Implementation of a Linear Tabling Mechanism

Neng-Fa Zhou
Department of Computer and Information Science,
Brooklyn College and Graduate Center,
The City University of New York
New York, NY 11210-2889, USA
zhou@sci.brooklyn.cuny.edu

Yi-Dong Shen
Department of Computer Science,
Chongqing University,
Chongqing, 400044, P.R.China
ydshen@cqu.edu.cn

Li-Yan Yuan, and Jia-Huai You
Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2H1
{yuan,you}@cs.ualberta.ca

Abstract

Delaying-based tabling mechanisms, such as the one adopted in XSB, are non-linear in the sense that the computation state of delayed calls has to be preserved. In this paper, we present the implementation of a linear tabling mechanism. The key idea is to let a call execute from the backtracking point of a former variant call if such a call exists. The linear tabling mechanism has the following advantages over non-linear ones: (1) it is relatively easy to implement; (2) it imposes no overhead on standard Prolog programs; and (3) the cut operator works for a certain class of useful tabled programs and thus it is possible to use the cut operator to express negation-as-failure and conditionals in those programs. The weakness of the linear mechanism is the necessity of *re-computation* for computing fixpoints. However, we have found that re-computation can be avoided for a certain portion of calls of directly-recursive tabled predicates. We have implemented the linear tabling mechanism in B-Prolog. Experimental comparison shows that B-Prolog is close in speed to XSB when re-computation can be avoided. Concerning space efficiency, B-Prolog is much better than XSB for some programs.

1 Introduction

Tabling [12, 14] in Prolog is a technique that can get rid of infinite loops for bounded-term-size programs and possible redundant computations in the execution of Prolog programs. The main idea of tabling is to memorize the answers to some calls and use the answers to resolve subsequent variant calls. Tabling has been found useful in many applications including program analysis, parsing, deductive databases, theorem proving,

model checking, and problem solving [14]. Although tabling can be added on top of Prolog [5], doing so is a burden on the programmers and can hardly achieve satisfactory performance. For this reason, it is mandatory that tabling be supported at the abstract machine level.

Currently, XSB is the only well-known Prolog system that supports tabling. The SLG [2] resolution adopted in XSB relies on the suspend/resume mechanism to do tabling. When a call (consumer), which is a variant of a former tabled call (producer), has used up the results in the table, it will be suspended. After the producer adds answers to the table, the execution of the consumer will be resumed. In contrast to the *linear* SLD resolution [7] where a new goal is always generated by extending the latest goal, SLG resolution is *non-linear*. The non-linearity of the suspend/resume mechanism considerably complicates the implementation and the handling of the cut operator. In SLG-WAM [9], the abstract machine adopted by XSB, the state of a consumer is preserved by freezing the stacks, i.e., by not allowing backtracking to reclaim the space on the stacks as is done in the WAM [13]. CHAT [4] preserves the state by copying part of it to a separate area and copying it back when the execution of the consumer needs to be resumed. In XSB, tabled calls are not allowed to occur in the scope of a cut.

Shen et al [10] proposed a strictly linear tabulated resolution, called SLDT in this paper, for Prolog. The main idea is as follows: Each tabled call can be a producer and a consumer as well. When there are answers available in the table for the call, the call consumes the answers; otherwise, it, like a usual Prolog call, produces answers by using clauses until a call that is a variant of some former tabled call occurs. In this case, the later call steals the choice point from the former call and turns to produce answers by using the remaining clauses of the former call. After a call produces an answer, it also consumes one. Answers in a table are used in a *first-generated-first-used* fashion. Backtracking is strictly chronological. The later call will be re-executed after all the available answers and clauses have been exhausted. Re-execution will stop when no new answer can be produced, i.e., when the fixpoint is reached.

To implement SLDT, we have extended the ATOAM [15], the abstract machine of B-Prolog. The extension of the abstract machine is straightforward thanks to the linearity of SLDT. Since no modification of the existing instructions and data areas is required, programs that do not use tabling are not affected. The implementation will be described in Section 3. While re-computation is necessary in general, we show in Section 4 that it can be avoided for a class of tabled calls. Most calls of predicates in deductive databases, such as *transitive closure* and *same generation*, belong to this class.

We have implemented the linear-tabling mechanism in B-Prolog. For the CHAT benchmark suite [4] for which re-computation is necessary, B-Prolog is 1/3 as fast as XSB. Nevertheless, for the well known programs in deductive database, B-Prolog is as fast as XSB. The experimental results will be presented in Section 5.

The readers are assumed to be familiar with logic programming and the WAM.

2 An Introduction to SLDT

In this section, we give a brief introduction to SLDT. The reader is referred to [10] for a formal description and a formal proof of the soundness and completeness of SLDT, and to [7] for definitions of SLD and related concepts.

Predicates in a tabled Prolog program are divided into *tabled* and *non-tabled* ones. Tabled predicates are explicitly declared by declarations in the following form:

`:-table` p_1/n_1, ..., p_k/n_k

where each p_i(i=1,...,k) is a predicate symbol and n_i is an integer that denotes the arity of p_i. A call of a tabled predicate is called a *tabled call*. Tabled calls are resolved by using SLDT, and non-tabled calls are resolved by using SLD. A tabled call that occurs first in an SLD tree is called a *pioneer*, and all subsequent calls that are variants of a pioneer are called *followers* of the pioneer. There is a table associated with every pioneer and its followers. Initially, the table is empty.

The SLDT resolution takes a tabled Prolog program and a goal, and constructs an SLD tree in the same left-to-right and depth-first fashion as the SLD resolution except when the selected call is a variant of some former call. In this case, we first use the answers in the table to resolve the call. After we exhaust all the answers, we resolve the call by using the remaining clauses of the latest former call. We say that the current call *steals* the choice point from the latest former call.

Backtracking is done similarly as in Prolog. When we backtrack to a tabled call, we use an alternative answer or a clause to resolve the call. After we exhaust all the answers and clauses, however, we cannot simply fail it since doing so we may risk losing answers. Instead, we decide whether it is necessary to re-execute the call starting from the first clause of the predicate. Re-execution will be repeated until no new answers can be generated, *i.e.*, when the fixpoint is reached.

In the following, we illustrate the behavior of SLDT using three examples.

Example 1

Consider resolving the query `?-reach(a,Y0)` against the following tabled program:

```
:-table reach/2.
reach(X,Y):-reach(X,Z),edge(Z,Y).   (C1)
reach(X,X).                         (C2)
reach(X,d).                         (C3)

edge(a,b).                          (C4)
edge(d,e).                          (C5)
```

We first apply the clause `C1` to the call `reach(a,Y0)` and obtain a new goal `N1: reach(a,Z1),edge(Z1,Y0)` where the subscripts are added to indicate the effects of

variable renaming (see Figure 1). As the call `reach(a,Z1)` is a follower of `reach(a,Y0)`, we choose `C2`, the backtracking point of `reach(a,Y0)`, and apply it to `reach(a,Z1)`, which results in a new child goal `N2:edge(a,Y0)`. As `reach(a,a)` is an answer to the call `reach(a,Z1)`, it is memorized in the table for `reach(a,Y0)`. We then resolve the call `edge(a,Y0)` by using the clause `C4`, which leads to an empty goal. So the answer `reach(a,b)` is added into the table for `reach(a,Y0)`. After these steps, we finish the leftmost branch of the tree as shown in Figure 1.

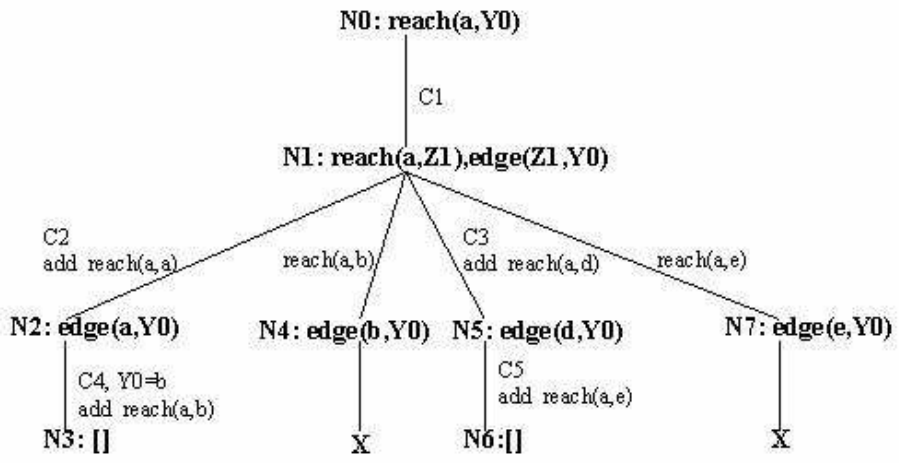

Figure 1: The SLD tree for the example.

Now consider backtracking. We first backtrack to the call `reach(a,Z1)` at node `N1`. This call has consumed `reach(a,a)` in the table. So, we use the next answer `reach(a,b)` to resolve it, which derive the goal to `N4:edge(b,Y0)`. Obviously, this goal will fail. So, we backtrack to the call `reach(a,Z1)` again. This time, as all answers in the table have been used, we use `C3` to resolve the call and obtain a new answer `reach(a,d)`, which is added to the table. After this step, the goal becomes `N5:edge(d,Y0)`. By using `C5` to resolve the call, we obtain another new answer `reach(a,e)`, which is added into the table. The goal now becomes empty, and the second answer `reach(a,b)` is returned to the top-level goal.

When more answers are required, we backtrack again to the call `reach(a,Z1)` at node `N1`. At this time, there is only one answer, `reach(a,e)`, remaining for the call. By using the answer to resolve the call, we obtain a goal `N7:edge(e,Y0)` which will fail immediately. By now, `reach(a,Z1)` has consumed all the answers and executed all the clauses. We re-execute the call but fail to produce any new answers. So, we fail it and backtrack to the pioneer `reach(a,Y0)`, which will consume the two remaining answers: `reach(a,d)` and `reach(a,e)`.

Example 2

In the previous example, re-computation produces no new answers. This example illustrates the necessity of re-computation. Consider the query `?-p(X0,Y0)` against the following program [10]:

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

q(X,Y):-p(X,Z),t(Z,Y).
q(a,b).

t(b,c).
```

There are two answers, namely, `p(a,b)` and `p(a,c)`, to the query. Without re-computation, the second answer `p(a,c)` would be lost.

Example 3

Before a call steals the choice point (call it C1) from a former variant call, the former call might have created some other choice points (call them C2) which locate to the left of C1 in the SLD tree[1]. If this is the case, the order of solutions will differ from that found by SLD resolution since C1 will be explored before C2. Consider the query `?-p(X),p(Y)` against the following program:

```
:-table p/1.
p(V):-t(V).
p(c).

t(a).
t(b).
```

First, we use the clause `p(V):-t(V)` to rewrite the subgoal `p(X)` to `t(X)`, and the fact `t(a)` to resolve the subgoal and bind `X` to `a`. Then, we turn to execute `p(Y)`. Since `p(Y)` is a variant of `p(X)`, the choice point of `p(X)` is stolen by `p(Y)`. We use the first answer in the table, i.e., `p(a)`, to resolve `p(Y)`. At this point, we get the first answer `(X=a,Y=a)`. To obtain the next answer, we backtrack to `p(Y)`. Since there is no answer remaining, we use the clause `p(c)`, which leads to the second solution `(X=a,Y=c)`. Note that if SLD resolution is used, the second answer obtained will be `(X=a,Y=b)`.

The order issue would not happen if we only allowed a call to steal a choice point from one of its ancestors. To do so, however, we have to check the ancestor/descendant relationship between two variant calls, which is expensive. It is acceptable in practice

[1] To say it more precisely, the corresponding branches of C2 locate to the left of the corresponding branches of C1 in the SLD tree.

to avoid this test because fixpoints are usually required for tabled predicates and thus the order of answers is not important.

3 Extending the ATOAM for Tabling

In the ATOAM [15], unlike in the WAM, arguments are passed through stack frames and only one frame is used for each predicate call. Frames for different types of predicates have different structures.

To implement SLDT, we introduce a new data area, called *table area*, for memorizing tabled calls and their answers, a new frame structure for tabled calls, and several new instructions for encoding the basic operations on tabled calls and the table area. We illustrate the instructions by examples. The reader is referred to the Appendix for their complete definitions.

3.1 The table area

For each pioneer and its followers, there is an entry in the *subgoal table* that contains the following four fields:

`Call:`	the predicate symbol and the arguments of the call.
`AR:`	the pointer to the frame of the latest variant call.
`Answers:`	list of available answers.
`Revised:`	whether or not new answers have been added.

The subgoal table is a hashing table that uses `Call` as the key. The `AR` field points to the frame of the latest variant call. It may take either of the following two values if it is not a pointer to a frame:

`NULL:`	the corresponding frame has been cut off by a cut.
`COMPLETE:`	the frame has been discarded after completion.

For a follower call, if the `AR` field of its table entry is `NULL`, then the call will have no choice point to steal from and the execution will start from the beginning of the predicate. If the `AR` field is `COMPLETE`, then all the answers of the call must have been produced in the table and thus no clause in the predicate need be executed.

The `Answers` field, which is called the *answer table* for the variant calls, stores the list of answers that are currently available for the calls. The answer table is also a hashing table, but the order of answers is preserved. The `Revised` is used to check whether or not re-computation needs to be continued. It is set to be `false` whenever the tabled predicate is executed or re-executed, and set to be `true` whenever an answer is added into the answer table. After the execution of the tabled predicate, `Revised` is checked. If it is `true`, then the predicate needs to be re-executed; otherwise, not.

3.2 Frames for tabled calls

The frame for a tabled call contains the following three slots in addition to the arguments, a copy of the arguments and the information stored in a choice point frame[2]:

`Table`	pointer to the table entry for the call.
`CurrA`	pointer to the answer that was last consumed.
`Pioneer`	Pointer to the frame of the pioneer.

If the tabled call is a pioneer, then an entry is added into the table , the `Table` slot of its frame is made to point to the entry, and the `Pioneer` slot is made to point to itself. If the call is a follower for which there is already an entry in the subgoal table, then the `Table` slot is made to point to the entry and the `Pioneer` slot is made to point to the frame of the pioneer. The first answer in an answer table is a dummy answer and `CurrA` is initialized to be a pointer to this dummy answer.

3.3 New instructions

There are four newly introduced instructions for tabled programs without cuts. The following example illustrates their meaning and how they are used.

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

The generated code is as follows:

```
    table_allocate 2,13,p/2,L2
L1: table_use_answer
    fork L2
    para_value y1
    para_value y2
    call q/2
    table_add_use_answer
L2: table_use_answer
    table_check_completion L1
```

where the instructions starting with `table_` are new. A new clause, called *completion-checking clause*, is added into the predicate. The two instructions at `L2` encode this clause.

[2]A choice point frame has the following slots: AR (parent frame), CP (continuation program point), TOP (top of the control stack), B (parent choice point), CPF (backtracking point), H (top of the heap), and T (top of the trail stack).

table_allocate 2, 13, p/2, L2

The instruction `table_allocate` is the first instruction in a tabled predicate. The operands are as follows: `2` is the arity, `13` is the size of the frame, `p/2` is the predicate symbol, and `L2` is the address to go to after all the clauses have been tried. For each call to the predicate, this instruction copies the arguments and allocates a frame for the call. Besides bookkeeping operations needed for backtracking, this instruction also does the following:

- If the call is a pioneer, then create an entry in the table, let the `Table` slot of the frame point to the table entry, and let the `AR` field in the table entry point to the frame. The backtracking point `CPF` of the frame is set to point to the next instruction.
- If the call is a follower for which there is already an entry in the subgoal table, then let the `Table` slot of the frame point to the table entry and take the following different actions according to the `AR` field in the table entry.
  - If the `AR` field is `NULL`, meaning that its frame has been discarded by a cut operator (see below), then treat the call as a pioneer, letting the `AR` field point to the current frame.
  - If the `AR` field is `COMPLETE`, meaning that all the answers have been produced for the call, then jump to `L2` and let the call consume the answers.
  - If the `AR` field points to a frame which must be the frame of the latest variant call, then we execute from the backtracking point stored in the frame and reset the backtracking point to be `L2`. This operation is what we call *stealing a choice point*.

table_use_answer

The `table_use_answer` instruction tries to use the next answer. If there are answers available, it unifies the original arguments of the call with the next answer and returns control to the continuation program point; otherwise, it does nothing before turning to execute the clause following the instruction.

The `fork L2` instruction resets the backtracking point `CPF` to be `L2`. So, `q/2` will not be executed on the next backtracking. The next three instructions following `fork L2` pass the arguments to the callee and start the execution of `q/2`.

table_add_use_answer

The `table_add_use_answer` instruction adds the current call, which becomes an answer, into the answer table if the call is not yet there and tries to use the next answer. If there is an answer in the table, then it does the same thing as `table_use_answer`, returning the answer to the caller. Otherwise, if no answer is available, then it triggers backtracking.

Note that this situation is possible since the answer being added may have been in the table and may have already been consumed by the call.

table_check_completion L1

As we mentioned above, `L2` is the address to go to after all the clauses have been executed. The `table_use_answer` instruction at `L2` returns all remaining answers. After that, the `table_check_completion L1` instruction determines whether or not the current predicate need be re-executed. If the `AR` field in the table entry for the call is `COMPLETE`, then discard the current frame and fail; if there was some new answers added into the table during the last round of execution, then re-execute the predicate starting from `L1`; otherwise, if no new answer was added into the table during the last round of execution, then set the `AR` field to be `COMPLETE`, discard the current frame, and fail.

3.4 Cut

The cut operator '!' in SLDT behaves in strictly the same way as that in the SLD resolution. Consider a cut in the clause

```
H:-L,!,R.
```

The cut `!` discards the choice points created for `H` and `L`. With tabling, however, we cannot just discard the choice points. We also have to cut the connection between the tabled calls in `L` and their table entries by properly updating the `AR` fields in the table entries. Otherwise, the `AR` fields in some table entries may become dangling pointers pointing to frames that no longer exist. We handle the cut in three different ways depending on the context in which the cut occurs:

- If `H` and all those called (directly or indirectly) by `L` are non-tabled, then we just treat the cut as a cut in a standard Prolog program, letting it discard the choice points created for `H` and `L`.
- If `H` is not tabled but there is at least one tabled predicate call in `L`, then we let the cut discard the choice points and cut the relationship between the tabled calls and their table entries by resetting the `AR` fields in the entries to be `NULL`. We introduce a new instruction, called `table_cut_outside`, to encode this type of cuts, where `outside` means that the cut does not reside in a tabled predicate.
- If `H` is a tabled predicate, then we let the cut discard the choice points created by `L`, cut the relationship between the tabled calls and their table entries, and set the backtracking point to be the address of the completion-checking clause. So, when the calls to the right of the cut fail, the `table_use_answer` and `table_check_completion` instructions will be executed. We introduce another new instruction, called `table_cut_inside` to encode this type of cuts, where `inside` means that the cut resides in a tabled predicate.

After the AR field in a subgoal table entry becomes NULL, the next variant call will be treated as a pioneer (see the table_allocate instruction).

Consider, as an example, a cut that does not reside in a tabled predicate but has tabled calls in its scope:

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

The call q(X) produces one answer, namely q(a), and binds X to a. After that, the cut discards the choice point for q(X) and disconnects q(X) and its table entry by setting the AR field of the entry to be NULL. Because the relationship was cut off, q(Y) will be executed like a pioneer. The answers returned to the query ?-p(X,Y) will be p(a,a) and p(a,b).

Consider, as another example, a cut that resides in a tabled predicate:

```
:- table q/1.
q(X):-!,q(X).
q(a).
```

Since there is no call appearing to the left of the cut, the cut just sets the backtracking point to be the address of the ending clause. When the call following the cut, which is a follower of the head, is executed, it will execute from the ending clause. Since no answer exists in the table, the follower will fail, which will cause the pioneer to fail too.

There is a case that we have not considered yet: what should we do if the AR field of the table entry of the current call is NULL when we execute the completion-checking clause? The following example illustrates this situation:

```
:- table p/1.
p(X):-q(X).
p(a).
p(b).
q(Y):-p(Y),!.
```

Consider the query p(X0). It is reduced to q(X0), which is re-written to p(X0),! by the last clause. The later p(X0) steals the choice point of the pioneer and executes from the second clause, i.e., p(a). After that, the cut sets the AR field of the table entry to NULL. Upon backtracking, the completion-checking clause will be executed for the pioneer. If the original table_check_completion instruction is used, we will lose the solution p(b)!

In general, a cut in a clause

```
q(...):-L,!,R
```

cannot be handled if there is a tabled predicate call p in L and q is called by p directly or indirectly. In other words, cuts that do not occur in cycles in the calling graph can be handled correctly. This is a significant progress considering that cuts have been ruled out completely from tabled Prolog systems. For a statement like "if there is a path in a graph, then do something", the XSB system has to preserve the status for computing all the paths even after one path is found.

4 Direct-Recursion Optimization (DRO)

Re-computation should be avoided if it is known to produce no new answers. It is an open problem to decide the exact class of predicates and calls for which re-computation is avoidable. In this section, we define a class of predicate calls for which re-computation is unnecessary and show how to optimize it.

Definition 1 A predicate is said to be *table-irrelevant* if it is not tabled and all the predicates it calls directly or indirectly are table-irrelevant.

Definition 2 A clause in a tabled predicate `p` is said to be *directly-recursive* if all the calls in the body either call `p` or call table-irrelevant predicates.

Definition 3 A tabled predicate is called a *DRO (Directly-Recursive Optimizable)* predicate if it consists of one directly-recursive clause and possibly several clauses whose bodies are table-irrelevant.

Theorem 4.1 *For a DRO predicate, re-computation is unnecessary for ancestor/descendant variant calls of the predicate.*

Let the following be the directly-recursive clause in a DRO predicate:

$P :- Q_1, ..., Q_i, P, Q_{i+1}, ..., Q_n$

In database terms, the relation `P` is defined as a join of `P` and Q_is. In deductive databases where rules are evaluated bottom-up, re-evaluation of recursive rules is needed to reach a fixpoint [1]. In SLDT, however, since the relations are joined tuple by tuple and a newly generated tuple is added into the relation `P` immediately, no re-execution is necessary. As long as a follower has no answer to be used in the join, no new answer can be produced for its pioneer. This guarantees that we can fail a follower safely after it exhausts all its answers and clauses.

Note that re-computation is still required if the follower is not a descendant but a sibling of the former variant call. Consider the following program:

```
:-table p/2.
q(A,B,C,D):-p(A,B),p(C,D).
```

Table 1: Comparing time efficiency.

Progs	tcl	tcr	sg	cs_o	cs_r	disj	gabriel	kalah	peep	pg	read
B/XSB	1.0	1.0	0.52	2.95	3.45	3.30	3.60	2.60	4.50	3.00	2.08

```
p(X,Z):-p(X,Y1),p(Y2,Z),Y1=Y2.
p(X,Y):-t(X,Y).

t(a,b). t(b,c). t(c,a).
```

Without re-computation of `p(C,D)`, the solution `q(a,b,a,a)` and many others would be lost.

We introduce another new instruction, called `table_end L`, which substitutes `table_check_compl` `L` instruction in the completion-checking clauses in DRO predicates. This instruction first checks whether the parent call and the pioneer of the current call are the same. If so, it behaves just like `table_check_completion` as though the fixpoint has been reached. Otherwise, if the parent and the pioneer are different, then `table_end L` behaves exactly the same as `table_check_completion`. Note that what is offered by the theorem is not fully exploited here. We only avoid re-computation for parent/child calls but not ancestor/descendant calls because we want to avoid the more expensive test of the ancestor/descendant relationship.

5 Performance Evaluation

Tables 1 compares the time efficiency of B-Prolog (version 6.0[3]) with that of XSB (version 2.4) for a set of programs. For XSB, the batched scheduling strategy was used. The `tcl` is the left-recursive version and the `tcr` is the right-recursive version of a program for computing the transitive closure of a graph. The `sg` computes the same-generation relation of the graph. The rest of the programs are taken from [4]. The numbers are the ratios of the time taken by B-Prolog to that taken by XSB. The comparison was done on a SPARC-10 workstation.

For the three small programs, namely, `tcl`, `tcr` and `sg`, B-Prolog has a time performance comparable with or better than that of XSB. These three programs are datalog (i.e., function-free) programs and require no or little re-computation[4].

For the CHAT suite, B-Prolog is on average about three times as slow as XSB. Two factors contribute to this result: First, DRO is applicable to none of the predicates in the

[3]The tabling mechanism in B-Prolog was implemented in 1998 and has been left untouched since then.

[4]No re-computation is necessary if the ancestor/descendent relation is checked. In B-Prolog, however, as only parent/child relation is checked, re-computation is still performed for `tcr` and `sg` even with the DRO optimization.

Table 2: Comparing stack space efficiency.

Progs	tcl	tcr	sg	cs_o	cs_r	disj	gabriel	kalah	peep	pg	read
B/XSB	0.43	0.96	1.03	2.40	3.43	2.09	0.79	0.87	0.07	0.12	0.05

Table 3: Comparing table space efficiency.

Progs	tcl	tcr	sg	cs_o	cs_r	disj	gabriel	kalah	peep	pg	read
B/XSB	0.46	0.72	1.64	0.80	0.79	0.93	0.86	0.79	0.70	0.89	0.84

suite and thus re-computation is necessary; and second, arguments of the tabled calls are complex terms for which the decision-tree data structure, called *trie*, adopted in XSB [11] should be much faster than hash tables employed in B-Prolog. It is difficult to draw a consistent conclusion from the figures. On the one hand, B-Prolog is on average twice as fast as XSB for standard Prolog programs, and on the other hand the trie data structure used in XSB is far more advanced than hash tables used in B-Prolog for managing the table area. Profiling the execution of the CHAT programs on B-Prolog shows that 75% of the time is spent in accessing the table area. Therefore, fast data structures for tabled calls are more important than a fast Prolog engine for these programs. This implies that the potential for improvement is very large in B-Prolog as long as tabling is concerned.

Table 2 compares the amounts of stack space required by B-Prolog and XSB to run the programs. For XSB, the CHAT area, which is used to preserve computation status, is also counted. For some programs, B-Prolg consumes more space because for each tabled call a copy is made on the control stack for producing answers. XSB requires more stack space for some programs because it keeps multiple paths of the search trees. For `read`, for example, XSB consumes more than 20 times stack space than B-Prolog.

Table 3 compares the amounts of the required table space. The table space area stores subgoals and answers. For each variant of tabled subgoals, only one copy needs to be stored. Also, the number of answers that need to be stored is independent of the tabling mechanism. Therefore, if the same data structures are used to manage data in the table area, the required table space should be the same on both systems. The figures, in fact, do not show much difference although different data structures are used in XSB and B-Prolog.

6 Related Work

Currently, there is only one widely available Prolog system, namely XSB, that supports tabling. In XSB, a very first tabled call is treated as a producer and all subsequent variant calls are treated as consumers. When execution backtracks to the producer, the

state of the consumers must be preserved. This non-chronological backtracking scheme does not go well with the WAM.

To simplify the implementation of tabling in XSB, Demoen and Sagonas proposed two alternative schemes for preserving the choice points of consumers and the related stacks [3, 4]. Instead of preserving the state of consumers by freezing the stack and heap, the new schemes save part of the state in a different area. These schemes are simpler than SLG-WAM and impose less overhead on the execution of standard Prolog programs, but the implementation is still complicated and the cut operator is left unhandled.

In this paper, we presented the implementation of a linear tabling mechanism. The linear feature brings several advantages: Firstly, a class of useful cuts can be handled correctly. This is a big step forward from the XSB system, where no tabled calls can reside in the scope of a cut. In XSB, for example, the sentence "if there is a path between two vertexes in a graph, then do something" cannot be described unless all the paths between the vertexes are found. Secondly, the implementation is relatively easy. Thirdly, the implementation imposes no overhead on the execution of standard Prolog programs. In comparison, the implementation of SLG-WAM [9] imposes about 10% overhead on standard Prolog programs. The overhead has been reduced significantly in CHAT [4], but an overhead-free implementation is yet to be achieved. The disadvantage of our scheme is the necessity of re-computation.

Recently, Guo and Gupta [6] proposed a tabling scheme that combines the advantages of SLG-WAM and SLDT. It reorders the alternative clauses for tabled calls such that non-looping clauses are executed before looping clauses [5] and only looping clauses need be re-executed.

7 Concluding Remarks

The need to extend Prolog to narrow the gap between declarative and procedural readings of programs has been urged long before [8]. Although tabling has been perceived as necessary to narrow the gap, it has not raised much interest among Prolog implementors compared with other extensions such as concurrency and constraint solving. One primary reason may be the lack of an easy-to-implement tabling mechanism. In this paper, we proposed a mechanism that is much simpler than the SLG-WAM. Our implementation has a comparable performance with XSB for programs that do not require re-computation, and is still slower than XSB for programs that require re-computation and/or manipulation of complex terms. We believe that the gap will be gone after more efficient data structures are adopted for tables and more effective techniques are invented for eliminating re-computation.

[5] A looping clause is a clause that generates a variant call.

Acknowledgement

This work was started while the first author visited University of Alberta in the summer of 1998 and most of the implementation work was done while the first author was with Kyushu Institute of Technology. The first author is supported in part by ONR grant N00014-96-1-1057, and the second author is supported in part by Chinese NNSF and Trans-Century Training Programme Foundation for the Talents by the Chinese Ministry of Education. We are grateful to Bart Demoen and Kostis Sagonas for kindly sharing with us the CHAT benchmark suite.

References

[1] Bancilhon, F. and Ramakrishnan, R.: An Amateur's Introduction to Recursive Query Processing Strategies, *SIGMOD'86*.

[2] Chen, W. and Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs, *J. ACM*, Vol.43, No.1, 20-74, 1996.

[3] Demoen, B. and Sagonas, K.: CAT: The Copying Approach to Tabling, *Proceedings of PLILP'98*, 1998.

[4] Demoen, B. and Sagonas, K.: CHAT: The Copy-Hybrid Approach to Tabling, *Proceedings of PADL'99, LNCS 1551*, pp.106-121, 1999.

[5] Fa, C.G. and Dietrich, S.W. : Extension Table Built-ins for Prolog, *Software Practice and Experience*, Vol.22, No.7, 573-597, 1992.

[6] Guo, H.F. and Gupta, G.: Efficient Tabling with Dynamic Alternative Reordering. In *Proc. of 2nd Workshop on Tabulation in Parsing and Deduction (TAPD)*, Sep. 2000.

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

[8] Parker, D.S., Carey, M. Jarke, M. Sciore, E. and Walker, A.: Logic Programming and Databases, *Expert Database Systems*, Kersobberg Larry, Ed., The Benjamin/Cummings Pub., 1986.

[9] Sagonas, K. and Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs, *ACM Transactions on Programming Languages and Systems*. Vol.20, No.3, 1998.

[10] Shen, Y.D., Yuan, L., You, J.H. and Zhou, N.F.: Linear Tabulated Resolution Based on Prolog Control Strategy, *Theory and Practice of Logic Programming (TPLP)*, Vol.1, No.1, pp.71-103, 2001.

[11] Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., and Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs, *J. Logic Programming*, vol. 38, pp.31-54, 1998.

[12] Tamaki, H. and Sato, T.: OLD Resolution with Tabulation, *Proc. of the Third ICLP*, LNCS 225, 84-98, 1986.

[13] Warren, D.H.D.: An Abstract Prolog Instruction Set, Technical Report 309, SRI International, 1983.

[14] Warren, D.S.: Memoing for Logic Programs, *CACM*, Vol.35, No.3, pp.93-111, 1992.

[15] Zhou, N.F.: Parameter Passing and Control Stack Management in Prolog Implementation Revisited, *ACM Transactions on Programming Languages and Systems*, Vol.18, No.6, 752-779, 1996.

Appendix I: Tabling Instructions

```
table_allocate Arity, Size, p, L:
     copy the arguments to the slots directly after the original arguments;
     allocate Size slots;
     tableEntry = lookupTable(AR,p,Arity);
     if (tableEntry == NULL){ /* AR is a pioneer */
         AR->Table = tableEntry = add a new table entry;
         AR->Pioneer = AR;
         P = NextInstruction;
     } else {
         AR->Table = tableEntry;
         if (tableEntry->AR==NULL){ /* treat AR as a pioneer */
             AR->Pioneer = AR;
             tableEntry->AR = AR;
             P = NextInstruction;
         } else if (tableEntry->AR==COMPLETE){
             P = L;
         } else {
             AR->Pioneer = tableEntry->AR->Pioneer;
             P = tableEntry->AR->CPF; /* steal backtracking point */
             tableEntry->AR->CPF = L;
             tableEntry->AR = AR;
        }
     }
     AR->CurrA = first answer in tableEntry;
     AR->CPF = P;
     do bookkeeping operations for backtracking;

table_add_use_answer:
     add the current answer to the table if it is not there
     if (AR->CurrA->Next!=NULL){
         goto consume_answer;
     }  else fail.

table_use_answer:
     if (AR->CurrA->Next!=NULL){ /* answer available */
     consume_answer:
         unify the answer with the original arguments
         AR->CurrA = AR->CurrA->Next;
         P = AR->CP; /* return control to the caller */
         AR = AR->AR;
     } else P = NextInstruction;

table_check_completion L:
     if (AR->Table->AR==COMPLETE){
        cut_fail;
     } else if (AR->Table->Revised==true){
         P = L;
     } else {
      fix_point_reached:
         AR->Table->AR = COMPLETE;
         B = AR->B; /* discard the current choice point frame */
         fail;      /* provoke backtracking */
     }

table_end L:
     if (AR->Pioneer == AR->AR){ /* AR is a child of the pioneer */
         goto fix_point_reached;
     } else goto table_check_completion;
```