An Efficient Implementation of Linear Tabling Based on Dynamic Reordering of Alternatives

Miguel Areias and Ricardo Rocha

CRACS & INESC-Porto LA, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
{miguel-areias,ricroc}@dcc.fc.up.pt

Abstract. Tabling is a technique of resolution that overcomes some limitations of traditional Prolog systems in dealing with recursion and redundant sub-computations. We can distinguish two main categories of tabling mechanisms: suspension-based tabling and linear tabling. In suspension-based tabling, a tabled evaluation can be seen as a sequence of sub-computations that suspend and later resume. Linear tabling mechanisms maintain a single execution tree where tabled subgoals always extend the current computation without requiring suspension and resumption of sub-computations. In this work, we present a new and efficient implementation of linear tabling, but for that we have extended an already existent suspension-based implementation, the YapTab engine. Our design is based on dynamic reordering of alternatives but it innovates by considering a strategy that schedules the re-evaluation of tabled calls in a similar manner to the suspension-based strategies of YapTab. Our implementation also shares the underlying execution environment and most of the data structures used to implement tabling in YapTab. We thus argue that all these common features allows us to make a first and fair comparison between suspension-based and linear tabling and, therefore, better understand the advantages and weaknesses of each.

Key words: Linear Tabling, Design, Implementation.

1 Introduction

Tabling [1] is an implementation technique that overcomes some limitations of traditional Prolog systems in dealing with redundant sub-computations and recursion. Tabling consists of storing intermediate answers for subgoals so that they can be reused when a repeated subgoal appears during the resolution process. Implementations of tabling are currently available in systems like XSB Prolog, Yap Prolog, B-Prolog, ALS-Prolog, Mercury and more recently Ciao Prolog. In these implementations, we can distinguish two main categories of tabling mechanisms: *suspension-based tabling* and *linear tabling*.

Suspension-based tabling mechanisms need to preserve the computation state of suspended tabled subgoals in order to ensure that all answers are correctly computed. A tabled evaluation can be seen as a sequence of sub-computations

that suspend and later resume. The environment of a suspended computation is preserved either by *freezing* the execution stacks, as in XSB [2] and Yap [3], by *copying* the execution stacks to separate storage, as in Mercury [4] and in the CAT model [5], or by using a mixed strategy as in the CHAT model [6]. Two more recent approaches, implemented in Yap [7] and Ciao Prolog [8], feature a higher-level implementation of suspension-based tabling. They apply source level transformations to a tabled program and then use external tabling primitives to provide direct control over the search strategy. In these proposals, suspension is implemented by leaving a *continuation call* [9] for the current computation in the table entry corresponding to the repeated call being suspended.

On the other hand, linear tabling mechanisms use iterative computations of tabled subgoals to compute fix-points. The main idea of linear tabling is to maintain a single execution tree where tabled subgoals always extend the current computation without requiring suspension and resumption of sub-computations. Two different linear tabling proposals are the SLDT strategy of Zhou *et al.* [10], as originally implemented in B-Prolog, and the DRA technique of Guo and Gupta [11], as originally implemented in ALS-Prolog. The key idea of the SLDT strategy is to let repeated calls execute from the backtracking point of the former call. The repeated call is then repeatedly re-executed, until all the available answers and clauses have been exhausted, that is, until a fix-point is reached. Current versions of B-Prolog implement an optimized variant of this strategy which tries to avoid re-evaluation of looping subgoals [12]. The DRA technique is based on dynamic reordering of alternatives with repeated calls. This technique tables not only the answers to tabled subgoals, but also the alternatives leading to repeated calls, the *looping alternatives*. It then uses the looping alternatives to repeatedly recompute them until reaching a fix-point.

Arguably, suspension-based mechanisms are considered to be more complicated to implement but, on the other hand, they are considered to obtain better results in general. A commonly referred weakness of linear tabling is the necessity of re-computation for computing fix-points. However, to the best of our knowledge, no rigorous and fair comparison between suspension-based and linear tabling was yet been done in order to better understand the advantages and weaknesses of each mechanism. The reason for this is that no single Prolog system simultaneously supports both mechanisms and thus, the available comparisons between both mechanisms cannot be fully dissociated from the strengths and weaknesses of the base Prolog systems on top of which they are implemented.

In this work, we present a new and efficient implementation of linear tabling, but for that we have extended an already existent suspension-based implementation, the YapTab engine [3], the tabling engine of Yap Prolog. Our linear tabling implementation is based on the DRA technique but it innovates by considering a strategy that schedules the re-evaluation of tabled calls in a similar manner to the suspension-based strategies of YapTab.

Our new implementation shares the underlying execution environment of the Yap Prolog system and most of the data structures used to implement tabling in YapTab. In particular, a critical component in the implementation of an ef-

ficient tabling system is the table space. Here we took advantage of YapTab's efficient table space data structures based on *tries* [13], that in our linear tabling proposal are used with minimal modifications. Our current design is also based on a scheduling strategy, *local scheduling* [14], supported by YapTab. We thus argue that all these common support features allows us to make a first and fair comparison between suspension-based and linear tabling and, therefore, better understand the advantages and weaknesses of each.

The remainder of the paper is organized as follows. First, we briefly describe the DRA technique and introduce its execution model. Next, we discuss our design decisions and provide the details for our implementation on top of the YapTab engine. At last, we present a detailed performance study and we end by outlining some conclusions.

2 Dynamic Reordering of Alternatives

The DRA linear tabling mechanism as proposed by Guo and Gupta [11] is based on the *dynamic reordering of alternatives with repeated calls* for incorporating tabling into an existing logic programming system. The DRA technique not only memorizes the answers for the tabled subgoal calls, but also the alternatives leading to repeated calls, the *looping alternatives*. It then uses the looping alternatives to repeatedly recompute them until a fix-point is reached. During evaluation, a tabled call can be in one of three possible states: *normal*, *looping* or *complete*. Figure 1 shows the state transition graph for DRA evaluation.

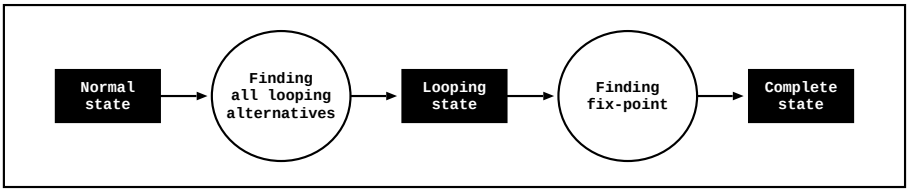

Fig. 1. State transition graph for DRA evaluation

Consider a tabled subgoal call C. Initially, C enters in normal state where it is allowed to explore the matching clauses as in standard Prolog. In this state, while exploring the matching clauses, the model checks for looping alternatives. If a repeated call is found[1] then the current clause for the first call to C will be memorized as a looping alternative. Essentially, the alternative corresponding to this call will be reordered and placed at the end of the alternative list for the call. As in a tabled evaluation repeated calls are not re-evaluated against the program clauses because they can potentially lead to infinite loops, the repeated call to C is then resolved by consuming the answers already available for the call

[1] A call repeats a previous call if they are the same up to variable renaming.

in the table space. In what follows we will refer to first calls to tabled subgoals as *generator calls* and to repeated calls to tabled subgoals as *consumer calls*.

Next, after exploring all the matching clauses, C goes into the looping state. From this point, it keeps trying the looping alternatives repeatedly until reaching a fix-point. If no new answers are found during one cycle of trying the looping alternatives, then we have reached a fix-point and we can say that C is completely evaluated. However, if a number of calls is mutually dependent, thus forming a *Strongly Connected Component* (or *SCC*), then completion is more complex and we can only complete the calls in a SCC together. SCCs are usually represented by the *leader call*. More precisely, the generator call which does not depends on older generators is the leader call. A leader call defines the next completion point, i.e., if no new answers are found during one cycle of trying the looping alternatives for the leader call, then we have reached a fix-point and we can say that all calls in the SCC are completely evaluated.

2.1 An Evaluation Example

We next illustrate in Fig. 2 the original principles of DRA tabled evaluation through an example. At the top, the figure shows the program code (the left box) and the final state of the table space (the right box). The program specifies a tabled predicate `t/2` defined by five clauses (alternatives `c1` to `c5`). The bottom sub-figure shows the evaluation sequence for the query goal `t(1,X)`. Generator calls are depicted by black oval boxes and consumer calls by white oval boxes.

The evaluation starts by inserting a new entry in the table space representing the generator call `t(1,X)` (step 1). Then, `t(1,X)` is resolved against the first matching clause, alternative `c1`, calling `t(2,X)` in the continuation. As this is a first call to `t(2,X)`, we insert a new entry in the table space representing `t(2,X)` and proceed as shown in the middle tree (step 2). `t(2,X)` is also resolved against the first matching clause, alternative `c3`, calling again `t(2,X)` in the continuation (step 3). Since `t(2,X)` is now a consumer call, we mark the clause in evaluation for the generator call, alternative `c3`, as a looping alternative for `t(2,X)`. Then, we try to consume answers but, as no answers are available for `t(2,X)`, the execution fails (step 4).

Next, we try the second matching clause for `t(2,X)`, alternative `c4`, thus calling `t(1,X)` (step 5). Since `t(1,X)` is also a consumer call, we mark the clauses in evaluation up to the generator call for `t(1,X)` as looping alternatives. This includes alternative `c1` for `t(1,X)` and alternative `c4` for `t(2,X)`. Then, we try to consume answers but, because no answers are available for `t(1,X)`, we fail (step 6). The last matching clause for `t(2,X)`, alternative `c5`, is then tried and we obtain a first answer for `t(2,X)`. The answer is inserted in the table space and, as we are following a local scheduling strategy, the execution fails (step 8).

We then backtrack again to the generator call for `t(2,X)` and because we have already explored all matching clauses, `t(2,X)` moves into the looping state. We have found a new answer for `t(2,X)`, so we must re-execute the looping alternatives `c3` and `c4` (step 9). In alternative `c3`, `t(2,X)` is called again as a consumer call (step 10). The answer `X=a` is forward to it but in the continuation

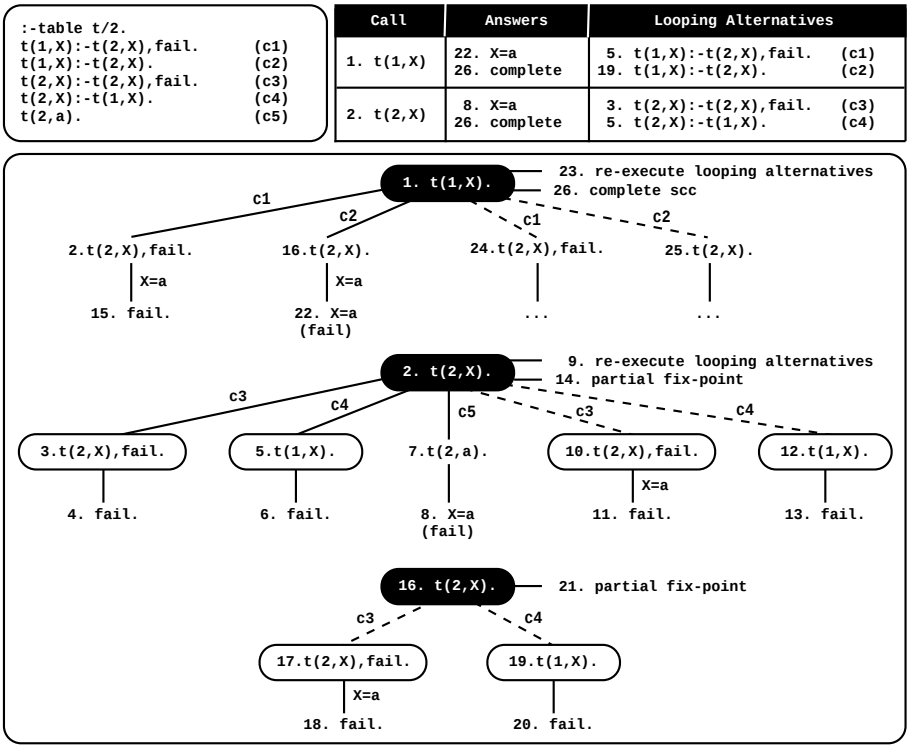

Fig. 2. A DRA tabled evaluation

the execution fails (step 11). In alternative `c4`, we repeat the situation in steps 5 to 6 and we fail for the same reasons (steps 12 to 13). The evaluation then backtracks to the generator call for `t(2,X)` and, because we have reached a partial fix-point (i.e., no answers were found when trying the looping alternatives), we check whether `t(2,X)` can complete (step 14). It cannot, because it depends on `t(1,X)` and thus it is not a leader call.

Next, as we are following a local scheduling strategy, the answer for `t(2,X)` should now be propagated to the context of the previous call. We thus propagate the answer `X=a` to the context of subgoal call `t(1,X)` but the execution fails in the continuation (step 15). Then, we try the second matching clause for `t(1,X)`, alternative `c2`, thus calling `t(2,X)`. Because `t(2,X)` has already reached the looping state, we proceed as shown in the bottommost tree with `t(2,X)` being resolved again against its looping alternatives (step 16). The evaluation then repeats the same sequence as in steps 10 to 14 (now steps 17 to 21), but now when the answer `X=a` is propagated to the context of `t(1,X)`, it originates a first answer for `t(1,X)` (step 22). We then backtrack to the generator call for `t(1,X)` and because we have already explored all matching clauses, `t(1,X)` moves into the looping state. We have found a new answer for `t(1,X)`, so we must re-execute the

looping alternatives `c1` and `c2` (step 23). The re-execution of these alternatives do not finds new answers for `t(1,X)` or `t(2,X)`. Thus, when backtracking again to `t(1,X)` we have reached a partial fix-point and because `t(1,X)` is a leader call, we can declare the two subgoal calls to be completed (step 26).

2.2 Re-Computation Issues

One advantage of the original DRA technique is that only the looping alternatives are recomputed. However, repeatedly retrying these alternatives may cause redundant computations: non-tabled calls are recomputed every time a looping alternative is tried, and repeated tabled calls re-consume all tabled answers every time they are called. Figure 3 shows the choice point stack at different steps of the DRA tabled evaluation of Fig. 2.

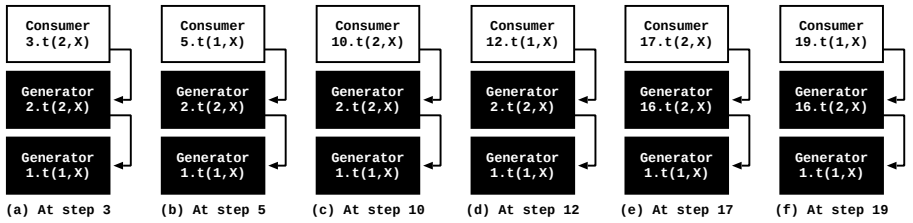

Fig. 3. DRA's choice point stack for the tabled evaluation of Fig. 2

Figures 3(c) and 3(d) reflect the decision made at step 9 in the evaluation of Fig. 2 of re-executing the looping alternatives `c3` and `c4`, and Figures 3(e) and 3(f) reflect the same decision made at step 16. Remember that the goal behind these decisions is to reach a partial fix-point in the evaluation of the corresponding tabled call. However, reaching a partial fix-point beforehand can be completely useless for non-leader calls when later the leader call re-executes itself its looping alternatives (which in turn leads the non-leader calls to re-execute again their looping alternatives). In fact, in the case of multiple dependent calls, reaching partial fix-points beforehand can cause a huge number of redundant computations.

We innovate by considering a strategy that schedules the re-evaluation of tabled calls in a similar manner to the suspension-based strategies of YapTab. In YapTab, only first calls to tabled subgoals allocate generator choice points and the fix-point check for completion is only done by leader calls (please refer to [3] for full details). Figure 4 illustrates YapTab's choice point stack for the same tabled evaluation of Fig. 2. In particular, Fig. 4(c) shows us that the whole evaluation requires just one generator choice point per call and only one and two consumer choice points for evaluating `t(1,X)` and `t(2,X)`, respectively.

Our proposal is thus to schedule the re-evaluation of non-leader tabled calls in such a way that the number of generator and consumer choice points is the same

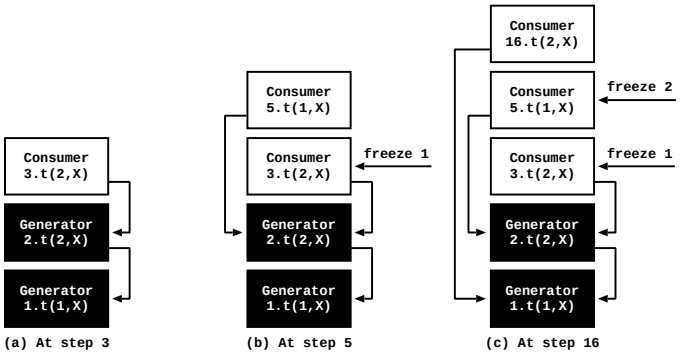

Fig. 4. YapTab's suspension-based choice point stack for the tabled evaluation of Fig. 2

as in YapTab, i.e., only first calls to tabled subgoals allocate generator choice points to execute alternatives and the fix-point check for completion is only done by leader calls (we get ride of the notion of partial fix-points). In particular, for the tabled evaluation of Fig. 2, this means that we do not re-execute the looping alternatives for `t(2,X)` at step 9 (`t(2,X)` is a non-leader call) and at step 16 (this call to `t(2,X)` is not the first call, the first one was at step 2). Instead, at both steps, we must consume the available answers for `t(2,X)`.

To correctly implement this strategy, note also that now the fix-point check is only done at the level of the leader call. This means that a leader call must re-execute its looping alternatives not only when new answers were found for it during the last traversal of the looping alternatives, but when new answers were found for any tabled call in the current SCC. Moreover, as in a DRA tabled evaluation the choice points are not frozen as in YapTab, we now consider that a tabled call is a first call every time we re-start a new round over the looping alternatives for the leader call. In particular, for the tabled evaluation of Fig. 2, this means that we re-execute the looping alternatives for `t(2,X)` only at step 24 (the call to `t(2,X)` at step 24 is the first call in the round over the looping alternatives for the leader call `t(1,X)` started at step 23).

3 Implementation Details

In YapTab, a key data structure in the table space organization is the *subgoal frame*. Subgoal frames are used to store information about each tabled call and to act like entry points to the data structures where answers are stored. We next enumerate the most relevant subgoal frame fields in our DRA implementation:

`SgFr_dfn`: is the *depth-first number* of the call. Calls are numbered incrementally and according to the order in which they appear in the evaluation.

`SgFr_state`: indicates the state of the subgoal. A subgoal can be in one of the following states: *ready*, *evaluating*, *loop_ready*, *loop_evaluating* or *complete*.

`SgFr_is_leader:` indicates if the call is a leader call or not. New calls are by default leader calls.

`SgFr_new_answers:` indicates if new answers were found during the normal state or during the execution of the last round trying the looping alternatives.

`SgFr_current_alt:` marks the alternative being evaluated.

`SgFr_stop_alt:` marks the looping alternative where we should stop when in looping state.

`SgFr_looping_alts:` is the pointer to the looping alternatives associated with the subgoal or `NULL` if no looping alternatives exist.

`SgFr_next_on_scc:` is the pointer to the subgoal frame corresponding to the previous tabled call in evaluation (i.e., with `SgFr_state` as *evaluating* or *loop_evaluating*) in the current SCC. It is used by the leader call to traverse the subgoal frames in order to mark them for re-evaluation or as completed. A global variable `TOP_SCC` always points to the youngest subgoal frame in evaluation in the current SCC.

`SgFr_next_on_branch:` is the pointer to the subgoal frame corresponding to the previous tabled call in the current branch that is in the normal state (i.e., with `SgFr_state` as *evaluating*) or that is a leader call. It is used to traverse the subgoal frames in order to detect looping alternatives and to detect non-leader calls. A global variable `TOP_BRANCH` always points to the youngest subgoal frame on the current branch.

We next show the pseudo-code for the main tabling operations in our DRA implementation. We start with Fig. 5 showing the pseudo-code for the *new answer* operation. The `new_answer()` procedure simply inserts the given answer `AW` in the answer structure for the given subgoal frame `SF` and, if the answer is new, it updates the `SgFr_new_answers` field to `TRUE`. We then implement a local scheduling strategy and always fail at the end.

```
new_answer(answer AW, subgoal frame SF) {
  if (answer_check_insert(AW,SF) == TRUE)
    SgFr_new_answers(SF) = TRUE                          // new answer
  fail()                                               // local scheduling
}
```

Fig. 5. Pseudo-code for the *new answer* operation

Figure 6 shows the pseudo-code for the *tabled call* operation. New calls to tabled subgoals are inserted into the table space by allocating the necessary data structures. This includes allocating and initializing a new subgoal frame to represent the given subgoal call (this is the case where the state of `SF` is *ready*). In such case, the tabled call operation then updates the state of `SF` to *evaluating*; saves the current alternative in the `SgFr_current_alt` field; adds `SF` to the current SCC and to the current branch; pushes a new generator choice point onto the local stack; and proceeds by executing the next instruction.

```
tabled_call(subgoal call SC) {
  SF = call_check_insert(SC)            // SF is the subgoal frame for SC
  if (SgFr_state(SF) == ready) {
    SgFr_state(SF) = evaluating
    SgFr_current_alt(SF) = PC                // PC is the program counter
    SgFr_next_on_scc(SF) = TOP_SCC               // add SF to current SCC
    SgFr_next_on_branch(SF) = TOP_BRANCH      // add SF to current branch
    TOP_SCC = TOP_BRANCH = SF
    store_generator_choice_point()
    goto execute(next_instruction())
  } else if (SgFr_state(SF) == loop_ready) {
    SgFr_state(SF) = loop_evaluating
    SgFr_current_alt(SF) = get_first_looping_alternative(SF)
    SgFr_stop_alt(SF) = SgFr_current_alt(SF)     // mark stop alternative
    SgFr_next_on_scc(SF) = TOP_SCC               // add SF to current SCC
    TOP_SCC = SF
    store_generator_choice_point()
    goto execute(SgFr_current_alt(SF))
  } else if (SgFr_state(SF) == evaluating ||
             SgFr_state(SF) == loop_evaluating) {
    mark_current_branch_as_a_looping_branch(SF)
    store_consumer_choice_point()
    goto consume_answers(SF)
  } else if (SgFr_state(SF) == complete)
    goto completed_table_optimization(SF)
}

mark_current_branch_as_a_looping_branch(subgoal frame SF) {
  subgoal frame aux_sf = TOP_BRANCH
  while (aux_sf && SgFr_dfn(aux_sf) > SgFr_dfn(SF)) {
    SgFr_is_leader(aux_sf) = FALSE
    mark_current_alternative_as_a_looping_alternative(aux_sf)
    aux_sf = SgFr_next_on_branch(aux_sf)
  }
  if (aux_sf)
    mark_current_alternative_as_a_looping_alternative(aux_sf)
}
```

Fig. 6. Pseudo-code for the *tabled call* operation

On the other hand, if the subgoal call is a repeated call, then the subgoal frame is already in the table space, and three different situations may occur. If the call is completed (this is the case where the state of `SF` is *complete*), the operation consumes the available answers by implementing the *completed table optimization* which executes compiled code directly from the answer structure associated with the completed call [13]. If the call is a first call in a new round over the looping alternatives for the leader call (this is the case where the state of `SF` is *loop_ready*), the operation updates the state of `SF` to *loop_evaluating*; loads the first looping alternative and marks it as the stopping alternative; adds `SF` to the current SCC; pushes a new generator choice point onto the local stack; and proceeds by executing the first looping alternative. Otherwise, the call is a consumer call (this is the case where the state of `SF` is *evaluating* or

loop_evaluating). In such case, the operation marks the current branch as a looping branch (in order to be able to re-execute that branch if new answers are found for the current call); pushes a new consumer choice point onto the local stack; and starts consuming the available answers. To mark the current branch as a looping branch, we follow the subgoal frames in the `TOP_BRANCH` chain up to the frame for the call at hand[2] and we mark the alternatives being evaluated in each frame as looping alternatives. Moreover, as the call at hand defines a new dependency for the current SCC, all intermediate subgoal frames in the `TOP_BRANCH` chain are also marked as non-leader calls.

Finally, we discuss in more detail how completion is detected in our DRA implementation. It proceeds as follows. After exploring the last program clause for a tabled call, from then on, every time we backtrack to a generator choice point for the call, we execute the *fix-point check* operation as shown next in Fig. 7. The fix-point check operation starts by checking if there are looping alternatives for the subgoal frame `SF` corresponding to the tabled call at hand. If so, it then checks if this is the first execution of the fix-point check operation for the call (the call is in normal state) or not (the call is in looping state). For first executions (this is the case where the state of `SF` is *evaluating*), the operation moves the call to looping state by updating the state of `SF` to *loop_evaluating*; removes `SF` from the current branch if the call is non-leader (this is the optimization that we mentioned in the previous footnote); loads the first looping alternative and marks it as the stopping alternative. For repeated executions (this is the case where the state of `SF` is *loop_evaluating*) it loads the next looping alternative[3].

Next, if we haven't reached the stop alternative, then the loaded looping alternative is executed. However, before doing that, we implement the following optimization. If the call at hand is a leader call with new answers found during the execution of the last alternative, we start a new round over the looping alternatives and mark the current alternative as the new stop alternative. Note that this is done even when the previous stop alternative wasn't still reached. The idea is to minimize the number of alternatives that need to be tried by starting new rounds as soon as possible. For example, consider that we have three looping alternatives and that the second looping alternative was the last in which we have found news answers. In such case, there is no point in trying again the third alternative in a new round over the looping alternatives because it is safe to only try the first and the second alternatives. When starting a new round, we need to reset the calls in the current SCC to the *loop_ready* state in order to allow their re-execution as first calls when called later.

Finally, if there is no more looping alternatives to try, we have reached a partial fix-point. If the call at hand is a leader call, then we can perform completion

[2] As an optimization, when a call is a non-leader call and moves to the looping state, it is removed from the `TOP_BRANCH` chain because there is no point in keeping it there. Thus, when this happens for the call at hand, we follow the subgoal frames in the `TOP_BRANCH` chain up to the first frame with a smaller `SgFr_dfn` value.

[3] The next alternative after the last one is the first alternative. Thus, in the cases where there is only one looping alternative, the next alternative is always the first.

```
fix-point_check(subgoal frame SF){
  if (SgFr_looping_alts(SF) != NULL) {
    if (SgFr_state(SF) == evaluating) {
      SgFr_state(SF) = loop_evaluating           // move to looping state
      if (SgFr_is_leader(SF) == FALSE)
        TOP_BRANCH = SgFr_next_on_branch(SF)     // remove SF from branch
      SgFr_current_alt(SF) = get_first_looping_alternative(SF)
      SgFr_stop_alt(SF) = SgFr_current_alt(SF)   // mark stop alternative
    } else                              // SgFr_state(SF) == loop_evaluating
      SgFr_current_alt(SF) = get_next_looping_alternative(SF)
    if (SgFr_is_leader(SF) && SgFr_new_answers(SF)) {  // start new round
      SgFr_new_answers(SF) = FALSE
      SgFr_stop_alt(SF) = SgFr_current_alt(SF)   // mark stop alternative
      while (TOP_SCC != SF) {               // reset calls in current SCC
        SgFr_state(TOP_SCC) = loop_ready
        TOP_SCC = SgFr_next_on_scc(TOP_SCC)
      }
      goto execute(SgFr_current_alt(SF))
    }
    if (SgFr_current_alt(SF) != SgFr_stop_alt(SF))
      goto execute(SgFr_current_alt(SF))
  }
  if (SF == TOP_BRANCH)
    TOP_BRANCH = SgFr_next_on_branch(SF)         // remove SF from branch
  if (SgFr_is_leader(SF)) {
    while (TOP_SCC != SF) {                                 // complete SCC
      SgFr_state(TOP_SCC) = complete
      TOP_SCC = SgFr_next_on_scc(TOP_SCC)
    }
    SgFr_state(SF) = complete
    TOP_SCC = SgFr_next_on_scc(SF)                  // remove SF from SCC
    goto completed_table_optimization(SF)             // local scheduling
  } else {
    if (SgFr_new_answers(SF)) {
      SgFr_new_answers(SF) = FALSE
      SgFr_new_answers(TOP_BRANCH) = TRUE   // propagate new answers info
    }
    goto consume_answers(SF)                            // local scheduling
  }
}
```

Fig. 7. Pseudo-code for the *fix-point check* operation

and mark all the calls in the current SCC as *complete*. At the end, as we are implementing a local scheduling strategy, we need to consume the set of answers that have been found. As the call is already completed, we can execute the completed table optimization. On the other hand, if the call at hand is not a leader call, we avoid re-executing the looping alternatives and, instead, we start acting like a consumer node. Before start consuming the available answers, we check if new answers were found during the traversal of the looping alternatives and, if this is the case, we propagate the new answers info to the previous subgoal frame on the `TOP_BRANCH` chain. By doing this, we ensure that the new answers info will be recursively propagated until reaching the leader call.

4 Experimental Results

To the best of our knowledge, YapTab is now the first tabling engine to support simultaneously suspension-based tabling and linear tabling. We have thus the conditions to make a first and fair comparison between both mechanisms. In what follows, we present a set of experiments comparing our DRA implementation against the original YapTab suspension-based implementation, both sharing the underlying execution environment of the Yap Prolog 6.0.0. To put the performance of our DRA implementation in perspective, we also compare it against the two most well-known tabling systems supporting suspension-based tabling and linear tabling, respectively XSB (version 3.2) and B-Prolog (version 7.3#2). The environment for our experiments was an Intel Core2 Quad CPU 2.83GHz with 2 GBytes of main memory and running the Linux kernel 2.6.24-24.

We used six different versions of the well-known `path/2` predicate, that computes the transitive closure in a graph, combined with several different configurations of `edge/2` facts, for a total number of 54 programs. The six versions of the path predicate include two right recursive, two left recursive and two double recursive definitions. Each pair has one definition with the recursive clause first and another with the recursive clause last. Regarding the edge facts, we used three configurations: a pyramid, a cycle and a grid configuration (Fig. 8 shows an example for each configuration). We experimented the pyramid and cycle configurations with depths 500, 1000 and 1500 and the grid configuration with depths 20, 30 and 40. We also experimented the left recursive definition of the `path/2` predicate with three different transition relation graphs usually used in Model Checking (MC) applications: the *i-protocol* (IP), *leader election* (LE) and *sieve* (SV) specifications[4]. All experiments find all the solutions for the problem.

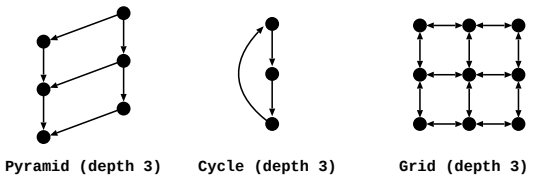

Fig. 8. Edge configurations for path definitions

Next, we show in Table 1 the running times ratios of YapTab, XSB and B-Prolog over our DRA implementation (YapTab+DRA) for all these configurations. YapTab+DRA, YapTab and XSB running times were all obtained using a local scheduling strategy. B-Prolog running times were obtained using *lazy scheduling* [12] (the local scheduling version of B-Prolog). The running times are the average of three runs. The experiments marked with *r.e.* in Table 1 for XSB mean that we got a run-time error.

[4] We didn't show results for the right and double recursive definitions because they took more than 5 hours to execute in YapTab and thus we aborted their execution.

Table 1. Running time ratios of YapTab, XSB and B-Prolog over YapTab+DRA

Predicate	Pyramid			Cycle			Grid			MC		
	500	1000	1500	500	1000	1500	20	30	40	IP	LE	SV
YapTab / YapTab+DRA												
left_first	0.67	0.67	0.73	0.61	0.68	0.72	0.60	0.52	0.58	0.55	0.56	0.53
left_last	0.59	0.63	0.62	0.60	0.64	0.67	0.64	0.54	0.52	0.56	0.51	0.54
right_first	0.99	0.99	1.03	0.70	0.59	0.69	0.25	0.16	0.12	–	–	–
right_last	1.05	1.00	0.99	0.83	0.72	0.74	0.26	0.17	0.13	–	–	–
double_first	0.51	0.49	0.53	0.59	0.58	0.58	0.57	0.56	0.59	–	–	–
double_last	0.52	0.51	0.51	0.57	0.57	0.58	0.57	0.56	0.56	–	–	–
XSB / YapTab+DRA												
left_first	0.61	0.56	0.58	0.83	0.78	0.69	0.71	0.66	0.65	1.05	1.52	0.80
left_last	0.64	0.58	0.62	0.79	0.68	0.79	0.81	0.66	0.63	1.05	1.50	0.69
right_first	1.26	1.32	1.44	1.01	1.05	1.03	0.39	0.29	0.23	–	–	–
right_last	1.23	1.36	1.34	1.06	1.01	0.98	0.41	0.30	0.24	–	–	–
double_first	0.92	0.89	0.90	1.00	0.98	*r.e.*	1.02	1.01	*r.e.*	–	–	–
double_last	0.92	0.90	0.89	1.00	0.97	*r.e.*	1.01	0.99	*r.e.*	–	–	–
B-Prolog / YapTab+DRA												
left_first	1.53	1.93	2.62	1.64	1.70	2.20	2.81	2.71	3.65	3.61	10.52	9.61
left_last	1.56	1.65	2.27	1.56	1.74	1.98	3.47	2.33	3.39	3.61	10.18	9.43
right_first	1.43	1.66	1.96	1.79	1.84	2.15	1.53	1.42	1.47	–	–	–
right_last	1.40	1.55	1.76	1.76	1.89	2.12	1.58	1.44	1.44	–	–	–
double_first	2.50	3.20	4.21	2.25	2.93	3.73	2.13	2.81	4.00	–	–	–
double_last	2.49	3.31	4.28	2.22	2.80	3.63	2.10	2.77	3.86	–	–	–

Globally, the results obtained in Table 1 indicate that YapTab+DRA is comparable to the YapTab and XSB suspension-based implementations and that YapTab+DRA clearly outperforms the B-Prolog linear tabling implementation.

In general, YapTab is around 1.5 to 2 times faster than YapTab+DRA in most experiments, including the three model checking specifications. The exception seems to be the right recursive definitions where for the pyramid configurations the running times are quite similar (with YapTab+DRA being faster in some cases) and for the grid experiments where YapTab is around 4 to 8 times faster than YapTab+DRA. The results also indicate that YapTab+DRA scales well when we increase the complexity of the problem being tested. In general, YapTab's ratio over YapTab+DRA is almost the same when we compare the pyramid configurations (depths 500, 1000 and 1500), the cycle configurations (depths 500, 1000 and 1500) or the grid configurations (depths 20, 30 and 40) between themselves. Again, the exception are the right recursive definitions with the grid configurations where YapTab's ratio over YapTab+DRA decreases proportionally to the complexity of the problem. Globally, best performance is achieved by the left recursive definitions. This is an interesting result because left recursion is the usual and *more correct* away to define tabled predicates. Note also that the path definitions that we have used are a kind of *worst-case scenarios* because most of the time they are exclusively doing tabled compu-

tations. If we have used more real-world applications, were the percentage of *standard* Prolog computation is higher, the ratios presented in Table 1 will be also proportionally higher.

The results for XSB are not so expressive as for YapTab and, in general, the difference between XSB running times and YapTab+DRA is clearly smaller. Globally, XSB achieves best performance for the right recursive definitions with the grid configurations. For the double recursive definitions and for the right recursive definitions with the cycle configurations the running times are quite similar. Surprisingly, YapTab+DRA obtains better results than XSB for the right recursive definitions with the pyramid configurations and for the left recursive definitions with the model checking specifications.

Regarding B-Prolog, Table 1 shows that YapTab+DRA is always faster than B-Prolog in these experiments and that, for almost all configurations, the ratio over YapTab+DRA shows a generic tendency to increase as the complexity of the problem also increases. In particular, for two of the model checking specifications, B-Prolog shows the worst results, being around 10 times slower for the leader election and the sieve specifications.

5 Conclusions

We have presented a new and very efficient implementation of linear tabling that shares the underlying execution environment and most of the data structures used to implement suspension-based tabling in YapTab. To the best of our knowledge, YapTab is now the first and single tabling engine to support simultaneously suspension-based tabling and linear tabling. Our linear tabling design is based on dynamic reordering of alternatives but it innovates by considering a strategy that schedules the re-evaluation of tabled calls in a similar manner to the suspension-based strategies of YapTab.

The results obtained with our approach are very interesting and very promising. Our experiments confirmed the idea that, in general, suspension-based mechanisms obtain better results than linear tabling. However, the commonly referred weakness of linear tabling of doing a huge number of redundant computations for computing fix-points was not such a problem in our experiments. We thus argue that an efficient implementation of linear tabling can be a good and first alternative to incorporate tabling into a Prolog system without tabling support.

Further work will include exploring the impact of applying our proposal to more complex problems, seeking real-world experimental results allowing us to improve and consolidate our current implementation. Moreover, since linear tabling does not require stack freezing or copying, it has a memory space advantage over suspension-based approaches and thus it would be interesting to study that memory impact in more detail. We also plan to expand our approach to support different linear tabling proposals like the SLDT strategy [10], as originally implemented in B-Prolog, and to support other optimizations, such as, remembering alternatives of non-tabled predicates at time of consumer calls to avoid the re-computation of the useless alternatives of non-tabled predicates too.

Acknowledgements

This work has been partially supported by the FCT research projects STAMPA (PTDC/EIA/67738/2006) and JEDI (PTDC/EIA/66924/2006).

References

1. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43**(1) (1996) 20–74
2. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems **20**(3) (1998) 586–634
3. Rocha, R., Silva, F., Santos Costa, V.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Conference on Tabulation in Parsing and Deduction. (2000) 77–87
4. Somogyi, Z., Sagonas, K.: Tabling in Mercury: Design and Implementation. In: International Symposium on Practical Aspects of Declarative Languages. Number 3819 in LNCS, Springer-Verlag (2006) 150–167
5. Demoen, B., Sagonas, K.: CAT: the Copying Approach to Tabling. In: International Symposium on Programming Language Implementation and Logic Programming. Number 1490 in LNCS, Springer-Verlag (1998) 21–35
6. Demoen, B., Sagonas, K.: CHAT: The Copy-Hybrid Approach to Tabling. Future Generation Computer Systems **16**(7) (2000) 809–830
7. Rocha, R., Silva, C., Lopes, R.: Implementation of Suspension-Based Tabling in Prolog using External Primitives. In: Local Proceedings of the 13th Portuguese Conference on Artificial Intelligence. (2007) 11–22
8. de Guzmán, P.C., Carro, M., Hermenegildo, M.V.: Towards a Complete Scheme for Tabled Execution Based on Program Transformation. In: International Symposium on Practical Aspects of Declarative Languages. Number 5418 in LNCS, Springer-Verlag (2009) 224–238
9. Ramesh, R., Chen, W.: Implementation of Tabled Evaluation with Delaying in Prolog. IEEE Transactions on Knowledge and Data Engineering **9**(4) (1997) 559–574
10. Zhou, N.F., Shen, Y.D., Yuan, L.Y., You, J.H.: Implementation of a Linear Tabling Mechanism. In: Practical Aspects of Declarative Languages. Number 1753 in LNCS, Springer-Verlag (2000) 109–123
11. Guo, H.F., Gupta, G.: A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In: International Conference on Logic Programming. Number 2237 in LNCS, Springer-Verlag (2001) 181–196
12. Zhou, N.F., Sato, T., Shen, Y.D.: Linear Tabling Strategies and Optimizations. Theory and Practice of Logic Programming **8**(1) (2008) 81–109
13. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38**(1) (1999) 31–54
14. Freire, J., Swift, T., Warren, D.S.: Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In: International Symposium on Programming Language Implementation and Logic Programming. Number 1140 in LNCS, Springer-Verlag (1996) 243–258