Speculative Computations in Or-Parallel Tabled Logic Programs

Ricardo Rocha[1], Fernando Silva[1], and Vítor Santos Costa[2]

[1] DCC-FC & LIACC
University of Porto, Portugal
{ricroc,fds}@ncc.up.pt
[2] COPPE Systems & LIACC
Federal University of Rio de Janeiro, Brazil
vitor@cos.ufrj.br

Abstract. Pruning operators, such as *cut*, are important to develop efficient logic programs as they allow programmers to reduce the search space and thus discard unnecessary computations. For parallel systems, the presence of pruning operators introduces the problem of *speculative computations.* A computation is named speculative if it can be pruned during parallel evaluation, therefore resulting in wasted effort when compared to sequential execution. In this work we discuss the problems behind the management of speculative computations in or-parallel tabled logic programs. In parallel tabling, not only the answers found for the query goal may not be valid, but also answers found for tabled predicates may be invalidated. The problem here is even more serious because to achieve an efficient implementation it is required to have the set of valid tabled answers released as soon as possible. To deal with this, we propose a strategy to deliver tabled answers as soon as it is found that they are safe from being pruned, and present its implementation in the OPTYap parallel tabling system.

1 Introduction

Logic programming is a programming paradigm based on Horn Clause Logic, a subset of First Order Logic. Given a theory (or program) and a query, execution of logic programs uses a simple theorem prover that performs refutation in order to search for alternative ways to satisfy the query. Prolog implements a refutation strategy called SLD resolution. Further, subgoals in a query are always solved from left to right, and that clauses that match a subgoal are always applied in the textual order as they appear in the program.

In order to make Prolog an useful programming language, Prolog designers were forced to introduce features not found within First Order Logic. One such feature is the cut operator. The cut operator adds a limited form of control to the execution by pruning alternatives from the computation. Cut is an asymmetric pruning operator because it only prunes alternatives to the right. Some Prolog systems also implement symmetric pruning operators, with a generic name of *commit*. In practice, pruning operators are almost always required when developing actual programs, because they allow programmers to reduce the search space and thus discard unnecessary computation.

Because their semantics are purely operational, pruning operators cause difficulties when considering alternative execution strategies for logic programs. The implementation of or-parallel systems is one example [1–4]. Namely, it has been observed that the presence of pruning operators during parallel execution introduces the problem of *speculative computations*. Ciepielewski defines speculative computations as *work which would not be done in a system with one processor* [5]. Alternatives picked for parallel execution, may later be pruned away by a cut. Earlier execution of such computations results in wasted effort when compared to sequential execution.

Pruning operators also raise questions in the context of tabling based execution models for Prolog. The basic idea behind tabling is straightforward: programs are evaluated by storing newly found answers of current subgoals in an appropriate data space, called the *table space*. New calls to a predicate check this table to verify whether they are repeated. If they are, answers are recalled from the table instead of the call being re-evaluated against the program clauses.

We can consider two types of cut operations in a tabling environment: cuts that do not prune alternatives in tabled predicates – *inner cut* operations, and cuts that prune alternatives in tabled predicates – *outer cut* operations. Inner cuts can be easily implemented in sequential systems. On the other hand, because tabling intrinsically changes the left-to-right semantics of Prolog, outer cuts present major difficulties, both in terms of semantics and of implementation.

In this work we address the problem of how to do inner pruning on systems that combine tabling with or-parallelism. Our interest stems from our work in the OPTYap system [6], to our knowledge the first available system that can exploit parallelism from tabled programs. Our experience has shown that many applications do require support for inner pruning. In contrast, outer pruning is not widely used in current tabling systems. Unfortunately, new problems arise even when performing inner pruning in parallel systems. Namely, speculative answers found for tabled predicates may later be invalidated. In the worst case, tabling such speculative answers may allow them to be consumed elsewhere in the tree, generating in turn more speculative computation and eventually cause wrong answers to occur. Answers for tabled predicates *can only be tabled when they are safe from being pruned*. On the other hand, finding and consuming answers is the natural way to get a tabled computation going forward. Delaying the consumption of valid answers too much may compromise such flow. Therefore, tabled answers *should be released as soon as it is found that they are not speculative*.

The main contribution of this paper is a design that allows the correct and efficient implementation of inner pruning in an or-parallel tabling system. To do so, we generalise Ali and Karlsson cut scheme [3], which prunes useless work as early as possible, to tabling systems. Our design allows speculative answers to be stored in advance into the table, but its availability is delayed. Answers will only be made available when proved to be not speculative.

The remainder of the paper is organised as follows. First, we discuss speculative computations in or-parallel systems and introduce the cut scheme currently implemented in OPTYap. Next, we discuss the problems arising with speculative tabled computations. Initially, we introduce the basic tabling definitions and the inner and outer cuts operations. After that, we present the support actually implemented in OPTYap to deal with speculative tabled computations. We end by outlining some conclusions.

2 Cut within the Or-Parallel Environment

Cut is a system built-in predicate that is represented by the symbol "!". Its execution results in pruning all the alternatives to the right of the current branch up to the scope of the cut. In a sequential system, cut only prunes alternatives whose exploitation has not been started yet. This does not hold for or-parallel systems, as cut can prune alternatives that are being exploited by other workers or that have already been completely exploited. Therefore, the cut semantics in a parallel environment introduces new problems. First, a pruning operation cannot always be completely performed if the branch executing the cut is not leftmost, because the operation itself may be pruned by the execution of other pruning operation in a branch to the left. Similarly, an answer for the query goal in a non-leftmost branch may not be valid. Last, when pruning we should stop the workers exploiting the pruned branches.

Ali showed that speculative computations can be completely banned from a parallel system if proper rules are applied [1]. However, such rules severely restrict parallelism. Hence, most parallel systems allow speculative computations. Speculative computations can be controlled more or less tightly. Ideally, we would prune all computations as soon as they become useless. In practice, deciding if a computation is still speculative or already useless can be quite complex when nested cuts with intersecting scopes are considered. We next discuss how cut executes in OPTYap (later we will discuss how cut affects the table).

2.1 Cut in OPTYap

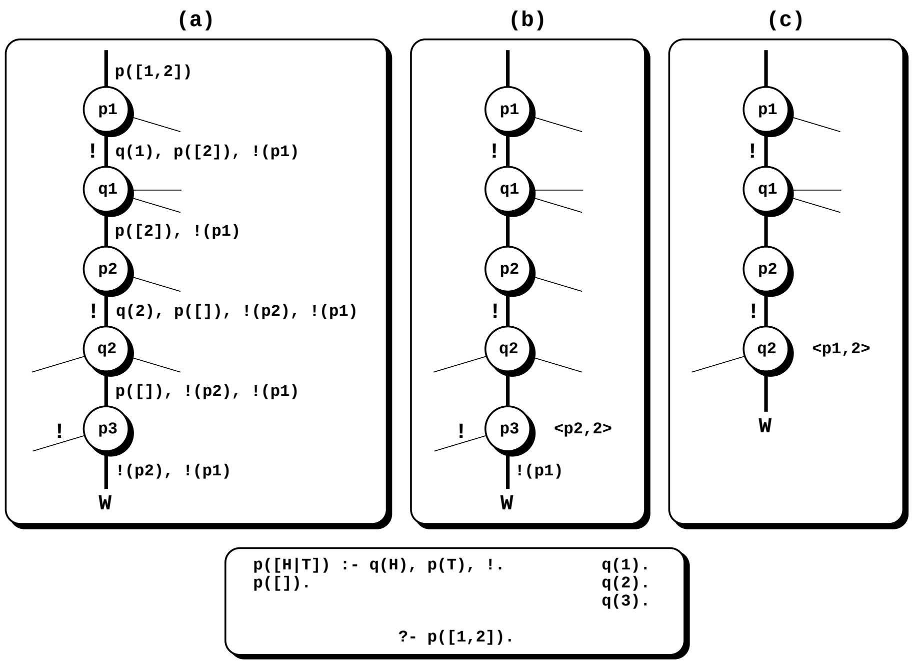
The OPTYap system builds on the or-parallel system YapOr [7] and on the tabling engine YapTab [8]. YapOr is based on the environment copying model for shared memory machines [9]. YapTab is a sequential tabling engine that extends Yap's execution model to support tabled evaluation for definite programs. YapTab's design is largely based on the ground-breaking XSB logic programming system [10], which implements the SLG-WAM [11]. OPTYap's execution model considers tabling as the base component of the system. Each computational worker behaves as a full sequential tabling engine. The or-parallel component of the system is triggered to allow synchronised access to the shared part of the search space or to schedule work.

OPTYap currently implements a cut scheme based on the ideas presented by Ali and Karlsson [3], designed to prune useless work as early as possible. The guiding rule is: *we cannot prune branches that would not be pruned if our own branch will be pruned by a branch to the left.* Thus, a worker executing cut must go up in the tree until it reaches either the *scope of the cut*, or, a node with *workers executing in branches to the left.* A worker may not be able to complete a cut if there are workers in branches to the left, because such workers can themselves prune the current cut. Such incomplete cuts are called *left pending.* In OPTYap, a cut is left pending on the youngest node $\mathcal{N}$ that has left branches. A pending cut can only be resumed when all workers to the left backtrack to $\mathcal{N}$. It will then be the responsibility of the last worker backtracking to $\mathcal{N}$ to continue the execution of the pending cut.

While going up, a worker may also find workers in branches to the right. If so, it sends them a signal informing that their branches have been pruned. Such workers must backtrack to the shared part of the tree and start searching for new work. Note that even if a cut is left pending in a node $\mathcal{N}$, there may be branches, older than $\mathcal{N}$, that correspond to useless work. OPTYap prunes these branches immediately. To illustrate how these branches can be detected we present in Fig. 1 a small example taken from [3]. For simplicity, the example ignores indexing and assumes that a node is always allocated for predicates defined by more than one clause. To better understand the example, we index the repeated calls to the same predicate by call order. For instance, the node representing the first call to predicate `p` is referred as $\mathtt{p}_1$, the second as $\mathtt{p}_2$ and successively. We also write $\mathtt{p}_\mathtt{n}^{(\mathtt{i})}$ to denote the *ith* alternative of node $\mathtt{p}_\mathtt{n}$. Note also that we use the symbol ! to mark the alternatives corresponding to clauses with cuts.

Figure 1(a) shows the initial configuration, where a worker $\mathcal{W}$ is computing the branch corresponding to $[\mathtt{p}_1^{(1)},\ \mathtt{q}_1^{(1)},\ \mathtt{p}_2^{(1)},\ \mathtt{q}_2^{(2)},\ \mathtt{p}_3^{(2)}]$. Its current goal is "$!(\mathtt{p}_2), !(\mathtt{p}_1)$", where $!(\mathtt{p}_2)$ means a cut with the scope $\mathtt{p}_2$ and $!(\mathtt{p}_1)$ means a cut with the scope $\mathtt{p}_1$. There are only two branches to the left, corresponding to alternatives $\mathtt{p}_3^{(1)}$ and ${\mathtt{q}_2}^{(1)}$. If there are workers within alternative ${\mathtt{p}_3}^{(1)}$, then $\mathcal{W}$ cannot execute any pruning at all because ${\mathtt{p}_3}^{(1)}$ is marked as containing cuts. A potential execution of a pruning operation in ${\mathtt{p}_3}^{(1)}$ will invalidate any cut executed in ${\mathtt{p}_3}^{(2)}$ by $\mathcal{W}$. Therefore, $\mathcal{W}$ saves a cut marker in $\mathtt{p}_3$ to indicate a pending cut operation (Fig. 1(b)). A cut marker is a two field data structure containing information about the scope of the cut and about the alternative of the node which executed the cut.

Let's now assume that there are no workers in alternative ${\mathtt{p}_3}^{(1)}$, but there are in alternative ${\mathtt{q}_2}^{(1)}$. Alternative ${\mathtt{q}_2}^{(1)}$ is not marked as containing cuts, but the continuation of $\mathtt{q}_2$ contains two pruning operations, $!(\mathtt{p}_2)$ and $!(\mathtt{p}_1)$. The worker $\mathcal{W}$ first executes $!(\mathtt{p}_2)$ in order to prune ${\mathtt{q}_2}^{(3)}$ and ${\mathtt{p}_2}^{(2)}$. This is a safe pruning operation because any pruning from ${\mathtt{q}_2}^{(1)}$ will also prune ${\mathtt{q}_2}^{(3)}$ and ${\mathtt{p}_2}^{(2)}$. At the same time $\mathcal{W}$ stores a cut marker in $\mathtt{q}_2$ to signal the pruning operation done. As we will see, for such cases, the cut marker is used to prevent unsafe future pruning operations from the same branch. Consider the continuation of the situation, $\mathcal{W}$



```
p([H|T]) :- q(H), p(T), !.        q(1).
p([]).                             q(2).
                                   q(3).

              ?- p([1,2]).
```

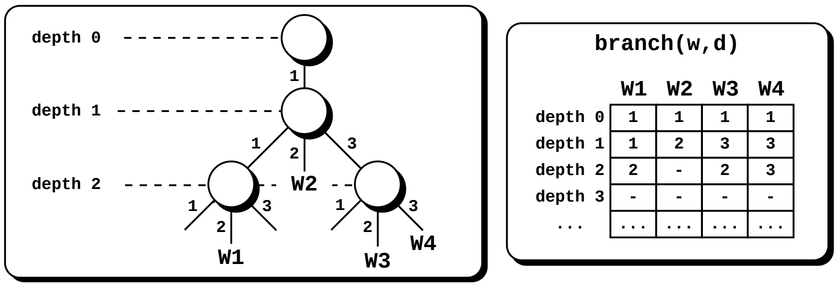
Fig. 1. Pruning in the or-parallel environment

tries to execute !($\mathtt{p_1}$) in order to prune $\mathtt{q_1}^{(2)}$, $\mathtt{q_1}^{(3)}$ and $\mathtt{p_1}^{(2)}$. However, this is a dangerous operation. A worker in $\mathtt{q_2}^{(1)}$ may execute the previous pruning operation, !($\mathtt{p_2}$), pruning $\mathcal{W}$'s branch but not $\mathtt{q_1}^{(2)}$, $\mathtt{q_1}^{(3)}$ or $\mathtt{p_1}^{(2)}$. Hence, there is no guarantee that the second pruning, !($\mathtt{p_1}$), is safe. The cut marker stored in $\mathtt{q_2}$ is a warning that this possibility exists. So, instead of doing pruning immediately, $\mathcal{W}$ updates the cut marker stored in $\mathtt{q_2}$ to indicate the new pending cut operation (Fig. 1(c)).

2.2 Tree Representation

To represent the shared part of the search tree, OPTYap follows the Muse approach [9] and uses *or-frames*. When sharing work, an or-frame is added per choice point being shared, in such a way that the complete set of or-frames form a tree that represents the shared part of the search tree. Or-frames are used to synchronise access to the unexploited alternatives in a shared choice point, and to store scheduling data. By default, an or-frame contains the following fields: the `OrFr_lock` field supports a busy-wait locking mutex mechanism that guarantees atomic updates to the or-frame data; the `OrFr_alt` field stores the pointer to the next unexploited alternative in the choice point; the `OrFr_members` field is a bitmap that stores the set of workers sharing the choice point; the `OrFr_node` field is a back pointer to the correspondent choice point; and the `OrFr_next` field is a pointer to the parent or-frame on the current branch.

Identifying workers on left branches or checking whether a branch is leftmost requires a mechanism to represent the relative positions of workers in the search tree. Our implementation uses a *branch*() matrix, where each entry $branch(w, d)$ corresponds to the alternative taken by worker w in the shared node with depth d of its current branch. Figure 2 shows a small example that clarifies the correspondence between a particular search tree and its matrix representation. Note that we only need to represent the shared part of a search tree in the matrix. This is due to the fact that the position of each worker in the private part of the search tree is not relevant when computing relative positions.



	W1	W2	W3	W4
depth 0	1	1	1	1
depth 1	1	2	3	3
depth 2	2	-	2	3
depth 3	-	-	-	-
...	...	...	...	...

Fig. 2. Search tree representation

To correctly consult or update the branch matrix, we need to know the depth of each shared node. We thus introduced a new data field in the or-frame data structure, the `OrFr_depth` field, that holds the depth of the corresponding node. By using the `OrFr_depth` field and the `OrFr_members` bitmap of each or-frame to consult the branch matrix, we can easily identify the workers in a node that are in branches at the left or at the right the current branch of a given worker.

Let us suppose that a worker $\mathcal{W}$ wants to check whether it is leftmost or at which node it ceases from being leftmost. $\mathcal{W}$ should start from the youngest shared node $\mathcal{N}$ on its branch, read the `OrFr_members` bitmap from the or-frame associated with $\mathcal{N}$ to determine the workers sharing the node, and investigate the branch matrix to determine the alternative number taken by each worker sharing $\mathcal{N}$. If $\mathcal{W}$ finds an alternative number less than its own, then $\mathcal{W}$ is not leftmost. Otherwise, $\mathcal{W}$ is leftmost in $\mathcal{N}$ and will repeat the same procedure at the next upper node on branch and so on until reaching the root node or a node where it is not leftmost.

2.3 Pending Answers

OPTYap also builds on a mechanism originally designed for a problem in or-parallel systems: an answer for the query goal may not be valid, if the branch where the answer was found may be pruned. At the end of the computation, only valid answers should be seen.

OPTYap addresses this problem by storing a new answer in the youngest node where the current branch is not leftmost. A new data field was therefore introduced in the or-frame data structure, the `OrFr_qg_answers` field. This field allows access to the set of pending answers stored in the corresponding node. Also, new data structures store the pending answers that are being found for the query goal in hand. Figure 3 details the data structures used to efficiently keep track of pending answers. Answers from the same branch are grouped into a common top data structure. The top data structures are organised by reverse branch order. This organisation simplifies the pruning of answers that became invalid in consequence of a cut operation to the left.

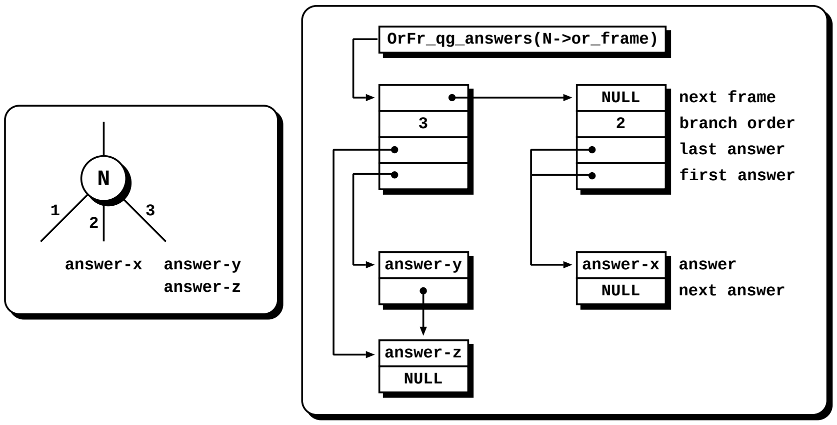


Fig. 3. Dealing with pending answers

When a node $\mathcal{N}$ is fully exploited and its corresponding or-frame is being deallocated, the whole set of pending answers stored in $\mathcal{N}$ can be easily linked together and moved to the next node where the current branch is not leftmost. At the end, the set of answers stored in the root node are the set of valid answers for the given query goal.

3 Cut within the Or-Parallel Tabling Environment

Extending the or-parallel system to include tabling introduces further complexity into cut's semantics. Dealing with speculative tabled computations and guaranteeing the correctness of tabling semantics, without compromising the performance of the or-parallel tabling system, requires very efficient implementation mechanisms. In this section, we present the OPTYap's approach. Before we start, we provide a brief overview of the basic tabling definitions and distinguish the two types of cut operations in a tabling environment: inner cuts and outer cuts.

3.1 Basic Tabling Definitions

Tabling is about storing intermediate answers for subgoals so that they can be reused when a repeated subgoal appears. Whenever a tabled subgoal S is first called, an entry for S is allocated in the table space. This entry will collect all the answers found for S. Repeated calls to *variants* of S are resolved by consuming the answers already stored in the table. Meanwhile, as new answers are generated, they are inserted into the table and returned to all variant subgoals. Within this model, the nodes in the search space are classified as either *generator nodes*, corresponding to first calls to tabled subgoals, *consumer nodes*, corresponding to variant calls to tabled subgoals, and *interior nodes*, corresponding to non-tabled subgoals.

Tabling evaluation has four main types of operations for definite programs. The *tabled subgoal call* operation checks if the subgoal is in the table and if not, inserts it and allocates a new generator node. Otherwise, allocates a consumer node and starts consuming the available answers. The *new answer* operation verifies whether a newly generated answer is already in the table, and if not, inserts it. The *answer resolution* operation consumes the next unconsumed answer from the table, if any. The *completion* operation determines whether a tabled subgoal is completely evaluated, and if not, schedules a possible resolution to continue the execution.

The table space can be accessed in different ways: to look up if a subgoal is in the table, and if not insert it; to verify whether a newly found answer is already in the table, and if not insert it; and to pick up answers to consumer nodes. Hence, a correct design of the algorithms to access and manipulate the table data is a critical issue to obtain an efficient implementation. Our implementation uses tries as the basis for tables, as proposed by Ramakrishnan *et al.* in [12].

Figure 4 shows the general table structure for a tabled predicate. Table lookup starts from the *table entry* data structure. Each table predicate has one such structure, which is allocated at compilation time. Calls to the predicate will always access the table starting from this point.

The table entry points to a tree of trie nodes, the *subgoal trie structure*. More precisely, each different call to the tabled predicate in hand corresponds to a unique path through the subgoal trie structure. Such a path always starts from the table entry, follows a sequence of subgoal trie data units, the *subgoal trie nodes*, and terminates at a leaf data structure, the *subgoal frame*.

Each subgoal frame stores information about the subgoal, namely an entry point to its *answer trie structure*. Each unique path through the answer trie data units, the *answer trie nodes*, corresponds to a different answer to the entry subgoal. To obtain the set of available answers for a tabled subgoal, the leaf answer nodes are chained in a linked list in insertion time order, so that we can recover answers in the same order they were inserted. The subgoal frame points to the first and last answer in this list. Thus, a consumer node only needs to point at the leaf node for its last consumed answer, and consumes more answers just by following the chain. To load an answer, the trie nodes are traversed in bottom-up order and the answer is reconstructed.

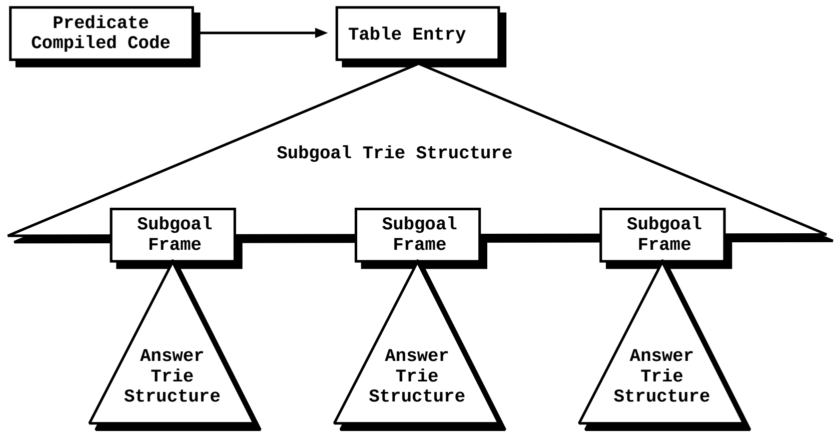
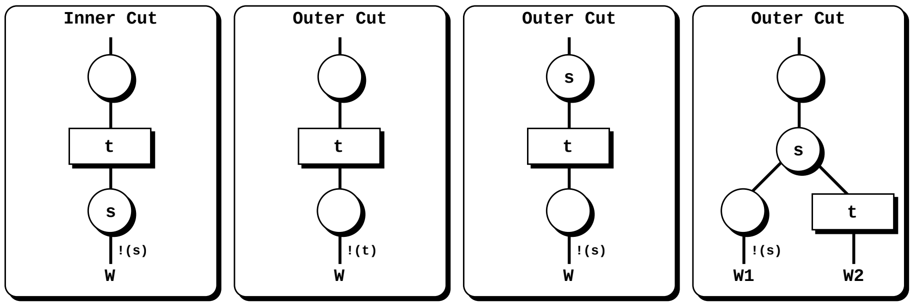


Fig. 4. Using tries to organise the table space

3.2 Inner and Outer Cut Operations

We consider two types of pruning in a tabling environment: cuts that do not prune alternatives in tabled predicates – *inner cut* operations, and cuts that prune alternatives in tabled predicates – *outer cut* operations. In Fig. 5 we illustrate four different situations corresponding to inner and outer cut operations. Below each illustration we present a block of Prolog code that may lead to such situations. For simplicity, we assume that `t` is the unique tabled predicate defined and that the "`...`" parts do not include `t`. Note that the rightmost situation only occurs if a parallel tabling environment is considered, as otherwise `t` will only be called if the cut operation in the first alternative of `s` is not executed.



```
:- table t/0.

t :- ..., s, ...
t :- ...

s :- ..., !, ...
s :- ...
```

```
:- table t/0.

t :- ..., !, ...
t :- ...
```

```
:- table t/0.

s :- ..., t, ..., !, ...
s :- ...

t :- ...
t :- ...
```

```
:- table t/0.

s :- ..., !, ...
s :- ..., t, ...

t :- ...
t :- ...
```

Fig. 5. The two types of cut operations in a tabling environment

Cut semantics for outer cut operations is still an open problem. A major problem is that of pruning generator nodes. Pruning generator nodes cancels its further completion and puts the table space in an inconsistent state. For sequential tabling, a simple approach is to delete the whole table data structures related with the pruned subgoal and recompute it from the beginning when it reappears. This can be safely done because when a generator is pruned all variant consumers are also pruned. On the other hand, for parallel tabling, it is possible that generators will execute earlier, and in a different branch than in sequential execution. In fact, different workers may execute the generator and the consumer goals. Workers may have consumer nodes while not having the corresponding generator nodes in their branches. Conversely, the owner of a generator node can have consumer nodes being executed by several different workers.

The intricate dependencies in a parallel tabled evaluation makes pruning a very complex problem. A possible solution to this problem can be moving the generator's role to a non-pruned dependent consumer node, if any, in order to allow further exploitation of the generator's unexploited branches. Such a solution will require that the other non-pruned consumer nodes recompute and update their dependencies relatively to the new generator node. Otherwise, if all dependent consumer nodes are also pruned, we can suspend the execution stacks and the table data structures of the pruned subgoal and try to resume them when the next variant call takes place. Further research is still necessary in order to study the combination of pruning and parallel tabling. Currently, OPTYap still does not support outer cut operations and for such cases execution is aborted. Outer cut operations are detected when a worker moves up in the tree either because it is executing a cut operation or it has received a signal informing that its branch have been pruned away by another worker.

3.3 Detecting Speculative Tabled Answers

As mentioned before, a main goal in the implementation of speculative tabling is to allow storing valid answers immediately. We would like to maintain the same performance as for the programs without cut operators. In this subsection, we introduce and describe the data structures and implementation extensions required to efficiently detect if a tabled answer is speculative or not.

We introduced a global bitmap register named `GLOBAL_pruning_workers` to keep track of the workers that are executing branches that contain cut operators and that, in consequence, may prune the current goal. Additionally, each worker maintains a local register, `LOCAL_safe_scope`, that references the youngest node that cannot be pruned by any pruning operation executed by itself.

The correct manipulation of these new registers is achieved by introducing a new instruction `clause_with_cuts` to mark the blocks of code that include cut instructions. During compilation, the code generated for the clauses containing cut operators is extended to include the `clause_with_cuts` instruction so that it is the first instruction to be executed for such clauses. When a worker loads a `clause_with_cuts` instruction, it executes the `clause_with_cuts()` procedure.

Figure 6 details the pseudo-code that implements the `clause_with_cuts()` procedure. It sets the worker's bit in the `GLOBAL_pruning_workers` register and, if the `LOCAL_safe_scope` register is younger than the current node, it updates the `LOCAL_safe_scope` register to refer to the current node. The current node is the resulting top node if a pruning operation takes place in the clause in hand.

```
clause_with_cuts() {
   if (LOCAL_safe_scope == NULL) {                      // first time here
      insert_into_bitmap(GLOBAL_pruning_workers, WORKER_id)
      LOCAL_safe_scope = B  // B is a pointer to the current choice point
   } else if (LOCAL_safe_scope is younger than B) {
      LOCAL_safe_scope = B
   }
}
```

Fig. 6. Pseudo-code for `clause_with_cuts()`

When a worker finds a new answer for a tabled subgoal, it first inserts it into the table space and then checks if the answer is safe from being pruned. When this is the case, the answer is inserted at the end of the list of available answers, as usual. Otherwise, if it is found that the answer can be pruned by another worker, its availability is delayed. Figure 7 presents the pseudo-code that implements the checking procedure.

```
speculative_tabled_answer(generator node G) {    // G is the generator...
   prune_wks = GLOBAL_pruning_workers  // ...for the answer being checked
   delete_from_bitmap(prune_wks, WORKER_id)
   if (prune_wks is not empty) {         // there are workers that may...
      or_fr = youngest_or_frame()        // ...execute pruning operations
      depth = OrFr_depth(or_fr)
      scope_depth = OrFr_depth(G->or_frame)
      while (depth > scope_depth) {           // check the branch till...
         alt_number = branch(WORKER_id, depth)        // ...the generator
         for (w = 0; w < number_workers; w++) {
            if (w is in prune_wks && w is in OrFr_members(or_fr) &&
                branch(w, depth) < alt_number &&
                OrFr_node(or_fr) is younger than LOCAL_safe_scope(w))
               return or_fr       // the answer can be pruned by worker w
         }
         or_fr = OrFr_next(or_fr)
         depth = OrFr_depth(or_fr)
      }
   }
   return NULL                    // the answer is safe from being pruned
}
```

Fig. 7. Pseudo-code for `speculative_tabled_answer()`

The procedure starts by determining if there are workers that may execute pruning operations. If so, it checks the safeness of the branch where the tabled answer was found. The branch only needs to be checked until the corresponding generator node, as otherwise it would be an outer cut operation. A branch is found to be safe if it is leftmost, or if the workers in the branches to the left cannot prune it. If it is found that the answer being checked can be speculative, the procedure returns the or-frame that corresponds to the youngest node where the answer can be pruned by a worker in a left branch. That or-frame is where the answer should be *left pending*. Otherwise, if it is found that the answer is safe, the procedure returns `NULL`.

3.4 Pending Tabled Answers

Tabled answers are inserted in advance into the table space. However, if a tabled answer is found to be speculative, its insertion in the list of available answers is delayed and the answer is left pending. This prevents unsafe answers to be consumed elsewhere in the tree. Only when it is found that a pending answer is safe from being pruned, it is released as a valid answer and inserted at the end of the list of available answers for the subgoal. Dealing with pending tabled answers requires efficient support to allow that the operations of pruning or releasing pending answers are efficiently performed.

Remember that speculative tabled answers are left pending in nodes. To allow access to the set of pending answers for a node, a new data field was introduced in the or-frame data structure, the `OrFr_tg_answers` field. New data structures were also introduced to efficiently keep track of the pending answers being found for the several tabled subgoals. Figure 8 details that data structure organisation.

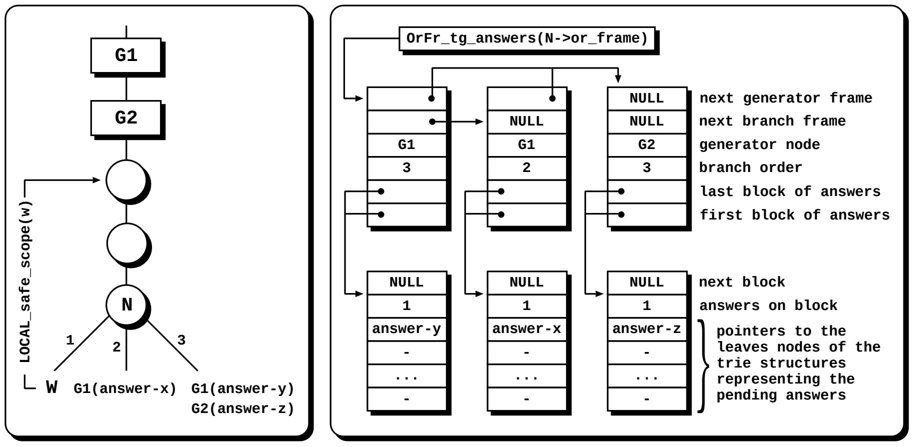


Fig. 8. Dealing with pending tabled answers

The figure shows a situation where three tabled answers, `answer-x`, `answer-y` and `answer-z`, were found to be speculative and therefore have all been left pending in a common node $\mathcal{N}$. $\mathcal{N}$ is the youngest node where a worker in a left branch, $\mathcal{W}$ in the figure, holds a `LOCAL_safe_scope` register pointing to a node older than $\mathcal{N}$.

Pending answers found for the same subgoal and from the same branch are addressed by a common top data structure. As the answers in the figure were found in different subgoal/branch pairs, three top data structures were required. `answer-x`, `answer-y` and `answer-z` were found, respectively, in branches 2, 3 and 3 for the subgoals corresponding to generator nodes $\mathcal{G}_1$, $\mathcal{G}_1$ and $\mathcal{G}_2$. These data structures are organised in older to younger generator order and by reverse branch order when they are for the same generator. Hence, each data structure contains two types of pointers to follow the chain of structures, one points to the structure that corresponds to the next younger generator node, while the other points to the structure that corresponds to the next branch within the same generator.

Blocks of answers address the set of pending answers for a subgoal/branch pair. Each block points to a fixed number of answers. By linking the blocks we can have a large number of answers for the same subgoal/branch pair. Note that the block data structure does not hold the representation of a pending answer, only a pointer to the leaf node of the answer trie structure representing the pending answer. As we will see, with this simple scheme, we can easily differentiate between occurrences of the same speculative answer in different branches. Figure 9 shows the procedure that OPTYap executes when a tabled answer is found.

```
tabled_answer(answer A, generator node G) {      // G is the generator...
   sf = subgoal_frame(G)                           // ...for the answer A
   leaf_node = insert_into_table_space(A, sf)
   if (leaf_node is a valid answer) {
      fail()           // already in the list of available answers for sf
   } else {
      or_fr = speculative_tabled_answer(G)
      if (or_fr == NULL)          // the answer is safe from being pruned
         valid_answer(leaf_node, sf)
      else
         left_pending(leaf_node, or_fr)
   }
}
```

Fig. 9. Pseudo-code for `tabled_answer()`

The procedure starts by inserting the answer in the table space. Then, it verifies if the answer is already tabled as a valid answer and, if so, the execution fails as usual. Otherwise, it checks if the answer is safe from being pruned. Being this the case, the answer is tabled as a valid answer. Otherwise, it is left pending.

Suppose now that we have an answer $\mathcal{A}$ left pending in a node $\mathcal{N}$ and that a new occurrence of $\mathcal{A}$ is found elsewhere. Two situations may happen: the new occurrence of $\mathcal{A}$ is also speculative or it is safe from being pruned. In the first case, $\mathcal{A}$ is left pending in a node of the current branch. This is necessary because there is no way to know beforehand in which branch $\mathcal{A}$ will be proved first to be not speculative, if in any. In the second case, $\mathcal{A}$ is released as a valid answer and inserted in the list of available answers for the subgoal in hand. Note that in this case, $\mathcal{A}$ still remains left pending in node $\mathcal{N}$. In any case, $\mathcal{A}$ is only represented once in the table space.

With this scheme, OPTYap implements the following algorithm in order to release answers as soon as possible: the last worker $\mathcal{W}$ leaving a node $\mathcal{N}$ with pending tabled answers, determines the next node $\mathcal{M}$ on its branch that can be pruned by a worker to the left. The pending answers from $\mathcal{N}$ that correspond to generator nodes equal or younger than $\mathcal{M}$ are made available (if an answer is already valid, nothing is done), while the remaining are moved from $\mathcal{N}$ to $\mathcal{M}$. Note that $\mathcal{W}$ only needs to check for the existence of $\mathcal{M}$ up to the oldest generator node for the pending answers stored in $\mathcal{N}$. To simplify finding the oldest generator node we organised the top data structures in older to younger generator order (please see Fig. 8).

On the other hand, when a node $\mathcal{N}$ is pruned, its pending tabled answers can be in one of three situations: only left pending in $\mathcal{N}$; also left pending in other nodes; or already valid answers. Note that for all situations no interaction with the table space is needed and $\mathcal{N}$ can simply be pruned away. Even for the first situation, we may keep the answers in the table and wait until completion, as in the meantime such answers can still be generated again in other branches. So, only when a subgoal is completed evaluated, it is required that the answer trie nodes representing speculative answers are removed from the table. As this requires traversing the whole answer trie structure, for simplicity and efficiency, this is only done in the first call to the tabled subgoal after it has been completed.

4 Conclusions

In this paper we discussed the management of speculative computations in or-parallel tabled logic programs. Our approach deals with inner pruning at a first step and we address speculative tabled computations by delaying the point at which their answers are made available in the table. With this support, OPTYap is now able to execute a wider range of applications without introducing significant overheads (less than 1%) for applications without cuts.

Support for outer cuts is a delicate issue. To our knowledge, the first proposal on outer cuts for sequential tabling was presented by Guo and Gupta in [13]. They argue that cuts in tabling systems are most naturally interpreted as a commit, and they define the cut operator in terms of the operational semantics of their tabling strategy [14], which is based on recomputation of so-called looping alternatives. In more recent work, Castro and Warren propose the demand-based *once* pruning operator [15], whose semantics are independent of the operational semantics for tabling, but which does not fully support cut. We believe that a complete design for outer cut operations in sequential tabling is still an open and, arguably, a controversial problem.

To fully support pruning in parallel tabling, further work is also required. We need to do it *correctly*, that is, in such a way that the system will not break but instead produce sensible answers according to the proposed sequential semantics, and *well*, that is, allow useful pruning with good performance.

Acknowledgments

This work has been partially supported by APRIL (POSI/SRI/40749/2001) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSI.

References

1. Ali, K.: A Method for Implementing Cut in Parallel Execution of Prolog. In: Proceedings of the International Logic Programming Symposium, IEEE Computer Society Press (1987) 449–456
2. Hausman, B.: Pruning and Speculative Work in OR-Parallel PROLOG. PhD thesis, The Royal Institute of Technology (1990)
3. Ali, K., Karlsson, R.: Scheduling Speculative Work in MUSE and Performance Results. International Journal of Parallel Programming **21** (1992) 449–476
4. Beaumont, A., Warren, D.H.D.: Scheduling Speculative Work in Or-Parallel Prolog Systems. In: Proceedings of the 10th International Conference on Logic Programming, The MIT Press (1993) 135–149
5. Ciepielewski, A.: Scheduling in Or-parallel Prolog Systems: Survey and Open Problems. International Journal of Parallel Programming **20** (1991) 421–451
6. Rocha, R., Silva, F., Santos Costa, V.: On a Tabling Engine that Can Exploit Or-Parallelism. In: Proceedings of the 17th International Conference on Logic Programming. Number 2237 in LNCS, Springer-Verlag (2001) 43–58
7. Rocha, R., Silva, F., Santos Costa, V.: YapOr: an Or-Parallel Prolog System Based on Environment Copying. In: Proceedings of the 9th Portuguese Conference on Artificial Intelligence. Number 1695 in LNAI, Springer-Verlag (1999) 178–192
8. Rocha, R., Silva, F., Santos Costa, V.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Proceedings of the 2nd Conference on Tabulation in Parsing and Deduction. (2000) 77–87
9. Ali, K., Karlsson, R.: Full Prolog and Scheduling OR-Parallelism in Muse. International Journal of Parallel Programming **19** (1990) 445–475
10. Sagonas, K., Swift, T., Warren, D.S.: XSB as an Efficient Deductive Database Engine. In: Proceedings of the ACM SIGMOD International Conference on the Management of Data, ACM Press (1994) 442–453
11. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems **20** (1998) 586–634
12. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38** (1999) 31–54
13. Guo, H.F., Gupta, G.: Cuts in Tabled Logic Programming. In: Proceedings of the Colloquium on Implementation of Constraint and LOgic Programming Systems. (2002)
14. Guo, H.F., Gupta, G.: A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In: Proceedings of the 17th International Conference on Logic Programming. Number 2237 in LNCS, Springer-Verlag (2001) 181–196
15. Castro, L.F., Warren, D.S.: Approximate Pruning in Tabled Logic Programming. In: Proceedings of the 12th European Symposium on Programming. Volume 2618 of LNCS., Springer Verlag (2003) 69–83