ICOT Technical Memorandum: TM-0699

TM-0699

Neck Cut Optimization: An Optimization Technique for the Shallow Backtracking

by
H. Tateno, H. Nakashima, S. Kondoh
(Mitsubishi) & K. nakajima

March, 1989

© 1989, ICOT

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

Neck Cut Optimization: An optimization Technique for the Shallow Backtracking

Hirokazu Tateno* Hiroshi Nakashima*
Seiich Kondoh* Katsuto Nakajima**

March 20, 1989

*: Mitubishi Electric Corporation
**: Institute for New Generation Computer Technology

Abstract

In many application programs of logic programming languages, shallow backtracking frequently occurs. The cost of shallow backtracking, however, is not so small in conventional implementations, because the choice points are manipulated in the same way as the deep backtracking. The clause indexing is powerful to reduce the cost of backtracking, but it is not always applicable.

So, we have exploited an optimization technique, Neck Cut Optimization, applicable to most of the *if-then-else* and *case-of* type predicates. It has been implemented on the PSI-II, a special purpose machine for the logic programming language, and improved the performance of application programs 10 to 17 %.

1 Introduction

Sequential inference machines PSI-I and PSI-II were developed as the tools of the Japanese fifth generation computer systems project [8], [4]. Many AI application programs, such as natural language recognition system, expert system , knowledge base management system , and operating system SIMPOS have been implemented on them [11].

In these programs, shallow backtracking frequently occurs, because *if-then-else*, *case-switch*, and *do-while* are mapped to the predicates containing control schemes in conventional language, such as multiple clauses. Since the choice points for these predicats are manipulated in the same way as deep backtracking, the cost of those simple control schemes is not small. Moreover, clause indexing are not always applicable to them, because their clauses are often sellected by built-in predicates [10].

This paper describes an optimization technique, Neck Cut Optimization, applicable to most of predicates selecting a clause by shallow backtracking. It greatly reduces the cost of choice point creation, backtracking, and cut operations.

This paper is organized as follows: Section 2 describes the mechanism and overhead of shallow backtracking in the conventional implementation ; section 3 describes Neck Cut Optimization in detail ; section 4 evaluates the performance improvement; section 5 discusses the language features and other works relating to the optimization technique; and section 6 gives conclusions.

2 Backtracking

This section briefly describes the backtracking mechanism in the conventional WAM, in order to point out the overhead of shallow backtracking.

2.1 Backtracking Mechanism of WAM

In the WAM, the choice point is manipulated for the backtracking. When a predicate is called and it has multiple candidate clauses, the `try(_me_else)` instruction

creates the choice point containing the following execution environment.

An: the argument registers

E: the current environment pointer

CP: the current continuation

B: the the pointer to previous choice point

BP: the address of the next alternative clause

TR: the current trail pointer

H: the current heap pointer

The instruction pushes the choice point onto `stack`, updates the choice point pointer `B`, and sets heap backtrack point `HB` to the current heap pointer for the trailing.

The choice point is referenced by the *fail* procedure, called when a failure occurs during the unification or built-in predicate execution. The procedure restores the registers (excepting `B`) to the values in the choice point. `Trail` is unwound as far as the trail pointer in the choice point, resetting the variables to unbound. Control is transferred to the next alternative clause, the first instruction of which is `retry(_me_else)` or `trust(_me_else)`.

The `retry(_me_else)` instruction updates the choice point entry with the address of the next alternative clause. The `trust(_me_else)` discards the choice point updating `B` and `HB` with the values for the previous choice point. The *cut* operation also discards the choice points created by the current and descendant predicates. It *tidies* `trail`, discarding entries corresponding to the discarded choice points in the most of systems, in order to minimize the growth of tit trail and avoid to leave dangling pointers.

2.2 Overhead of Shallow Backtracking

Some operations manipulating the choice point are redundant in the case of *shallow* backtracking, caused by the failure of the head unification. Figure 1 shows

the predicate *member/2* and its compiled codes. In its execution, its second clause is selected until the list element matched to the second argument is found. Since clause indexing can not be applied to it, the first clause is tried and backtracking occurs.

```
        member([E|_],E):-!.
        member([_|L],E):- member(L,E).

member/2:
    try_me_else         C2
    get_list            A1
    unify_local_value   A2
    unify_void          1
    cut
    proceed
C2:
    trust_me_else       fail
    get_list            A1
    unify_void          1
    unify_variable      A1
    execute             member/2
```

Figure 1: member/2

When the head unification of the first clause fails, the *fail* procedure restores registers. This operation, however, is almost redundant because the value of the all registers, but TR, are the same as those of the choice point. Moreover, the choice point itself is redundant, because it is discarded by the `trust_me_else` instruction for the second clause.

On the other hand, if the head unification of the first clause succeeds, the choice point is discarded by the *cut* operation. Thus, all the contents of the choice point are redundant, but those of TR and BP, whether or not the unification succeeds.

Shallow backtracking is caused by not only the failure of the head unification but also that of the built-in predicate. Figure 2 shows the predicate *split/4* of the quick-sort program and its compiled codes. Assuming that the goal X<Y is compiled to the abstract instruction `less_than`, shallow backtracking will occur if the car of the first argument is not less than the second argument.

In this case, restoring registers is not totally redundant, because;

- An the argument registers A1 and A3 is updated by the optimized register allocation.
- H is incremented to generate a new list cell.
- TR is incremented to trail the binding of the third argument.

For the other registers, however, the contents of the choice point are redundant too. Moreover, if the compiler abandons the optimal register allocation and generates two put_value instruction for L1 and L2, save and restore operations of the argument registers become unnecessary.

3 Neck Cut Optimization

3.1 Predicates to be optimized

In the execution of the predicates in the previous section, their choice points are eliminated by the *cut* or trust(_me_else), before the first (non built-in) goal is called. Thus, the size of the choice points can be reduced, because the values of most of the registers are not changed until trusting.

The *Neck Cut Optimization* is applied to those predicates. That is, it is applicable if the following conditions are satisfied.

1. All clauses, but the last one, of the predicate have the *cut*.
2. The *cut* performed at *neck*, that is, before the first goal is called. *Simple* predicate call, however, is allowed before the *cut*.

The *simple* predicates will be built-in predicates, such as for the type checking, comparison, arithmetic operation, and so on. Those built-in predicates should not change the execution environment other than that of the unification. For example, most of built-in predicates of the PSI-II are classified as *simple*.

Any unifications are allowed in the criteria given above. If the output unification is not allowed, the choice point manipulation is simplified further. The content of the

choice point, in fact, becomes only the address of the next alternative clause. It seems that the restriction is too severe and few predicates have the chance to be optimized. However, if the mode information is given, the predicates which satisfy the criterion will increase considerably(discussed later). Even if not, the optimization is applicable to the typical cases of *inner clause or*. Assume the *split/4* is rewritten as figure 3(a), and the compiler converts it into two predicates shown in figure 3(b). The predicate *split_2/5* satisfies the criterion.

Those criteria are examined at compile time. So, the compiler generates special codes, described later.

3.2 Minimizing Choice Point

The *Neck Cut Optimization* reduces the size of choice points. The following registers are not saved into the choice point.

An: The compiler allocates argument registers avoiding the destruction until the neck cut. Note that overwriting of the dereference result is not allowed.

E: The environment allocation is delayed after the neck cut.

CP: No goals are not called.

B: Choice point is never nested. The necessity of the trailing is checked in the other way than that of the usual cases (described the next sub-section). `HB` is also left unmodified.

The restriction of the argument register allocation will degrade the performance. However, it is expected that the degradation will not be serious, because the optimal allocation is allowed for the last clause, which is often the *recursive* clause.

Thus, the reduced choice point consists only three entries for `BP`, `TR`, and `H`. The entries for `TR` and `H` is reduced if the output unification are not performed.

Since the choice point is small and never nested, it may be allocated on the special registers insted of `stack`.

3.3 Trailing

Since the criteria of the optimization allow any unifications in head (and by the *simple* predicates), the trailing is necessary for the output unification. The trailing mechanism, however, can be optimized in two ways, one of which is suitable for the implementations on general purpose machines, and another for special purpose machines.

The optimization method for general purpose machines stands on the fact that *all* bindings to the unbound variables should be trailed in the clause which cannot be deterministic. (Strictly speaking, the binding to the element of the compound term created in the head is not necessary, but allowed, to be trailed.) That is, for the head unifications in these clauses, the compiler can generates special codes from which the trail checking is removed. Moreover, the checking can be romoved from the head unifications of all clauses but the last one, because cut will tidy redundant addresses trailed by the posibbly determinate clauses. This technique also make it possible to leave `B` and `HB` unchanged when the reduced choice point is generated and removed.

Another method optimizes the tidying of `trail`. This method requires a (small) stack buffer beside the main memory, to which the address of the unbound variable is *always* pushed when the variable is instantiated. When the reduced choice point is created, the pointer of the buffer is cleared, and `B` and `HB` are left unchanged. The unifier trails the variable address in the usual way, that is, comparing it with `B` or `HB` which point the *previous* backtrack point. Thus, the buffer contains addresses of the variables to be unbound by the fail procedure referencing the reducced choice point. And `trail` only contains the addresses which is necessary after the neck cut, which will not perform any operations for tidying. It is easy to implement the buffer without any overheads, because the address will be pushed during the value is written in the memory. The overflow check mechanism for the buffer can be removed, if it is allowed to change the choice point type dynamically. The compiler can count up the maximum number of the variable binding performed until the neck cut, assuming the general unifier binds some constant number variables (one should be enough). If the count is larger than the

size of the buffer, ten or so, the compiler gives up the optimization. And the general unifier, `get_value`, make ordinary choice point, if it binds multiple variables, or, more roughly, both arguments are compound terms. According to the statistics reported in [6,7], giving up the optimization, either at compile and run time, will be very rare.

3.4 Implementation

Since the optimization causes various types of choice point, it is required for some instructions and procedures to identify the referencing choice point type. The *choice point mode*, containing one of the following values, is introduced for the identification.

- *normal*

  The current choice point is the same as ordinary implementations.

- *fast*

  The current choice point contains `BP`, `TR`, and `H`.

- *very-fast*

  The current choice point contains `BP` only.

In order to set the mode, the ordinary `try(_me_else)` instruction is replaced with one of the following new instructions, according to the predicate type.

- **`normal_try(_me_else)`**

  Same as ordinary `try(_me_else)`, except for setting the mode to *normal*.

- **`fast_try(_me_else)`**

  Save `TR` and `H` to the special registers, named `BTR` and `BH`. Set the special register, named `AP`, to the address of the next clause. Set the mode to *fast*.

- **`very_fast_try(_me_else)`**

  Set `AP` to the address of the next clause, and set the mode to *very-fast*.

The fail procedure is modified for the mode dependent execution.

- `fail`

  In normal mode, same as ordinary one. In *fast* mode, unbind variables according to the trailing method, restore `TR` and `H` to the contents of `BTR` and `BH`, and jump to the content of `AP`. In *very-fast* mode, jump to the content of `AP`.

There are two methods to implement the other instructions referencing the choice point, `retry(_me_else)`, `trust(_me_else)` and `cut`. The one method is to introduce the following instructions for the *fast* mode.

- `fast_retry(_me_else)`

  Update `AP` with the address of the next clause.

- `fast_trust(_me_else)`

  Set the mode to *normal*.

- `fast_neck_cut`

  Set the mode to *normal*, and tidy `trail` if the trailing method requires it.

The instructions `fast_retry(_me_else)` and `fast_trust(_me_else)` are also used for the *very-fast* mode. The neck cut for the *very-fast* mode is translated to `fast_trust_me_else`. These instructions are fairly simple, because they are free from mode-dependent operations.

However, it is not allowed to change the mode dynamically.

The other method will be used, if the dynamic mode change is required for the trailing optimization using the small stack buffer, or the dynamic predicate call such as *freeze* in Prolog-II (discussed later). In this method, ordinary `retry(_me_else)`, `trust(_me_else)` and `cut` are modified for the mode-dependent execution. That is, they perform ordinary operations in *normal* mode, or act as `fast_xxx` in *fast* or *very-fast* mode.

Figure 4,5,6 show the optimized compiled codes for *member/2*, *split/4* and *split_2/5*, according to the former method.

4 Performance Evaluation

The neck cut optimization has been implemented on PSI-II. The implementation is slightly different from the above description in the following points.

1. The trailing is not optimized.
2. The *very-fast* mode is implemented according to [5]. In this method, the application condition is more restricted.

And the instructions `retry_(me_else)`, `trust_(me_else)` and `cut` perform mode-dependent execution.

Table 1 shows the statistics of the type of choice points and the performance improvement, in the following four simple benchmarks and three real application programs.

`member1000`

Member/4 shown in figure 1, searching a 1000-element list, whose last element matched to the second argument.

`qsort50`

Quick sort program using split/4 shown in figure 2. The sorted data are the same as those of [1].

`qsort50'`

Quick sort program using split/4 shown in figure 3.

`8queen`

Eight queens program in [1].

`compiler`

The compiler for PSI-II, compiling `qsort`

`assembler`

The microprogram assembler for PSI-II.

SIMPOS

Basic operations, including window and file manipulation, in SIMPOS.

The second to forth columns of the table show the ratios of choice point types. The last column is the performance improvement ratio based on the measured execution time with and without the optimization.

The evaluation results show that the many choice points are reducible. The ratios of the reduction are similar to the results in [12], in which the choice point type is determined at run time. This shows that the application criteria of the optimization cover the most of predicates which cause shallow backtracking.

The ratio of the *very-fast* mode choice points seems to be surprisingly high in real application programs. The result may be overestimated a little, because these programs are written by professional programmers who know what codes are generated. However, the ratio of *avoidable choice points* in [9], which are classified as *very-fast* mode at run time, is also considerably high.

The performance improvement, 9 to 17 %, shows that the optimization is effective and worthwhile for real application programs.

Table 1: performance evaluation of the programs

programs	normal mode	fast mode	very-fast mode	improvement ratio
member	0 (%)	100 (%)	0 (%)	21 (%)
qsort50	0	100	0	15
qsort50'	0	0	100	34
8queen	12	88	0	22
compiler	20	20	60	17
assembler	56	21	23	9
SIMPOS	35	20	35	10 x

5 Expansion of Neck Cut Optimization

5.1 Related Works

In the implementation of PSI-I, which is not a WAM, an optimization method for shallow backtracking are adopted [4]. In this method, a stack frame similar to the choice point is created when the first goal of the clause, selected by shallow backtracking, are called. Similar methods for WAM is proposed in [10] and [6], and the actual implementation and performance evaluation is described in [12]. The main difference between those methods and ours is the timing of the choice point classification. In their methods, the `neck` instruction is inserted at the end of the head unification, and classifies the choice point at run time, while we classify it at compile time. That is, `try(_me_else)` create the choice point partially, and `neck` completes it if it should be *normal*.

This dynamic classification make it possible to optimize any predicates regardless of the type. The restriction to the neck cut predicates in ours, however, is not so severe, as described in the section 4. The static classification will be advantageous, because it removes the type checking from run time operations. If the dynamic mode change is allowed as the implementation on PSI-II, however, this advantage will relatively decrease.

Other differences are the optimization of the trailing and *very-fast* mode, which will be applicable to their implementations. The argument register allocation in [12] is also different from ours, because the destruction of the argument registers is not allowed even in the last clause. In order to reduce the overhead caused by the non-optimal allocation and `neck`, two code streams, according to the deterministic and non-deterministic cases, are generated.

The other way to reduce the overhead will be to allow the destruction to the *possibly* deterministic clauses, which is the target clause of the `switch` instructions. The `neck` instruction (or `make_normal` in our implementation) should be inserted just after `retry_me_else`, in order to create the *normal* choice point when the clause is selected

by backtracking. This technique will be effective if the first argument (or the others used for indexing) is not unbound in most cases.

5.2 Dynamic Predicate Call

Some languages have the dynamic predicate call mechanism, such as *freeze* in Prolog-II and *exception* in KL0 [3]. These mechanisms bring a troublesome problem to the optimizing compilation, because predicates, hooked to the variable binding or the exceptional cases of built-ins, may be called before the neck cut.

This problem is solved by changing the choice point mode dynamically. That is, when the unifier or built-in predicates detect the condition to call the hooked predicate and the mode is not *normal*, they create the *normal* choice point and change the mode as `get_value` in section 3.3. This mechanism is implemented on PSI-II, and requires the mode-dependent execution to `retry(_me_else)` etc.

5.3 Mode Information

If the mode information is available, the chance to apply the *very-fast* mode optimization will increase considerably. For example, if the *split/4* shown in figure 2 have the mode declaration, `(+,+,-,-)`, the output unifications in the first and second clauses can be postponed after the neck cut (if the *freeze* mechanism is not adopted). Thus, there are no output unifications before the cut, because the first argument should be instantiated.

The mode *declaration*, however, does not always provide sufficient mode *information*. For example, *member/4* shown in figure 1 cannot optimized even if `(+,+)` is declared, because the first argument may not be a ground term, and its car may be an unbound variable. Moreover, the declaration will often be harmful, because incorrect declarations change the semantics of the program and make the debugging hard. So, it is required for the optimization to get the correct and sufficient mode *information*, by the practical mode inference system.

6 Conclusion

In this paper, an optimization technique for shallow backtracking are described. The performance evaluation, based on the actual implementation on PSI-II, is also presented and shows the following.

- A wide range of predicates in real application programs satisfy the criteria of the optimization.
- The amount of the predicates which satisfy the more restricted criteria, the inhibition of the output unification, is considerably large.
- The optimization improves the performance of real application programs, 9 to 17 %.

We are now modifying our implementation, adopting the optimization of the trailing, relieved criteria of the *very-fast* mode, and other ideas proposed in this paper.

7 Acknowledgments

We would like thank to Mats Carlsson, Herve' Touati and Ralph Butler for their helpful comments on our work. They also kindly gave us the information about their works.

References

[1] D.H.D. Warren, Applied Logic - Its Use and Implementation as Programming Tool, PhD thesis, University of Edinburgh, 1977.

[2] D.H.D. Warren, An Abstract Prolog Instruction Set, Technical Note 309, SRI.

[3] T. Chikayama, et al., Fifth Generation Kernel Language, in Proc. of the Logic Programming Conf., 1983

[4] M. Yokota, et al., A Microprogrammed Interpreter for the Personal Sequential Machine, in Proc. of FGCS'84.

[5] Takagi, et al., If-Then-Else Optimization, ICOT TM-104.(in Japanese)

[6] M. Meier, Shallow Backtracking in Prolog Programs, ECRC Internal Report IR-LP-1113, 1986.

[7] R. Onai, et al., Analysis of Sequential Prolog Programs, in Journal of Logic Programming, No.2.

[8] H. Nakashima, et al., Hardware Architecture of the Sequential Inference Machine : PSI-II, in Proc. of SLP'87.

[9] H. Touati, An Empirical Study of the Warren Abstract Machine, in Proc of SLP'87.

[10] E. Tick, Studies in Prolog Architectures, Technical Report CSL-TR-87- 329, Stanford University, 1987.

[11] T. Kurozumi, et al., Present Status and Plans for Research and Development, in Proc. of FGCS'88.

[12] M. Carlsson, On the Efficiency of Optimising Shallow Backtracking in Compiled Prolog, will appear in Proc. of ICLP'89, 1989.

```
        split([],_,[],[]):-!.
        split([X|L1],Y,[X|L2],L3):-
            X < Y, !, split(L1,Y,L2,L3).
        split([X|L1],Y,L2,[X|L3]):- split(L1,Y,L2,L3).

split/4:
        switch_on_term  C1a,C1,L1,fail
L1:
        try             C2
        trust           C3
C1a:
        try_me_else     C2a
C1:
        get_nil         A1
        get_nil         A3
        get_nil         A4
        cut
        proceed
C2a:
        retry_me_else   C3a
C2:
        get_list        A1
        unify_variable  X5
        unify_variable  A1
        get_list        A3
        unify_value     X5
        unify_variable  A3
        less_than       X5,A2
        cut
        execute         split/4
C3a:
        trust_me_else fail
C3:
        get_list        A1
        unify_variable  X5
        unify_variable  A1
        get_list        A4
        unify_value     X5
        unify_variable  A4
        execute         split/4
```

Figure 2: split/4

```
split([],_,[],[]):-!.
split([X|L1],Y,L2,L3):- ( X <Y ==>
                L2 = [X|L2a], split(L1,Y.L2a,L3) ;
                L3 = [X|L3a], split(L1,Y.L2,L3a) )
                    (a)

split_1([],_,[],[]):-!.
split_1([X|L1],Y,L2,L3):- split_2(L1,Y,L2,L3,X).
split_2(L1,Y,L2,L3,X):- X <Y,!,
                L2 = [X|L2a], split_1(L1,Y,L2a,L3).
split_2(L1,Y,L2,L3,X):-
                L3 = [X|L3a], split_1(L1,Y,L2,L3a).
                    (b)
```

Figure 3: split/4 using inner clause or

```
member/2 :
        fast_try_me_else        C2
        get_list                A1
        unify_value             A2
        unify_void              1
        fast_neck_cut
        proceed
C2:
        fast_trust_me_else      fail
        get_list                A1
        unify_void              1
        unify_variable          A1
        execute                 member/2
```

Figure 4: Optimized Compiled Codes member/2

```
split/4 :
        switch_on_term  C1a,C1,L1,fail
L1:
        fast_try                C2
        fast_trust              C3
C1a:
        fast_try_me_else        C2a
C1:
        get_nil                 A1
        get_nil                 A3
        get_nil                 A4
        fast_neck_cut
        proceed
C2a:
        fast_retry_me_else      C3a
C2:
        get_list                A1
        unify_variable          X5
        unify_variable          X6
        get_list                A3
        unify_value             X5
        unify_variable          X7
        less_than               X5,A2
        fast_neck_cut
        put_value               X6,A1
        put_value               X7,A3
        execute                 split/4
C3a:
        fast_trust_me_else      fail
C3:
        get_list                A1
        unify_variable          X5
        unify_variable          A1
        get_list                A4
        unify_value             X5
        unify_variable          A4
        execute                 split/4
```

Figure 5: Optimized Compiled Codes split/4

```
split_2/5:
        very_fast_try_me_else   C2
        less_than               A5,A2
        fast_trust_me_else      fail        ; for neck cut
        get_list                A3
        unify_value             A5
        unify_variable          A3
        execute                 split_1/4
C2:
        fast_trust_me_else      fail
        get_list                A4
        unify_value             A5
        unify_variable          A4
        execute                 split_1/4
```

Figure 6: Optimized Compiled Codes split_2/5