PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT. © ICOT, 1988

Cut and Side-Effects in Or-Parallel Prolog

Bogumil Hausman*, Andrzej Ciepielewski*
Alan Calderwood†

*Swedish Institute of Computer Science
Box 1263
S-164 28 Kista
Sweden

†Department of Computer Science
University of Manchester
Manchester M13 9PL
U.K.

Abstract

Two contrasting approaches are possible when Prolog is implemented on a multiprocessor system. One is to treat the implementation as porting to yet another machine and say that the language is still strictly Prolog. Another approach is to seek new extralogical features more suitable for the new execution strategy. We introduce some new extralogical predicates and compare them to the corresponding Prolog predicates. We then discuss some implementation options for Prolog I/O, database and cut predicates. Finally, we propose an algorithm for execution of cut which is a good compromise between implementation complexity and amount of speculative work allowed.

1. Introduction

One of the first results of the Gigalips project, a collaborative effort between SICS, Manchester University and ANL in Illinois, is the Aurora or-parallel Prolog system running on shared-memory multiprocessors (Lusk et al. 1988). The system implements a mixture of the depth-first and the breadth-first search strategies. Computation can be seen as processing of a search tree by a team of workers, with each worker following the depth-first strategy whenever possible.

One of the initial goals of the project is to demonstrate that existing Prolog programs can be run much faster in parallel with no, or minimal, changes, producing exactly the same results as on sequential systems. Producing the same results means giving the same set of results and in the same order irrespective of the system. Many of the extralogical predicates of Prolog, like I/O predicates, database predicates and cut, are very strongly dependent on the depth-first execution strategy. We had to rethink their meaning and implement them in the or-parallel setting.

Another goal of the project is to experiment with new language features. One of the things we are looking at are "asynchronous" versions of built-in predicates, like I/O and database predicates, not depending on Prolog's particular execution order. By using the new predicates we would loose the strong equivalence of results obtained on different systems. However, it is often not important in which order the results are presented, and the programs written without considering Prolog's execution order might come closer to the ideal of logic programming. Besides, introducing predicates more suitable for or-parallel execution can win efficiency.

The main contribution of this paper is the presentation of an implementation of cut and "ordinary" side-effect predicates. We also make a small contribution to the discussion about suitable extensions of Prolog by presenting a set of examples using the new "asynchronous" primitives.

The paper is organized as follows. In Section 2 we introduce the asynchronous predicates, show how they can be used, and compare them with the ordinary Prolog predicates. In Sections 3, 4, 5 and 6 we discuss different implementation options of cut and ordinary side-effect predicates and describe our choices.

2. Using Prolog on an Or-Parallel System

One of our goals is to be able to run the same Prolog program on any hardware, parallel or sequential. On the other hand Prolog is not the ultimate logic programming language. Moving to a different type of hardware asks for reevaluation of the language, maybe mostly of its extra-logical features like side-effect predicates and control predicates. We do not claim to have a final answer, we rather want to show some "natural" extensions which have been discussed in the Gigalips project and elsewhere, and which implementation we are considering.

In this section we compare uses and semantics of the ordinary Prolog predicates and their "asynchronous" versions, and also indicate some minimal requirements placed on the implementation.

In the examples below, the procedures which can be executed in or-parallel are declared as *parallel*.

2.1 I/O Predicates and Database Predicates

Input/output predicates in Prolog enforce order in which data is read or written. Program 1(a) prints all permutations of a list of numbers in the order decided by the depth first execution of Prolog. This program uses the *ordinary write* predicate. Often it is not important to present the solutions in any particular order. Program 1(b) run on our system would present solutions in some unknown order. The program uses the *asynchronous write* predicate (*asynch_write*). Expressed in implementation terms, *write* can be executed first when the branch it is in becomes the leftmost one in the search tree, while *asynch_write* can be executed anywhere in the tree.

(a)
```
all_permutations(Xs) :-
        permutation(Xs,Zs),
        write(Zs),
        fail.

permutation(Xs,[Z|Zs]) :-
        select(Z,Xs,Ys),
        permutation(Ys,Zs).
permutation([],[]).

:- parallel select/3.
select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]) :- select(X,Ys,Zs).
```

(b)
```
all_permutations(Xs) :-
        permutation(Xs,Zs),
        asynch_write(Zs),
        fail.
```

Program 1. Two versions of all_permutations program. We assume that the action of *asynch_write* is atomic.

Not only input/output predicates depend on a particular execution order of Prolog. Database predicates like *assert* or *retract* depend on the ordering even stronger. An *ordinary assert* or *retract* can be executed first when the branch it is in becomes the leftmost one in the search tree. We do not want to go into the complex issues of modifying or-parallel Prolog programs "on the fly" in general. There are, though, some uses of *assert* and *retract* which are solely for controlling searches. For example, in the program for playing Mastermind presented in (Sterling and Shapiro 1986) *assert* ensures that the search space is traversed from left to right even when executed on an or-parallel system. This requirement can be relaxed, because it is not important in what order the solutions are found. In the version of the Mastermind program presented in Program 2 *asynchronous assert* (*asynch_assert*) is used. There are no restrictions on when *asynch_assert* can be executed.

The "parallel" version of the Mastermind program has been shown to us by Peter Szeredi.

```
solve(Code):-
        length(Code,N),
        asynch_abolish(tried/1),
        asynch_assert(tried([])),
        guess(N,Guess),
        lock,
        tried(Tried_Guesses),
        unlock,
        test(Tried_Guesses,Guess,N),
        ask(Code,Guess,[B,C]),
        lock,
        asynch_abolish(tried/1),
        asynch_assert(
                tried([[B,C,Guess]|Tried_Guesses])),
        unlock,
        B=N.

guess(0,[]) :- !.
guess(N,[Q|Qs]) :-
        pick(Q),
        N1 is N-1,
        guess(N1,Qs).

:- parallel pick/1.
pick(0).
pick(1).
pick(2).
pick(3).
pick(4).
pick(5).
pick(6).
pick(7).

test(Tried_Guesses,Guess,N) :-
        tests if Guess is consistent with all of the
        answers already made

ask(Code,Guess,[B,C]) :-
        asks the user for the number of bulls and
        cows in Guess
```

Program 2. "Parallel" version of the Mastermind program.

In Program 2 *asynch_assert* is used to store the list of tried guesses. Notice the problems caused by the atomicity requirements, the system predicates for locking and unlocking guarantee mutually exclusive access to the critical section of the program.

The database predicates can also be used to collect answers generated in set predicates. In the *find_all* program in Program 3(a) we used *ordinary assert* and *retract* which means that even when executed on or-parallel system the answers are added to the database in the order decided by the depth first execution. We are interested in a set of answers, thus it is not important in which order they are added to the database. In Program 3(b) *asynchronous assert* and *retract* along with a unique functor name are used to store the answers in order they are generated. The only restriction is that if the *Goal* invokes procedures containing cuts, we store only the answers which would have been returned on a sequential system (Hausman and Ciepielewski 1988).

```
(a)
  :- parallel Goal.
  find_all(X,Goal,Xs) :-
        find_all_dl(X,Goal,Xs\[]).

  find_all_dl(X,Goal,Xs) :-
        asserta($instance($mark)),
        Goal,
        asserta($instance(X)),
        fail.
  find_all_dl(X,Goal,Xs\Ys) :-
        retract($instance(X)),
        reap(X,Xs\Ys), !.

  reap(X,Xs\Ys) :-
        X ≠ $mark,
        retract($instance(X1)), !,
        reap(X1,Xs\[X|Ys]).
  reap($mark,Xs\Xs).

(b)
  :- parallel Goal.
  find_all(X,Goal,Xs) :-
        unique_functor(F),
        find_all_dl(X,Goal,Xs\[],F).

  find_all_dl(X,Goal,Xs,F) :-
        Goal,
        T =.. [F,X],
        asynch_assert(T),
        fail.
  find_all_dl(X,Goal,Xs\Ys,F) :-
        reap(Xs\Ys,F), !.

  reap(Xs\Ys,F) :-
        T =.. [F,X],
        asynch_retract(T), !,
        reap(Xs\[X|Ys],F).
  reap(Xs\Xs,_).

  unique_functor(Functor) :-
        generates a new unique functor
```

Program 3. Two versions of find_all program.

In Section 5 (implementing database predicates) we shall discuss a safe way of using database predicates in general.

2.2 Control Predicates

Execution of Prolog programs can be controlled by the *cut* predicate even in an or-parallel system. Examining *cut* and ordinary side-effects from the implementation point of view, we can see that the rule delaying execution of side-effects until their branches become leftmost ensures that only those results that would be produced during sequential execution will be seen. The shell program shown as Program 4(a) would produce exactly the same sequence of results in our system as in a sequential one. Program 4(b) is a version of the interactive shell in Program 4(a) with the difference that now we are not interested in the same sequence of results as in the sequential system, the asynchronous predicates are used and the results appear in order they are found. As in the second version of the Find_all program the only restriction is that if the *Goal* invokes procedures containing cuts, we present only those results that would be produced during sequential execution (the results that would not be cut).

The *cut* in *shell_solve* was replaced by another control predicate, *cavalier commit (|)*, proposed by Ross Overbeek (Warren 1987, Butler et al. 1988). *Cavalier commit*, as we understand it, commits to the first solution irrespective of the clause order, it does not interact with side-effects, and does not prohibit other branches in its scope from producing results if they happen to be faster.

Cavalier commit can increase efficiency of or-parallel execution when we need only one solution to the given goal and we do not care which clause is used. Cut in Program 5(a) commits the execution of *Goal* to the first alternative found in the depth first search while cavalier commit in Program 5(b) can commit the execution to any alternative.

In the rest of the paper we describe the implementation of cut and ordinary side-effects in the or-parallel setting. We postpone discussion of an implementation of cavalier commit and other possible control primitives until their role in the language is better understood. For further details see (Hausman and Ciepielewski 1988).

```
:- parallel Goal.
shell :-
      shell_prompt,
      read(Goal),
      shell(Goal).

shell(exit) :- !.
shell(Goal) :-
      ground(Goal), !,
      shell_solve_ground(Goal),
      shell.
shell(Goal) :-
      shell_solve(Goal),
      shell.

shell_solve(Goal) :-
      Goal,
      write(Goal), nl,
      write('Next Solution? '),
      read(Answer),
      check(Answer), !.
shell_solve(Goal) :-
      write('No (more) solutions'), nl.

shell_solve_ground(Goal) :-
      Goal, !,
      write('Yes'), nl.
shell_solve_ground(Goal) :-
      write('No'), nl.

check(yes) :- fail.
check(no).
shell_prompt :- write('Next Command? ').
```

Program 4(a). "Sequential" version of an interactive shell.

```
:- parallel Goal.
shell :-
      shell_prompt,
      read(Goal),
      shell(Goal).

shell(exit) :- !.
shell(Goal) :-
      ground(Goal), !,
      shell_solve_ground(Goal),
      shell.
shell(Goal) :-
      shell_solve(Goal),
      shell.

shell_solve(Goal) :-
      Goal,
      lock,
      asynch_write(Goal), nl,
      asynch_write('Next Solution? '),
      asynch_read(Answer),
      unlock,
      check(Answer), |.
shell_solve(Goal) :-
      write('No (more) solutions'), nl.
```

Program 4(b). "Parallel" version of an interactive shell.

```
(a)
  :- parallel Goal.
  not(Goal) :-
        Goal, !,
        fail.
  not(Goal).

(b)
  :- parallel Goal.
  not(Goal) :-
        Goal, |,
        fail.
  not(Goal).
```

Program 5. Two versions of "negation as failure".

3. Prerequisites

Programs are compiled to WAM-like instructions (Carlsson 1987), but the code can still be modified during execution. The state of the computation during the execution of a program is represented by a tree.

A node represents the state of the computation on the corresponding branch at the time a non-deterministic predicate has been invoked. Among other things a node contains explicit information about still available alternatives. We assume that at the tip of each branch there is a node containing the current state, and shall call such nodes *embryonic*. An embryonic node is converted into an ordinary node when a non-deterministic predicate is invoked. A node can be removed from the tip of a branch, which corresponds to backtracking in sequential Prolog. The execution tree of a program is represented in such a way that the parent of a node, its siblings and children can be reached from each node. Each branch of a tree is divided into a private and a public parts. The public part of a branch can be dynamically extended. We shall call the oldest private node the *sentry* node. There is always a sentry node on a branch, though it might be embryonic. A sample tree is pictured in Figure 1.

A tree of a program is processed by several workers (be it processes or processors) and each node contains explicit information which workers are active in the subtree rooted at the node. The workers proceed independently of each other except when the nodes in the public part of the tree are accessed. Actions on the public nodes are considered atomic. Atomicity is obtained by using a lock associated with each node. A worker active in its private part of the branch can be viewed as an instance of sequential WAM.

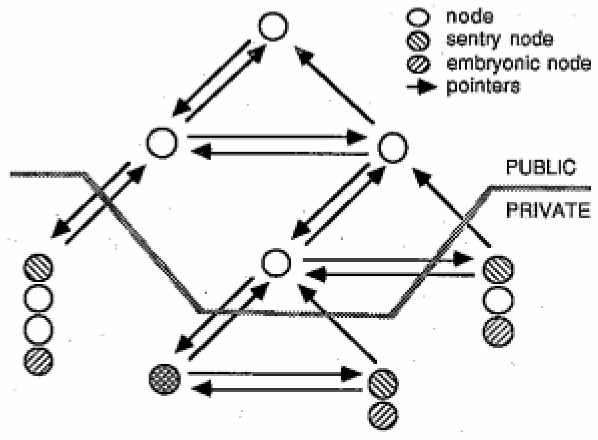

Figure 1. An abstract representation of a search tree.

The whole Aurora system is described in (Lusk et al. 1988), and the schedulers used in (Butler et al. 1988) and (Calderwood 1988a).

In the rest of this paper we assume for simplicity that all predicates are declared as parallel.

4. I/O Predicates

In this section we consider the ordinary I/O predicates.

In or-parallel systems, in contrast to the sequential ones, there are usually several branches being executed simultaneously. Any of the branches could contain I/O predicates. As stated in the previous section, an I/O predicate can be executed only in the leftmost branch. If an I/O predicate is reached before its branch becomes leftmost, processing of the branch must be suspended until all the branches to its left are completed.

Before executing an I/O predicate a worker checks if the branch is leftmost by looking for left sibling nodes starting from the worker's sentry node. When a left sibling node is found the worker's branch is suspended and the sibling node gets a reference to the tip node in the branch. When the node causing suspension is to be removed the reference is stored in the next left sibling node. If there are no more left sibling nodes the suspended branch is activated. To avoid the repeated checking of nodes all the way to the root, the worker which finds that its branch is leftmost marks all the nodes in the branch as leftmost. The leftmost test will stop as soon as a node marked as leftmost is found. The root node is always marked as leftmost.

The algorithms expressing suspension and activation are presented in Figure 2. *Suspend* is invoked with a reference to the current sentry node as argument before an I/O predicate is called. *Activate* is invoked when a node is about to be removed from the tree by a worker searching for work. *Leftmost* is invoked with two arguments, the first is the current node and the second the reference to the suspended branch, when invoked in *suspend* both arguments reference the sentry node.

```
suspend(n) = if not leftmost(n, n) then
                 suspend this branch.

bool leftmost(n, b) =
     if marked_as_leftmost(n) then
          return(true)
     else if exists n.leftsibling then
          {n.leftsibling.suspended_branch := b
          return(false)}
     else if leftmost(n.parent, b) then
          {mark_as_leftmost(n)
          return(true)}.

activate(n) =
     if branch to the right is suspended and
          leftmost(n, n.suspended_branch) then
               activate the suspended branch.
```

Figure 2. Basic algorithms for suspension and activation.

The test "branch to the right is suspended" is implemented using an explicit reference (*suspended_branch*) to the tip node in a suspended branch. The reference is stored in the node causing suspension, when the leftmost test fails. When the node causing suspension is to be removed, the reference is copied to the next node which causes the leftmost test to fail.

The suspension and activation processes are illustrated in Figure 3.

When the worker active below sentry node 7 executes leftmost(7), it suspends and node 6 gets reference to the suspended branch (reference to node 7) (A). When another worker removes node 6, the reference to node 7 is copied into the node 2 (B). When node 2 is removed, node 7 becomes leftmost and the branch is activated. At the same time nodes 1, 3, 4 and 7 are marked as leftmost.

Our algorithms are neutral to what actually happens with the worker on the suspended branch. It can either busy-wait on the suspended branch, or mark the nodes in the branch as suspended and

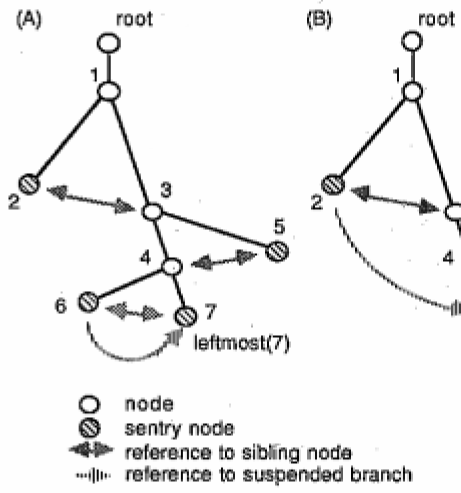

Figure 3. A tree with a suspended branch.

search for work. Activation would then mean either reactivating the busy-waiting worker or marking the nodes in the branch as leading to work.

5. Database Predicates

To keep the sequential semantics of database predicates in an or-parallel system we need some synchronization, because execution of a database predicate could affect goals in the whole search tree (Example 1).

Example 1

```
    :- parallel q, p.
    :- dynamic z0.

(1) q :- r0.
(2) q :- p.
(3) q :- r1, z0.

(4) p :- z0.
(5) p :- z1, asserta(z0).
(6) p :- z3, z0.
```

When executing on an or-parallel system, the *asserta(z0)* in clause (5) could affect the computation of subgoals *z0* in clauses (3), (4) and (6), whereas only execution in clauses (3) and (6) should be influenced.

We assume that all the procedures which can be modified by adding or deleting individual clauses are declared as *dynamic* (as e.g. in Quintus Prolog). We shall call the goals modifying the dynamic predicates *hard side-effects*, and the goals invoking dynamic predicates *unsafe goals*.

To keep the right ordering (i.e. sequential Prolog ordering) between the *unsafe* and *hard side-effects* goals, the execution of such predicate is not allowed until the branch containing it becomes leftmost in the tree. To achieve this the algorithms introduced for I/O predicates can be used.

By invoking the *suspend* procedure as part of all *hard side-effects* and *unsafe goals* we also solve the problem of the *call* predicate invoking one of the mentioned goals. For example the effect of *assert(P)* and *call(X)* where *X* is bound to *assert(P)* will be then the same.

Which predicates are *hard side-effects* or *unsafe* can be decided on the compile time.

6. Cut

The main difficulty in implementing cut correctly on an or-parallel system is caused by the case of several workers executing cuts pruning overlapping parts of the execution tree. The problem is illustrated by the examples in Figures 4 and 5.

If in Figure 4 the cut $!^{(2)}$ in nodes G, D or E is reached before the cut $!^{(3)}$ in node F, care must be taken that the cut $!^{(2)}$ does not prune the third branch in *p*. If that happened, the semantics of cut would be violated, because during the sequential execution the *choose(a1)* predicate succeeds and the cut $!^{(2)}$ in the second clause of *p* is not reached because of the failure of *validate(a1)*. Looking at the execution tree, one can see that when the cut $!^{(2)}$ is reached it is in the scope of the cut $!^{(3)}$. The cut $!^{(2)}$ has a larger scope than the cut $!^{(3)}$, as it cuts to a higher node in the tree, and their scopes overlap.

If in Figure 5 the cut $!^{(2)}$ in node F is reached before the cut $!^{(3)}$ in node E, care must be taken that the cut $!^{(2)}$ does not prune the third branch in *p* and the second branch in *w*, because during the sequential execution a *choose(a1)* predicate succeeds and the cut $!^{(2)}$ in the second clause of *p* in node F is not reached.

The general rule is that, if a cut is in a scope of another cut, the cut with the larger scope must not prune branches that would not be pruned anyway by the cut with the smaller scope. Those branches can be pruned when the branch leading to the cut with the smaller scope fails.

```
p :- veryslow1, !(1).
p :- q, !(2).
p :- speculative1.

q :- choose(X), !(3), validate(X).
q :- speculative2
q :- speculative3.

choose(a1).
choose(a2).
validate(a2).
```

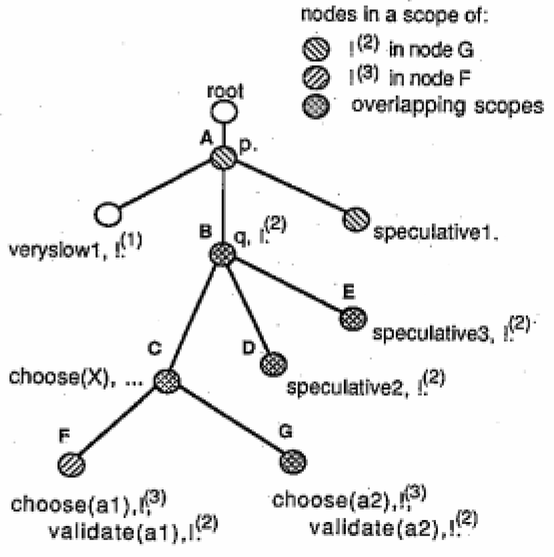

Figure 4. Cuts with different overlapping scopes. The clauses with cuts belong to predicates with many matching clauses. Active tasks are associated with the tip nodes.

```
p :- veryslow1, !(1).
p :- w, !(2).
p :- speculative1.

w :- q.
w :- speculative2.

q :- choose(X), !(3), validate(X).

choose(a1).
choose(a2).
validate(a2).
```

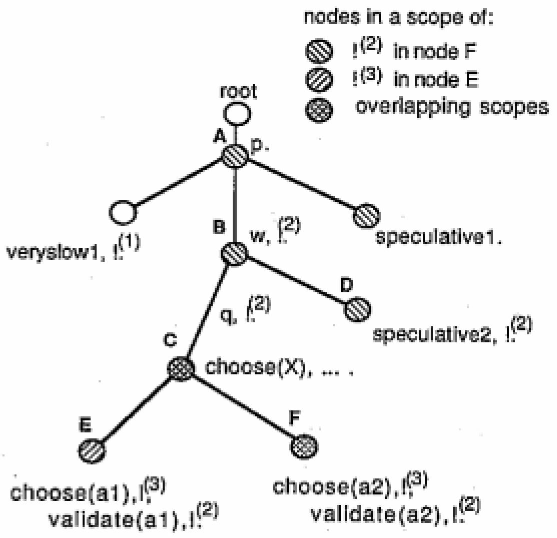

Figure 5. Cuts with different overlapping scopes. The clause with cut $!^{(3)}$ is the only matching clause of predicate *q*. Active tasks are associated with the tip nodes.

If correctness were the only consideration, then the cut's semantics could be implemented simply by suspending its execution until the branch it is in becomes leftmost. Another consideration is speculative work. Work in any branch which can be cut away is speculative. The simple solution proposed above lets computations in all branches in the scope of the suspended cut go on until the branch containing the suspended cut becomes leftmost. The restarted cut then prunes away branches in its scope.

A more ambitious plan (Calderwood 1988b) with corresponding implementation (Carlsson 1988) would be to prune all the branches in the scope of the cut in the subtree in which the branch with the cut is leftmost, and the branches rooted at the right siblings of the root of the subtree, and then suspend execution of the cut until the branch becomes leftmost in the subtree belonging to the predicate containing the cut. Note that in this and in the previous solution, execution of a cut might be suspended even if it is not in a scope of any other cut, which is clearly sub-optimal.

A better solution, with respect to the amount of speculative work performed, would be to prune at once all branches which would be pruned anyway by cuts with smaller scopes. The difference between this and the previous alternative is that the execution of the cut will be suspended only if it is in the scope of a cut with a smaller scope, and first when the scope boundary of the "smaller" cut is crossed.

A fourth solution (Ali 1987) would be to delay execution of any branch until it can no longer be affected by cuts. The solution prevents all speculative work, but at the same time limits the amount of parallelism severely.

The four schemes are illustrated in Figure 6.

p :- veryslow, $!^{(1)}$.
p :- q, $!^{(2)}$.
p :- speculative1.

q :- slow, $!^{(3)}$, fail.
q :- r.
q :- speculative2.

r :- slow.
r :- fast.
r :- speculative3.

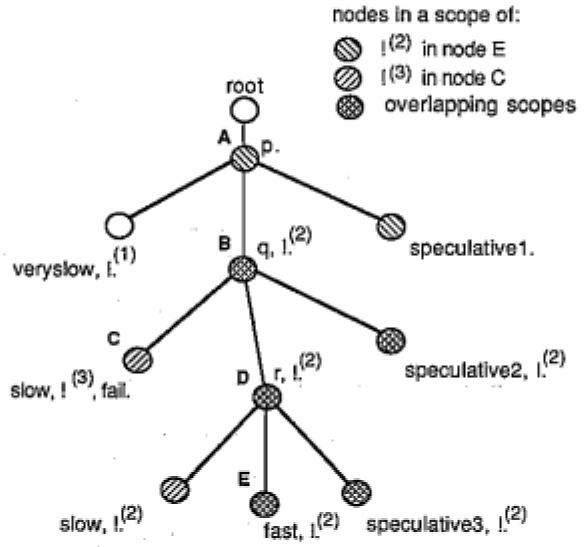

Figure 6. Speculative work in the four schemes for implementing cut. When the cut in node E is executed then: solution 1 - *speculative2* and *speculative3* continue; solution 2 - *speculative3* is cut, *speculative2* continues; solution 3 - *speculative2* and *speculative3* are cut. In solution 4 we do not start execution of the second and third branches of *p* until *veryslow* in *p* fails.

Before describing our implementation of the third scheme we have to explain the way cut is usually implemented, and define scope more precisely. On the WAM instruction level the cut predicate has an argument which is a reference to the last node created before the predicate containing the cut is invoked. We shall call the child of the argument node the *cut level*. The scope of a cut consists of nodes in the subtrees to the right of the path between the place on the branch where the cut is invoked and its *cut level*. Execution of a cut prunes all nodes in its scope.

All cuts in a predicate with many clauses, have the cut level corresponding to the node created when the predicate is invoked. The cut level of all cuts in a predicate with a one clause is the next node to be created after invoking the predicate. In general, the determinacy property of a predicate (one or more matching clauses) is dynamic due to indexing, and the rules deciding which node is the cut level must be applied appropriately.

The key to the implementation of the chosen scheme is the scope information.

6.1 Scope Information

There is one piece of scope information per node, the *cut level flag* which indicates whether the corresponding node is at the cut level of some cut. When a predicate with many matching clauses containing cut is invoked the cut level flag is set in the node created for the predicate. When a predicate with one matching clause containing cut is invoked the cut level flag is set in the embryonic node of the corresponding branch. Finally the flag is not set at all if the matching clauses do not contain any cuts.

The search trees from Figures 4 and 5 with the scope information of predicate *q* are shown in Figure 7.

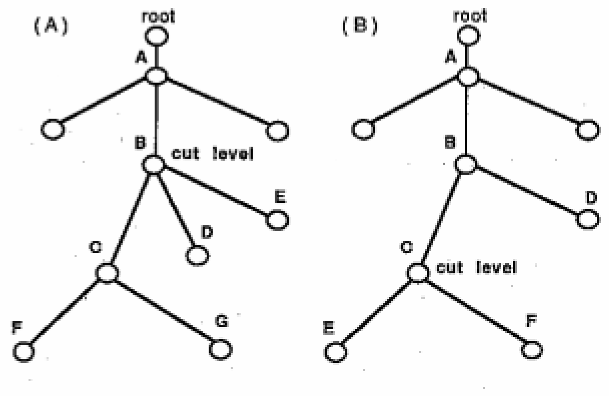

Figure 7. The search trees from Figure 4 (A) and Figure 5 (B) with the scope information of cut $!^{(3)}$ in predicate *q*. (Remember that the predicate *q* in Figure 4 is different from the one in Figure 5)

A worker executing cut is expected to prune all nodes in the scope of cut starting from the sentry node up to the cut level, traversing the path node by node. The worker can always prune all nodes up to a cut level of a "smaller" cut, because if the "smaller" cut were executed the pruned nodes would be pruned anyway. Then the worker can cut higher if all left branches leading to the "smaller" cut have failed.

In terms of the scope information the worker can cut higher if it is leftmost in the subtree rooted at the cut level node of the "smaller" cut (the node is marked with the cut level flag).

The scope information is used in the *cut_public* procedure which is invoked when a cut to a node in the public part of the tree is executed. Its arguments are references to the parent of the cut level node and to the sentry node. We describe only actions taken in the public part of the branch, because actions taken in the private part are the same as during pure depth first execution. In order to simplify the description, the algorithms do not contain calls to the mutual exclusion primitives (lock and unlock).

```
cut_public(c, b) =
    if not b.parent = c then
        {b.alternatives_available := false
        if exists b.right_sibling then
            prune_trees(b.right_sibling)
        if in_scope_of_cut(c, b) then
            b.parent.alternatives_available := false
        else
            cut_public(c, b.parent)}
    else
        b.alternatives_available := false.

bool in_scope_of_cut(c, b) =
    if b.parent.parent = c then
        return(false)
    else if b.parent.cut_level and
            not leftmost below b.parent then
        {suspend until leftmost below b.parent
        return(true)}
    else
        return(false).

prune_trees(n) =
    prune_tree(n)
    if exists n.right_sibling then
        prune_trees(n.right_sibling).
```

Figure 8. Algorithms for cut.

The invocation of *prune_tree(n)* in *prune_trees* interrupts all the workers active in a subtree rooted at *n*, the subtree is removed and the workers look for new tasks outside the pruned part of the tree. A worker executing the test *leftmost below node* invokes the *leftmost* procedure from Figure 2 with the difference that now the worker checks for left sibling nodes only up to the *node* and does not mark nodes as leftmost.

The scheme for cut can be further optimized. If there is no available work to take, the worker executing a cut to be suspended can proceed with the rest of its branch, and the pruning is completed by a scheduler at the time the suspended cut would have been activated. Another possible optimization is that the worker which ought to suspend proceeds if there are no nodes to be pruned.

The presented implementation of cut is optimal for procedures where all clauses contain cuts (except possibly the last), and is still correct, but not optimal, for more general case when the procedures contain more than one clause without cut. For example consider Figure 4. The procedure *q* contains two clauses without cut (the second and the third), and if the first clause of *q* fails before reaching the cut $!^{(3)}$ (it cannot happen in this example but let us assume it does) the worker executing the cut $!^{(2)}$ in node E does not have to wait for *speculative2* to fail before pruning *speculative1*.

There is also another inefficiency, when there are no more branches with pending cuts, then either the branches have failed or the cuts have already been executed, and the corresponding generated scope information is out of date and should be ignored. For further details and the optimized version of our implementation of cut see (Hausman 1988).

7. Conclusions

The proposed implementation of *ordinary* side-effects predicates and cut guarantees the same set of results and in the same order, whether a program is executed sequentially or in or-parallel.

The proposed algorithm for implementing cut is a good compromise between implementation complexity and the amount of speculative work performed. We expect that its advantages, as compared to the other schemes mentioned, will be especially clear for large programs with deep and bushy trees.

The overhead introduced by the algorithms consists of the compile time overhead which occurs only once for each program, and the run time overhead which occurs only when side-effect and cut predicates are executed.

The *asynchronous* versions of side-effect predicates introduced here are useful as a programming tool and can increase the efficiency of or-parallel execution. It must be further investigated what control primitives should be added to the language (*cavalier commit* is one of the alternatives), and how they will interact with cut and all types of side-effects.

Acknowledgements

We wish to thank all colleagues within the Gigalips Project and SICS for useful remarks and discussions and thanks are due in particular to Per Brand, Peter Szeredi and David H. D. Warren.

References

Khayri A.M. Ali. *A Method for Implementing Cut in Parallel Execution of Prolog*. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 449-456, 1987

Ralph Butler, Terry Disz, Ewing Lusk, Robert Olson, Ross Overbeek and Rick Stevens. *Scheduling Or-Parallelism: an Argonne Perspective.* To appear in *Proceedings of the Fifth International Logic Programming Conference and Fifth Symposium on Logic Programming 1988*, February 1988

Alan Calderwood. *Aurora - the Manchester Scheduler*. May 1988. Internal Report, Gigalips Project

Alan Calderwood. *Cut, Commit and Side Effects in Or-Parallel Prolog*. Internal Report, Gigalips Project/personal communication, 1988

Mats Carlsson. *Implementation of Cut in Or-Parallel Prolog*. Personal communication, April 1988

Mats Carlsson. *Internals of Sicstus Prolog Version 0.6*. November 1987. Internal Report, Gigalips Project

Bogumil Hausman. *Cut and Speculative Work in Or-Parallel Prolog*. SICS Research Report, Swedish Institute of Computer Science, July 1988

Bogumil Hausman and Andrzej Ciepielewski. *Control Primitives and Side-Effects in Or-Parallel Prolog*. SICS Research Report, Swedish Institute of Computer Science, 1988

Ewing Lusk, David H. D. Warren, Seif Haridi, Ralph Butler, Alan Calderwood, Terrence Disz, Robert Olson, Ross Overbeek, Rick Stevens, Peter Szeredi, Per Brand, Mats Carlsson, Andrzej Ciepielewski and Bogumil Hausman. *The Aurora Or-Parallel Prolog System.* To appear in *Proceedings of International Conference on Fifth Generation Computer Systems 1988*, April 1988

Leon Sterling and Ehud Shapiro. *The Art of Prolog*. pages 336-337, MIT Press, 1986

David H. D. Warren. *The SRI Model for Or-Parallel Execution of Prolog - Abstract Design and Implementation Issue.* In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92-102, 1987