partly due to the way the SICStus compiler collapses certain WAM sequences into single byte codes: the *neck* instruction was inserted into such a sequence (see page 3), and the inner loop size grew from 4 to 7 byte codes. A native code implementation would probably not display this anomalous behaviour. In a native code implementation, strategy SB1 could be the best choice, especially since the need to keep the code size down is more urgent than in a bytecode implementation.

Meier reports only 3–4% slowdown of naive reverse.

## 6 Comparison with other work

Our implementation is much like Meier's. The main difference is that Meier introduces new WAM registers which are written by the *try* and *try_me_else* instructions, read when shallow backtracking occurs, and copied into a choicepoint by the *neck* instruction. No partial choicepoint is constructed. On hardware like the MC68020, however, the extra WAM registers would have to be stored in memory anyway because of shortage of machine registers. This was our motivation for building a partial choicepoint instead.

Meier's abstract machine always preserves the ***E*** register up to the first *call* instruction by maintaining the top of the environment stack in a register ***TE*** and using two sets of *unify* instructions: one for head unification which accesses permanent variables using ***TE***, and another for the clause body which uses ***E***. One set of *unify* instructions suffices for us.

The design of Van Roy et.al. uses up to four entrypoints per clause

whereas ours uses at most two entrypoints. The scope of the shallow backtracking optimisation is somewhat restricted in their design, as head unification for "nondeterministic entries" is not allowed to bind variables.

Tateno et.al. [4] describe a design similar to ours but restricted to predicates in which all clauses but the last one have a cut before the first general body goal. Their design allows head unifications to bind variables and handles the trail specially for such bindings.

To the author's knowledge, all published speedup figures for optimised shallow backtracking have been for toy programs. We ran the same toy programs to see if the results could be reproduced. In all cases, the published figures were rather better than our results. For example, Meier uses the *memberchk* predicate as an example where shallow backtracking is extremely advantageous, and reports a 65% performance improvement. Using strategy SB3, we measured a speedup factor of 1.34 over strategy DB:

```
memberchk(X, [X|_]) :- !.
memberchk(X, [_|Xs]) :- memberchk(X, Xs).
```

Van Roy et.al. report a speedup factor of as much as 2.6 for the *min_list* predicate, listed below. Again using SB3 we measured a speedup of 1.22 over DB:

```
min_list([X], X).
min_list([X|L], M) :-
         min_list(L, Y),
         minimum(X, Y, M).

minimum(A, B, A) :- A<B, !.
minimum(A, B, B).
```

## 7 Conclusions

The main contribution of this paper is to present actual measurements of the effectiveness of optimising shallow backtracking in non-trivial Prolog programs, running on a highly optimised Prolog system. The measurements show that this optimisation is worthwhile over a wide range of applications, yielding a speedup of 7%–15%. We expect the optimisation to be more important for or-parallel implementations than for sequential ones. We showed that program transformation can significantly increase the scope of the optimisation.

Three approaches to this optimisation were presented, two of which involve generating two code streams for each compiled clause. We showed that if only a single code stream is generated, the overhead on deterministic programs can be intolerable if executed by a bytecode emulator. We expect that in a native code implementation, with its more urgent need to keep the code size down, the overhead on deterministic programs may be tolerable.

A great deal of extra analysis can be done in a procedure compiler, at the cost of increased compilation time, since a procedure compiler knows precisely which clauses may be used nondeterministically and which may not. Mode declarations provide further information in this respect.

## Acknowledgements

The author is indebted to Carl Kesselman, Hiroshi Nakashima and the referees for their comments, which substantially improved the presentation.

This research would not have been possible without the support of my family.

## References

[1] M. Carlsson, *Freeze, Indexing and Other Implementation Issues in the WAM*, Proc. Fourth International Conference on Logic Programming, pp. 40–58, MIT Press, 1987.

[2] M. Carlsson, J. Widén, *SICStus Prolog User's Manual*, SICS Research Report R88007B, October, 1988.

[3] M. Meier, *Shallow Backtracking in Prolog Programs*, Internal report, ECRC, 1987.

[4] H. Tateno, H. Nakashima, S. Kondo, K. Nakajima, *Neck Cut Optimization: An Optimization technique for the Shallow Backtracking*, Internal report, Mitsubishi Electric Corporation and ICOT, 1989.

[5] E. Tick and D.H.D. Warren, *Towards a pipelined Prolog processor*, in Proc. International Symposium on Logic Programming, pp. 29–40, IEEE Computer Society, 1984.

[6] E. Tick, *Studies in Prolog Architectures*, Technical Report No. CSL-TR-87-329, Stanford University, June 1987.

[7] P. Van Roy, B. Demoen, and Y.D. Willems, *Improving the execution speed of compiled Prolog with modes, clause selection, and determinacy*, Proc. TAPSOFT'87: International Joint Conference on Theory and Practice of Software Development, pp. 111–125, Springer–Verlag, 1987.

[8] D.H.D. Warren, *IMPLEMENTING PROLOG—compiling predicate logic programs*, D.A.I. Research Report 39, University of Edinburgh, May, 1977.

[9] D.H.D. Warren, *An Abstract Prolog Instruction Set* , Technical Note 309, SRI International, 1983.

[10] E. Lusk, D.H.D. Warren, S. Haridi et.al., *The Aurora Or-Parallel Prolog System*, in Proc. International Conference on Fifth Generation Computer Systems, pp. 819–830, ICOT, Tokyo, 1988.