Or-Parallelism within Tabling

Ricardo Rocha Fernando Silva Vítor Santos Costa[*]

DCC-FC & LIACC, University of Porto,
Rua do Campo Alegre, 823 - 4150 Porto, Portugal
{ricroc,fds,vsc}@ncc.up.pt

Abstract. One important advantage of logic programming is that it allows the implicit exploitation of parallelism. Towards this goal, we suggest that or-parallelism can be efficiently exploited in tabling systems and propose two alternative approaches, Or-Parallelism within Tabling (OPT) and Tabling within Or-Parallelism (TOP).
We concentrate on the fundamental concepts of an environment copying based model to implement the OPT approach and introduce the data structures and algorithms necessary to extend the YapOr Or-Parallel system, in order to obtain a parallel tabling system.
Keywords: Parallel Logic Programming, Or-Parallelism, Tabling.

1 Introduction

Prolog is an extremely popular and powerful logic programming language. Prolog execution is based on SLD resolution for Horn clauses. This strategy allows efficient implementation, but suffers from fundamental limitations, such as in dealing with infinite loops and redundant subcomputations. SLG resolution [3] is a tabling based method of resolution that overcomes some limitations of traditional Prolog. The method evaluates programs by storing newly found answers of current subgoals in a table. The method then uses this table to verify for repeated subgoals. Whenever such a repeated subgoal is found, the subgoal's answers are recalled from the table instead of being resolved against the program clauses. SLG resolution can thus reduce the search space for logic programs and in fact it has been proven that it can avoid looping and thus terminate for all programs that construct bounded depth terms. The XSB-Prolog system [11] was the first Prolog system to implement SLG resolution.

One important advantage of logic programming is that it allows the implicit exploitation of parallelism. This is true for SLD-based systems, and should also apply for SLG-based systems. A first proposal on how to exploit implicit parallelism in tabling systems was Freire's table-parallelism [6]. In this model, each tabled subgoal is associated with a new computational thread, a *generator thread*, that will produce and add the answers into the table. Threads that call a tabled

[*] The first author is thankful to PRAXIS and Fundação para a Ciência e Tecnologia for their financial support. This work was partially funded by the Melodia project (JNICT PBIC/C/TIT/2495/95).

subgoal will asynchronously consume answers as they are added to the table. This model is limitative in that it restricts exploitation of parallelism to just one implicit form of parallelism present in logic programs. Ideally, we would like to exploit maximum parallelism and take maximum advantage of current technology for parallel and tabling systems.

We observe that tabling is still about exploiting alternatives to find solutions for goals, and that or-parallel systems have precisely been designed to achieve this goal efficiently. We therefore propose two computational models, the OPT and TOP models, that combine or-parallelism and tabling by considering all open alternatives to subgoals as being amenable to parallel exploitation, be they from tabled or non-tabled subgoals. The OPT approach considers that tabling is the base component of the system, this is, each worker can be considered like a full sequential tabling engine. The or-parallel component of the system is only triggered when a worker runs out of alternatives to exploit. The TOP model unifies or-parallel suspension and suspension due to tabling. A suspended subgoal can wake up for several reasons, such as new alternatives having been found for the subgoal, the subgoal becoming leftmost, or just for lack of non speculative work in the search tree. The TOP approach considers that each worker can be considered like a sequential WAM engine, hence only managing a logical branch of the search tree, and not several branches.

In this paper we compare both models and focus on the design and implementation of the OPT model for combining or-parallelism and tabling. We chose the OPT model mainly because we believe it gives the highest degree of orthogonality between or-parallelism and tabling, thus simplifying initial implementation issues. The implementation framework is based on YapOr [9], an or-parallel system that extends Yap's sequential execution model to exploit implicit or-parallelism in Prolog programs. YapOr is based on the environment copy model, as first implemented in Muse [1]. We describe the main issues that arise in supporting tabling and its parallelization through copying. The implementation of tabling is largely based on the XSB engine, the SLG-WAM, however substantial differences exist in particular on the algorithms for restoring a computation, determining the leader node and completion operation. All these and the extended data structures already take into account the support of combined tabling and or-parallelism.

In summary, we briefly introduce the implementation of tabling. Next, we discuss the fundamental issues in supporting or-parallelism for SLG resolution and propose two alternative approaches. Then, we present the new data structures and algorithms required to extend YapOr to support tabling and to allow for a combined exploitation of or-parallelism and tabling. This corresponds to the implementation work already done and terminate by outlining some conclusions.

2 Tabling Concepts and the SLG-WAM

The SLG evaluation process can be modeled by a *SLG-forest* [10]. Whenever a tabled subgoal S is called for the first time, a new tree with root S is added to the

SLG-forest. Simultaneously, an entry for S is allocated in the *table space*. This entry will collect all the answers generated 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 to S, they are inserted into the table and returned to all variant subgoals. Within this model, the nodes in the search tree are classified as either *generator* nodes, that is, first calls to tabled subgoals, *consumer* nodes, that consumer answers from the table space, and *interior* nodes, that are evaluated by the standard SLD resolution.

Space for a tree can be reclaimed when its root subgoal has been *completely evaluated*. A subgoal is said to be completely evaluated when all possible resolutions for it have been made, that is, when no more answers can be generated and the variant subgoals have consumed all the available answers. Note that a number of root subgoals may be mutually dependent, forming a *strongly connected component* (or *SCC*), and therefore can only be completed together. The completion operation is thus performed by the *leader* of the SCC, that is, by the oldest root subgoal in the SCC, when all possible resolutions have been made for all root subgoals in the SCC. Hence, in order to efficiently evaluate programs one needs an efficient and dynamic detection scheme to determine when both the subgoals in an SCC and the SCC itself have been completely evaluated.

For definite programs, tabling based evaluation has four main types of operations: (i) *Tabled Subgoal Call*, creates a generator node; (ii) *New Answer*, verifies whether a newly generated answer is already in the table, and if not, inserts it; (iii) *Answer Return*, consumes an answer from the table; and *Completion* determines whether an SCC is completely evaluated, and if not, schedules a possible resolution to continue the execution.

The implementation of tabling in XSB was attained by extending the WAM into the SLG-WAM, with minimal overhead. In short, the SLG-WAM introduces special instructions to deal with the operations above and two new memory areas: a table space, used to save the answers for tabled subgoals; and a *completion stack*, used to detect when a set of subgoals is completely evaluated.

Further, whenever a consumer node gets to a point in which it has consumed all available answers, but the correspondent tabled subgoal has not yet completed and new answers may still be generated, it must suspend its computation. In the SLG-WAM the suspension mechanism is implemented through a new set of registers, the *freeze registers*, which freeze the WAM stacks at the suspension point and prevent all data belonging to the suspended branch from being erased. A suspended consumer is resumed by restoring the registers saved in the corresponding node and by using an extension of the standard trail, the *forward trail*, to restore the bindings of the suspended branch. The resume operation is implemented by setting the `failure_continuation` field of the consumer nodes to a special `answer_return` instruction. This instruction is responsible for resuming the computation, guaranteeing that all answers are given once and just once to every variant subgoal. Through failure and backtracking to a consumer node, the `answer_return` instruction gets executed and resuming takes place.

It is upon the leader of an SCC to detect its completion. This operation is executed dynamically and must be efficiently implemented in order to minimize overheads. To achieves this, the SLG-WAM sets the `failure_continuation` field of a generator node to a special `completion` instruction whenever it resolves the last applicable program clause for the correspondent subgoal. The `completion` instruction thus ensures the total and correct evaluation of the subgoal search space. This instruction is executed through backtracking. In the default XSB scheduling strategy (*batched scheduling* [7]), it resumes the consumer node corresponding to the deeper variant subgoal with unconsumed answers. The computation will consume all the newly found answers, backtrack to other consumer nodes (higher up in the chain of consumer nodes) with unconsumed answers, and fail to the generator node, until no more answers are left to consume. At this point, if that node is the leader of its SCC, a fixpoint is reached, all dependent subgoals are completed, and the subgoal can be marked completed. Otherwise, the computation will fail back to the previous node and the fixpoint check will later be executed in an upper generator node.

3 Parallel Execution of Tabled Programs

Ideally, we would like to exploit maximum parallelism and take maximum advantage of current technology for tabling systems. We propose that all alternatives to subgoals should be amenable to parallel exploitation, be they from normal or tabled subgoals, and that or-parallel frameworks can be used as the basis to do so. This gives an unified approach with two major advantages. First, it does not restrict parallelism to tabled subgoals and, second, it can draw from the very successful body of work in implementing or-parallel systems. We believe that this approach can result in efficient models for the exploitation of parallelism in tabling-based systems. We envisage two different models to combine or-parallelism and tabling:

Or-Parallelism within Tabling (OPT) In this approach, parallelism is exploited between independent tabling engines, that share alternatives. Tabling is the base component of the system, this is, each worker can be considered a full sequential tabling engine and should spend most of its computation time exploiting the search tree involved in such an evaluation. It thus can allocate all three types of nodes, fully implement suspension of tabled subgoals, and resume subcomputations to consume newly found answers. Or-parallelism is triggered when a worker runs out of alternatives to exploit. In the OPT approach, unexploited alternatives should be made available for parallel execution, regardless of whether they originate from a generator, consumer or interior node. Therefore, parallelism can and should stem from both tabled and non-tabled subgoals.

From the viewpoint of or-parallelism, the OPT approach generalizes Warren's multi-sequential engine framework for the exploitation of or-parallelism. Or-parallelism now stems from having several engines that implement SLG-resolution, instead of implementing Prolog's SLD-resolution.

Fig. 1 gives an example of this approach for the following small program and the query `?- a(X)`. We assume two workers, `W1` and `W2`.

```
:- table a/1.            b(1).
a(X) :- a(X).            b(X) :- ...
a(X) :- b(X).            b(X) :- ...
```

Consider that worker `W1` executes the query goal. It first inserts an entry for the tabled subgoal `a(X)` into the table and creates a generator node for it. The execution of the first alternative leads to a recursive call for `a/1`, thus the worker creates a consumer node for `a/1` and backtracks. The next alternative finds a non-tabled subgoal `b/1` for which an interior node is created. The first alternative for `b/1` succeeds and an answer for `a(X)` is therefore found (`a(1)`). The worker inserts the newly found answer in the table and then starts exploiting the next alternative of `b/1`.

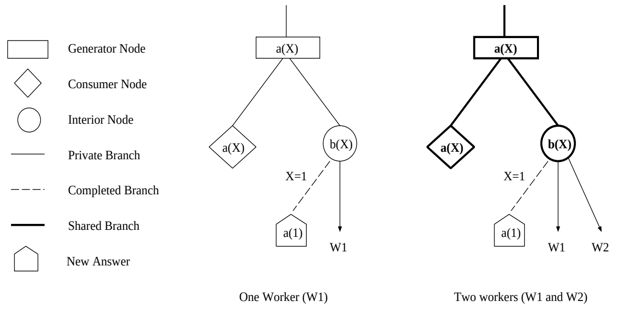

Fig. 1. Sharing work in a SLG tree.

At this point, worker `W2` moves in to share work. Consider that worker `W1` decides to share work up to its last private node (that is, the interior node for `b/1`). The two workers will share three nodes: the generator node for `a/1`, the consumer node for `a/1` and the interior node for `b/1`. Worker `W2` takes the next unexploited alternative of `b/1` and from now on, both workers can quickly find further answers for `a(X)` and any of them can restart the shared consumer node.

Tabling Unified with Or-Parallelism (TOP) We have seen that in tabling based systems subgoals need to suspend on other subgoals obtaining the full set of answers. Or-parallel systems also need to suspend, either while waiting for leftmostness in the case of side-effects, or to avoid speculative execution. The need for suspending introduces an important similarity between tabling and or-parallelism. The TOP approach unifies or-parallel suspensions and suspensions due to tabling. When exploiting parallelism between branches in the search tree, some branches may be suspended, say, because they are speculative or not leftmost nodes, or because they are consumer nodes waiting for more solutions, while others are available for parallel execution.

Fig. 2 shows parallel execution for the previous program under this approach. The left figure shows that as soon as `W1` suspends on consumer node for `a/1`, it makes the current branch of the search tree public and backtracks to the upper node. The consumer node for `a/1` can only be resumed after answers to `a(X)` are found. In the left figure an answer for subgoal `a(X)` was found. So, worker `W2` can choose whether to resume the consumer node with the newly found answer or to ask worker `W1` to share his private nodes. In this case we represent the situation where worker `W2` resumes the consumer node.

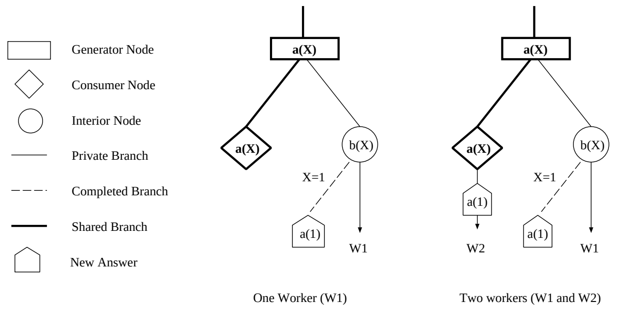

Fig. 2. Unifying suspension in parallelized tabling execution.

Comparing the Two Models The TOP model is a very attractive model, as it provides a clean and systematic unification of tabling and or-parallel suspensions. Workers have a clearly defined position, because a worker always occupies the tip of a single branch in the search tree. Everything else is shared work. It also has practical advantages, such as the fact that in this approach we can guarantee a suspended branch will only appear once, instead of possibly several times for several workers.

On the other hand, as suspended nodes are always shared in or-parallel systems, the unified suspension may result in having a larger public part of the tree which may increase overheads. Besides, to support all forms of suspension with minimal overhead, the unified suspension mechanism must be implemented efficiently. Moreover, support for the tabling component in the TOP approach requires a slightly different tabling engine than SLG-WAM. The recently proposed CAT model [4] seems to fulfill best the requirements of the TOP approach, since it assumes a linear stack for the current branch and uses an auxiliary area to save the suspended nodes. Therefore, in order to implement the TOP approach using CAT, we should adopt, for the or-parallel component, an environment copying model (such as used in Muse) as it fits best with the kind of operations CAT introduces.

In this regard, the OPT approach offers some interesting advantages. It enables different combinations for or-parallelism and tabling, giving implementors the highest degree of freedom. For instance, one can use the SLG-WAM for tabling, and environment copying [1] or binding arrays [5] for or-parallelism. Moreover, the OPT approach reduces to a minimum the overlap between or-parallelism and tabling, as we only have a tabling system extended with an or-parallel component. In TOP, we have a standard Prolog system extended with a tabling/or-parallel component.

In this work, we focus on the design and implementation of the OPT approach, adopting the SLG-WAM for tabling and environment copying for or-parallelism. Our choice seems the most natural as we believe the OPT approach gives the highest degree of orthogonality between or-parallelism and tabling. The hierarchy of or-parallelism within tabling results in a property that one can take advantage of to structure, and thus simplify, scheduler design and initial implementation issues.

Overview of the Model In our model, a set of workers, will execute a tabled program by traversing its search tree, whose nodes are entry points for parallelism. Each worker physically owns a copy of the environment (that is, the stacks) and shares a large area related to tabling and scheduling. During execution, the search tree is implicitly divided in public and private regions. Workers in the private region execute nearly as in sequential tabling. A worker with excess of work (that is, with private nodes with unexploited alternatives or unconsumed answers) when prompted for work by other workers, shares some of their private nodes. When a worker shares work with another worker, the incremental copy technique is used to set the environment for the requesting worker. Whenever a worker backtracks to a public node it synchronizes to perform the usual actions that are executed in sequential tabling. For the generator and interior nodes it takes the next alternative, and for the consumer nodes it takes the next unconsumed answer. If there are no alternatives or no unconsumed answers left, then the worker executes the *public completion*[1] operation.

4 Extending YapOr to Support Tabling

We next discuss the main data structures to extend YapOr to support parallel tabling. In the initial design, we will only consider table predicates without any kind of negative calls.

Data Areas The data areas necessary to implement the complete or-parallel tabling system are shown in the memory layout depicted in Fig. 3. The memory is divided into a global shared area and into a number of logically private areas, each owned by a single *worker* in the system. The private areas contains the standard WAM stacks, as required for each worker. The global shared area includes four main sub-areas, that we describe next.

The *or-frame space* is required by the or-parallel component in order to synchronize access to shared nodes [1] and to store scheduling information. The *table space* is required by the tabling component. It contains the table structure

[1] A public completion operation is a completion operation executed in a shared node.

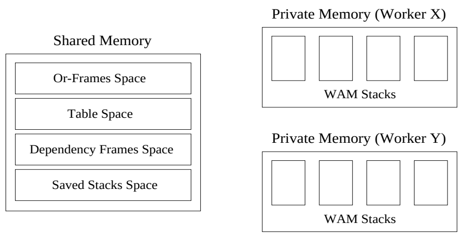

Fig. 3. Memory layout for the proposed OPT approach.

and has to be stored in shared memory for fast access to tabled subgoals by all the workers.

The *dependency frames space* is a novel data structure, designed to support or-parallel tabling. Dependency frames extend consumer nodes by holding supplementary information about a suspension point. They must be implemented in shared memory in order to permit synchronization among workers in the management of the suspension points, and in order to avoid unnecessary copying of information when a consumer node is shared between workers. This data structure is discussed in more detail below.

The *saved stacks space* preserves stacks of suspended branches. This space is required by a purely or-parallel implementation to save worker's suspension on builtins that require to be leftmost, or *voluntarily* abandon speculative work in favor of work that is not likely to be pruned [2]. In both cases, workers suspend their current work by copying their stacks into the saved stacks space, and start searching for other work. When we execute tabling in parallel, workers that share nodes can have dependencies between them. Thus, sometimes it may be necessary to delay the execution of a public completion operation until no more dependencies exist. On the other hand, in order to allow the parallel exploitation of other alternatives, as required by the environment copy model, it is necessary to maintain the stacks coherent with those of the workers that share the same nodes. These two objectives can be achieved by saving in the saved stacks space the parts of the stacks that correspond to the branches in the range of the public completion operation.

Table Space The design and implementation of the data structures and algorithms to efficiently access the table space is one of the critical issues in the design of a tabling system. Next, we provide a brief description of our implementation. It uses tries as the basis for tables as proposed in [8]. Tries provide complete discrimination for terms and permit a lookup and possible insertion to be performed in a single pass through a term. Tries are also easily parallelizable.

The table space can be accessed in different ways during the course of an evaluation: 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; to pick up answers from the table to consumer nodes; and finally to mark subgoals as completed during a completion operation.

Fig. 4 presents the table data structures for a particular predicate `t/2` after the execution of some `tabled_subgoal_call` and `new_answer` instructions.

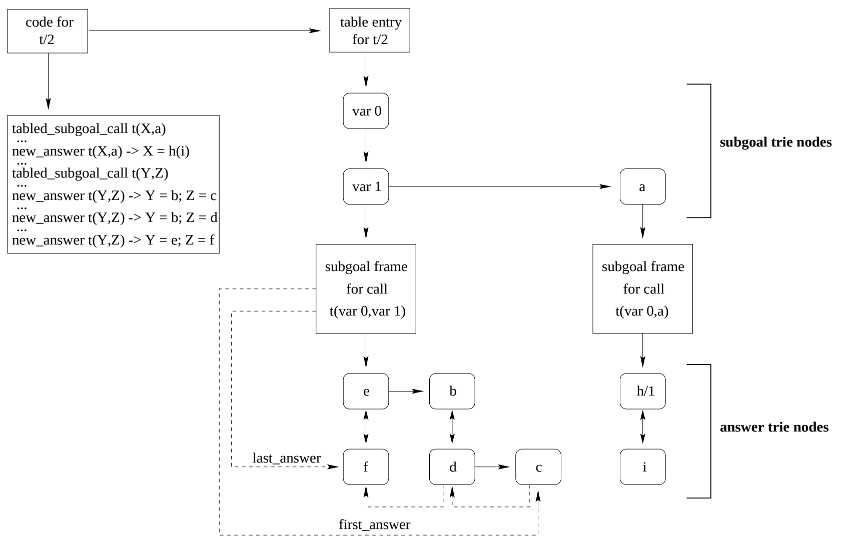
code for t/2
table entry for t/2
tabled_subgoal_call t(X,a)
...
new_answer t(X,a) -> X = h(i)
...
tabled_subgoal_call t(Y,Z)
...
new_answer t(Y,Z) -> Y = b; Z = c
...
new_answer t(Y,Z) -> Y = b; Z = d
...
new_answer t(Y,Z) -> Y = e; Z = f
var 0
var 1
a
subgoal trie nodes
subgoal frame for call t(var 0,var 1)
subgoal frame for call t(var 0,a)
e
b
h/1
f
d
c
i
last_answer
first_answer
answer trie nodes

Fig. 4. Using tries to organize the table space.

The subgoal frames delimit the subgoal and answer trie nodes space. They are used to easily access the answers found for a particular subgoal and to efficiently check if new answers have been found for a particular consumer node, through its `last_answer` pointer. The subgoal frames are also used by the completion operation to mark a subgoal as completed.

Each invocation of the `tabled_subgoal_call` instruction leads to either finding a path through the subgoal trie nodes, always starting from the table entry, until reaching a matching subgoal frame, or to creating the correspondent path of subgoal trie nodes, otherwise. Each invocation of the `new_answer` instruction corresponds to the definition of a path through the answer trie nodes, starting from the corresponding subgoal frame. In the example, the subgoal trie node with value `var_0` and the answer trie nodes with value `b` belong to several paths. This design property merges the paths that have the same first arguments. Note also that each trie node can only hold atoms, variables, or functors. Thus we need two nodes to represent the answer `h(i)`, one for the functor `h/1` and the other for the argument `i`.

Accessing and updating the table space must be carefully controlled in a parallel system. We want to maximize parallelism, whilst minimizing overheads. Read/write locks are the ideal data structure for this purpose. One can have a single lock for the table, thus only enabling a single writer for the whole table, one lock per table entry allowing one writer per procedure, one lock per path allowing one writer per call, or one lock per every trie node to attain most parallelism. Experimental evaluation is required to find the best solution.

Dependency Frames The dependency frame is the key data structure required to control suspension, resumption and completion of subcomputations. This data structure serves to: save information about the suspension point; connect consumer nodes with the table space, to search for and to pick up new answers; and form a dependency graph between consumer nodes, to efficiently check for leader nodes and perform completion.

Fig. 5 shows an example of an evaluation involving dependency frames. The sub-figure to the left presents the dependencies between the predicates involved.

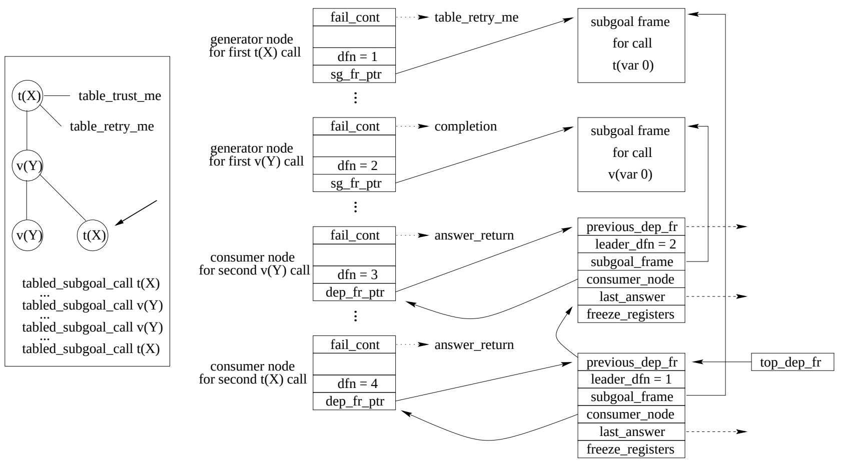

Fig. 5. Table data structures dependencies.

The first instance of `tabled_subgoal_call` searches the table space for the corresponding subgoal `t(X)`. Supposing this is the first call to the subgoal, it must allocate a subgoal frame and create a generator choice point. In our scheme, a generator choice point is simply a standard choice point plus a new `sg_fr_ptr` field, pointing to the newly allocated subgoal frame. The pointer is used to add new solutions and to check for completion. In order to mark the precedence between choice points in the left-to-right, top-down search, we introduce an extra field, `dfn` (depth first number), in all choice points. This field is not strictly necessary in the implementation because we can achieve the same goal by using the addresses of the choice points, however it simplifies the description of the model.

Following the example, an analogous situation occurs with the first call to subgoal `v(Y)`. The repeated call to subgoal `v(Y)` allocates a dependency frame and creates a consumer choice point. A consumer choice point is a standard choice point plus an extra `dep_fr_ptr` field, pointing to the newly allocated dependency frame.

A dependency frame contains six fields. The `previous_dep_fr` field points to the previous allocated dependency frame. The last dependency frame is pointed by the global variable `top_dep_fr`. The `leader_dfn` field stores the `dfn` of the bottommost leader node (details are presented next). The `subgoal_frame` field is a pointer to the correspondent subgoal frame. The `consumer_node` field is a back pointer to the consumer node. The `last_answer` field is a pointer to the last consumed answer, and is used in conjunction with the `subgoal_frame` field to check if new answers have been found for the subgoal call. The `freeze_registers` fields stores the current top positions of each of the stacks. When a completion instruction is executed with success, we consult the `freeze_registers` of the resulting top dependency frame to restore the top positions of each stack and release space. Note that in the SLG-WAM these registers are kept at the nodes were completion may take place, that is the generator nodes. In our case, this solution is not adequate as we may execute completion in any node.

Finally, the second call to `t(X)` implies an analogous situation to the previous one. A dependency frame and a consumer choice point are allocated and the `top_dep_fr` is updated.

5 The Flow of Control

A tabling evaluation can be seen as a sequence of suspension and resumptions of subcomputations. A computation suspends every time a consumer node has consumed all available answers and resumes when new solutions are found.

Restoring a Computation Every time a consumer node is allocated, its failure continuation pointer is made to point to an `answer_return` instruction. The instruction is executed through failure to the node, and guarantees that every answer is consumed once and just once. Before resuming a computation, it is necessary to restore the WAM registers and the variable bindings at the suspension point. The WAM register values are saved in the consumer node and the variable bindings are saved in the *forward trail.* The forward trail is an extension of the standard WAM trail that includes variable bindings in trail operations.

The SLG-WAM uses a forward trail with three fields per each frame. They record the address of the trailed variable (as the standard WAM trail), the value to which the variable was bound and a pointer to the parent trail frame which permits to chain correctly the variables through the current branch, hence jumping across the frozen segments (see [10] for more details). In our approach we only use the first two fields to implement the forward trail, thus spending less space in the trail stack. As Yap already uses the trail to store information beyond the normal variable trailing (to control dynamic predicates and multi-assignment variables), we extend this information to also control the chain between the different frozen segments. In terms of computational complexity the two approaches are equivalent. The main advantage of our scheme is that Yap already tests the trail frames to check if they are of a special type, and so we do not introduce further overheads in the system. It would be the case if we had chosen to implement the SLG-WAM approach.

Leader Nodes In sequential tabling, only generator nodes can be leader nodes, hence only they perform completion. In our design any node can be a leader. We must therefore check for leader nodes whenever we backtrack to a private generator node or to any shared node. If the node is leader we perform completion, otherwise we simply fail and backtrack. We designed our algorithms to quickly determine whether a node is a leader.

```
find_bottommost_leader_node () {
   leader_dfn = direct_dependency_dfn();
   aux_dep_fr = top_dep_fr;
   while (aux_dep_fr != NULL &&
          leader_dfn < DependencyFrame_consumer_dfn(aux_dep_fr)) {
      if (leader_dfn >= DependencyFrame_leader_dfn(aux_dep_fr))
         return DependencyFrame_leader_dfn(aux_dep_fr);
      aux_dep_fr = DependencyFrame_previous_dep_fr(aux_dep_fr);
   }
   return leader_dfn;
}
```

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

Fig. 6 presents the pseudo-code to initialize the `leader_dfn` field when we allocate a new dependency frame. The `direct_dependency_dfn()` function returns the `dfn` of the depth-most node that contains in one of the branches below it, the generator node of the variant subgoal call. In sequential tabling, as we have all nodes, the depth-most node that contains the generator node for a particular subgoal call, is the generator node itself. Hence, the `dfn` returned is the value of the `dfn` field within the generator node. Fig. 7 illustrates the dependencies between the several depth first numbers involved in the `find_bottommost_leader_node()` algorithm, in the case of a parallel evaluation. In order for a worker to test if a certain node, of its own branch, is leader or not, it has to check if the `dfn` field of the node is equal to the `leader_dfn` field found in the `top_dep_fr` register. In Fig. 7, workers 1 and 2 have leader nodes at generator node `a` and interior node `b` respectively.

Public Completion The correctness and efficiency of the completion algorithm appears to be one of the most important points in the implementation of a combined tabling/or-parallel system. Next we present a design for our OPT-based implementation.

If a leader node is shared and contains consumer nodes below it, this means that it depends on branches explored by other workers. Thus, even after a worker having backtracked to a leader node, it may not execute the completion operation

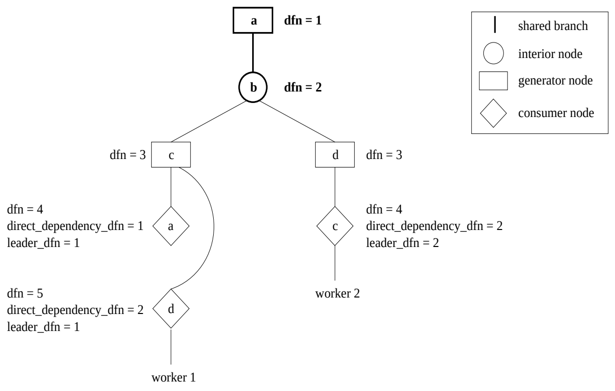

Fig. 7. *Depth first number* dependencies.

immediately. The reason for this is that the other workers can still influence the leader branch. As a result, it becomes necessary to suspend[2] the leader branch, and therefore, allow the current worker to continue execution. This suspension mechanism includes, saving the correspondent part of the stacks to the *save stacks space* (putting the respective reference in the leader node) and readjusting the freeze registers. The save stacks space resides in a shared area to allow any other worker to complete a suspended branch. This scheme also enables the current worker to proceed its execution. If the worker would not suspend the leader branch, hence not saving the stacks in the shared space, then a future sharing work operation would damage the stack areas related to the leader branch and therefore would make the completion operation unworkable. An alternative would be for the worker responsible for the leader branch to wait until no one else could influence it and only then complete the branch. Obviously, this is not an efficient strategy and besides it may carry us into a deadlock situation.

Fig. 8 shows the pseudo-code for the `public_completion()` operation. This operation is executed when a worker backtracks to a shared interior/generator node with no more alternatives left, or to a shared consumer node without unconsumed answers. The last worker leaving a node is responsible for checking and collecting all the suspension branches that have unconsumed answers. To resume a suspension branch a worker needs to copy the saved stacks to the correct position in its own stacks. Thus, for a worker to resume immediately a suspension branch, it has first to suspend its current branch and only later restart it. This has the disadvantage that the worker has to make two suspensions and resumptions instead of just one. Hence, we adopted the strategy of resuming the collected branches only when the worker finds itself in a leader node position. Here, a worker either completes the correspondent branch or suspends it. In both

[2] The notion of suspension in this context is obviously different from the one presented for tabling.

```
public_completion (node N) {
   if (last worker in node N)
      for all suspension branches SB stored in node N
         if (exists unconsumed answers for any consumer node in SB)
            collect (SB) /* to be resumed later */
   if (N is a leader node)
      if (exists unconsumed answers for any consumer node below node N)
         backtrack_through_new_answers() /* as in SLG-WAM */
      if (suspension branches collected)
         suspend_current_branch()
         resume (a suspension branch)
      else if (not last worker in node N)
         suspend_current_branch()
      else if (hidden workers in node N)
         suspend_current_branch()
      else
         complete_all()
   else   /* not leader */
      if (consumer nodes below node N)
         increment hidden workers in node N
   backtrack
}
```

Fig. 8. Pseudo-code for `public_completion()`.

situations, the stacks do not contain frozen segments below the leader node and therefore we do not have to pay extra overheads to resume a collected branch.

Whenever a node finds that it is a leader, it starts to check if there are no consumer nodes with unconsumed answers below. If there is such a node, it resumes the computation to the deeper consumer node with unconsumed answers. To check if there is a node with unconsumed answers, we can follow the dependency frames chain corresponding to the consumer nodes below, and check for one such that the `last_answer` pointer is different from the one stored on the subgoal frame pointed by the `subgoal_frame` field.

When a worker backtracks from a node and that node stays preserved in its stacks, the worker has to increment the *hidden workers* counter of the node. This counter indicates the number of workers that are executing in upper nodes and still contain the node. These workers can influence the node, if a resume operation takes place and it includes the node.

In this public completion algorithm, a worker only completes a leader node when there is nothing that can influence its branches. This only happens, when there are no suspended branches collected and there are no workers in the node, be they physically present or hidden. Completing a node includes marking the tabled subgoals involved as completed, releasing memory space and readjusting the freeze registers.

The public completion scheme proposed has two major advantages. One is that there is no communication or explicit synchronization between workers, therefore it reduces significantly possible overheads. The second advantage is that the leader nodes are the only points where we suspend branches. This is very important, since it reduces the number of check points we have to make in order to ensure that all answers are consumed in a suspended branch, to just one. This check point is executed by the last worker that leaves a node. Besides, in upper nodes we do not need to care about the uncollected suspended branches stored in the bottom nodes because we know that no upper branches can influence them.

6 Conclusions

In this paper we presented two approaches to combine or-parallelism and tabling, and focused on the design of data structures and algorithms to implement the OPT approach. These new structures and algorithms were built from within the environment copying based system, YapOr, in order to obtain a parallel tabling system. Currently, we have sequential tabling and or-parallelism functioning separately within the same system. However, we already have in the system all the data structures necessary to combine their execution and conforming with the description given in this paper. We are now working on adjusting the system for parallel execution of tabling. This will require changes to the current scheduler in order to efficiently and correctly execute tabled logic programs in parallel.

References

1. K. A. M. Ali and R. Karlsson. Full Prolog and Scheduling OR-Parallelism in Muse. *Intern. Journal of Parallel Programming*, 19(6), Dec. 1990.
2. K. A. M. Ali and R. Karlsson. Scheduling Speculative Work in Muse and Performance Results. *Intern. Journal of Parallel Programming*, 21(6), Dec. 1992.
3. W. Chen and D. S. Warren. Query Evaluation under the Well Founded Semantics. In *Proc. of PODS'93*, pages 168–179, 1993.
4. B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. In *Proc. of PLILP/ALP'98*. Springer-Verlag, Sep. 1998.
5. E. Lusk et. al. The Aurora Or-parallel Prolog System. In *Proc. of FGCS'88*, pages 819–830. ICOT, Nov. 1988.
6. J. Freire, R. Hu, T. Swift, and D. S. Warren. Exploiting Parallelism in Tabled Evaluations. In *Proc. of PLILP'95*, pages 115–132. Springer-Verlag, 1995.
7. J. Freire, T. Swift, and D. S. Warren. Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In *Proc. of PLILP'96*, pages 243–258. Springer-Verlag, Sep. 1996.
8. I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Tabling Mechanisms for Logic Programs. In *Proc. of ICLP'95*, pages 687–711. The MIT Press, June 1995.
9. R. Rocha, F. Silva, and V. S. Costa. YapOr: an Or-Parallel Prolog System based on Environment Copying. Technical Report DCC-97-14, DCC-FC & LIACC, University of Porto, Dec. 1997.
10. K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *Journal of ACM Transactions on Programming Languages and Systems*, 1998. To appear.
11. K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *Proc. of ACM SIGMOD International Conference on the Management of Data*, pages 442–453, May 1994.