A Method for Implementing Cut in Parallel Execution of Prolog

by

Khayri A M Ali

SICS R87001
Research Report
ISSN 0283-3638

A Method for Implementing Cut in Parallel Execution of Prolog

Khayri A M Ali
Logic Programming Systems
SICS
P.O. Box 1263
S-163 13 Spånga, Sweden

Feb. 1987

ABSTRACT

A method for implementing cut in parallel execution of Prolog is presented. It takes advantages of the efficient implementation of cut in the sequential WAM. It restricts the parallelism, however, it is simple and adds a small extra overhead over the sequential scheme. The method can be used in parallel execution of Prolog on shared and nonshared memory multiprocessors.

1. Introduction

Most practical Prolog programs contain cuts. The cut is often used to increase the efficiency of programs and to prevent the consideration of alternate solutions. The most common semantics of cut (known as the *asymmetric cut*) is as follows.

The cut operation commits all choices made since the predicate in which the cut occurs was invoked, and causes other alternatives to be discarded.

A discussion on different operational semantics for cuts and different implementation approaches used on the sequential WAM [Warr83] is found in [Carl87].

In the sequential WAM, only one processor works on all alternative clauses (branches). The branches are selected from left-to-right in depth-first manner. The cut in the leftmost branch of a choice point (CP) is always encountered before the cuts in the right branches of the same CP. So, it is simple and efficient to find and to discard the right branches.

In Or-parallel execution of Prolog, alternative branches with cut of the same predicate can be processed by different processors. In this case, it is possible for the processor working on a non-leftmost branch with cut to encounter cut before the processor working on the leftmost branch with cut. The system should eventually cut away all branches right to the leftmost branch with cut.

A parallel implementation of cut based on the above approach allows high parallelism. However, it requires a mechanism for keeping track of processors working on different branches in order to stop those processors working on the branches to the right of the leftmost branch with cut. Such a mechanism is expected to be much more expensive in comparison with the one used in the sequential implementation. The other disadvantage of the above approach is that processors can be utilised badly; unneeded work may be assigned to idle processors instead of the needed work.

In this paper we present a scheme for parallel implementation of cut taking advantage of the efficient sequential implementation of cut in WAM [Carl87]. Our scheme restricts the parallelism. However, it is simple and efficient - it has a small extra overhead over the sequential scheme. It also avoids bad utilisation of processors. We assume herein the reader is familiar with WAM [Warr83] and the implementation of cut in WAM [Moss86, Carl87].

The structure of this paper is as follows. The next section introduces our idea of a parallel implementation of the asymmetric cut. Section 3 introduces our scheme. Section 4 gives an example of compiling predicates with cut, and discusses the problems of creating two CP frames for one predicate and the solutions. Section 5 discusses possible optimizations.

2. Basic Idea

Suppose that the clauses in a given predicate p contain at least one cut. These clauses are partitioned according to the textual order into groups G1, G2, ... Gn according to the following rule [Ali86]:

"each group is either a set of clauses in which none of the clauses contains cut, or a set of clauses in which each clause contains at least one cut, where no two contiguous groups may form one group."

Let us first consider the case in which G1, G3, ... have the clauses which contain cuts, and G2, G4, ... have the clauses which do not contain cut.

i) When a call is made to p, the clauses of p will be executed sequentially until either the last cut in any clause of G1 is executed, or the first clause of G2 is about to be

executed.

ii) In the latter case of i), the clauses of G2 can be executed in parallel while the remaining clauses (in G3, G4, ... Gn) will be executed sequentially until either the last cut in any clause of G3 is executed, or the first clause of G4 is about to be executed, and so on.

In the other case G1, G3, ... have clauses containing no cut. When a call is made to p, we get a situation similar to ii) where G2 is replaced by G1, G3 by G2, and G4 by G3. That is, the clauses of G1 can be executed in parallel while clauses in G2, G3, ... will be executed sequentially until either the last cut in any clause of G2 is executed or the first clause of G3 is about to be executed.

That is, a subtree may be processed in sequential mode or in parallel mode depending on if its current leftmost branch contains cut or not respectively.

Example 1:

Assume a predicate p in the following structure:

```
(1)     p :- q.
(2)     p :- q1.
(3)     p :- q2.
(4)     p :- q3, !, r, !, s.
(5)     p :- q4.
(6)     p :- !, q5, v.
(7)     p :- q6, !, u.
```

P is partitioned into 4 clause groups:

G1 has clauses (1), (2) and (3),
G2 has (4),
G3 has (5), and
G4 has (6) and (7).

When p is invoked, the three clauses (1), (2) and (3) can be processed in parallel, and the remaining clauses (4) - (7) will be executed sequentially until either (a) passing through the last cut in clause (4), or (b) failing before passing the first cut in clause (4). That is, four processors can work in parallel: one for each of the first three clauses and the fourth processor for the remaining clauses (see Figure 1).

In case (a), if s has the following predicate:

```
(8)     s :- r.
(9)     s :- r1.
(10)    s :- !, r2.
(11)    s :- r3, !, r4, !, r5.
```

The clauses (8) and (9) can be processed in parallel, and clauses (10) and (11) will be executed sequentially but in parallel with (8) and (9).

In case (b), clauses (6) and (7) will be processed sequentially but in parallel with clause (5).

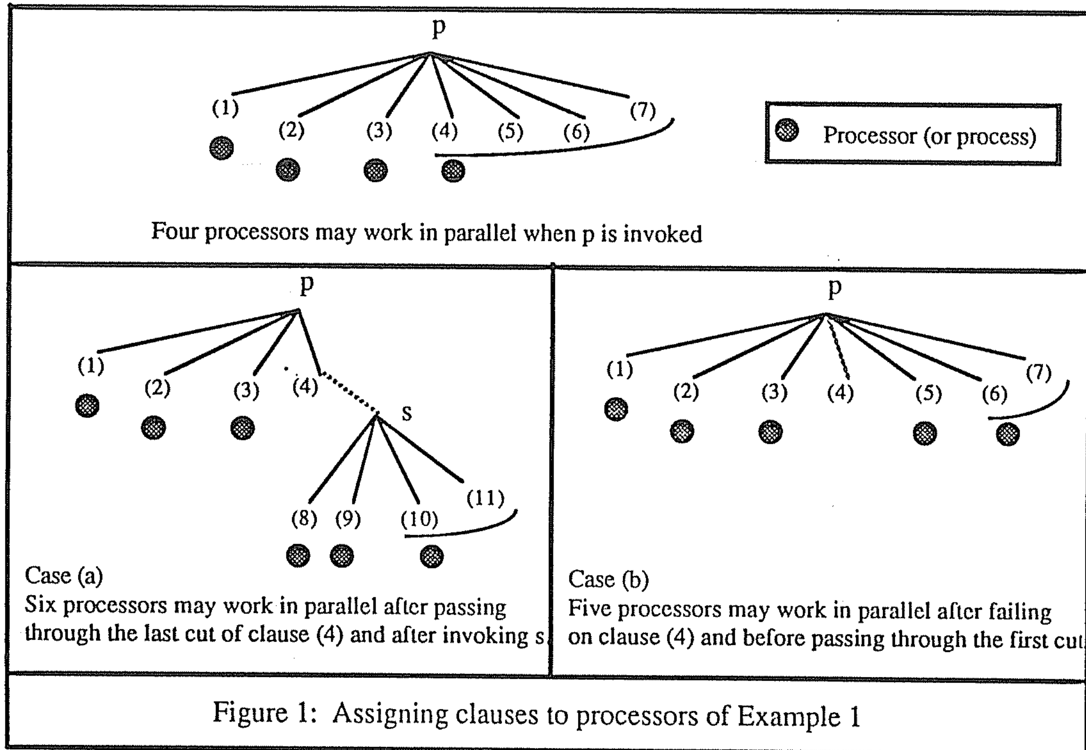

Figure 1: Assigning clauses to processors of Example 1

Processor modes and switching rules:

Let us assume a Prolog program is executed by a system having a number of processors, with shared or nonshared memory. Each of these processors is either in the sequential mode or in the parallel mode. Initially, all processors are in the parallel mode. Processors switch from one mode to the other according to the following rules:

- A processor switches from the parallel mode to the sequential mode when it starts processing the first clause of a clause group G containing cut.

- A processor switches from the sequential mode to the parallel mode when either the last cut in one clause of the group G is encountered, or all clauses of the group G fail.

We notice that a processor may have a number of CPs just before switching to the sequential mode. Processors, in both modes, can split up untried choices of CPs and assign them to other processors according to the following rules:

- A processor in the sequential mode can split up all untried choices it had before the predicate with G was invoked.

- A processor in the parallel mode can split up all untried choices it has.

- When a CP having an untried branch with cut splits up, the leftmost branch with cut along with the branches to the right are assigned to one processor.

In the next section we give a mechanism that allows each processor to switch from one mode to the other according to the above rules and to identify alternative choices that can be split up.

3. A Parallel Scheme

Since our goal is to take advantage of the efficient sequential implementation of cut in WAM, we investigate, in Section 3.1, one of the sequential schemes and discuss why it is not enough for supporting a parallel implementation of cut presented in Section 2. Then, we discuss possible modifications to the sequential scheme. In Section 3.2, we introduce our scheme.

3.1. Preliminaries

We take a scheme presented by Carlsson [Carl87], which uses a separate stack for holding the CP frames. (The idea of this scheme was originally proposed by Venken [Venk84].) In this scheme, an instruction

choice A_n: $A_n := B$;
where A_n is an argument register and B is a WAM register pointing to the last CP frame,

begins the code for each predicate using cut. An instruction

cut V_n: $B := V_n$; *tidy trail;*
where the argument Vn contains the saved value of B,

is used for each occurrence of cut.

This mechanism is not enough for parallel implementation of cut for the following reasons:

1. It does not distinguish between the last cut and a non-last cut in a clause.

2. Since no CP frame is created for a single clause predicate, the sequential scheme can not detect the last cut in the first clause using cut. To illustrate this problem let us take an example with nested cuts. Assume the B register has the value B' when p(a, b) is invoked in the following code segment, and the current processor is in the parallel mode:

```
        ... , p(a, b), ...

p(X, Y) :- s(X), !, q(Y).
s(X) :- r(X), !, q1(X).
r(X) :- v(X), !, q2(X).

q1(a). q2(a).   v(a).
```

The current processor switches to the sequential mode at the beginning of execution of the clause p/2. The processor should switch back to the parallel mode when cut in the clause p/2 is encountered. Since no CP frame is created for any of the above clauses, B = B' at the beginning execution of any of the first three clauses and when any of the three cuts is encountered. How does the system distinguish between the last cut in the clause p/2, s/1, and r/1? So, the sequential cut mechanism should be modified in order to detect the last cut in the first clause (in this example, cut in the clause p/2).

One solution is to create a CP frame for each predicate using cut even for a single clause predicate, and to add a new instruction for last cut. When the first clause using cut is about to be processed, the respective CP frame is marked and an extra flag set ON indicating that the processor is in the sequential mode. Each last cut removes its respective CP frame and the recent ones. When the marked CP is removed, the processor switches to the parallel mode by resetting the flag OFF. In this solution, when the last clause of a predicate is selected and that clause uses cut, the respective CP frame can not be removed before passing through the last cut in the clause or failing on that clause.

The disadvantage of this solution is that the extra CP frames require large space and processing time overhead.

Another possible solution for detecting the last cut in the first clause is as follows. Assume that each clause invocation creates an environment frame (ENV frame). When the first clause with cut is processed, the respective ENV frame is marked and the current processor switches to the sequential mode. On processing a last cut, the current ENV frame is tested. If it is marked, so it corresponds to the first clause with cut and the processor switches to the parallel mode. If it is not marked, the current last cut does not correspond to the first clause with cut.

The main disadvantage of the second solution is that in the efficient implementation of WAM, an ENV frame is created only when there are at least two goals in the clause body. So, it is not efficient to create an ENV frame for clauses that have one or no goal in the body.

3.2. The Scheme

In our scheme herein, the CP-stack (and of course the trail stack) is used as in the first solution discussed in Section 3.1. But, **only one word**, called mark (M) frame, is created in the CP-stack for each predicate using cut. When an M frame is created, a pointer to the last M frame is saved in it. Also, when a CP frame is created, a pointer to the last CP frame is saved in it. That is, in the CP-stack there are two chains of different frames: one for M frames and the other for CP frames.

An M frame is removed from the CP-stack when either all clauses of the predicate are processed, or when the last cut in a clause is encountered. CP frames are created and removed with the same sequential mechanism. We distinguish between CPs for predicates using cut and for not using cut by marking those corresponding to the former ones. This distinction is necessary to recognize two different situations that generate an M frame and then a CP frame. (The first situation occurs, when a single clause predicate with cut invokes a multi-clause predicate with no cut. The second situation corresponds to a multi-clause predicate using cut.)

We assume that each CP-stack has three registers:

- B: points to the last CP frame (the same as in WAM),
- T: points to the last M frame, and
- S: points to the M frame corresponding to the first clause using cut.

Initially, B := S := T := Bottom of the CP-stack (called <u>Low</u>). The S register is also used to indicate the current execution mode: when S > <u>Low</u>, the current processor is in the sequential mode, otherwise in the parallel mode. Figure 2 shows a snapshot of the CP-stack and the associated registers.

We assume also that the compiler generates an instruction called *mark* that begins code of every clause with cut. When a mark instruction is executed and the current processor is in the parallel mode, the processor switches to the sequential mode by copying T into S. It is always true that when a mark instruction is executed T > <u>Low</u>. This is because (1) on each invocation of a predicate using cut, an M frame is created and T points to it, and (2) the respective mark instructions are executed after creation of the M frame.

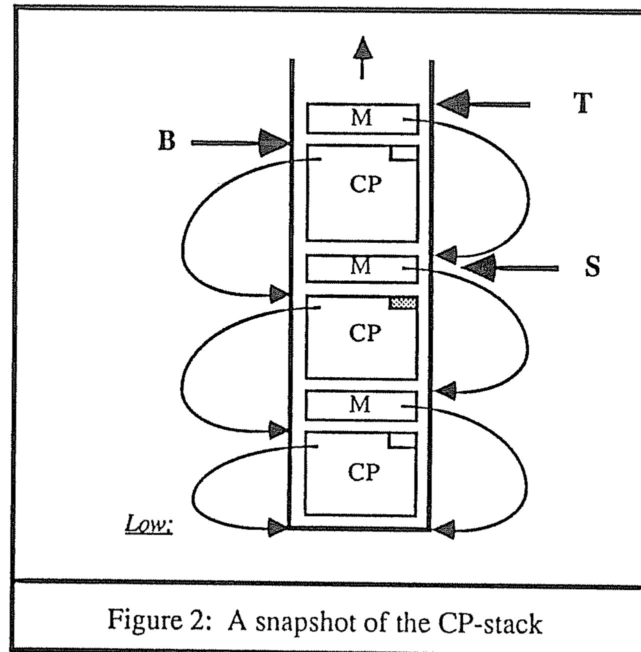

Figure 2: A snapshot of the CP-stack

Now, we can introduce our scheme as follows.

(a) Initially:
B, T and S, point to the bottom of the CP-stack i.e.,

B := T := S := <u>Low</u>

(b) On invoking a predicate using cut:
For each invocation of a predicate using cut, an M frame is created in the CP-stack. We modify the semantics of choice instruction as follows. It first creates an M frame on the CP-stack and saves the register T in it. Then, a pointer to the created M frame (i.e. the new value of register T) is copied into the argument register A_n. That is, the new semantics of choice instruction is as follows.

choice A_n:
T := create an M frame on the top of CP-stack and save in it the register T
A_n := T

(c) On invoking a clause with cut:
An instruction *mark* begins the code of every clause with cut. This instruction switches the current processor to the sequential mode if it is not already in this mode. This is done by copying T into S only if S = <u>Low</u>

(As mentioned above, it is always true that, *"there is a choice instruction which is executed before a mark instruction"*. Therefore, when a mark instruction is executed, T must be greater than <u>Low</u>.) The semantics of mark instruction is as follows.

mark:
If S = <u>Low</u> Then S := T

(d) On creation of a new CP frame:

In the standard implementation of Prolog, the *try label* instruction creates a new CP frame in the CP-stack, saves the current state in it and executes the code at address *label.* In our implementation, we use the try instruction with the above semantics for predicates with no cut. A new instruction called *mtry* is used for predicates with cut. The semantics of the mtry instruction is as follows. It first tests the processor mode. If the processor is in the parallel mode (i.e. S = <u>Low</u>) and the first alternative clause uses cut (mtry + mark), the processor switches to the sequential mode by copying T into S. Otherwise, the semantics of the mtry is the same as in WAM but the created CP frame is marked. In the SICStus implementation, there is no need to save the current B in the created CP frame, but in our implementation we need that.

The try instruction is defined by

```
try label:
   B := create a CP frame on the top of CP-stack and save in it the current state including B,
   process the first alternative clause of the created CP frame,
```

and the mtry instruction is defined by

```
mtry label:
   If S = low
   Then
         If *label = mark                         {*x means contents of memory address x}
         Then S := T
   B := create and mark a CP frame on the top of CP-stack and save in it the current state including B,
   process the first alternative clause of the created CP frame.
```

(e) Non-last cut:

Each *cut* operator, which does not correspond to a last cut in a clause, will be executed as follows. First it works the same as in the sequential WAM with restoring T instead of B. Second all CP frames created after invoking the predicate in which the cut occurs are discarded. That is,

```
cut Vn:
         T := Vn
         tidy trail
         If T < B
         Then B := the first CP frame below T
```

(f) Last cut:

We assume the compiler generates an instruction, called *lastcut,* which corresponds to the last cut in a clause. When this instruction is executed, it first works as in (e), then it tests if T and S point to the same M frame. If so, the current processor leaves the sequential mode by assigning <u>Low</u> to S. Last it removes the current M frame. The semantics of lastcut is as follows.

```
lastcut Vn:
         The same as in (e).
         If S = T Then S := Low
         T := *T
```

(g) On failure:

If there are no more alternative clauses, the current processor terminates and looks for a new job from another processor. If there is an alternative clause (i.e. B > <u>Low</u>), the following actions will be performed:

- All M frames more recent than the current CP will be removed.
- If the processor is in the parallel mode (i.e. S = Low) and the next alternative clause uses cut, the processor switches to the sequential mode (by copying T into S) and the next alternative is taken.
- If the processor is in the sequential mode and the new T > S, the next alternative is taken. (Whatever the next branch is the processor remains in the sequential mode.)
- If the processor is in the sequential mode and S > T and the next alternative clause uses cut, the next alternative is taken and T is copied into S. That is, some M frames are removed from the CP-stack.
- If the processor is in the sequential mode and S > T and the next alternative clause does not use cut, the processor switches to the parallel mode and the next alternative is taken.
- If the processor is in the sequential mode and S = T and the current CP frame is not marked, the next alternative is taken.
- If the processor is in the sequential mode and S = T and the next alternative clause does not use cut and the current CP frame is marked, the processor switches to the parallel mode and the next alternative is taken.

The semantics of failure is as follows.

```
If B = Low
Then terminate
Else
        If B < T Then T := the first M frame below B
        If S = Low
        Then                        {B > Low  and  S = Low}
                If next alternative containing cut (i.e. retry+mark or trust+mark)
                Then S := T
        Else                        {B > Low  and S > Low}
                If T < S
                Then                {B > Low  and  S > Low  and  T < S}
                     If next alternative containing cut (i.e. retry+mark or trust+mark)
                     Then S := T
                     Else S := Low
                Else                {B > Low  and  S > Low  and  T ≥ S}
                     If S = T
                     Then           {B > Low  and  S > Low  and  T = S}
                          If marked(B)  and  next alternative with no cut
                          Then S := Low
        process the next alternative clause
```

We notice that, when the taken alternative is the last one of the CP (i.e. *trust* instruction), the CP frame is removed from the stack exactly the same as in WAM.

(h) On splitting:
If the current processor is in the parallel mode (i.e. S = Low), all CP frames in the CP-stack can be split up. Otherwise, CP frames below S only can be split up. These CP frames can be split up and assigned to the other processors in any order. The only necessary condition is that when a CP with cut (marked CP) splits up, the leftmost branch with cut along with the right branches assign to one processor.

One possible scheme for assigning alternative branches of CP frames, that can be split up, to the other processors is as follows.

0. Select a CP frame c.
1. If the next alternative clause of c does not use cut, it is assigned to another processor.

2. If the next alternative clause uses cut, the alternative and the remaining alternatives of c are assigned to another processor along with the respective M frame. (In this case, pointers to the M frame will be updated.)
3. If all alternative clauses at c are assigned to other processors, another CP frame is taken and split up exactly the same as in 1 and 2.
4. Step 3 is repeated until either no more CP frames can be split up, or no processor is idle.

If it is possible to distinguish between parallel and sequential CP frames as in ANLWAM [Over86], the parallel CP frames in the part of the CP-stack, that can split up, split up and assign to the other processors.

4. On Compiling Indexing and Cut

This section gives an example of compiling a predicate with cut. Then it discusses the problem of creating more than one CP frame for one predicate when using the two-level indexing and disjunctions.

In WAM, nine instructions are used for clause indexing:

```
try L
retry L
trust L

try_me_else L
retry_me_else L
trust_me_else_fail

switch_on_term Lv, Lc, Ll, Ls
switch_on_constant N, Table
switch_on_structure N, Table
```

where the first six instructions are used for managing choicepoints and the last three for discriminating on the first argument.

In the scheme presented in Section 3, the first three instructions for managing choicepoints are assumed. It is easy to extend our scheme to include also the instructions *try_me_else*, *retry_me_else* and *trust_me_else_fail* (the next instruction to be tested is the one below the current instruction). However, in some WAM implementations as in SICStus [Carl86], the first three indexing instructions are only used for managing choicepoints, since the instructions *try_me_else*, *retry_me_else* and *trust_me_else_fail* can be replaced by *try/ retry/ trust.*

The following example shows the generated code for a predicate p of four clauses when the third clause contains cut.

```
p:   choice An            {each predicate with cut begins with choice instruction}
     mtry  C1             {the mtry instruction is used to create a marked CP frame}
     retry C2
     retry C3
     trust C4

C1:  <code for the first clause>
C2:  <code for the second clause>
C3:  mark                 {each clause with cut begins with mark instruction}
     <code for the third clause>
C4:  <code for the fourth clause>
```

If the clause C3 had three cuts, it would compile to:

```
C3:  mark
     ....
     cut  Vn                          {first cut in a clause}
     ....
     cut  Vn                          {second cut in a clause}
     ....
     lastcut  Vn                      {last cut in a clause}
     ....
```

In the remaining part of this section we discuss problems with indexing and creating more than one CP for one predicate.

In our scheme presented in Section 3, we assumed that at most one CP frame for each predicate, and that once a CP is established, a simple test will be done in order to know the next alternative clause using cut or not. For example, on backtracking if the next alternative from the current CP is not the last one (i.e. retry L instruction), we do the following simple test:

Is *L = mark

in order to know the next alternative uses cut or not.

We do not assume more than one CP frame for one predicate (two-level indexing scheme) for the following reasons:

1. If two CP frames are created for one predicate with cut, these two frames should be treated as one unit when such frames split up. A problem arises when one of such frames (the innermost) has cut and is assigned to another processor. In this case, when the cut is executed, our mechanism can not remove alternatives that reside in another processor's CP-stack.

2. The second problem with the two-level indexing is the possibility of performing a large test operation in order to know the next alternative uses cut or not. For instance, If the outer CP frame is created and the next instruction (at *L) is one of the switch instructions. In this case, the first argument should be tested, then the target instruction, and possibly more instructions should be investigated.

In one-level indexing scheme by Carlsson [Carl87], there is at most one CP created for any predicate. It discriminates first on the type of the first argument, and second, when appropriate, on its principle functor. A CP is then needed only for non-singleton sets. In Carlsson's scheme, switching instructions are used only before establishing a CP. (The switching instructions: *switch_on_constant* and *switch_on_structure* are slightly modified.) So, our scheme in Section 3, works fine with the one-level indexing scheme.

The other source, that may generate two-level indexing, is disjunctions in a predicate. A problem arises only when cuts are used inside disjunctions, as in the following example:

```
p(X) :- q(X), ((r(X), !, s(X)); t(X)), u(X).
p(X) :- v(X).
```

In general, two different operational semantics are used for such cuts:

1. A cut commits the innermost disjunction in which it occurs.
2. A cut commits the entire predicate in which it occurs.

Our scheme works with the first semantics (and not with the second) by transforming each innermost disjunction to a new predicate. The above p/1 predicate transforms to

```
p(X) :- q(X), p'(X), u(X).
p(X) :- v(X).

p'(X) :- r(X), !, s(X).
p'(X) :- t(X).
```

Carlsson [Carl87] and O'Keefe [O'Kee85] pointed out that constructs with cuts inside disjunctions represent doubtful programming style.

5. Optimization

In this section we discuss a possible optimization to our scheme. None of the operations: cut, lastcut, mark, mtry, or choice is used in any predicate not using cut. However, some extra work on backtracking, will be performed even for programs not using cut. The following three tests are performed before taking the next alternative branch:

(1) If B < T	{the result of this test is always false for programs with no cut}
(2) If S = *Low*	{the result of this test is always true for programs with no cut}
(3) If next alternative using cut	{the result of this test is always false for programs with no cut}

For programs with no cut, no M frame is created and T is always equal to *Low*. So, one possible optimization that requires only one test on backtracking is as follows. T first is checked against *Low*, if they are equal, then there is no M frame in the CP-stack and B points to the top of the CP-stack. The next alternative clause is just taken without any further test.

As long as no M frame is created (i.e. T = *Low*), this optimization reduces the extra work. However, once an M frame is created on the CP-stack (i.e. T > *Low*), this extra test is added on backtracking.

Marked lines with "•" at the left in the following code are the added parts to the previous definitions in Section 3. On failure is modified to

```
  If B = Low
  Then terminate
  Else
•       If T = Low
•       Then process the next alternative clause
•       Else
            If B < T Then T := the first M frame below B
            Exactly the same remaining code as in Section 3 (g)
```

6. Conclusions and Discussions

We have presented a parallel scheme for implementing the asymmetric cut in parallel execution of Prolog on either shared [Over86, Ciep87, Warr86], or nonshared memory [Ali86] architectures. The scheme takes advantages of the efficient implementation of cut in the sequential WAM. Three new instructions (lastcut, mtry and mark) are added to WAM and operations that manage choicepoints are modified. This modification adds a small extra overhead over the sequential WAM. The scheme works with one-level indexing [Carl87]. For cuts inside disjunctions, our scheme supports the operational semantics of such a cut as follows. A cut commits the innermost disjunction in which it occurs.

The scheme restricts the parallelism, which is in some situations good and bad in others. It is good when many small tests are needed and many or large branches will be discarded. It is bad when we get a situation in which a large tree is enforced to be executed sequentially due to cut, and one of the right branches has a needed solution.

A detailed experimental study of different ways on practical large programs is required in order to conclude which implementation method gives better overall performance of the system; a simple one that restricts parallelism with small overhead as ours, or a general sophisticated one that allows more, or all possible parallelism with more memory space and processing time overhead.

Acknowledgments

The author would like to thank Mats Carlsson, Dan Sahlin, and Andrzej Ciepielewski for discussing and refining the idea reported herein.

References

[Ali86] K. A. M. Ali, "Or-parallel execution of Prolog on a multi-sequential machine", Accepted for publication in *International Journal of Parallel Processing*.

[Carl86] M. Carlsson, "SICStus: Preliminary Specification", SICS, Working paper April 1986.

[Carl87] M. Carlsson, "On Compiling Indexing and Cut for the WAM", Accepted for publication in the 4th International Conference on Logic Programming, 1987.

[Ciep87] A. Ciepielewski, "Or-parallel execution of Prolog on shared memory multiprocessor systems", SICS, (in preparation).

[Moss86] C. Moss, "CUT & PASTE - defining the impure primitives of Prolog", Proc. 3ICLP, London, 1986.

[O'Kee85] R. A. O'Keefe, " On the treatment of cuts in Prolog source-level tools", Proc. IEEE Symposium on Logic Programming, Boston, 1985.

[Over86] R. Butler, E. L. Lusk, R. Olson, R. Overbeek, " ANLWAM: A Parallel Implementation of the Warren Abstract Machine", Technical Report ANL-86- , Argonne National Laboratory, Argonne, Aug. 1986.

[Venk84] R. Venken, "A Prolog Meta-Interpreter for Partial Evaluation and its Application to Source-to-Source Transformation and Qurey Optimization", Proc. ECAI 1984.

[Warr83] D. H. Warren, "An abstract Prolog instruction set", Technical Note 309, SRI International, Menlo Park, CA, Oct. 1983.

[Warr86] D. H. Warren, "Or-Parallel Execution Models of Prolog", Technical Note, Depart. of Computer Science, University of Manchester, Dec. 1986

SICS Research Reports

Box 1263
S-164 28 Kista
Sweden

R 86001 Yoeli, M. and B. Pehrson, *Behavior-Preserving Reductions of Communicating System Nets*, 1986

R 86002 Hausman, B., *A Simulator of the OR-Parallel Token Machine*, 1986

R 86003 Ciepielewski, A. and B. Hausman, *Performance Evaluation of a Storage Model for OR-Parallel Execution of Logic Programs*, 1986 (Also located in the Proceedings of the 1986 Symposium on Logic Programming, September 22-25, 1986, Salt Lake City, Utah. pp. 246-257. IEEE Computer Society Press)

R 86004 Karjoth, G., P. Sjödin and S. Weckner, *A Sophisticated Environment for Protocol Simulation and Testing,* 1987

R 86005C Hallnäs, L., *Partial Inductive Definitions,* 1988 (Revised version of R 86005 "On the Interpretation of Inductive Definitions" and R 86005B)

R 86006B Khayri A. M. Ali, *OR-Parallel Execution of Prolog on a Multi-Sequential Machine,* 1986 (Revised version of R 86006) (Also located in the International Journal of Parallel Programming. Vol. 15, No. 3, June 1986, pp. 189-214.)

R 86007 Wærn, A., *Process Models of Logic Programs: a Comparison,* 1987

R 86008 Mathieu, P., *On the Learning of Functional Dependencies in Deductive Databases,* 1986 (an extended abstract) (A more comprehensive version, in French, in the Proceedings of the 1st Spanish Congress on Artificial Intelligence and Databases, Blanes, 1985)

R 86009B Appleby, K., M. Carlsson, S. Haridi and D. Sahlin, *Garbage Collection for Prolog Based on WAM*, 1988 (Revised version of R 86009) (To be published in Communications of the ACM, June 1988)

R 86010 Hallnäs, L., *Generalized Horn Clauses,* 1986 (No longer distributed; replaced by R 88005)

R 86011 Carlsson, M., *On Compiling Indexing and Cut for the WAM,* 1987 (Related papers, see R 86012)

R 86012 Carlsson, M., *An implementation of "dif" and "freeze" in the WAM,* 1986 (Extended version *Freeze, Indexing, and Other Implementation Issues in the WAM* located in Logic Programming: Proceedings of the Fourth International Conference, Vol. 1, pp. 40-58. MIT Press, 1987)

R 86013 Elshiewy, N., *Time, Clocks and Committed Choice Parallelism for Logic Programming of Real Time Computations,* 1987 (Related paper *Logic Programming for Real Time Control of Telecommunication Switching Systems* to appear in the Journal of Logic Programming, 1988)

R 87001 Khayri A. M. Ali, *A Method for Implementing Cut in Parallel Execution of Prolog,* 1987 (Also located in Proceedings of the 1987 Symposium on Logic Programming, August 31-September 4, 1987, San Francisco, California. pp. 449-456. IEEE Computer Society Press.)

R 87002 Rayner, M. and S. Janson, *Epistemic Reasoning, Logic Programming, and the Interpretation of Questions,* 1987 (Also located in Natural Language Understanding and Logic Programming, II. pp. 301-318. North-Holland Press.)

R 87003 Gunningberg, P., *Innovative Communication Processors: A Survey,* 1987

R 87004 Gadener, C., M. Lidén and J. Riboe, *ORPWAM An Implementation Study, Part 1*, 1987 (out-of-print; no longer published).

R 87005 Khayri A. M. Ali and S. Haridi, *Global Garbage Collection for Distributed Heap Storage Systems,* 1987 (Also located in the International Journal of Parallel Programming. Vol. 15, No. 5, October 1986, pp. 339-387.)

R 87006 Hausman, B., A. Ciepielewski and S. Haridi, *OR-parallel Prolog Made Efficient on Shared Memory Multiprocessors,* 1987 (Also located in Proceedings of the 1987 Symposium on Logic Programming, August 31-September 4, 1987, San Francisco, California. pp. 69-79. IEEE Computer Society Press.)

R 87007 Holmgren, F. and A. Wærn, *A Scheme for Compiling GHC to Prolog Using Freeze,* 1987

R 87008 Sahlin, D., *Making Garbage Collection Independent of the Amount of Garbage,* 1987 (Appendix to SICS Research Report R 86009)

R 87009 Sjödin, P., *Optimizing Protocol Implementations for Performance - A Case Study,* 1987 (Author's Licentiate thesis at Uppsala University, Sweden)

R 87010B Franzén, T., *Algorithmic Aspects of Intuitionistic Propositional Logic,* 1988 (Revised version of R 87010)

R87011 Mathieu, P., *Towards the Realization of the Database Designer's Apprentice. Part 1: General Introduction,* 1987

R 88001 Rayner, M., Å. Hugosson and G. Hagert, *Using a Logic Grammar to Learn a Lexicon,* 1988 (Also located in SCAI'88: Proceedings of the first Scandinavian Conference on Artificial Intelligence, Tromso, Norway, March 9-11, 1988. pp.143-154)

R 88002 Franzén, T., *Logic Programming and the Intuitionistic Sequent Calculus,* 1988

R 88003 Rayner, M. and Å. Hugosson, *Reasoning about Procedural Programs in a Chess Ending,* 1988

R 88004 Wærn, A., *An Implementation Technique for the Abstract Interpretation of Prolog,* 1988.

R 88005 Hallnäs, L. and P. Schroeder-Heister, *A Proof-Theoretic Approach to Logic Programming. I. Generalized Horn Clauses*, 1988

R 88006 Nordmark, E. and P. Gunningberg, *SPIMS: A tool for protocol implementation performance measurements*, **to be published**

R 88007 Carlsson, M. and J. Widén, *SICStus Prolog User's Manual,* 1988

R 88008 Kreuger, P., *A Higher Order Logic Parser for Natural Language Implemented in Lambda Prolog*, 1988

R 88009 Rayner, M., *On the applicability of default logic: two short papers*, 1988

R 88010 Khayri A. M. Ali, *OR Parallel Execution of Horn Clause Programs Based on WAM and Shared Control Information*, 1986

R 88011 Khayri A. M. Ali, *OR-Parallel Execution of Prolog on BC-Machine*, 1988