A Simple Scheme for Implementing Tabled Logic Programming Systems based on Dynamic Reordering of Alternatives*

Hai-Feng Guo
Department of Computer Science
SUNY Stony Brook
haifeng@cs.sunysb.edu

Gopal Gupta
Department Computer Science
University of Texas at Dallas
gupta@utdallas.edu

Abstract

Tabled logic programming (LP) systems have been applied to elegantly and quickly solving very complex problems (e.g., *model checking*). However, techniques currently employed for incorporating tabling in an existing LP system are quite complex and require considerable change to the LP system. We present a simple technique for incorporating tabling in existing LP systems based on *dynamically reordering* clauses containing *variant* calls at runtime. Our simple technique allows tabled evaluation to be performed with a *single* SLD tree and without the use of complex operations such as *freezing* of stacks and heap. It can be incorporated in an existing logic programming system with a small amount of effort. Our scheme also facilitates exploitation of parallelism from tabled LP systems. Results of incorporating our scheme in the commercial ALS Prolog system are reported.

1 Introduction

Traditional logic programming systems (e.g., Prolog) use SLD resolution [14] with the following *computation strategy* [14]: subgoals of a resolvent are tried from left to right and clauses that match a subgoal are tried in the textual order they appear in the program. It is well known that SLD resolution may lead to non-termination for certain programs, even though an answer may exist via the declarative semantics. In fact, this is true of any "static" computation strategy that is adopted. That is, given any static computation strategy, one can always produce a program that will not be able to find the answers due to non-termination even though finite solutions may exist. In case of Prolog, programs containing certain types of left-recursive clauses are examples of such programs.

To get around this problem, researchers have suggested computation strategies that are *dynamic* in nature coupled with recording solutions in a *memo table*. By a dynamic computation strategy we mean that the decision regarding which clause to use next for resolution is taken based on runtime properties, e.g., the nature and type of goals in the current resolvent. OLDT [21] is one such computation strategy. In OLDT resolution, solutions to certain subgoals are recorded in a *memo table* (heretofore referred to simply as a *table*). Such a call that has been recorded in the table is referred to as a *tabled call*. In OLDT resolution, when a tabled call is encountered, computation is started to try the alternative branches of the original call and to compute solutions, which are then recorded in the table. These solutions are called *tabled solutions* for the call. When a call to a subgoal that is identical to a previous call is encountered while computing a tabled call—such a call is called a variant call and may possibly lead to non-termination if SLD resolution is used—the OLDT resolution strategy will not expand it as SLD resolution will, rather the solutions to the variant call will only be obtained by matching it with tabled solutions. If any solutions are found in the table,

*This research is supported by NSF grants CCR 99-00320, CCR 98-20852, CDA-9729848, EIA 98-10732, EIA 9729848, and INT 9904063. Hai-Feng Guo is supported by a Post-doctoral fellowship from NSF.

they are *consumed* one by one just as a list of fact clauses by the variant call, each producing a solution for the variant call. After consuming, the computation of the variant subgoal is suspended until some new solutions appear in the table. This consumption and suspension continues, until we can detect that all the solutions for the tabled call have been generated and a *fixpoint* reached.

Tabled logic programming systems have been put to many innovative uses. A tabled logic programming system can be thought of as an engine for efficiently computing fixpoints. Efficient fixpoint computation is critical for many applications. These applications include:

1. Model checking of software systems: model checkers based on tabled LP systems such as XSB are comparable in speed to state-of-the-art model checkers but are considerably easier to program [17].

2. Efficient implementation of deductive databases: Deductive databases based on tabling can be as much as 10 times faster than those based on bottom-up evaluation and magic sets [23].

3. Program Analysis: The fixpoint engine of tabled logic programming systems provides a generic framework for quickly developing abstract interpreters [16] and program analysis systems [7].

4. Non-monotonic reasoning: tabled LP systems can support more powerful forms of negations [6], thus making logic programming more expressive.

The ability to table calls and solutions results in a more complete logic programming system. Thus tabling should be an indispensable part of any Prolog system. However, this has not happened, mainly, we believe, due to the techniques that have been traditionally used to incorporate tabling in existing LP systems. Traditionally, OLDT has been implemented by a combination of computation suspension via *stack freezing* and maintaining a forest of SLD trees (e.g., the XSB system) [29, 5, 22]. Due to this maintenance of forest of SLD trees, freezing of stacks, suspensions and resumption, implementing OLDT in this way can be quite complex. Also, the freezing of stacks results in space overheads. Several man-years have been invested in the design and development of the XSB system [5, 20, 8, 9, 11, 28] due to this complexity. This investment in effort has indeed yielded results, turning XSB into an extremely efficient tabled LP system most widely in use today.

Other techniques that are variants of OLDT and that are simpler to implement and incur less space overhead, such as SLDT, have been recently proposed and incorporated in the B-Prolog system [25]. However, in SLDT because of the execution strategy used, ensuring that all solutions to a tabled call have been found can incur considerable overhead.

In this paper, we present a novel, simple scheme for incorporating tabling in a standard logic programming system. Our scheme, which is based on *dynamic reordering of alternatives* that contain variant calls, allows one to incorporate tabling in an existing logic programming system with very little effort. Using our scheme we were able to incorporate tabling in the commercial ALS Prolog system [3] in a few weeks of work. The time efficiency of our tabled ALS (TALS) system is comparable to that of the XSB system and B-Prolog. The space efficiency of our system is comparable to that of B-Prolog and XSB with local scheduling and better than that of XSB with batch scheduling (batch scheduling is XSB's current default scheduling strategy). Unlike traditional implementations of tabling [5], our scheme works with a single SLD tree without requiring suspension of goals and freezing of stacks. Additionally, no extra overhead is incurred for non-tabled programs. Intuitively, our scheme builds the search tree as in normal Prolog execution based on SLD, however, when a variant tabled call is encountered, the branch that lead to that variant call is "moved" to the right of the tree. Essentially, branches of the search tree are reordered during execution to avoid exploring potentially non-terminating branches.

The principal advantage of our technique is that because of its simplicity it can be incorporated very easily and without sacrificing efficiency in an existing Prolog system. This can have important consequences, given that tabling is so important for many serious applications of logic programming (e.g., model checking [17]).

In our dynamic alternative reordering strategy, not only are the solutions to variant calls tabled, the alternatives leading to variant calls are also memorized in the table (these alternatives, or clauses, containing variant calls are called *looping alternatives* in the rest of the paper). A tabled call first

tries its non-looping alternatives (tabling any looping alternatives that are encountered along the way). Finally, the tabled call repeatedly tries its looping alternatives until it reaches a fixpoint. This has the same effect as shifting branches with variant calls to the right in the search tree. The simplicity of our scheme guarantees that execution is not inordinately slowed down (e.g., in the B-Prolog tabled system [25], a tabled call may have to be re-executed several times to ensure that all solutions are found), nor considerable amount of memory used (e.g., in the XSB tabled system [5] a large number of stacks/heaps may be frozen at any given time), rather, the raw speed of the Prolog engine is available to execute even those programs that contain variant calls.

An additional advantage of our technique for implementing tabling is that parallelism can be naturally exploited. In traditional tabled systems such as XSB, the ideas for parallelism have to be reworked and a new model of parallelism derived [10, 19]. In contrast, in a tabled logic programming system based on dynamic reordering, the traditional forms of parallelism found in logic programming (or-parallelism and and-parallelism) can still be exploited. Work is in progress to augment the or-parallel ALS system [3, 12] (currently being developed by us [24, 13]) with tabling [13].

A disadvantage of our approach is that certain *non-tabled* goals occurring in looping alternatives may be computed more than once. However, this recomputation can be eliminated by the use of tabling, automatic program transformation, or more sophisticated reordering techniques (see later).

2 SLD and OLDT Resolution

Prolog was initially designed to be a declarative programming language [14], that is, a logic program with a correct declarative semantics should also get the same results through procedural semantics. However, the operational semantics of standard Prolog systems that adopt SLD resolution (leftmost-first selection rule and a depth-first search rule) is not close to their declarative semantics. The completeness of SLD resolution ensures that given a query, the solutions implied by the program, if they exist, can be obtained through computation paths in the SLD tree [14]. However, standard Prolog systems with a pre-fixed computation rule may only compute a subset of these solutions due to problems with non-termination.

Example 2.1 *Consider the following program:*

```
r(X, Y) :- r(X, Z), r(Z, Y).        (1)
r(X, Y) :- p(X, Y), q(Y).           (2)
p(a, b).   p(a, d).   p(b, c).
q(b).      q(c).
:- table r/2.
:- r(a, Y).
```

Following the declarative semantics of logic programs (e.g., employing bottom-up computation), the example program 2.1 above should produce two answers $Y{=}b$ and $Y{=}c$. However, standard Prolog system will go into an infinite loop for this program. It is Prolog's computation rule that causes the inconsistency between its declarative semantics and procedural semantics. With the leftmost-first selection rule and depth-first search rule, Prolog systems are trapped in an infinite loop in the SLD-tree even though computation paths may exist to the solutions. It seems that breadth-first search strategy may solve the problem of infinite-looping, and it does help in finding the first solution. However, if the system is required to find all the solutions and terminate, breadth-first search is not enough, since the SLD tree may contain branches of infinite length.

To get around this problem, a tabled evaluation strategy called OLDT is used in tabled logic programming systems such as XSB. In the most widely available tabled Prolog systems, XSB, OLDT is implemented in the following way[1]. When a call to a tabled predicate is encountered for the first time, the current computation is suspended and a new SLD tree is built to compute the answers to this tabled call. The new tree is called a *generator*, while the old tree (which led to the tabled call) is called a *consumer* w.r.t. the new tabled call. When a call that is a variant of a previous call—and

[1]The current XSB system uses SLG resolution, which is OLDT augmented with negation.

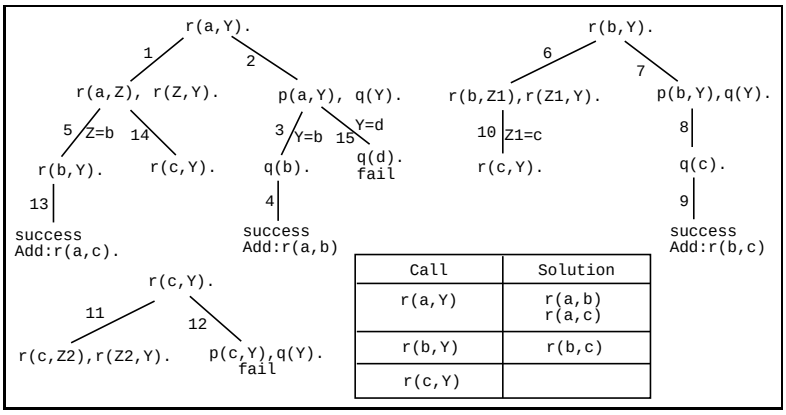

Figure 1: An OLDT Tree

that may potentially cause infinite loop under SLD—is encountered in the generator SLD tree, XSB first consumes the tabled solutions of that call (i.e., solutions that have already been computed by the previous call). If all the tabled solutions have been exhausted, the current call is suspended until some new answers are available in the table. Finally, the solutions produced by the generator SLD tree are consumed by the consumer SLD tree after its execution is resumed. In XSB, the suspension of the consumer SLD tree is realized by freezing the stacks and heap. An implementation based on suspension and freezing of stacks is quite complex to realize as well as can incur overhead in terms of time and space. Considerable effort is needed to make such a system very efficient. In this paper, we present a simple scheme for incorporating tabling in a Prolog system in a small fraction of this time. Additionally, our system is comparable in efficiency to existing systems w.r.t. time and space.

The OLDT resolution forest for example 2.1 following XSB style execution is shown in figure 1. (The figure also shows the memo-table used for recording solutions; the numbers on the edges of the tree indicate the order in which XSB will generate those edges). Compared to SLD, OLDT has several advantages (i) A tabled Prolog system avoids redundant computation by memoing the computed results. In some cases, it can reduce the time complexity of a problem from exponential to polynomial. (ii) A tabled Prolog system terminates for all queries posed to bounded term-sized programs that have a finite least fixpoint. (iii) Tabled Prolog keeps the declarative and procedural semantics of definite Prolog programs consistent.

3 Dynamic Reordering of Alternatives (DRA)

We present a simple technique for implementing tabling that is based on *dynamic reordering of looping alternatives* at runtime, where a *looping alternative* refers to a clause that matches a tabled call containing a recursive variant call. Intuitively, our scheme works by reordering the branches in SLD trees. Branches containing variant calls are moved to the right in the SLD tree for the query. In our scheme, a tabled call can be in one of three possible states: *normal state*, *looping state*, or *complete state*. The state transition graph is shown in figure 2.

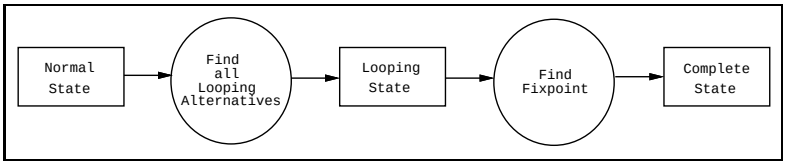

Figure 2: State Transition Graph

Consider any tabled call $\mathcal{C}$, normal state is initially entered when $\mathcal{C}$ is first encountered during the computation. This first occurrence of $\mathcal{C}$ is allowed to explore the matched clauses as in a standard Prolog system (normal state). In normal state, while exploring the matching clauses, the system tables all the solutions generated for the call $\mathcal{C}$ in this state and also checks for variants of $\mathcal{C}$. If

```
(1)  solve(true,_).
(2)  solve((A,B),master) :- solve(A,master), solve(B,master).
(3)  solve(A,master) :-
(4)       (tabled(A) ->
(5)           (state(A,normal) ->      % A in normal execution state
(6)                ( clause(A,Cl),
(7)                  (isLoopingAlt(A,Cl) ->
(8)                      addLoopAlt(A,Cl), Cl = (A:-A,B)
(9)                      solve(A, slave), solve(B, master)
(10)                 ;   Cl = A:-B, solve(B, master)
(11)                 ),
(12)                 addTableSol(A,_ )
(13)               ; setState(A,looping), solve(A,master)
(14)               )
(15)         ;  (state(A,looping) ->     % A in looping execution state
(16)               getLoopAltList(A, LoopAltList),
(17)               solve_loop_alt(LoopAltList,IsNewSolFnd),
(18)               (var(IsNewSolFnd) ->
(19)                   setState(A,complete)
(20)                 ; solve(A,master))
(21)            ; look_up_table(A)     % A in complete execution state
(22)            )
(23)         ),
(24)       ; clause(A,Cl),  % non tabled call
(25)         solve(Cl,master)
(26)      ).
(27)  solve(A,slave) :- look_up_table(A).
(28)  solve_loop_alt([], _).
(29)  solve_loop_alt([ (A:-A,B) | RestList], FndFlag) :-
(30)             solve(A, slave), solve(B, master),
(31)             addTableSol(A, Flag),
(32)             solve_loop_alt(RestList, FndFlag).
```

Figure 3: A Meta-interpreter for DRA

a variant is found, the current clause that matches the original call to $\mathcal{C}$ will be memorized, i.e., recorded in the table, as a *looping alternative*. This call will not be expanded at the moment because it can potentially lead to an infinite loop. Rather it will be solved by consuming the solutions from the table that have been computed by other alternatives. To achieve this, the alternative corresponding to this call will be reordered and placed at the end of the alternative list in the choice-point. A failure will be simulated and the alternative containing the variant will be backtracked over. After exploring all the matched clauses (some of which were possibly tabled as looping alternative), $\mathcal{C}$ goes into its looping state. From this point, tabled call $\mathcal{C}$ keeps trying its tabled looping alternatives repeatedly (by again putting the alternative at the end of the alternative list after it has been tried) until $\mathcal{C}$ is completely evaluated. If no new solution is added to $\mathcal{C}$'s tabled solution set in any one cycle of trying its tabled looping alternatives, then we can say that $\mathcal{C}$ has reached its fixpoint.

$\mathcal{C}$ enters its *complete state* after it reaches its fixpoint, i.e., after all solutions to $\mathcal{C}$ have been found. In the complete state, if the call $\mathcal{C}$ is encountered again later in the computation, the system will simply use the tabled solutions recorded in the table to solve it. In other words, $\mathcal{C}$ will be solved simply by consuming its tabled solution set one after another as if trying a list of facts.

Much research has been devoted to evaluating recursive queries in the field of deductive databases

[4]. Intuitively, the DRA scheme can be thought roughly equivalent to the following deductive query evaluation scheme for computing fixpoints of recursive programs: (i) first find all solutions to the query using only *non-recursive clauses* in a top-down fashion, (ii) use this initial solution set as a starting point and compute (semi-naively) the fixpoint using the recursive clauses in a bottom up fashion. By using the initial set obtained from top-down execution of the query using non-recursive clauses, only the answers to the query are included in the final fixpoint. Redundant evaluations are thus avoided as in Magic set evaluation. The proof of correctness of DRA is based on formally showing its equivalence to this evaluation scheme [13], and is omitted here due to lack of space.

Fig 3 shows a meta-interpreter that formally illustrates the DRA scheme. To keep the presentation and the meta-interpreter simple, we assume that all variant calls occur in left-recursive clauses and that there are no nested tabled calls dependent on each other. The first call to a tabled predicate (termed `master` call) is distinguished from subsequent calls to the variant (termed `slave` calls). We also assume that the table exists as a global data structure. The `solve/2` goal takes different actions depending on whether execution is in `master` mode or `slave` mode. The input to the interpreter is a goal `A`. The goal `clause(A,Cl)` nondeterministically finds the matching clause, `Cl`, for the goal `A`. The goal `tabled(A)` checks if `A` has been declared as tabled or not, `state(A,X)` checks to see if the `A`'s execution status is `X` (one of `normal`, `looping`, or `complete`), while the `setState(A,X)` changes the execution state of the goal `A` to `X` (one of `normal, looping, or complete`). The goal `addTableSol(A,Flag)` adds a solution for goal `A` to the table; if the solution is a new one, the `Flag` is set to a ground value, otherwise, it is left unbound. The goal `look_up_table(A)` looks up solutions for the goal `A` in the table that have been recorded so far. The goal `isLoopingAlt(A,Cl)`, checks if the clause `Cl` is a looping alternative w.r.t. goal `A`, while `addLoopAlt(A,Cl)` records `Cl` as a looping alternative of goal `A`. The meta-interpreter is pretty self-explanatory. If the goal is tabled, then if execution state is normal (lines 5-13), a matching clause for the goal is non-deterministically found. If the matching clause is a looping alternative (line 7), this fact is recorded (line 8), and the variant is executed in the slave mode, i.e., it can only be resolved against tabled solutions (line 9, first call), while the goals following the variant are executed normally in `master` mode (line 9, second call). If the matching clause is not a looping alternative, it is executed normally. In either case, if a solution is found, it is tabled (line 12). After all matching clauses have been seen, execution state is set to `looping` (line 13). If the execution state is `looping` (lines 15-22), the collected looping alternatives are retrieved, and executed (line 17, 28-32)). All variant calls are executed in the `slave mode` (line 30). If no new solution is found while executing the looping alternatives (`IsNewSolFnd` is still unbound), the state of execution is set to complete (line 19). If the state of execution is `complete` (line 21), then all solutions are to be found in the table, and thus only table look up is to be used for resolution. The DRA scheme is next illustrated with examples.

Example 3.1 *Consider resolving the following program and its evaluation using DRA:*

```
r(X, Y) :- r(X, Z), p(Z, Y).     (1)
r(X, Y) :- p(X, Y).              (2)
r(X, Y) :- r(X, Z), q(Z, Y).     (3)
p(a, b).   p(b, c).   q(c, d).
:- table r/2.
:- r(a, Y).
```

Figure 4 gives the computation tree produced by DRA for example 3.1 (note that the labels on the branch refer to the clause used for creating that branch). Both clause (1) and clause (3) need to be tabled as looping alternatives for the tabled call `r(a, Y)` (this is accomplished by operations `a_add:(1)` and `a_add:(3)` shown in Figure 4). The second alternative is a non-looping alternative that produces a solution for the call `r(a,Y)` which is recorded in the table (via the operation `s_add` shown in the Figure). The query call `r(a, Y)` is a master tabled call (since it is the first call), while all the occurrences of `r(a, Z)` are slave tabled calls (since they are calls to variant of `r(a,Y)`). When the call `r(a, Y)` enters its looping state, it keeps trying the looping alternatives repeatedly until the solution set does not change any more, i.e., until `r(a, Y)` is completely evaluated (this is accomplished by trying a looping alternative, and then moving it to the end of the alternatives list).

Note that if we added two more facts: `p(d,e)` and `q(e,f)`, then we'll have to go through the two looping alternatives one more time to produce the solutions `r(a,e)` and `r(a,f)`.

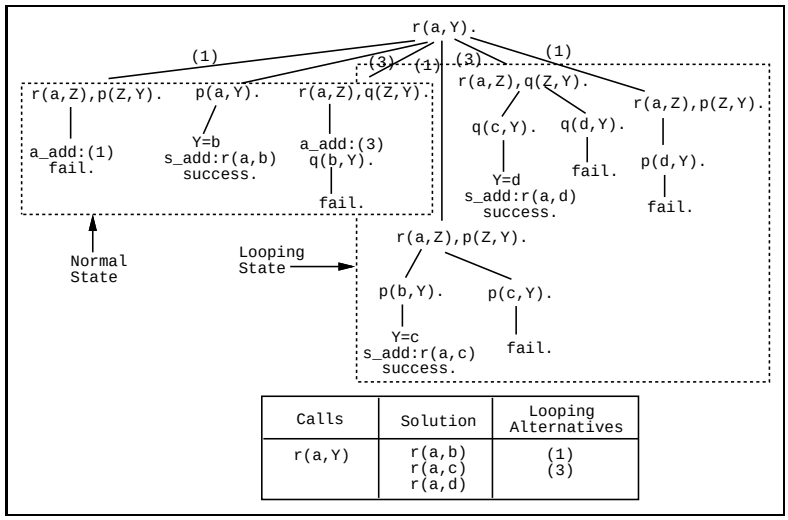

Figure 4: DRA for Example 3.1

Intuitively, given a tabled call $\mathcal{C}$, the DRA scheme first finds all the solutions for $\mathcal{C}$ using clauses not containing variant calls (non-looping alternative). Once this set of solutions is computed and tabled, it is treated as a set of facts, and used for computing rest of the solutions from the clauses containing variant calls (looping alternatives). The process stops when no new solutions can be computed via the looping alternatives, i.e., a fixpoint is reached. In this regard, the DRA scheme is similar to the way fixpoint is computed during bottom-up execution [14, 27] of logic programs, except that the base solution set on which the clauses containing variants are repeatedly applied is obtained by executing the call w.r.t. non-recursive clauses. In contrast, the base solution set in pure bottom up computation is assumed as empty in the beginning [14]. Thus, in example 3.1 above, we find the initial solution set $\mathtt{S}_0 = \{\mathtt{r(a,b)}\}$ from clause (2). Clauses (1) and (3) are then repeatedly used to generate the following sequence:

$\mathtt{S}_0 = \{\mathtt{r(a,b)}\}$
$\Delta\mathtt{S}_1 = \{\mathtt{r(a,c)}\}$ $\quad \mathtt{S}_1 = \{\mathtt{r(a,b)}, \mathtt{r(a,c)}\}$ (clause 1)
$\Delta\mathtt{S}_2 = \{\mathtt{r(a,d)}\}$ $\quad \mathtt{S}_2 = \{\mathtt{r(a,b)}, \mathtt{r(a,c)}, \mathtt{r(a,d)}\}$ (clause 3)
$\Delta\mathtt{S}_3 = \{\ \}$ $\quad \mathtt{S}_3 = \{\mathtt{r(a,b)}, \mathtt{r(a,c)}, \mathtt{r(a,d)}\}$

The final fixpoint is $\{\mathtt{r(a,b)}, \mathtt{r(a,c)}, \mathtt{r(a,d)}\}$. Note that $\Delta\mathtt{S}_i$ stands for the set difference $\mathtt{S}_i - \mathtt{S}_{i-1}$.

An important problem that needs to be addressed in any tabled system is *detecting completion.* When there are multiple tabled calls occurring simultaneously during the computation, and results produced by one tabled call may depend on another's, then knowing when the computation of a tabled call is complete (i.e., all solutions have been computed) is quite hard. Completion detection based on finding *strongly connected components* (SCC)[2] has been implemented in the TALS system (details are omitted due to lack of space and can be found elsewhere [13]). Completion detection is very similar to the procedure employed in XSB and the issues are illustrated in the next two examples.

Example 3.2 *Consider resolving the following program with DRA:*

```
r(X, Y) :- r(X, Z), r(Z, Y).      (1)
r(X, Y) :- p(X, Y), q(Y).         (2)
p(a, b).   p(a, d).   p(b, c).
q(b).      q(c).
```

[2]A directed graph is **strongly connected** if every two vertices are reachable from each other. The **strongly connected components (SCCs)** of a graph are the equivalent classes of vertices under the "are mutually reachable" relation.

```
:- table r/2.
:- r(a, Y).
```

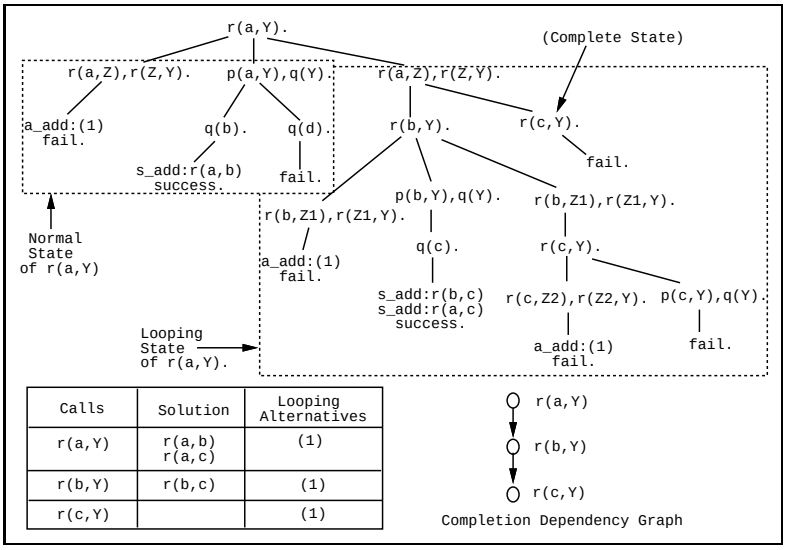

Figure 5: DRA for Example 3.2

As shown in the computation tree of Figure 5, the tabled call `r(b, Y)` is completely evaluated only if its dependent call `r(c, Y)` is completely evaluated, and `r(a, Y)` is completely evaluated only if its dependent calls, `r(b, Y)` and `r(c, Y)`, are completely evaluated. Due to the depth-first search used in TALS, `r(c, Y)` always enters its complete state ahead of `r(b, Y)`, and `r(b, Y)` ahead of `r(a, Y)`. The depth-first strategy with alternative reordering guarantees for such dependency graphs (i.e., graphs with no cycles) that dependencies can be satisfied without special processing during computation. However, these dependencies can be cyclic as in the following example.

Example 3.3 *Consider resolving the following program with DRA:*

```
r(X, Y) :- p(X, Z), r(Z, Y).      (1)
r(X, Y) :- p(X, Y).               (2)
p(a, b).   p(b, a).
:- table r/2.
:- r(a, Y).
```

Figure 6 shows the complete computation tree of example 3.3. In this example, two tabled calls, `r(a, Y)` and `r(b, Y)`, are dependent on each other, forming a SCC in the completion dependency graph. It is not clear which tabled call is completely evaluated first. A proper semantics can be given to the program only if all tabled calls in a SCC reach their complete state simultaneously. According to depth-first computation strategy, the least deep tabled call of each SCC should be the last tabled call to reach its fixpoint in its SCC. To detect completion correctly, the table is extended to record the least deep tabled call of each SCC, so that the remaining calls in the SCC can tell whether they are in the complete state by checking the state of the least deep call. The state of a tabled call can be set to "complete" only after its corresponding least deep call is in a complete state. In this example, there are two occurrences of `r(b, Y)` during the computation. In its first occurrence, `r(b, Y)` can not be set to "complete" even though it reaches a *temporary* fixpoint after exploring its looping alternative, because it depends on the tabled call `r(a, Y)`, which is not completely evaluated yet. If the call `r(b, Y)` is set to "complete" state at this point, a solution `r(b, b)` will be lost. Only after the tabled call `r(a, Y)` is completely evaluated during its looping state, can the tabled call `r(b, Y)` (within the same SCC with `r(a, Y)`) be set to complete state.

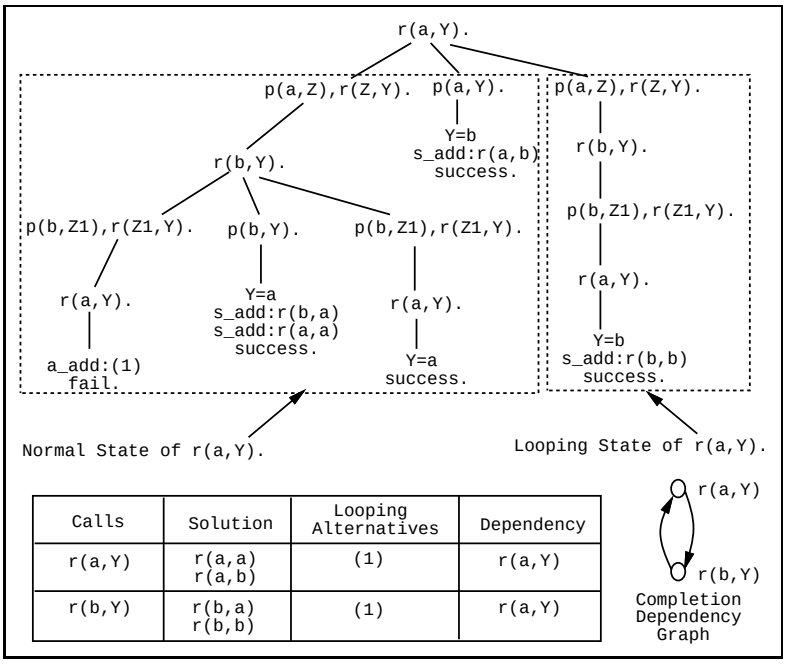

Figure 6: Example 3.3

4 Implementation

The DRA scheme can be easily implemented on top of an existing Prolog system. TALS is an implementation of DRA on top of the commercial ALS Prolog system. In the TALS system, tabled predicates are explicitly declared. Tabled solutions are consumed incrementally to mimic semi-naive evaluation [2, 4]. Memory management and execution environment can be kept the same as in a regular Prolog engine. Two main data structures, *table* and *tabled choice-point stack*, are added to the TALS engine. The table data structure is used to keep information regarding tabled calls such as the list of tabled solutions and the list of looping alternatives for each tabled call, while tabled choice-point stack is used to record the properties of tabled call, such as whether it is a master call (the very first call) or a slave call (call to the variant in a looping alternative). The master tabled call is responsible for exploring the matched clauses, manipulating execution states, and repeatedly trying the looping alternatives and solutions for the corresponding tabled call, while slave tabled calls only consume tabled solutions. The allocation and reclaiming of master and slave choicepoints is similar to regular choicepoints, except that the former have a few extra fields to manage the execution of tabled calls.

Very few changes are required to the WAM engine of a Prolog system to implement the DRA scheme (more implementation details can be found elsewhere [13]). We introduce six new WAM instructions, needed for tabled predicates: `table_try_me_else`, `table_retry_me_else`, `table_trust_me`, `table_loop`, `table_consume`, and `table_save`. We differentiate between tabled calls and non-tabled calls at compile-time, and generate appropriate type of WAM `try` instructions. For regular calls, the WAM `try_me_else`, `retry_me_else`, and `trust_me_else`, instructions are generated to manage the choicepoints, while for tabled calls, these are respectively modified to `table_try_me_else`, `table_retry_me_else`, and `table_trust_me_else` instructions. Every time `table_try_me_else` is invoked, we have to check if the call is a variant of a previous call[3]. This is easily accomplished by comparing the starting code address and the arguments of the current call to those of calls currently tabled. If the call is a variant, the address of the WAM code corresponding to this clause is recorded in the table as a looping alternative. The variant call is treated as a *slave* tabled call, which will only

[3] `table_try_me_else` is always the first WAM instruction executed for non-deterministic tabled calls; for tabled predicates with a single clause definition (deterministic), a dummy failing clause is automatically added to make that predicate non-deterministic.

consume tabled solutions if there are any in the table, and will not explore any matched clauses. The next-alternative-field of the slave choicepoint is changed to `table_consume` so that it repeatedly consumes the next available tabled solutions. If the call is a new tabled call, it will be added into the table by recording the starting code address and its arguments information. This new tabled call is treated as a *master* tabled call, which will explore the matched clauses and generate new solutions. The continuation instruction of a master tabled call is changed to a new WAM instruction `table_save`, which checks if generated solution is new. If so, the new solution is tabled, and execution continues with the sub-goal after the tabled call as in normal Prolog execution. When the last matched alternative of the master choicepoint is tried by first executing the `table_trust_me` instruction, the next-alternative-field of the master choicepoint is set to the instruction `table_loop`, so that after finishing the last matched alternative, upon backtracking, the system will enter the looping state to try the looping alternatives. After a fixpoint is reached, and all the solutions have been computed, this instruction is changed to the WAM `trust_me_fail` instruction, which de-allocates the choicepoint and simulates a failure, as in normal Prolog execution.

5 Recomputation Issues in DRA

The main idea in the TALS system is to compute the base solution set for a tabled call using clauses not containing variants, then repeatedly applying the clauses with variants (looping alternatives) on this base solution set until the fixpoint of the tabled call is reached. Due to the looping alternatives being repeatedly executed, certain *non-tabled* goals occurring in these clauses may be unnecessarily re-executed. This recomputation can affect the overall efficiency of the TALS system. Non-tabled calls may be redundantly re-executed in the following three situations:

(i) While trying a looping alternative, the whole execution environment has to be built again until a slave tabled choicepoint is created. Consider the looping alternative:

```
p(X, Y) :- q(X), p(X, Z), r(Z, Y).
...
:- table p/2.
:- p(a, X).
```

Suppose `p/2` is a tabled predicate, while `q/1` and `r/2` are not. Then each time this alternative is tried, `q(X)` has to be computed since it is not a tabled call. That is, the part between the master tabled call and slave tabled call has to be recomputed when this alternative is tried again.

(ii) False looping alternatives may occur and may require recomputation. Consider the program below:

```
p(1).
p(2).
:- table p/1.
:- p(X), p(Y).
```

After the first goal `p(X)` gets the solution `p(1)`, a variant call of `p(X)`, namely, `p(Y)`, is met. According to the DRA scheme, the explored clause is then tabled as a looping alternative. However, all the matched clauses `p(1)` and `p(2)` are determinate facts, which will not cause any looping problem. The reason we falsely think that there is a looping alternatives is because it is difficult to tell whether `p(Y)` is a descendant of `p(X)` or not. Even worse, the false looping alternatives will generate the solutions in a different order from those generated by SLD resolution. In standard Prolog, the solution sequence is "`X=1, Y=1`", "`X=1, Y=2`", "`X=2, Y=1`", and "`X=2, Y=2`", while if DRA is applied, the solution sequence should be "`X=1, Y=1`", "`X=2, Y=1`", "`X=2, Y=2`" and "`X=1, Y=2`".

This problem of false looping alternative is also present in XSB and B-Prolog.

(iii) A looping alternative may have multiple clause definitions for its non-tabled subgoals. Each time a looping alternative is re-tried, all the matching clauses of its non-tabled subgoals have to be computed. For example:

```
p(a, b).
p(X, Y) :- p(X, Z), q(Z, Y).   (1)
p(X, Y) :- t(X, Y).            (2)
t(X, Y) :- p(X, Z), s(Z, Y).   (3)
t(X, Y) :- s(X, Y).            (4)
...
:- table p/2.
:- p(a, X).
```

For the query `p(a, X)`, clause `(1)` and clause `(2)` are two looping alternatives. Consider the second looping alternative. The predicate `p(X, Y)` is reduced to the predicate `t(X, Y)`, which has two matching clauses. The first matching clause of `t(X, Y)`, clause `(3)`, leads to a variant call of `p(X, Y)`, while the second matching clause, clause `(4)`, is a determinate clause. However, each time the looping alternative, clause `(2)`, is re-tried, both matching clauses for the predicate `t(X, Y)` are tried. However, because clause (4) does not leads to any variant of the tabled call, this recomputation is a waste.

For the first case, fortunately, this recomputation can be avoided by explicitly tabling the predicate `q/1`, so that `q(X)` can consume the tabled solutions of `q` instead of recomputing them. XSB does not have this problem with recomputation, because XSB freezes the whole execution environment, including the computation state of `q(X)`, when the variant call `p(X,Z)` is reached. This freezing of the computation state of `q(X)` amounts to implicitly tabling it.

The second case can be solved by finding the scope of the master call. If we know that `p(Y)` is out of the scope of `p(X)`, we can compute `p(X)` first, then let the variant call `p(Y)` only consume the tabled solutions. However, one assumption is that the tabled call `p(X)` has a finite fixpoint and thus can be completely evaluated.

The final case can be handled in several ways. One option is to table the specific computation paths leading to the variants of a previous tabled call instead of the whole looping alternative. However, tabling the computation paths will incur substantial overhead. Second option is to table the non-tabled predicates, such as `t(X, Y)`, so that the determinate branches of `t(X, Y)` will not be re-tried. A third option is to unfold the call to `t(X,Y)` in the clause (2) of predicate `p` so that the intermediate predicate `t(X,Y)` is eliminated.

Thus, all cases where non-tabled goals may be redundantly executed can be eliminated. Note that tabling of goals `q(X)` in case (i) and of goal `t(X,Y)` in case (iii) can be done automatically. The unfolding in case (iii) can also be done automatically.

6 Related Work

The most mature implementation of tabling is the XSB [29, 22] system from SUNY Stony Brook. As discussed earlier, the XSB system implements OLDT by developing a forest of SLD trees, suspension of execution via freezing of corresponding stacks/heap, and resumption of execution via their unfreezing. Recently, improvements of XSB, called CAT [9] and CHAT [8], that reduce the amount of storage locked up by freezing, have been proposed. Of these, the CHAT system seems to achieve a good balance between time and space overhead since it only freezes the heap, the state of the other stacks is captured and saved in a special memory area (called CHAT area).

Because of considerable investment of effort in design and optimization of the XSB system [5, 20, 8, 9, 28], XSB has turned out to be an extremely efficient system. The modified WAMs that have been designed [28, 20], the research done in scheduling strategies [11] for reducing the number of suspensions and reducing space usage [9, 8] are crucial to the efficiency of the XSB system. Ease of

implementation and space efficiency are the main advantages of DRA. The scheme based on DRA is quite simple to implement on an existing WAM engine, and produces performance that is comparable to XSB.

Recently, another implementation of a tabled Prolog system based on SLDT has been reported [25]. This implementation has been done on top of the existing Prolog system called B-Prolog. The main idea behind SLDT is as follows: when a variant is recursively reached from a tabled call, the active choice-point of the original call is transferred to the call to the variant (the variant *steals* the choice-point of the original call, using the terminology in [25]). Suspension is thus avoided (in XSB, the variant call will be suspended and the original call will produce solutions via backtracking) and the computation pattern is closer to SLD. However, because the variant call avoids trying the same alternatives as the previous call, the computation may be incomplete. Thus, repeated *recomputation* [25] of *tabled calls* is required to make up for the solutions lost and to make sure that the fixpoint is reached. SLDT does not propose a complete theory regarding when a tabled call is completely evaluated, rather it relies on blindly recomputing the tabled calls to ensure completeness. Additionally, if there are multiple clauses containing recursive variant calls, the variant calls may be encountered several times in one computation path. Since each variant call executes from the backtracking point of a former variant call, a number of solutions may be lost. These lost solutions have to be found by recomputation. This recomputing may have to be performed several times to ensure that a fixpoint is reached.

Observe that the DRA (used in TALS) is not an improvement of SLDT (used in B-Prolog) rather a completely new way of implementing a tabled LP system (DRA and SLDT were both conceived independently). The techniques used for evaluating tabled calls and for completion detection in the TALS system are quite different (even though implementations of both DRA and SLDT seem to be manipulating choicepoints). B-Prolog is efficient only for certain types of very restricted programs (referred to as *directly recursive* in [25]; i.e., programs with a single recursive rule with a single variant call). For programs with multiple recursive rules or with multiple variant calls, the B-Prolog system can be quite overhead-prone. Also, on the surface it may appear that both B-Prolog and DRA recompute goals, however, note that while B-Prolog *recomputes tabled and non-tabled* goals resulting in considerable overhead, DRA *only recomputes non-tabled* goals. Recomputation of non-tabled goals in DRA can be avoided (see section 5), while recomputation of tabled goals in B-Prolog is inevitable.

7 Performance Results

The TALS system has been implemented on top of the WAM engine of the commercial ALS Prolog system. It took us less than two man-months to implement the dynamic reordering of alternatives (DRA) scheme (with semi-naive evaluation) along with full support for complex terms on top of commercial ALS Prolog system (tabled negation is not yet supported, work is under way). Our performance data indicates that in terms of time and space efficiency, our scheme is comparable to XSB and B-Prolog. The main advantage of the DRA scheme is that it can be incorporated relatively easily in existing Prolog systems. Note that the most recent releases of XSB (version 2.3) and B-Prolog (version 5.0) were used for performance comparison. All systems were run on a machine with 700MHz Pentium processor and 256MB of main memory. XSB and B-Prolog are the only two publicly available tabled LP systems that we are aware of. Note that comparing systems is a tricky issue since all three systems employ a different underlying Prolog engine. Table 1 shows the performance of the three systems on regular Prolog programs (i.e., no predicates are tabled) and gives some idea regarding the relative speed of the engines employed by the 3 systems (arithmetic on the ALS system is slow, which is the primary reason for its poor performance on the 10-Queens and Knight benchmarks compared to other systems). Note that `Sg` is the "cousin of the same generation" program, `10-Queen` is the instance of N-Queen problem, `Knight` is the Knight's tour, `Color` is the map-coloring problem, and `Hamilton` is the problem of finding Hamiltonian cycles in a graph. Note that all figures for all the systems on all of the benchmarks are for all solution queries.

Table 2 compares the time efficiency among XSB, B-Prolog, and TALS system. These benchmarks

Benchmarks	10-Queen	Sg	Knight	Color	Hamilton
XSB	0.441	0.301	2.63	0.08	1.18
B-Prolog	0.666	0.083	3.15	0.233	2.667
TALS	2.46	0.19	11.26	0.38	1.48

Table 1: Running Time (Seconds) on Non-tabled Programs

Benchmark	cs_o	cs_r	disj	gabriel	kalah	peep	pg	read	sg
TALS	0.16	0.37	0.26	0.72	0.42	0.52	0.29	5.94	0.04
XSB-b	0.081	0.16	0.05	0.06	0.05	0.18	0.05	0.23	0.06
XSB-l	0.071	0.13	0.041	0.05	0.04	0.131	0.041	0.18	0.05
B-Prolog	0.416	0.917	0.233	0.366	0.284	1.417	0.250	0.883	0.084

Table 2: Running Time (Seconds) on Tabled Programs

are taken from the CHAT suite of benchmarks distributed with XSB and B-Prolog.[4] Most of these benchmarks table multiple predicates many of whom use structures. For XSB, timings for both batch scheduling (XSB-b) and local scheduling (XSB-l) are reported (Note that batch scheduling is currently the default scheduling strategy in XSB).

In general, the time performance of TALS on most of the CHAT benchmarks is worse than that of XSB, however, it is not clear how much of it is due to the differences in base engine speed, and how much is due to TALS' recomputation of non-tabled goals leading up to looping alternatives (the fix for this described in section 5 could not be used, as the CHAT benchmarks are automatically generated from some preprocessor and are unreadable by humans). However, except for `read` the performance is comparable, (i.e., it is not an order of magnitude worse). With respect to B-Prolog the time-performance is mixed. For programs with multiple looping alternatives TALS performs better than B-Prolog.

Benchmark	cs_o	cs_r	disj	gabriel	kalah	peep	pg	read	sg
TALS	8360	8438	12193	17062	23520	6800	20084	20426	2226
XSB-b	11040	13820	10012	30356	43628	1148296	436012	1600948	3096
XSB-l	6992	8584	6876	23156	9564	19448	16324	125342	3540
B-Prolog	21040	38592	16484	37596	61288	96884	64232	72916	1664

Table 3: Total Space Usage in Bytes (Excluding Table Space)

Tables 3, 4 and 5 compare the space used by TALS, XSB (both batch and local scheduling), and B-Prolog systems. Table 3 shows the total space used by the system. This space includes total stack and heap space used as well as *space overhead* to support tabling (but excluding space used for maintaining the table). The *space overhead* to support tabling in case of TALS includes the extra space needed to record looping alternatives and extra fields used in master and slave choicepoints. In case of both XSB-l and XSB-b, the figure includes the CHAT space used. For B-Prolog it is difficult to separate this overhead from the actual heap + stack usage. *Space overhead* incurred is separately reported in Table 4. As can be noticed from Table 3, the space performance of TALS is significantly better than that of XSB-b (for some benchmarks, e.g., peep, pg and read, it is orders of magnitude better). It is also better than the space performance of B-Prolog (perhaps due to the extra space used during recomputation in B-Prolog) and is comparable in performance to XSB-l.

[4]Note that benchmarks used in Table 1 will not benefit much from tabling, except for sg, so a different set of benchmarks is used; most of the benchmarks used in Table 2 cannot be executed under normal Prolog.

Benchmark	cs_o	cs_r	disj	gabriel	kalah	peep	pg	read	sg
TALS	672	750	213	190	376	976	420	2666	342
XSB-b	2544	4016	2568	16172	16784	1132596	363872	1356672	0
XSB-l	696	1392	1632	10848	1612	7732	7768	63720	0
B-Prolog	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Table 4: Space Overhead for Tabling in Bytes

Benchmark	cs_o	cs_r	disj	gabriel	kalah	peep	pg	read	sg
TALS	21056	21400	6488	7244	13496	17256	3852	15404	25128
XSB-b	26572	27072	22768	199948	35784	22688	15876	48032	47568
XSB-l	25356	25858	21592	19076	34160	21920	15108	45944	42448
B-Prolog	20308	20396	20104	16492	26884	15260	13860	38388	69740

Table 5: Table Space Usage in Bytes

For completeness sake, we also report the actual space used in maintaining the actual table for each of the 4 cases in Table 5.

Note that XSB with local scheduling (XSB-l) has much better performance than XSB with batch scheduling (XSB-b). XSB-l *evaluates all solutions to a tabled predicate (i.e., the generator) before returning control to the consumer.* While it considerably improves the space performance compared to XSB-b, XSB with local scheduling is meant for computing fixpoints and database type all-solutions queries. For Prolog like computation, where a user may only be interested in a single solution, XSB-l will compute the entire set regardless. Thus, in the case of the benchmarks above, the time to compute a single solution will be considerably less for XSB-b, TALS and B-Prolog, but for XSB-l it will be the same time as for computing all-solutions. Perhaps for this reason, we speculate that batch scheduling (and not local scheduling) is XSB's default scheduling strategy.

8 Incorporating Parallelism

Because of the simplicity of the DRA scheme used in the TALS system, parallelism can be easily incorporated. In fact, the parallel models of execution that have been developed for ordinary Prolog can be used largely unchanged. Two forms of parallelism have been traditionally identified in logic programming: or-parallelism and and-parallelism. Or-parallelism arises when multiple matching clauses of a goal are tried in parallel. And-parallelism arises when conjunctive goals in the current resolvent are executed in parallel. Both forms of parallelism can be easily incorporated in the TALS system, though, here we only briefly discuss incorporating or-parallelism.

Or-parallelism arises when different alternatives of a call are tried in parallel. Essentially, the multiple branches of the search tree are pursued in parallel. In the or-parallel TALS system, the multiple branches running in or-parallel will have to share the table. In a shared memory multiprocessor implementation, this table will have to be recorded in a shared memory space. The rest of the machinery for or-parallelism can be the same as in an or-parallel Prolog system [12]. Note that in an or-parallel system the looping alternatives can be tried in parallel. Or-parallelism can thus help in reaching the fix-point faster. An or-parallel implementation of TALS is currently in progress.

It should be noted that parallelism is not easy to incorporate in an implementation based on forest of SLD trees. In [10], a new form of parallelism, called *table parallelism* is identified that corresponds to exploring the multiple SLD trees that arise in OLDT simultaneously. Traditional or-parallelism can be exploited within each of these SLD-trees. Due to the presence of multiple SLD trees, the parallel execution model for exploiting table-parallelism and or-parallelism can be quite complex and hard to implement [19]. In DRA, in contrast, table-parallelism just shows up as

ordinary or-parallelism, and, thus, we believe, is easier to realize.

9 Conclusion and Future Work

The advantages of DRA can be listed as follows: (i) It can be easily implemented on top of an existing Prolog system without modifying the kernel of WAM engine in any major way; (ii) It works with a single SLD tree without suspension of goals and freezing of stacks resulting in less space usage; (iii) Unlike SLDT, it avoids blindly recomputing subgoals (to ensure completion) by remembering looping alternatives; (iv) Unlike XSB with local scheduling it produces solutions for tabled goals incrementally while maintaining good space and time performance (v) Parallelism can be easily incorporated in the DRA model.

Our alternative reordering strategy can be thought of as a dual [15] of the Andorra-principle [26]. In the Andorra model of execution, goals in a clause are reordered leading to a considerable reduction in search space and better termination behavior. The reordering of subgoals is done based on runtime properties. Likewise, our tabling scheme based on reordering alternatives (which correspond to clauses) also reduces the size of the computation (since solutions for tabled call once computed are remembered) and results in better termination behavior.

Our scheme is quite simple to implement. We were able to implement it on top of an existing Prolog engine (ALS Prolog) in a few weeks of work. Performance evaluation of our implementation shows that it is comparable in performance to well-engineered tabled systems such as XSB, yet it is considerably easier to implement. Work is in progress to add support for tabled negation and or-parallelism, so that large and complex applications (e.g., model-checking) can be tried.

Acknowledgments

None of this work would have been possible without all the research that already exists on tabled LP system, in particular, on the XSB system. We are grateful to Profs. David Warren, I.V. Ramakrishnan, and C. R. Ramakrishnan of SUNY Stony Brook for discussions, and to Ken Bowen of ALS, Inc., for providing us with the source code of the ALS Prolog system, and to Charles Houpt for explaining to us its intricate details. Thanks are also due to Kostis Sagonas of Uppsala University, Bart Demoen of KU Leuven, and Neng-Fa Zhou of CUNY Graduate Center for discussion and help with benchmarks and performance evaluation. Thanks also to Enrico Pontelli and Karen Villaverde of New Mexico State University for their help in understanding the source code of ALS.

References

[1] Warren Abstract Machine: A Tutorial. MIT Press, 1991.

[2] I. Balbin, K. Ramamohanarao. A Generalization of the Differential Approach to Recursive Query Evaluation. J. of Logic Prog 4(3): 259-262 (1987)

[3] K. Bowen, C. Houpt, et al. ALS Prolog System. www.als.com.

[4] F. Bancilhon, R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proc. ACM SIGMOD*, 1986, pp. 16-50.

[5] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM, 43(1):20-74, January 1996.

[6] W. Chen, T. Swift, and D. Warren. Efficient top-down computation of queries under the well-founded semantics. J. Logic Programming, 24(3):161-199, September 1995.

[7] M. Codish, B. Demoen, and K. Sagonas. Semantics-based program analysis for logic-based languages using XSB. International J. of Software Tools for Tech. Transfer, 2(1):29-45, Nov. 1998. Springer Verlag.

[8] B. Demoen, K. F. Sagonas. Heap Garbage Collection in XSB: Practice and Experience. Practical Aspects of Declarative Languages 2000. Springer Verlag LNCS 1753. pp. 93-108.

[9] B. Demoen and K. Sagonas. CHAT: the Copy-Hybrid Approach to Tabling. Practical Aspects of Declarative Languages: LNCS 1551, pages 106-121, Texas, Jan. 1999. Springer Verlag.

[10] J. Freire, R. Hu, T. Swift, D. Warren. Exploiting Parallelism in Tabled Evaluations. PLILP 1995: 115-132. LNCS 982.

[11] J. Freire, T. Swift, D. S. Warren. Beyond Depth-First Strategies: Improving Tabled Logic Programs through Alternative Scheduling. J. of Functional and Logic Prog. 1998(3). MIT Press.

[12] G. Gupta, E. Pontelli. Stack-splitting: A Simple Technique for Implementing Or-parallelism in Logic Programming Systems. In *Proc. Int'l Conf. on Logic Programming*, 1999.

[13] Hai-Feng Guo. *High Performance Logic Programming.* Ph.D. thesis. Oct. 2000.

[14] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[15] E. Pontelli, G. Gupta. On the Duality between Or-parallelism and And-parallelism. In *European Conference on Parallel Processing '95*, Springer LNCS 966, pp. 43-54.

[16] C. R. Ramakrishnan, S. Dawson, and D. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems: A Case Study. In *Proc. ACM PLDI.* 1996.

[17] Efficient Model Checking using Tabled Resolution. Y.S. Ramakrishnan, et al. In *Proceedings of Computer Aided Verification (CAV'97).* 1997.

[18] P. Rao, I. V. Ramakrishnan, et al. Efficient table access mechanisms for logic programs. Journal of Logic Programming, 38(1):31-54, Jan. 1999.

[19] Ricardo Rocha, Fernando M. A. Silva, Vtor Santos Costa. Or-Parallelism within Tabling. PADL 1999: 137-151. Springer LNCS 1551.

[20] K. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. ACM TOPLAS, 20(3):586 - 635, May 1998.

[21] Hisao Tamaki, T. Sato. OLD Resolution with Tabulation. In *ICLP* 1986. pp. 84-98.

[22] T. Swift and David S. Warren. An Abstract Machine for SLG Resolution: Definite Programs. *SLP* 1994: 633-652.

[23] K. Sagonas, T. Swift, and D. Warren. XSB as an efficient deductive database engine. In Proc. of SIGMOD 1994 Conference. ACM, 1994.

[24] K. Villaverde, H-F. Guo, E. Pontelli, G. Gupta. Incremental Stack Splitting and Scheduling in the Or-parallel ALS System. Tech Report. UT Dallas. Dec. 2000.

[25] Neng-Fa Zhou, et al. Implementation of a Linear Tabling Mechanism. *PADL* 2000: 109-123. Springer LNCS 1753.

[26] V. Santos Costa et al. R. Yang. Andorra-I: A Parallel Prolog system that transparently exploits both And- and Or-Parallelism. In *Proceedings of ACM PPoPP*, Apr. '91, pp. 83-93.

[27] J. D. Ullman. Bottom-Up Beats Top-Down for Datalog. ACM Principles of Database System. 1989. pp. 140-149.

[28] D. S. Warren. The XWAM: A machine that integrates Prolog and deductive database query evaluation. TR 89/25, SUNY at Stony Brook, 1989.

[29] XSB Inc. www.sourceforge.net and www.xsb.com.