DOI: 10.2298/CSIS121129066A

Batched Evaluation of Linear Tabled Logic Programs

Miguel Areias and Ricardo Rocha

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

Abstract. Logic Programming languages, such as Prolog, provide a high-level, declarative approach to programming. Despite the power, flexibility and good performance that Prolog systems have achieved, some deficiencies in Prolog's evaluation strategy - SLD resolution - limit the potential of the logic programming paradigm. Tabled evaluation is a recognized and powerful technique that overcomes SLD's susceptibility in dealing with recursion and redundant sub-computations. In a tabled evaluation, there are several points where we may have to choose between different tabling operations. The decision on which operation to perform is determined by the scheduling algorithm. The two most successful tabling scheduling algorithms are *local scheduling* and *batched scheduling*. In previous work, we have developed a framework, on top of the Yap Prolog system, that supports the combination of different *linear tabling strategies* for local scheduling. In this work, we propose the extension of our framework to support batched scheduling. In particular, we are interested in the two most successful linear tabling strategies, the DRA and DRE strategies. To the best of our knowledge, no other Prolog system supports both strategies simultaneously for batched scheduling. Our experimental results show that the combination of the DRA and DRE strategies can effectively reduce the execution time for batched evaluation.

Keywords: logic programming, linear tabling, scheduling.

1. Introduction

Logic programming provides a high-level, declarative approach to programming. Arguably, Prolog is one of the most popular and powerful logic programming languages. Ideally, one would want Prolog programs to be written as logical statements first, and for control to be tackled as a separate issue. In practice, the operational semantics of Prolog is given by SLD resolution [9], an evaluation strategy particularly simple but that suffers from fundamental limitations, such as in dealing with recursion and redundant sub-computations. Unfortunately, the limitations of SLD resolution mean that Prolog programmers must be concerned with SLD semantics throughout program development.

Tabling [4] is a proposal that overcomes SLD limitations in dealing with recursion and redundant sub-computations. Tabling based models are able to

reduce the search space, avoid looping, and always terminate for programs with the *bounded term-size property*[1]. In a nutshell, tabling consists of storing intermediate solutions for subgoals so that they can be reused when a similar subgoal appears during the execution of a program and, for that, the calls and the solutions to tabled subgoals are stored in a global data structure called the *table space*. Work on tabling, as initially implemented in the XSB system [11], proved its viability for application areas such as Natural Language Processing, Knowledge Based Systems, Model Checking, Program Analysis, among others.

In a tabled evaluation, there are several points where we may have to choose between continuing forward execution, backtracking, consuming solutions from the table, or completing subgoals. The decision on which operation to perform is determined by the scheduling strategy. Whereas a strategy can achieve very good performance for certain applications, for others it might add overheads and even lead to unacceptable inefficiency. The two most successful strategies are *local scheduling* and *batched scheduling* [7]. Local scheduling tries to complete subgoals as soon as possible. When new solutions are found, they are added to the table space and the evaluation fails. Solutions are only returned when all program clauses for the subgoal at hand were resolved. Batched scheduling favors forward execution first, backtracking next, and consuming solutions or completion last. It thus tries to delay the need to move around the search tree by batching the return of solutions. When new solutions are found for a particular tabled subgoal, they are added to the table space and the evaluation continues.

The main difference between the two strategies is that in batched scheduling, variable bindings are immediately propagated to the calling environment when a solution is found. For some situations, this may result in creating complex dependencies between subgoals and in having more memory space requirements. On the other hand, since local scheduling delays solutions, it does not benefit from binding propagation, and instead, when explicitly returning the delayed solutions, it incurs an extra overhead for copying them out of the table.

Currently, the tabling technique is widely available in systems like XSB Prolog [14], Yap Prolog [12], B-Prolog [15], ALS-Prolog [8], Mercury [13] and Ciao Prolog [5]. 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 solutions are correctly computed. A tabled evaluation can be seen as a sequence of sub-computations that suspend and later resume. Linear tabling mechanisms use iterative computations of tabled subgoals to compute fix-points and, for that, they maintain a single execution tree without requiring suspension and resumption of sub-computations. For that reason, linear tabling mechanisms have less memory space requirements and can be implemented with less disruption of an existing Prolog engine. On the other hand, linear tabling mechanisms can be arbitrarily

[1] A logic program has the bounded term-size property if there is a function $f : N \rightarrow N$ such that whenever a query goal Q has no argument whose term size exceeds n, then no term in the derivation of Q has size greater than $f(n)$.

slower than suspension-based tabling. However, they are still very competitive on a large number of examples. In particular, for batched scheduling, they may have an additional advantage since, with suspension-based tabling, some evaluations may require very large amounts of space.

In previous work, we have developed a framework, on top of the Yap Prolog system, that supports the combination of different linear tabling strategies for local scheduling [1, 2]. As these strategies optimize different aspects of the evaluation, they were shown to be orthogonal to each other for local scheduling. In this work, we propose the extension of our framework, to combine different linear tabling strategies, but for batched scheduling. In particular, we are interested in the two most successful linear tabling strategies, the DRA and DRE strategies [2]. To the best of our knowledge, no other Prolog tabling system supports both strategies simultaneously for batched scheduling. Extending our framework from local scheduling to batched scheduling should be, in principle, smooth but, as we will see, there are some relevant details that have to be considered in order to ensure a correct and efficient integration of the DRA and DRE strategies with batched scheduling. In more detail, this integration required changes to the table space data structures, to the tabling operations and a new mechanism to support the propagation of solutions in reevaluation rounds.

Our experimental results show that the combination of the DRA and DRE strategies can effectively reduce the execution time for batched evaluation. When compared with Yap's suspension-based mechanism, 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 such support.

The remainder of the paper is organized as follows. First, we briefly introduce the basics of tabling and describe the execution model for standard linear tabled evaluation using batched scheduling. Next, we present the DRA and DRE strategies and discuss how they optimize different aspects of the evaluation. We then describe the most relevant implementation details regarding the integration of the two strategies on top of the Yap Prolog system. Finally, we present experimental results and we end by outlining some conclusions.

2. Standard Linear Tabled Evaluation

Tabling works by storing intermediate solutions for tabled subgoals so that they can be reused when a similar[2] (or repeated) call appears. In a nutshell, first calls to tabled subgoals are considered *generators* and are evaluated as usual, using SLD resolution, but their solutions are stored in a global data space, called the *table space*. Similar calls to tabled subgoals are considered *consumers* and

[2] For the sake of simplicity, we are assuming a *variant-based tabling* mechanism, where two terms are considered to be similar if they are the same up to variable renaming. Alternatively, *subsumption-based tabling* mechanisms consider that two terms are similar if one term *subsumes* (is more general than) the other [6].

are not reevaluated against the program clauses because they can potentially lead to infinite loops, instead they are resolved by consuming the solutions already stored for the corresponding generator. During this process, as further new solutions are found, we need to ensure that they will be consumed by all the consumers, as otherwise we may miss parts of the computation and not fully explore the search space.

A generator call *C* thus keeps trying its matching clauses until a fix-point is reached. If no new solutions are found during one round of trying the matching clauses, then we have reached a fix-point and we can say that *C* is completely evaluated. However, if a number of subgoal calls is mutually dependent, thus forming a *Strongly Connected Component (SCC)*, then completion is more complex and we can only complete the calls in a SCC together [11]. SCCs are usually represented by the *leader call*, i.e., the generator call which does not depend on older generators. A leader call defines the next completion point, i.e., if no new solutions are found during one round of trying the matching clauses for the leader call, then we have reached a fix-point and we can say that all subgoal calls in the SCC are completely evaluated.

We next illustrate in Fig. 1 the standard execution model for linear tabling using batched scheduling. 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 defines two tabled predicates, *a/1* and *b/1*, each defined by two clauses (clauses *c1* to *c4*). The bottom sub-figure shows the evaluation sequence, numbered in order of evaluation, for the query goal *a(X)*. Generator calls are depicted by black oval boxes and consumer calls by white oval boxes.

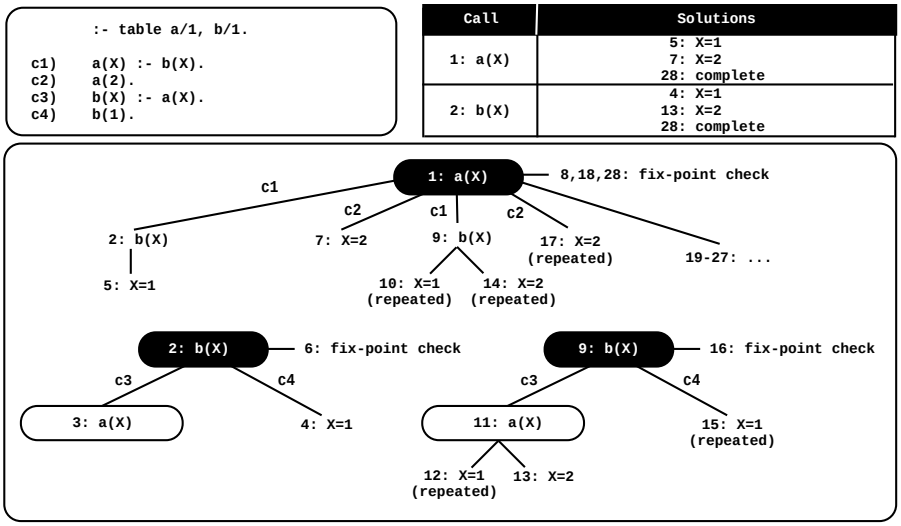

Fig. 1. A standard linear tabled evaluation using batched scheduling

The evaluation starts by inserting a new entry in the table space representing the generator call *a(X)* (step 1). Then, *a(X)* is resolved against its first matching clause, clause *c1*, calling *b(X)* in the continuation. As this is a first call to *b(X)*, we insert a new entry in the table space representing *b(X)* and proceed as shown in the bottom left tree (step 2). Subgoal *b(X)* is also resolved against its first matching clause, clause *c3*, calling again *a(X)* in the continuation (step 3). Since *a(X)* is a repeated call, we try to consume solutions from the table space, but at this stage no solutions are available, so execution fails.

We then try the second matching clause for *b(X)*, clause *c4*, and a first solution for *b(X)*, $\{X{=}1\}$, is found and added to the table space (step 4). We then follow a batched scheduling strategy and the evaluation continues with *forward execution* [7]. With batched scheduling, new solutions are immediately returned to the calling environment, thus the solution for *b(X)* should now be propagated to the context of the previous call, which also originates a first solution for *a(X)*, $\{X{=}1\}$ (step 5).

The execution then fails back to node 2 and we check for a fix-point (step 6), but *b(X)* is not a leader call because it has a dependency (consumer node 3) to an older call, *a(X)*. Remember that we reach a fix-point when no new solutions are found during the last round of trying the matching clauses for the leader call. Then, we try the second matching clause for *a(X)* and a second solution for it, $\{X{=}2\}$, is found and added to the table space (step 7). We then backtrack again to the generator call for *a(X)* and because we have already explored all matching clauses, we check for a fix-point (step 8). We have found new solutions for both *a(X)* and *b(X)* in this round, thus the current SCC is scheduled for reevaluation.

The evaluation then repeats the same sequence as in steps 2 to 3 (now steps 9 to 11), but since we are following a batched scheduling strategy, we first consume the solutions already available for *b(X)* (this will be further explained later in section 4), which leads to a repeated solution for *a(X)* (step 10). Tabling does not store duplicate solutions in the table space. Instead, repeated solutions fail. Next, the evaluation moves to the consumer call of *a(X)* (step 11). Solution $\{X{=}1\}$ is first forwarded to it, which originates a repeated solution for *b(X)* (step 12) and thus execution fails. Then, solution $\{X{=}2\}$ is also forward to it and a new solution for *b(X)* is found (step 13) and propagated to *a(X)*, which leads to a repeated solution for *a(X)* (step 14).

In the continuation, we find another repeated solution for *b(X)* (step 15) and we fail a second time in the fix-point check for *b(X)* (step 16). Again, as we are following a batched scheduling strategy, the solutions for *b(X)* were already all propagated to the context of *a(X)*, thus we can safely backtrack to the generator call for *a(X)*. Because we have found a new solution for *b(X)* during this last round, the current SCC is scheduled again for reevaluation (step 18). The reevaluation of the SCC does not find new solutions for both *a(X)* and *b(X)* (steps 19 to 27). Thus, when backtracking again to *a(X)* we have reached a fix-point and because *a(X)* is a leader call, we can declare the two subgoal calls to be completed (step 28).

3. Linear Tabling Strategies

The standard linear tabling mechanism uses a naive approach to evaluate tabled logic programs. Every time a new solution is found during the last round of evaluation, the complete search space for the current SCC is scheduled for reevaluation. However, some branches of the SCC can be avoided, since it is possible to know beforehand that they will only lead to repeated computations, hence not finding any new solutions. Next, we present two different strategies for optimizing the standard linear tabled evaluation. The common goal of both strategies is to minimize the number of branches to be explored, thus reducing the search space, and each strategy tries to focus on different aspects of the evaluation to achieve it.

3.1. Dynamic Reordering of Alternatives

The key idea of the *Dynamic Reordering of Alternatives (DRA)* strategy, as originally proposed by Guo and Gupta [8], is to memoize the clauses (or alternatives) leading to consumer calls, the *looping alternatives*, in such a way that when scheduling an SCC for reevaluation, instead of trying the full set of matching clauses, we only try the looping alternatives.

Initially, a generator call *C* explores the matching clauses as in standard linear tabled evaluation and, if a consumer call is found, the current clause for *C* is memoized as a looping alternative. After exploring all the matching clauses, *C* enters the *looping state* and from this point on, it only tries the looping alternatives until a fix-point is reached. Figure 2 uses the same program from Fig. 1 to illustrate how DRA evaluation works.

The evaluation sequence for the first SCC round (steps 2 to 7) is identical to the standard evaluation of Fig. 1. The difference is that this round is also used to detect the alternatives leading to consumers calls. We only have one consumer call at node 3 for *a(X)*. The clauses in evaluation up to the corresponding generator, call *a(X)* at node 1, are thus marked as looping alternatives and added to the respective table entries. This includes alternative *c3* for *b(X)* and alternative *c1* for *a(X)*. As for the standard strategy, the SCC is then scheduled for two extra reevaluation rounds (steps 9 to 15 and steps 17 to 23), but now only the looping alternatives are evaluated, which means that the clauses *c2* and *c4* are ignored.

3.2. Dynamic Reordering of Execution

The second strategy, that we call *Dynamic Reordering of Execution (DRE)*, is based on the original SLDT strategy, as proposed by Zhou et al. [16]. The key idea of the DRE strategy is to give priority to the program clauses and, for that, it lets repeated calls to tabled subgoals execute from the *backtracking clause of the former call*. A first call to a tabled subgoal is called a *pioneer* and repeated calls are called *followers* of the pioneer. When backtracking to a pioneer or a follower, we use the same strategy and we give priority to the exploitation of the

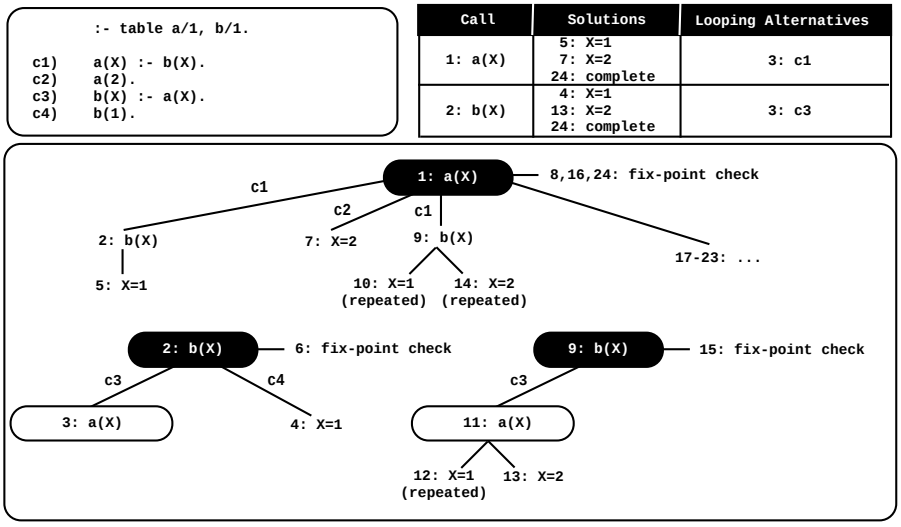

Fig. 2. A linear tabled evaluation using batched scheduling with DRA evaluation

remaining clauses. The fix-point check operation is still performed by pioneer calls. Figure 3 uses again the same program from Fig. 1 to illustrate how DRE evaluation works.

As for the standard strategy, the evaluation starts with (pioneer) calls to *a(X)* (step 1) and *b(X)* (step 2), and then, in the continuation, *a(X)* is called repeatedly (step 3). With DRE evaluation, *a(X)* is now considered a follower and thus we *steal* the backtracking clause of the former call at node 1, i.e., clause *c2*. The evaluation then proceeds as for a generator call (right upper tree in Fig. 3), which means that new solutions can be generated for *a(X)*. We thus try clause *c2* and a first solution for *a(X)*, $\{X{=}2\}$, is found and added to the table space (step 4). Then, we follow a batched scheduling strategy and the solution $\{X{=}2\}$ is propagated to the context of *b(X)*, which originates the solution $\{X{=}2\}$ (step 5), and to the context of *a(X)*, which leads to a repeated solution (step 6).

As both matching clauses for *a(X)* were already taken, the execution backtracks to the pioneer node 2. Next, we find a second solution for *b(X)* (step 7), which is then propagated, leading also to a second solution for *a(X)* (step 8). In step 9, we check for a fix-point, but *b(X)* is not a leader call because it has a dependency (follower node 3) to an older call, *a(X)*. We then backtrack to the pioneer call for *a(X)* and because we have already explored the matching clause *c2* in the follower node 3, we check for a fix-point. Since we have found new solutions during the last round, the current SCC is scheduled for reevaluation (step 10). As the full set of solutions was already found during the first round, the reevaluation of the SCC does not find any further solutions (steps 11 to 19), and thus the evaluation can be completed at step 20.

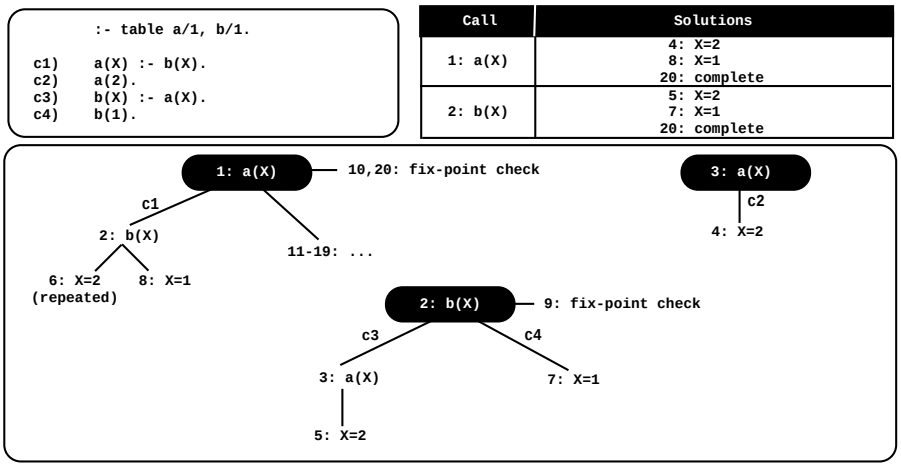

Fig. 3. A linear tabled evaluation using batched scheduling with DRE evaluation

4. Propagation of Solutions in Reevaluation Rounds

In the previous sections, one could observe that tabling does not store duplicate solutions in the table space and, instead, repeated solutions fail. This is how tabling avoids unnecessary computations and looping for duplicate solutions. However, since repeated solutions also fail in reevaluation rounds, this means that, in fact, a solution is only propagated once, i.e., in the round it is first found, which might be not sufficient to ensure the completeness of the evaluation. To solve this problem, in a reevaluation round, we start by propagating (consuming) the solutions already available for the subgoal call at hand. Alternatively, we could propagate the solutions at the end, after the fix-point check procedure, but by doing that some solutions will be propagated more than once in the same round, which is worthless.

In the previous examples, for simplicity of explanation, we have omitted some steps regarding the propagation of solutions in the leader call since, for all the examples, one propagation per solution was enough to correctly compute the corresponding evaluations. To better illustrate the importance of the propagation of solutions in reevaluation rounds and, in particular, for the leader call, Fig. 4 shows a new example, using again the same program from Fig. 1, but for the query goal *a(X1), b(X2)*. For simplicity of explanation, we consider a standard linear tabled evaluation, i.e., without DRA and DRE support. In order to have a common representation of variables between the program code, the evaluation and the table space, the different calls to both *a/1* and *b/1* are presented using a generic variable *X*, instead of the *real* variables *X1* and *X2*.

In the first round of the evaluation (steps 1 to 12), the solutions found for *a(X)*, at steps 5 and 9, are propagated to the context of *a(X1)* and, in the continuation, *b(X2)* consumes (note that at this point *b(X2)* is a repeated call

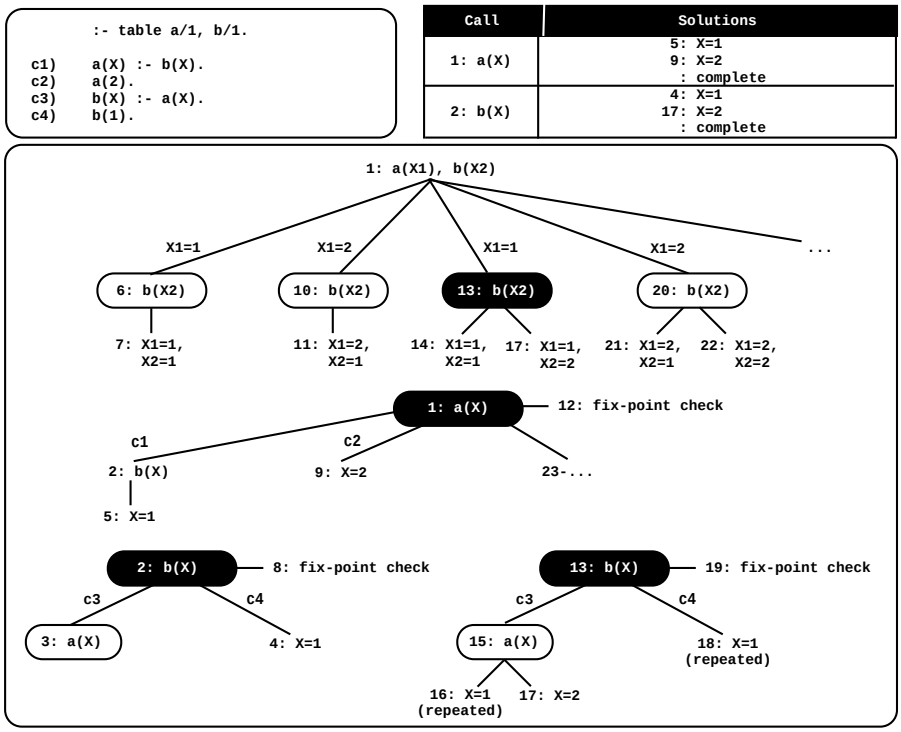

Fig. 4. Propagation of solutions in reevaluation rounds using batched scheduling

to *b(X)*) the available solution found at step 4, which originates the solutions {*X1=1, X2=1*} (step 7) and {*X1=2, X2=1*} (step 11) for the top query goal.

Next, in the second round of the evaluation, the leader call starts by propagating its first solution, calling *b(X2)* in the continuation (step 13). Since this is the first call to *b(X)* in this round, *b(X2)* also starts by propagating its current solution (step 14). Then, when reevaluating the program clauses for *b(X)* (steps 15 to 18), a new solution {*X=2*} is found (step 17). The combination of this new solution with the previous solutions for *a(X1)* originates two new solutions, {*X1=1, X2=2*} (step 17) and {*X1=2, X2=2*} (step 22), for the top query goal.

Notice that without the propagation of solutions for the leader call (steps 13 and 20), no further solutions had been found for the top query goal. In particular, the solution {*X=2*} for *b(X)* would have been found in the context of *a(X)* (similarly to the solution {*X=1*} found at step 4) but, since this originates a repeated solution for *a(X)*, the computation will fail. By failing for *a(X)*, we cannot combine the new solution for *b(X)* with the previous solutions for *a(X1)* at the top query goal. Hence, this fact, i.e., the fact that tabling fails for repeated solutions, can lead to a collateral effect where it can be blocking forward execution. To solve this problem, in a reevaluation round, we start by propagating all the available solutions.

5. Implementation Details

This section describes the implementation details regarding the extension of our framework to support batched scheduling, with particular focus on the table space data structures and on the tabling operations.

5.1. Table Space

To implement the table space, Yap uses *tries* which is considered a very efficient data structure to implement the table space [10]. Tries are trees in which common prefixes are represented only once. Tries provide complete discrimination for terms and permit look up and insertion to be done in a single pass.

In more detail, a trie is a tree structure where each different path through the *trie nodes* corresponds to a term described by the tokens labeling the traversed nodes. For example, the tokenized form of the term *p(X,1,f(Y))* is the sequence of 5 tokens *p/3*, VAR_0, 1, *f/1* and VAR_1, where each variable is represented as a distinct VAR_i constant [3]. Two terms with common prefixes will branch off from each other at the first distinguishing token. Consider, for example, a second term *p(Z,1,b)*. Since the main functor, token *p/3*, and the first two arguments, tokens VAR_0 and 1, are common to both terms, only one additional node will be required to fully represent this second term in the trie, thus allowing to save three trie nodes in this case.

As other tabling engines, Yap uses two levels of tries: one for the subgoal calls and other for the computed solutions. A tabled predicate accesses the table space through a specific *table entry* data structure. Each different subgoal call is represented as a unique path in the *subgoal trie* and each different solution is represented as a unique path in the *solution trie*. Contrary to subgoal tries, solution trie paths hold just the substitution terms for the free variables that exist in the argument terms of the corresponding subgoal call [10]. An example for a tabled predicate *p/3* is shown in Fig. 5.

Initially, the table entry for *p/3* points to an empty subgoal trie. Then, the subgoal *p(X,1,Y)* is called and three trie nodes are inserted to represent the arguments in the call: one for variable *X* (VAR_0), a second for integer 1, and a last one for variable *Y* (VAR_1). Since the predicate's functor term is already represented by its table entry, we can avoid inserting an explicit node for *p/3* in the subgoal trie. Then, the leaf node is set to point to a subgoal frame, from where the answers for the call will be stored. The example shows two answers for *p(X,1,Y)*: $\{X{=}VAR_0,\ Y{=}f(VAR_1)\}$ and $\{X{=}VAR_0,\ Y{=}b\}$. Since both answers have the same substitution term for argument *X*, they share the top node in the answer trie (VAR_0). For argument *Y*, each answer has a different substitution term and, thus, a different path is used to represent each.

When adding answers, the leaf nodes are chained in a linked list in insertion time order, so that the recovery may happen the same way. In Fig. 5, we can observe that the leaf node for the first answer (node VAR_1) points (dashed arrow) to the leaf node of the second answer (node *b*). To maintain this list, two fields in the subgoal frame data structure point, respectively, to the first and last answer

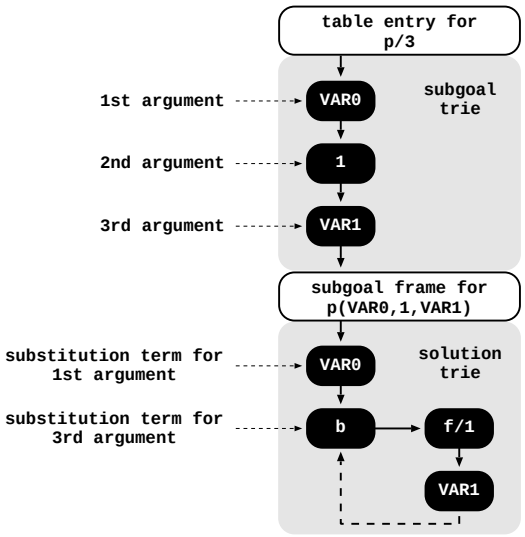

Fig. 5. Table space organization

of this list (for simplicity of illustration, these pointers are not shown in Fig. 5). When consuming answers, a consumer node only needs to keep a pointer to the leaf node of its last loaded answer, and consumes more answers just by following the chain. Answers are loaded by traversing the trie nodes bottom-up (again, for simplicity of illustration, such pointers are not shown in Fig. 5).

A key data structure in this organization is the *subgoal frame*. Subgoal frames are used to store information about each tabled subgoal call, namely: the entry point to the solution trie; the state of the subgoal (*ready*, *evaluating* or *complete*); support to detect if the subgoal is a leader call; and support to detect if new solutions were found during the last round of evaluation. The DRA and DRE strategies extend the subgoal frame data structure with the following extra information [2]: support to detect, store and load looping alternatives; two new states used to detect generator and consumer calls in reevaluating rounds (*loop_ready* and *loop_evaluating*); the pioneer call; and the backtracking clause of the former call. In more detail, the most relevant subgoal frame fields in our implementation are:

***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_prev_on_scc*:** points to the subgoal frame corresponding to the previous 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 reevaluation or as completed. A global variable *TOP_SCC* always points to the youngest subgoal frame in evaluation in the current SCC.

***SgFr_prev_on_branch*:** points to the subgoal frame corresponding to the previous call in the current branch that is in the first round (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 in the current branch.

SgFr_new_solutions: indicates if new solutions were found during the execution of the current round.

***SgFr_first_solution*:** points to the leaf trie node corresponding to the first available solution.

***SgFr_last_solution*:** points to the leaf trie node corresponding to the last available solution.

***SgFr_last_consumed*:** marks the last solution consumed in a generator (pioneer or follower) call (supports the propagation of solutions, as discussed in section 4).

5.2. Tabling Operations

We next introduce the pseudo-code for the main tabling operations required to support batched scheduling with DRA and DRE evaluation.

We start with Algorithm 1 showing the pseudo-code for the new solution operation. Initially, the operation simply inserts the given solution *SOL* in the solution trie structure for the given subgoal frame *SF* (line 1) and, if the solution is new, it updates the *SgFr_new_solutions* subgoal frame field to *TRUE* (line 2) and proceeds with forward execution as usual. Otherwise, the solution is repeated and execution fails (line 4).

Algorithm 1 $new\_solution(solution\ SOL,\ subgoal\ frame\ SF)$

```
1: if solution_check_insert(SOL, SF) = true then {new solution}
2:     SgFr_new_solutions(SF) ← true
3: else
4:     fail()
```

Next, in Algorithm 2, we show the pseudo-code for the tabled call operation. Initially, the operation starts by inserting the given subgoal call *SC* in the subgoal trie structure, from where a subgoal frame *SF*, representing the given call, is obtained (line 1). New calls to tabled subgoals are inserted into the table space by allocating the necessary data structures, which includes a new subgoal frame (this is the case where the state of *SF* starts to be *ready*). In

such case, the tabled call operation then stores a new generator node[3] (line 3); updates the state of *SF* to *evaluating* (line 4); defines a new SCC (lines 5-6); adds *SF* to the current branch (lines 7-8); and proceeds by executing the current alternative (line 9).

Algorithm 2 $tabled\_call(subgoal\ call\ SC)$

```
 1: SF ← subgoal_check_insert(SC) {SF is the subgoal frame for the subgoal call SC}
 2: if SgFr_state(SF) = ready then {new call}
 3:    store_generator_node()
 4:    SgFr_state(SF) ← evaluating
 5:    SgFr_prev_on_scc(SF) ← TOP_SCC {new SCC}
 6:    TOP_SCC ← SF
 7:    SgFr_prev_on_branch(SF) ← TOP_BRANCH {add to current branch}
 8:    TOP_BRANCH ← SF
 9:    goto evaluate(current_alternative())
10: else if SgFr_state(SF) = complete then {already evaluated}
11:    goto completed_table_optimization(SF)
12: else if SgFr_state(SF) = loop_ready then {first call in reevaluation round}
13:    store_generator_node()
14:    SgFr_state(SF) ← loop_evaluating
15:    SgFr_prev_on_scc(SF) ← TOP_SCC {new SCC}
16:    TOP_SCC ← SF
17:    SgFr_last_consumed(SF) ← SgFr_first_solution(SF)
18:    if DRA_mode(SF) then
19:       goto consume_solutions_and_reevaluate(SF, first_looping_alternative())
20:    else
21:       goto consume_solutions_and_reevaluate(SF, first_alternative())
22: else if SgFr_state(SF) = evaluating or SgFr_state(SF) = loop_evaluating then
23:    mark_current_branch(SF)
24:    if DRE_mode(SF) and has_unexploited_alternatives(SF) then
25:       store_follower_node()
26:       if DRA_mode(SF) and SgFr_state(SF) = loop_evaluating then
27:          goto consume_solutions_and_reevaluate(SF, next_looping_alternative())
28:       else
29:          goto consume_solutions_and_reevaluate(SF, next_alternative())
30:    else
31:       store_consumer_node()
32:       goto consume_solutions(SF)
```

On the other hand, if the subgoal call is a repeated call, then the subgoal frame *SF* is already in the table space, and three different situations may occur. First, if the call is already evaluated (this is the case where the state of *SF* is *complete*), the operation consumes the available solutions by implementing the

[3] Generator, consumer and follower nodes are implemented as regular choice points extended with some extra fields related to the table space data structures.

completed table optimization [10] which executes compiled code directly from the solution trie structure associated with the completed call (line 11).

Second, if the call is a first call in a reevaluating round (this is the case where the state of *SF* is *loop_ready*), the operation stores a new generator node (line 13); updates the state of *SF* to *loop_evaluating* (line 14); defines a new SCC (lines 15-16); and resets the *SgFr_last_consumed* field to the first solution (line 17). Then, it executes the *consume_solutions_and_reevaluate()* procedure in order to consume the available solutions before reevaluate the matching alternatives. This procedure, consumes all the available solutions for the subgoal, starting from the first solution, and, when no more solutions are to be consumed, it starts with the evaluation of the first matching alternative, which for DRA is the first looping alternative (lines 18-21).

Third, if the call is a repeated call (this is the case where the state of *SF* is *evaluating* or *loop_evaluating*), the operation first calls the *mark_current_branch()* procedure (please see Algorithm 3 next) in order to mark the current branch as a non-leader branch and, if in DRA mode, also mark the current branch as a looping branch (line 23). Next, if DRE mode is enabled and there are unexploited alternatives (i.e., there is a backtracking clause for the former call), it stores a follower node (line 25) and proceeds by consuming the available solutions before executing the next looping alternative or the next matching alternative, according to whether the DRA mode is enabled or disabled for the subgoal (lines 26-29). Otherwise, it stores a new consumer node and starts consuming the available solutions (lines 31-32).

Algorithm 3 shows the details for the *mark_current_branch()* procedure. To mark the current branch as a non-leader branch and, if in DRA mode, as a looping branch, we follow the *TOP_BRANCH* chain and for all intermediate generator calls in evaluation up to the generator call for *SF*, we mark them as non-leader calls (note that the call at hand defines a new dependency for the current SCC) and we mark the alternatives being evaluated by each call as looping alternatives.

Algorithm 3 $mark\_current\_branch(subgoal\ frame\ SF)$

```
1: aux_sf ← TOP_BRANCH
2: while SgFr_dfn(aux_sf) > SgFr_dfn(SF) do
3:    SgFr_is_leader(aux_sf) ← false
4:    if DRA_mode(aux_sf) then
5:       mark_current_alternative_as_looping_alternative(aux_sf)
6:    aux_sf ← SgFr_prev_on_branch(aux_sf)
7: if DRA_mode(aux_sf) then
8:    mark_current_alternative_as_looping_alternative(aux_sf)
```

Finally, we discuss in more detail how completion is detected with batched scheduling. Remember that after exploring the last matching clause for a tabled

call, we execute the *fix-point check* operation. Algorithm 4 shows the pseudo-code for its implementation.

Algorithm 4 $fix\_point\_check(subgoal\ frame\ SF)$

```
1: if (SgFr_is_leader(SF)) then
2:    if SgFr_new_solutions(SF) then {start a new round}
3:       for all SG such that SG in current SCC do
4:          SgFr_state(SG) ← loop_ready
5:       SgFr_state(SF) ← loop_evaluating
6:       TOP_SCC ← SF
7:       SgFr_new_solutions(SF) ← false
8:       SgFr_last_consumed(SF) ← SgFr_first_solution(SF)
9:       if DRA_mode(SF) then
10:         goto consume_solutions_and_reevaluate(SF, first_looping_alternative())
11:      else
12:         goto consume_solutions_and_reevaluate(SF, first_alternative())
13:   else {reached a fix-point}
14:      for all SG such that SG in current SCC do {complete all subgoals in SCC}
15:         SgFr_state(SG) ← complete
16:      TOP_SCC ← SgFr_prev_on_scc(SF)
17:      fail()
18: else {not a leader call}
19:   if SgFr_state(SF) = evaluating then {first round}
20:      TOP_BRANCH ← SgFr_prev_on_branch(SF)
21:   if SgFr_new_solutions(SF) then {propagate new solutions}
22:      SgFr_new_solutions(current_leader(SF)) ← true
23:   SgFr_new_solutions(SF) ← false
24:   fail()
```

The fix-point check operation starts by verifying if the subgoal at hand is a leader call. If it is leader and has found new solutions during the last round, then the current SCC is scheduled for a reevaluation round (lines 3-12). This includes updating the state for all subgoals in the current SCC, updating the *TOP_SCC* variable to the current subgoal frame and resetting the *SgFr_new_solutions* field to *FALSE* (lines 3-7). Then, as for a first call in a reevaluating round in the *tabled call* operation, it also resets the *SgFr_last_consumed* field to the first solution (line 8) and executes the *consume_solutions_and_reevaluate()* procedure (lines 9-12).

On the other hand, if the subgoal is leader but no new solutions were found during the current round, then we have reached a fix-point. All subgoals in the current SCC are thus marked as completed, the *TOP_SCC* variable is updated to the next subgoal frame and the evaluation fails (lines 14-17).

Otherwise, the subgoal is not a leader call. Then it removes itself from the *TOP_BRANCH* chain (lines 19-20), propagates the new solutions information to the current leader of the SCC (lines 21-22), resets the *SgFr_new_solutions* field

to *FALSE* (line 23) and then fails (line 24). Note that, with batched scheduling, we can safely fail since all the solutions were already propagated to the context of the calling environment. Moreover, since the *SgFr_new_solutions* flag is propagated to the leader of the SCC, the leader will mark the SCC for a reevaluation round, which means that the current subgoal will be called again, and so it will start by consuming its solutions.

As an optimization, a non-leader call *C* executing the fix-point check operation can be removed beforehand from the *TOP_BRANCH* chain (lines 19-20 in Algorithm 4) since we already know that it is a non-leader call and have marked its looping alternatives. Thus, when we execute the *mark_current_branch()* procedure in a reevaluation round for a call *C*, then *C* might have been removed from the chain in a previous fix-point check operation. This is the reason why we need to follow the subgoal frames in the *TOP_BRANCH* chain up to the first subgoal frame with a smaller *SgFr_dfn* value than *C* (while loop on Algorithm 3).

6. Experimental Results

To the best of our knowledge, Yap is now the first tabling engine that integrates and supports the combination of different linear tabling strategies using batched scheduling. We have thus the conditions to better understand the advantages and weaknesses of each strategy when used solely or combined. In what follows, we present experimental results comparing linear tabled evaluation with and without DRA and DRE support, using batched scheduling. To put our results in perspective, we have also included experiments for the B-Prolog linear tabling system [15] and for the YapTab suspension-based tabling system [12], both using batched scheduling. In fact, for B-Prolog, we used its *eager scheduling mode*, which is similar to batched scheduling. The environment for our experiments was a PC with a 2.83 GHz Intel(R) Core(TM)2 Quad CPU and 8 GBytes of memory running the Linux kernel 3.0.0-16-generic. We used B-Prolog version 7.5 and Yap version 6.0.7[4].

For benchmarking, we used three sets of programs. The **Model Checking** set includes three different specifications and transition relation graphs usually used in model checking applications: **IProto**, the transition relation graph for the i-protocol specification defined for a correct version (fix) with a huge window size (w = 2); **Leader**, the transition relation graph for the leader election specification defined for 5 processes; and **Sieve**, the transition relation graph for the sieve specification defined for 5 processes and 4 overflow prime numbers. The **Path Right** set implements the right recursive definition of the well-known $path/2$ predicate, that computes the transitive closure in a graph, using three different edge configurations. Figure 6 shows an example for each configuration. We experimented the **Pyramid** and **Cycle** configurations with depths 1000, 2000 and 3000 and the **Grid** configuration with depths 20, 30 and 40. We chose this set of experiments because the $path/2$ predicate implements a relatively easy to

[4] Source code available from http://cracs.fc.up.pt/node/5121

understand pattern of computation and its right recursive definition creates several inter-dependencies between tabled subgoals. The **Warren** set is a variation of the left recursive definition of the path problem for a linear graph (see Fig. 6), where the *path/2* clauses are duplicated to be used with the labels *a* and *b*. This problem was kindly suggested by David S. Warren as a way to stress the performance of a linear tabling system. All benchmarks find all the solutions for the problem.

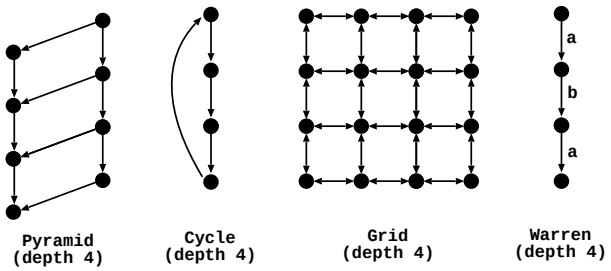

Fig. 6. Edge configurations used with the second and third set of problems

In Table 1, we show the execution time, in milliseconds, for standard linear tabling (column **Std**) and the respective execution time ratios for DRA and DRE evaluation, solely and combined (column **DRA+DRE**), B-Prolog and YapTab, using batched scheduling, for the **Model Checking**, **Path Right** and **Warren** sets of problems. Ratios higher than 1.00 mean that the respective strategy has a positive impact on the execution time, when compared with standard linear tabling. The ratio marked with $n.c.$ for B-Prolog means that we are *not considering* it in the average results (for some reason, we failed in executing this benchmark). The results are the average of five runs for each benchmark.

In addition to the results presented in Table 1, we also collected several statistics regarding important aspects of the evaluation. In Table 2, we show some of these statistics for standard linear tabling and the respective performance ratios when compared with the other models, for a subset of the benchmarks. We used the **Leader** specification for the **Model Checking** set, the configurations **Pyramid** and **Cycle** with depth 2000 and **Grid** with depth 30 for the **Path Right** set, and the configuration with depth 600 for the **Warren** set.

The statistics in Table 2 measure how the mixing with SLD (non-tabled) computations can affect the base performance of our benchmarks. For that, we extended the tabled predicates, at the beginning and at the end of each clause, with dummy SLD (non-tabled) predicates, which we named *sldi/0*, with $0 < i \leq 2n$, where n is the number of clauses defining the tabled predicate. For example, the extended definition for the *path/2* predicate is:

```
path(X,Z) :- sld1, edge(X,Y), path(Y,Z), sld2.
path(X,Z) :- sld3, edge(X,Z), sld4.
```

Table 1. Execution time, in milliseconds, for standard linear tabling and the respective execution time ratios for DRA and DRE evaluation, solely and combined, B-Prolog and YapTab, using batched scheduling (for the linear tabling models, best ratios are in bold)

Benchmark	Std	DRA	DRE	DRA+DRE	B-Prolog	YapTab
Model Checking						
IProto	2,874	**1.00**	0.50	0.93	0.36	2.39
Leader	5,355	**1.01**	0.40	0.99	0.13	2.83
Sieve	35,218	**1.00**	0.46	0.93	0.16	3.19
Average ratio		1.00	0.45	0.95	0.22	2.80
Path Right - Pyramid						
1000	983	**1.87**	0.89	1.49	1.04	1.90
2000	3,897	**1.88**	0.89	1.49	0.69	1.94
3000	9,043	**1.91**	0.89	1.53	$n.c.$	1.98
Path Right - Cycle						
1000	687	**1.27**	0.96	1.22	**1.27**	1.89
2000	2,793	**1.27**	0.97	1.22	0.91	1.82
3000	6,048	**1.29**	0.95	1.22	0.70	2.05
Path Right - Grid						
20	221	**1.33**	0.97	1.27	1.09	2.10
30	1,344	**1.32**	0.99	1.30	1.02	2.22
40	4,578	**1.31**	0.97	1.26	0.76	2.34
Average ratio		**1.50**	0.94	1.33	0.93	2.03
Warren						
400	2,673	1.02	**64.26**	**64.26**	0.34	126.09
600	9,496	0.99	**87.28**	**87.28**	0.35	162.61
800	23,163	1.00	112.88	**116.98**	0.35	216.88
Average ratio		1.00	**87.93**	**89.51**	0.35	168.53

The rows in Table 2 show the number of times each dummy SLD predicate is called for the corresponding benchmark. We can read these numbers as an estimation of the performance ratios that we will obtain if the execution time of the corresponding SLD predicate clearly overweights the execution time of the other computations. Note that the odd SLD predicates (such as **sld1** and **sld3**) correspond to re-executions of a clause and that the even SLD predicates (such as **sld2** and **sld4**) correspond to new solution operations. In our experiments, the **sld2** predicate (placed at the end of the first tabled clause) is the one that can potentially have a greater influence in the performance ratios as it clearly exceeds all the others in the number of times it is called (see Table 2).

7. Discussion

Analyzing the general picture of Table 1, the results show that DRA evaluation is able to reduce the execution time for the **Path Right** problem set (1.50 times faster, on average) but has no impact for the other two sets, when compared with standard evaluation. The results also indicate that DRE evaluation has a negative impact in the execution time for the **Model Checking** and **Path Right**

Table 2. Number of calls to the dummy SLD predicates for standard linear tabling and the respective ratios for DRA and DRE evaluation, solely and combined, B-Prolog and YapTab, using batched scheduling (for the linear tabling models, best ratios are in bold)

Benchmark	Std	DRA	DRE	DRA+DRE	B-Prolog	YapTab
Model Checking - Leader						
sld1	3	1.00	0.75	1.00	1.00	3.00
sld2	1,153,026	1.00	0.40	1.00	1.00	2.00
sld3	3	**3.00**	0.75	**3.00**	**3.00**	3.00
sld4	3	**3.00**	0.75	**3.00**	**3.00**	3.00
Path Right - Pyramid 2000						
sld1	7,999	**2.00**	1.00	**2.00**	**2.00**	2.00
sld2	37,951,017	**2.38**	0.86	1.73	**2.38**	2.38
sld3	7,999	**2.00**	1.00	**2.00**	**2.00**	2.00
sld4	23,988	**2.00**	1.00	**2.00**	**2.00**	2.00
Path Right - Cycle 2000						
sld1	6,002	1.00	1.00	1.00	1.00	3.00
sld2	18,003,000	**1.29**	1.00	**1.29**	**1.29**	2.25
sld3	6,002	**3.00**	1.00	**3.00**	**3.00**	3.00
sld4	10,000	**2.50**	1.00	**2.50**	**2.50**	2.50
Path Right - Grid 30						
sld1	2,702	1.00	1.00	1.00	0.18	3.00
sld2	13,851,534	1.29	1.00	**1.30**	0.30	2.21
sld3	2,702	**3.00**	1.00	1.02	**3.00**	3.00
sld4	17,400	**2.50**	1.00	1.27	**2.50**	2.50
Warren - 600						
sld1/sld3	302	1.00	**100.67**	**100.67**	1.00	302.00
sld2/sld4	18,044,650	1.00	66.98	**100.42**	1.00	201.17
sld5/sld7	302	**302.00**	100.67	**302.00**	**302.00**	302.00
sld6/sld8	90,600	**302.00**	100.67	**302.00**	**302.00**	302.00

sets but, on the other hand, it can significantly reduce the execution time for the **Warren** set (more than 80 times faster, on average). We next discuss in more detail each strategy.

DRA: the results for DRA evaluation show that the strategy of avoiding the exploration of non-looping alternatives in reevaluation rounds is quite effective in general and does not add extra overheads when not used. The results also show that, for the **Path Right** set, DRA is more effective for programs without loops, like the **Pyramid** configurations, than for programs with larger SCCs, like the **Cycle** and **Grid** configurations. On Table 2, we can observe that the number of dummy SLD computations is, in fact, effectively reduced with DRA evaluation.

DRE: for the **Model Checking** set, DRE evaluation is around two times slower than standard evaluation and, for the **Path Right** set, DRE has no significant impact for all the configurations. Table 2 confirms that, the strategy of allocating follower nodes, adds an extra complexity to the evaluation for the **Model Checking** set (the number of dummy SLD calls is higher) and that

it has no impact for the **Path Right** set (the number of dummy SLD calls is identical to standard evaluation). For the **Warren** set, DRE evaluation produces the most interesting results. Note that, this is the set of benchmarks where suspension-based tabling (the YapTab system) is far more faster than standard linear tabling (168.53 times faster, on average) and the difference increases as the depth of the problem also increases. However, DRE evaluation is able to reduce this huge difference to a minimum. On average, DRE evaluation is 87.93 times faster than standard evaluation and the scalability, as the depth of the problem increases, is similar to the one observed for YapTab. Table 2 confirms this behavior for DRE and YapTab evaluations (the number of dummy SLD calls is clearly lower than standard evaluation).

Regarding the combination of both strategies (DRA+DRE), our experiments show that, in general, the best of both worlds is always present in the combination. The results in Table 1 show that, by combining both strategies, DRA is able to avoid DRE behavior for the **Model Checking** and **Path Right** sets. Still, the results for DRA+DRE are slightly worst than DRA used solely. For the **Warren** set, the results show that, by combining both strategies, it is possible to reduce even further the execution time when compared with DRE used solely. In particular, one can observe that, for depths 400 and 600, the execution times are the same but, for depth 800, DRA+DRE evaluation outperforms DRE used solely.

The statistics on Table 2 confirm that, in general, the best of both worlds is always present in the combination. The exceptions are the **sld2** predicate, for the **Pyramid 2000** configuration, and the **sld3** and **sld4** predicates, for the **Grid 30** configuration. On the other hand, for the **Warren 600** configuration, the **sld1**/**sld3** predicates are executed the same number of times as for DRE used solely, the **sld5** to **sld8** predicates are executed the same number of times as for DRA used solely, and the **sld2** and **sld4** predicates are executed less times than both strategies used solely, which is explained by the fact that the fix-point is achieved in less rounds (statistics not shown here).

Regarding the comparison with the B-Prolog linear tabling system, the results in Table 2 suggest that B-Prolog implements a DRA-based evaluation strategy since the statistics for B-Prolog and DRA evaluation are all the same, except for the **sld1** and **sld2** predicates in the **Grid 30** configuration. However, the execution times in Table 1 show that our DRA implementation is always faster than B-Prolog in these experiments and that, for almost all configurations, the ratio difference shows a generic tendency to increase as the depth of the problem also increases.

For all experiments, the results obtained for the YapTab suspension-based system clearly outperform the standard linear tabled evaluation but, for our DRA+DRE implementation, they are globally comparable. On average, YapTab is around 2 times faster than DRA+DRE evaluation, including the **Warren** problem set, where YapTab shows a huge difference for standard linear tabling. The results also indicate that our implementation scales as well as YapTab when we increase the depth of the problem being tested.

8. Conclusions

We have presented a new linear tabling framework that integrates and supports batched scheduling with DRA and DRE evaluation, solely or combined. We discussed how these strategies can optimize different aspects of a tabled evaluation and we presented the relevant implementation details for their integration on top of the Yap system.

Our experimental results were very interesting and very promising. In particular, the combination of DRA with DRE showed the potential of our framework to effectively reduce the execution time of the standard linear tabled evaluation. When compared with YapTab's suspension-based mechanism, 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 such support.

Further work will include adding new strategies/optimizations to our framework, and exploring the impact of applying our strategies to more complex problems, seeking real-world experimental results, allowing us to improve and consolidate our current implementation.

Acknowledgments. This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Technology) within projects LEAP (FCOMP-01-0124-FEDER-015008) and PEst (FCOMP-01-0124-FEDER-022701). Miguel Areias is funded by the FCT grant SFRH/BD/69673/2010.

References

1. Areias, M., Rocha, R.: An Efficient Implementation of Linear Tabling Based on Dynamic Reordering of Alternatives. In: International Symposium on Practical Aspects of Declarative Languages. pp. 279–293. No. 5937 in LNCS, Springer-Verlag (2010)
2. Areias, M., Rocha, R.: On Combining Linear-Based Strategies for Tabled Evaluation of Logic Programs. Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue 11(4–5), 681–696 (2011)
3. Bachmair, L., Chen, T., Ramakrishnan, I.V.: Associative Commutative Discrimination Nets. In: International Joint Conference on Theory and Practice of Software Development. pp. 61–74. No. 668 in LNCS, Springer-Verlag (1993)
4. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM 43(1), 20–74 (1996)
5. Chico, P., Carro, M., Hermenegildo, M.V., Silva, C., Rocha, R.: An Improved Continuation Call-Based Implementation of Tabling. In: International Symposium on Practical Aspects of Declarative Languages. pp. 197–213. No. 4902 in LNCS, Springer-Verlag (2008)
6. Cruz, F., Rocha, R.: Retroactive Subsumption-Based Tabled Evaluation of Logic Programs. In: European Conference on Logics in Artificial Intelligence. pp. 130–142. No. 6341 in LNAI, Springer-Verlag (2010)

7. 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. pp. 243–258. No. 1140 in LNCS, Springer-Verlag (1996)
8. 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. pp. 181–196. No. 2237 in LNCS, Springer-Verlag (2001)
9. Lloyd, J.W.: Foundations of Logic Programming. Springer-Verlag (1987)
10. 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), 31–54 (1999)
11. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems 20(3), 586–634 (1998)
12. Santos Costa, V., Rocha, R., Damas, L.: The YAP Prolog System. Journal of Theory and Practice of Logic Programming 12(1 & 2), 5–34 (2012)
13. Somogyi, Z., Sagonas, K.: Tabling in Mercury: Design and Implementation. In: International Symposium on Practical Aspects of Declarative Languages. pp. 150–167. No. 3819 in LNCS, Springer-Verlag (2006)
14. Swift, T., Warren, D.S.: XSB: Extending Prolog with Tabled Logic Programming. Theory and Practice of Logic Programming 12(1 & 2), 157–187 (2012)
15. Zhou, N.F.: The Language Features and Architecture of B-Prolog. Journal of Theory and Practice of Logic Programming 12(1 & 2), 189–218 (2012)
16. Zhou, N.F., Shen, Y.D., Yuan, L.Y., You, J.H.: Implementation of a Linear Tabling Mechanism. In: Practical Aspects of Declarative Languages. pp. 109–123. No. 1753 in LNCS, Springer-Verlag (2000)

Miguel Areias received his B.Sc. and M.Sc. degrees in Computer Science from Faculty of Science of the University of Porto, in 2008 and 2010, respectively. He is currently pursuing the Ph.D. degree at the University of Porto. He is a Researcher in the Center for Research in Advanced Computing Systems (CRACS), where he has been since 2008 under the supervision of Prof. Dr. Ricardo Rocha. His research interests lie on Parallelism, Concurrency, Multithreading and Tabling mechanisms applied to Logic Programs.

Ricardo Rocha is an Assistant Professor at the Department of Computer Science, Faculty of Sciences, University of Porto, Portugal and a researcher at the CRACS & INESC-Porto LA research unit. He received his PhD degree in Computer Science from the University of Porto in 2001 and his main research topics are the Design and Implementation of Logic Programming Systems, Tabling in Logic Programming and Parallel and Distributed Computing. Another areas of interest include Inductive Logic Programming, Probabilistic Logic Programming and Deductive Databases. He is also one of the main developers of Yap Prolog system, and in particular of the execution models that support tabling and parallel evaluation. He has published more than 50 refereed papers in journals and international conferences, has supervised 11 MSc students and has leading

role in two national projects: project STAMPA, funded with 150,000 Euros, and project LEAP, funded with 115,000 Euros. Currently, he also serves the ALP Newsletter as area co-editor for the topic on Implementation.

Received: November 29, 2012; Accepted: August 12, 2013.