On Applying Or-Parallelism to Tabled Evaluations

Ricardo Rocha   Fernando Silva   Vítor Santos Costa

Technical Report Series: DCC-97-2

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto

Rua do Campo Alegre, 823 4150 Porto, Portugal

Tel: +351+2+6001672 – Fax: +351+2+6003654

http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

On Applying Or-Parallelism to Tabled Evaluations*

Ricardo Rocha   Fernando Silva   Vítor Santos Costa

{ricroc,fds,vsc}@ncc.up.pt

Departamento de Ciência de Computadores & LIACC
Faculdade de Ciências, Universidade do Porto
Rua do Campo Alegre, 823 4150 Porto, Portugal

June 25, 1997

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 then focus on OPT approach where environment copying is used to implement or-parallelism. We give the necessary data structures and data areas and describe an algorithm for the public completion operation.

Keywords: Parallel Logic Programming, Or-parallelism, Tabling.

1 Introduction

Prolog is an extremely popular and powerful logic programming language. Prolog is widely used to program symbolic computing applications in areas such as Artificial Intelligence, Natural Language, Knowledge Based Systems, Machine Learning, Database Management or Expert Systems. Prolog's popularity was also sparked by the success in Prolog compilation technology which has enabled Prolog programs to run nearly as fast as their C counterparts [War83, Van90].

Prolog execution is based on SLD resolution for Horn clauses. This execution strategy allows efficient implementation, but suffers from fundamental limitations, such as in dealing with infinite loops and redundant subcomputations. Even the extension of SLD resolution to support negation-by-failure (SLDNF) has not proved to be appropriate in important areas of application, such as Deductive Databases, Non-Monotonic Reasoning and models for executable specifications, such as Model Checking.

SLG resolution [CW93] is a tabling based method of resolution that overcomes some limitations of SLDNF. 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 with finite models.

*to appear in the Post-ICLP'97 Workshop on .

The XSB-Prolog system is a Prolog system that implements tabling, providing a sequential implementation of SLG resolution for the well-founded semantics [SSW94, SSW96]. The system was attained by extending the WAM into the SLG-WAM, an abstract machine designed to fully integrate Prolog execution and tabling with minimal overhead.

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 table-parallelism [FHSW95]. 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 subgoal will asynchronously consume answers as they are added to the table. Table-parallelism resembles the Linda's tuple-space model, in that it views the table as a shared data structure through which cooperating agents may synchronize and communicate. As the tuple-space model, it is amenable to shared and distributed memory implementations. Note that table-parallelism requires a deep redesign of tabling systems, including new scheduling strategies and a new completion algorithm. Note also that in this scheme parallelism results from having many tabled subgoals, and that parallelism arising from many alternatives to tabled subgoals or from having non-tabled subgoals will be hard to exploit. Last, load balancing for this scheme can be difficult if the number of tabled subgoals is small, as some subgoals may have much larger search spaces than others. Even for larger numbers of subgoals scheduling may be difficult, as dependencies between tabled subgoals can be quite intricate. We would expect that choosing which subgoals to allocate to which processors even harder problem than for traditional parallel systems.

Ideally, we would like to exploit maximum parallelism and take maximum advantage of current technology for parallel and tabling systems. In order to do so an important observation is that tabling is still about exploiting alternatives to find solutions for goals. Or-parallel systems are precisely designed to accelerate exploitation of alternatives. Our suggestion is 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 large experience in implementing or-parallel systems. We believe that this approach can be an efficient model for the exploitation of parallelism in tabling-based systems.

One of interesting characteristic of tabling-based systems is that some 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, and results in two different approaches to exploiting or-parallelism in a tabled system.

The first approach is to consider tabled computations to be computations that have unexploited alternatives. These alternatives can be stolen by other workers (processors) and exploited in parallel, without knowledge on how they were generated. This principle is particularly close to the environment copying model, as used in the Muse system [AK90, Kar92]. The second approach 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 unified suspension mechanism must be in this case sufficiently efficient to support all forms of suspension with minimal overhead. This approach seems therefore closer for the SRI model [War87a], used in Aurora [LBD+88, Car90], a model designed in a way that suspension can be implemented with little overhead.

In this paper, we first briefly introduce the general concepts underlying tabling and its implementation using the SLG-WAM. Next, we discuss the fundamental issues in supporting or-parallelism for SLG resolution and discuss the two alternative approaches. Following in the vein of Muse [AK90] and YapOr [Roc96] we present a computation model based in environment copying and designed to require the least changes to the XSB-engine. We discuss the new data structures and data areas that are required for this model, and present a completion algorithm for this model.

2 Tabling Concepts and the SLG-WAM

The SLG evaluation process can be modeled by an SLG-forest. Whenever a tabled subgoal, S is called for the first time a new tree with root S is added to this forest. Simultaneously, an entry for S is allocated in the table of answers. 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 consumer subgoals. Within this model the nodes in the trees are classified as follows:

Generator (Producer) nodes, nodes that use program clause resolution to produce answers.

Active (Consumer) nodes, nodes that consume answers from the table.

Interior nodes, nodes corresponding to predicates not tabled and that are evaluated by SLD resolution.

Space for a tree can be reclaimed when all possible resolutions for its root node have been made, that is, when the subgoal has been *completely evaluated*. Note that a number of subgoals may be mutually dependent, forming a *strongly connected component*, or (SCC), and therefore can only be completed together. This completion is done by the *leader* of the SCC, which is the oldest subgoal in the SCC, when all possible resolutions have been made for all 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.

Tabling-based evaluation has four main types of operation:

Tabled Subgoal Call, looks up the subgoal in the table. If the subgoal is not found, inserts it into the table.

New Answer, verifies whether a newly generated answer is already in the table and if not inserts it into the table.

Answer Return, consumes an answer already in the table.

Completion, determines whether an SCC is completely evaluated.

The XSB system implements SLG resolution for the well-founded semantics. This implementation was attained by extending the WAM into the SLG-WAM, with minimal overhead. The SLG-WAM contains three main memory areas: the usual WAM stacks; 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.

Active (consumer) nodes must suspend when they get to a point in which they have consumed all available answers but the correspondent tabled subgoal has not yet completed and new answers may still be generated. In the SLG-WAM the suspension mechanism is implemented through a new set of registers, which freeze the WAM stacks in the suspension point and prevent all data belonging to the suspended branch from being erased. A suspended branch is resumed by restoring the information saved in the corresponding active node and by using a forward trail to restore the bindings of the suspended branch.

The SLG-WAM implements this strategy by setting the *failure continuation* field in the active 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 active node. Through failure and backtracking to an active 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 dynamically executed and must be efficient implemented in order to minimize overheads. The SLG-WAM achieves this by setting the *failure continuation* field to a `completion` instruction when a generator node resolves the last applicable program clause for the correspondent subgoal. This instruction is responsible for ensuring the total and correct evaluation of the subgoal search space. The instruction is executed through backtracking, and in the default XSB-scheduling it restarts the deeper active node with an unconsumed answer from this subgoal. The computation thus send newly found answers to all active nodes, backtrack through answers, and fail to the generator node until no more unconsumed answers are left. At this point, if the generator node is the leader of its SCC, a fix-point is reached and all dependent subgoals are completed. Otherwise, the computation will fail back to the previous node and the fix-point check will later be executed by the leader of the SCC.

3 Or-Parallelism and Tabled Evaluations

The key idea in our proposal is that we want to explore in parallel the available alternatives, be they from generator nodes, active nodes, or interior nodes. For efficiency reasons we are most interested in multi-sequential systems [War87b], that is, in systems where workers (or engines, or agents, or processors) compute independently in the search tree, and mainly communicate with each other to fetch work. Assuming that most of the time workers will be doing resolution, we present two major approaches to the problem.

Or-Parallelism within Tabling (OPT)

The first approach, that we shall name as Or-Parallelism within Tabling (OPT), considers that workers are full SLG-WAM engines. In other words, workers will spend most of their time executing as if they were sequential SLG-WAM engines, creating all three types of nodes, fully implementing suspension of tabled subgoals, and sending answers from generator to active nodes. Only when workers run out of alternatives to exploit they will need to take alternatives from other workers nodes. In the OPT approach any unexploited alternative can be taken regardless of whether the node it originates from is a generator, active or interior node. Parallelism thus arises from both tabled and non-tabled subgoals. Figure 1 gives an example for the following small program and the query `a(X)`.

```
:- table a/1.

a(X) :- a(X).
a(X) :- b(X).

b(1).
b(X) :- ...
b(X) :- ...

?- a(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 an active 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`.

At this point, a second 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 active node for `a/1` and the interior node for `b/1`.

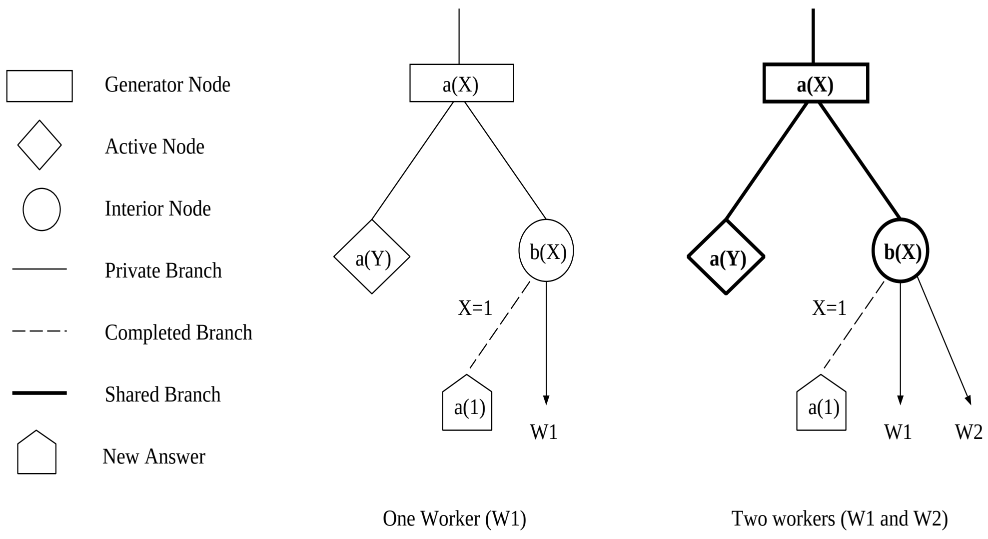

Figure 1: Sharing work in a SLG tree.

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 active node.

The OPT approach fits naturally with environment-copying based systems such as Muse. In these systems the act of sharing work can be seen as copying the stacks from one worker to another. Shared nodes also receive a pointer to a shared data area that contains synchronization and communication data between workers. As we explain in detail later this area can also be used to communicate information regarding the SLG execution.

Tabling within Or-Parallelism (TOP)

The second approach, that we shall name as Tabling within Or-Parallelism (TOP), considers that there is a shared tree and that workers only manage a logical branch, not a whole part of the tree. For instance, when worker W1 suspends an active node A, W1 backtracks and takes an alternative branch in the upper node. In the TOP approach the suspended node A thus stops being the responsibility of W1 and becomes, instead, *shared work* that anyone can take. Before W1 leaves the active node A it has to make the whole branch public.

Figure 2 shows parallel execution for the same program under this approach. The left figure shows that as soon as W1 suspends on active node for `a/1`, it makes the current branch of the search tree public and backtracks to the upper node. The active 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 active 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 took the shared work.

The major advantage of the TOP approach is that it unifies the notion of suspension. In other words, the system can handle suspensions from or-parallelism and from tabling in the same framework. Moreover, the state of the worker is more clearly defined, because a worker occupies the tip of a single branch in the search tree. Last, implementations of this approach should also spend less memory, because a suspended branch will only appear once, instead of possibly several times for several workers. On the other hand, for the TOP approach to work efficiently, suspension must be an efficient operation for both or-parallelism and tabling. This means that this approach is more natural for, say, binding arrays models where suspending is not very costly, than for environment copying based implementations. Moreover, the TOP approach requires significant changes to the XSB-engine

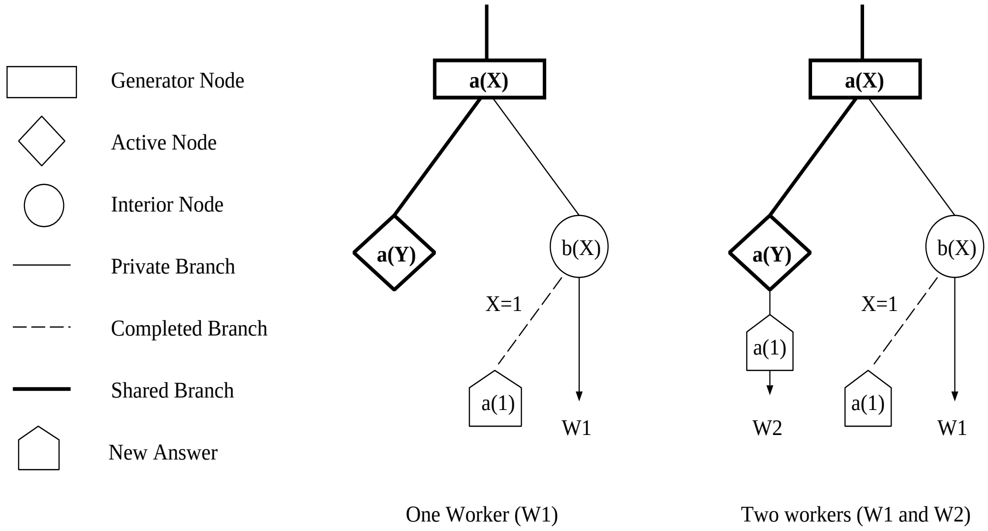

Figure 2: Unifying suspension in parallelized tabling execution.

in order to support the unified suspension and results in having a larger public part of the tree which may increase overheads.

4 Implementing Or-Parallelism in a Tabled Evaluation

The basis of our work will be the YapOr system [Roc96], an Or-Parallel Prolog system based on the Yap-Prolog engine and on environment copying as originally implemented in the Muse system. In order to support tabling in our system, as in XSB Prolog, the or-parallelism within tabling approach is the most natural one. As explained before, the OPT approach gives the highest degree of orthogonality between or-parallelism and tabling and should thus simplify the implementation issues.

In our model, a set of workers, one per PE, 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 WAM stacks) and shares a large area related to tabling and scheduling, that are required to obtain a normal execution of goals in parallel.

During execution, the search tree is implicitly divided in public and private regions. Workers in the private region execute nearly as the sequential SLG-WAM. A worker with excess of work (that is, with private nodes with unexploited alternatives or unconsumed answers) when prompted for work by other workers, makes public 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 SLG-WAM. For the generator and interior nodes it takes the next alternative, and for the active nodes it takes the next unconsumed answer. If there are no alternatives or no unconsumed answers left, then the worker will check if it has completed the node and all the other branches below it (that is, it realizes public completion).

During normal execution, the workers will have to execute the four main type of operations required by tabling. However, in order to guarantee the correct functioning of these operations, when exploiting the search tree in parallel, the workers have to be able to synchronize and communicate among themselves. Furthermore, the completion operation is even more complex since, with the parallel evaluation, the position of an active node relative to its generator node is not so easily determined and this influences the way a node can be leader. Moreover, being a leader node may not be enough to complete it.

Next we introduce the new data structures and data areas that are necessary for the implementation of combined or-parallelism and tabling and detail an algorithm for the completion operation.

Data Areas

The data areas used by the parallel implementation are shown in the memory layout depicted in figure 3. The memory is divided into a shared area and a number of private areas, corresponding to the number of workers in the system.

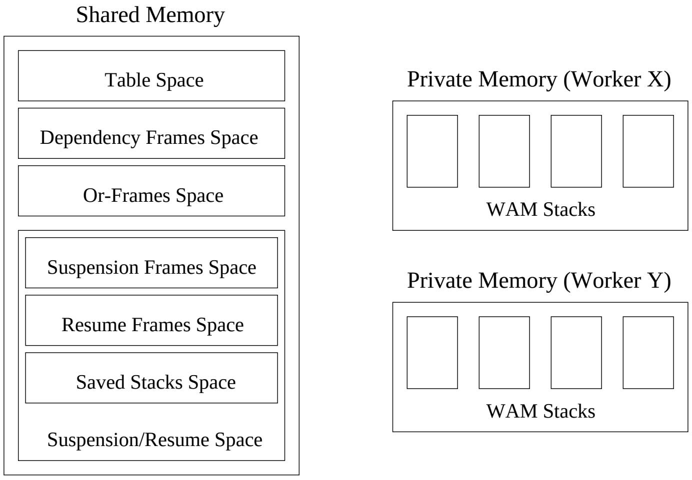

Figure 3: Memory layout for the OPT approach.

The shared area is divided into several sub-areas. The *or-frames space* is inherited from the or-parallel implementation and is used to synchronize access to shared nodes. The *table space* is inherited from the sequential tabling implementation and is implemented in shared memory to simplify communication among workers whenever they are exploiting the same table subgoals. The other shared spaces are introduced to support our model.

Introducing or-parallelism in a tabling system will cause workers to have active nodes for tabled predicates, while not having the corresponding generator nodes. Conversely, the owner of a generator node can have active nodes being executed in different workers. This may induce complex dependencies between workers, therefore requiring a more elaborate completion operation – a *public completion operation* – in which a completion procedure not only involves the private branch of a worker, but also the branches from other workers within the dependency chain. The *dependency frames space* holds some of the frames required by the public completion algorithm, the dependency frames. Each dependency frame holds information about an active node and can be shared by several workers that share the corresponding node. Each worker maintains a list of dependency frames to keep track of the active nodes in its branch of the tree.

A worker can find that a certain node in its own branches is leader for the whole active nodes below it, if those active nodes only depend from tabled subgoals that can be found in branches below the leader node. Since the worker can have dependencies with other workers who share the same node, it may not be able to complete immediately the part of the branch involved. Thus, it may be necessary to suspend the completion operation, to resume it later when 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 other workers. These two goals can be achieved by saving in a different memory area the parts of the stacks that correspond to the part of the search tree that has to be suspended. This memory area is a *suspension/resume space* in shared memory divided in three smaller spaces. A *saved stacks space* to save the corresponding stacks, a *suspension frames space* to hold the necessary information about the suspended branch, and a *resume frames space* to hold pointers to the collected suspended frames that have to be resumed.

Public Completion

Detection of completion in a sequential model is a non-trivial problem. In a parallel model the difficulties are even more considerable, hence implementation of an efficient algorithm requires great care in order to keep to a minimum the synchronization requirements between workers.

In our algorithm, a unique depth-first number (*DFN*) is associated with every node. If the node is an active node, then it is also necessary to allocate a *dependency frame* and to attach it to the list of dependency frames of the worker. One of the most important fields in the dependency frame is the *DFNLink* field. For a node C, this field stores the depth-first number for the depth-most node D such that node D is an ancestor of node C, and one of the branches below node D contains the generator node for the current active node (see Figure 4).

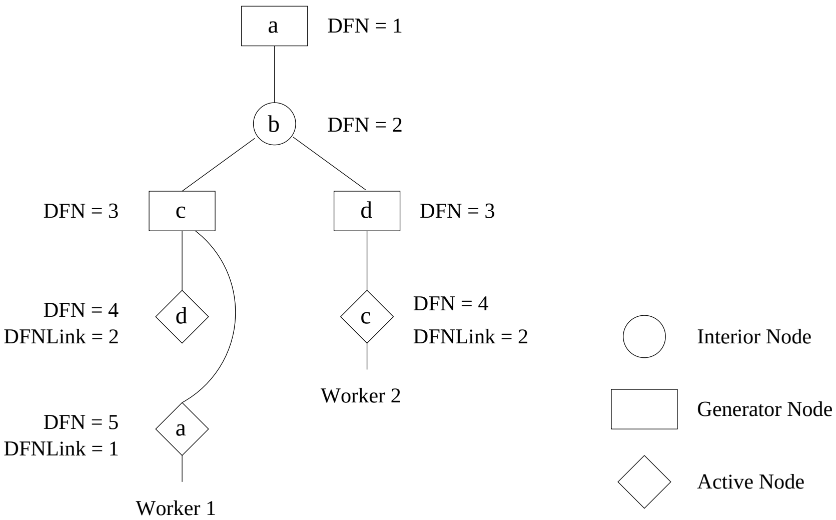

Figure 4: *DFN* and *DFNLink* fields.

For a worker to test whether a certain node N of its own branch is leader, it has to check whether the *DFN* for node N is equal to the smallest *DFNLink* field found in the dependency frames of the active nodes below node N. For instance, in figure 4 the workers 1 and 2 have leader nodes at generator node a and interior node b respectively. Notice that all kinds of nodes can be leaders in a worker's branch.

When a leader node contains active nodes below it and is shared by other workers, this means that it depends on branches explored by other workers. Thus, even after a worker finds a leader node, it may not execute the completion operation immediately since there may be other workers sharing the node that can influence the leader branch. For instance, these workers may still find new answers for an active node in the leader's branch, in which case the leader must be resumed to consume the new answers. As a result, it becomes necessary to suspend the leader branch. This includes saving all stacks to the saved-stacks space, associating the suspension frame that has been allocated with the respective node and re-adjust the freeze registers. Note that a suspension only makes sense if the worker contains active nodes in the leader branch, otherwise no stack portions exist to be saved.

The pseudo-code of the `public_completion` instruction is presented in figure 5. This instruction implements the algorithm to synchronize the completion operation in the public region of the search tree and is executed by a worker when it backtracks to a shared node without alternatives or unconsumed answers left.

```
public_completion (node N)

   if (last worker in node)
      for all not collected suspension frames SF stored in node N
         if (exists unconsumed answers for any dependency frame in SF)
            collect (SF) /* to be resumed later */
   if (leader on that node)
      for all dependency frames DF below node N
         if (DF have unconsumed answers)
            backtrack_through_new_answers() /* as in SLG-WAM */
      if (suspension frames collected)
         suspend_current_branch()
         resume (old collected suspension frame)
      else if (N.HiddenWorkers != 0)
         suspend_current_branch()
      else if (last worker in node)
         complete_all()
      else
         suspend_current_branch()
   else   /* not leader */
      if (dependency frames below node N)
         N.HiddenWorkers ++
   backtrack
```

Figure 5: Public completion instruction.

The algorithm starts to collect all suspension-frames stored in the node, that have unconsumed answers for any dependency frame in it. These frames are only resumed later, when the worker finds itself in a leader node[1] (the current node or an upper one). To resume a suspension frame a worker only needs to copy the saved stacks to the correct position of its own stacks.

A suspension frame is allocated to guarantee that if new answers are derived for at least one of its dependency frames, they will be consumed. When a suspension is being resumed, it can produce new answers to tabled subgoals. So, when the worker backtracks in the newly resumed branch, it will search for other suspension frames that need to be resumed. On the other hand, it will not be necessary to collect or resume a suspension frame when no new answers are found for it. The frame will be released when the `complete_all` instruction is executed.

When a worker backtracks from a node and the node contains active nodes below it, the worker has to preserve the stacks representing that part of the branch in order to allow for those stack portions to be used later on. The SLG-WAM uses freeze registers to mark the preserved stacks. When executing the `public_completion` instruction in a non-leader node, the worker has also to increment a counter, named *HiddenWorkers*, in the or-frame associated with the node. This counter indicates the number of workers that still have the node in their stacks and are executing in an upper node.

A worker only completes all the branches present in a leader node when it is the last worker in that node, there are no hidden workers and there are no frames to resume. Completing a node includes marking all tabled subgoals exploited here as completed, readjust the freeze registers and release all memory spaces involved.

5 Conclusions

In this paper we presented a model for exploiting or-parallelism in tabling based logic programming systems. We suggested that there are two major alternatives to tackle the problem. We presented the fundamental concepts of an environment copying based approach that we believe offers advantages in terms of implementation simplicity and efficiency. Other then the issues discussed here, support for or-

[1] If a worker resumes immediately a suspension frame, it has to suspend its current branch and restart it later after having resumed the suspension frame. Hence, we need to make two suspensions and resumes instead of one.

parallelism in tabling systems requires further research in areas such as scheduling and parallelization of table access.

Work has started on implementing the ideas presented in this paper.

Acknowledgments

This work is partially funded by the PROLOPPE project (grant PRAXIS/3/3.1/TIT/24/94) and by the MELODIA JNICT project (grant PBIC/C/TIT/2495/95). Ricardo Rocha is sponsored by the PRAXIS XXI program administered by JNICT - Junta Nacional de Investigação Científica e Tecnológica, Portugal. We would also like to thank the anonymous referees for their helpful comments.

References

[AK90] Khayri A. M. Ali and Roland Karlsson. The Muse Approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, April 1990.

[Car90] Mats Carlsson. *Design and Implementation of an OR-Parallel Prolog Engine*. PhD thesis, The Royal Institute of Technology, Stockholm, March 1990.

[CW93] D. Chen and D. S. Warren. Query evaluation under the well-founded semantics. In *Proc. of 12th PODS*, pages 168–179, 1993.

[FHSW95] Juliana Freire, Rui Hu, Terrance Swift, and David S. Warren. Exploiting Parallelism in Tabled Evaluations. Technical report, Department of Computer Science, SUNY at Stony Brook, 1995.

[Kar92] Roland Karlsson. *A High Performance OR-parallel Prolog System*. PhD thesis, The Royal Institute of Technology, Stockholm, March 1992.

[LBD+88] Ewing Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, David H. D. Warren, A. Calderwood, P. Szeredi, Seif Haridi, P. Brand, M. Carlsson, A. Ciepelewski, and B. Hausman. The Aurora Or-parallel Prolog System. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 819–830. ICOT, Tokyo, November 1988.

[Roc96] Ricardo Jorge Rocha. Um Sistema Baseado na Cópia de Ambientes para a Execução de Prolog em Paralelo. Dissertação de Mestrado, Departamento de Informática da Universidade do Minho, Julho 1996.

[SSW94] K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994.

[SSW96] K. Sagonas, T. Swift, and D. S. Warren. An Abstract Machine for Computing the Well-Founded Semantics. In *Proceedings of the 1996 International Conference and Symposium on Logic Programming, Bonn, Germany*, pages 274–288. The MIT Press, September 1996.

[Van90] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, November 1990.

[War83] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[War87a] D. H. D. Warren. The SRI Model for Or-Parallel Execution of Prolog—Abstract Design and Implementation Issues. In IEEE, editor, *The 1987 Symposium on Logic Programming*, pages 92–102, 1987.

[War87b] David H. D. Warren. Or-Parallel Execution Models of Prolog. In *TAPSOFT'87, The 1987 International Joint Conference on Theory and Practice of Software Development, Pisa, Italy*, pages 243–259. Springer-Verlag, March 1987.