On Improving the Efficiency of Deterministic Calls and Answers in Tabled Logic Programs

Miguel Areias and Ricardo Rocha

DCC-FC & CRACS
University of Porto, Portugal
{miguel-areias,ricroc}@dcc.fc.up.pt

Abstract. The execution model on which most tabling engines are based allocates a choice point whenever a new tabled subgoal is called. This happens even when the call is deterministic. However, some of the information from the choice point is never used when evaluating deterministic tabled calls with batched scheduling. Moreover, when a deterministic answer is found for a deterministic tabled call, the call can be completed early and the corresponding choice point can be removed. Thus, if applying batched scheduling to a long deterministic computation, the system may end up consuming memory and evaluating calls unnecessarily. In this paper, we propose a solution that tries to reduce this memory and execution overhead to a minimum. Our experimental results show that, for deterministic tabled calls and tabled answers with batched scheduling, it is possible not only to reduce the memory usage overhead, but also the running time of the execution.

Keywords: Tabling, Deterministic Calls and Answers, Implementation.

1 Introduction

Tabling [1] is an implementation technique that overcomes some limitations of traditional Prolog systems in dealing with redundant sub-computations and recursion. Implementations of tabling are now widely available in systems like XSB Prolog, Yap Prolog, B-Prolog, ALS-Prolog, Mercury and more recently Ciao Prolog. The increasing interest in tabling led to further developments and proposals that improve some practical deficiencies of current tabling execution models in key aspects of tabled evaluation like re-computation, scheduling and memory recovery. The discussion we address in this work also results from practical deficiencies that we have found in the execution data structures and algorithms used to deal with deterministic tabled calls and answers if applying batched scheduling [2].

The execution model on which most tabling engines are based allocates a choice point whenever a new tabled subgoal is called. This happens even when the call is deterministic, i.e., *defined by a single matching clause*. This is necessary since the information from the choice point is crucial to correctly implement some tabling operations. However, some of this information is never used when evaluating deterministic tabled calls with batched scheduling. Moreover, when an answer found for a deterministic tabled call is known to be a deterministic answer, i.e., *be the single matching answer for the call*, then the call can be completed early [3] and the corresponding choice point can be removed.

Thus, if tabling is applied to a long deterministic computation, the system may end up consuming memory and evaluating calls unnecessarily. In this paper, we propose a solution that tries to reduce this memory and execution overhead to a minimum. We will focus our discussion on a concrete implementation, the YapTab system [4], an efficient suspension-based tabling engine that extends the state-of-the-art Yap Prolog system to support tabled evaluation for definite programs, but our proposal can be generalized and applied to other suspension-based tabling engines. Our experimental results show that, for deterministic tabled calls and tabled answers with batched scheduling, it is possible not only to reduce the memory usage overhead, but also the running time of the execution.

The remainder of the paper is organized as follows. First, we briefly introduce the main background concepts about tabled evaluation. Next, we discuss in more detail how YapTab compiles and dynamically indexes deterministic tabled calls and how deterministic tabled answers can be handled. We then describe how we have extended YapTab to provide engine support to efficiently deal with deterministic tabled calls and answers. Finally, we present some experimental results and we end by outlining some conclusions.

2 Basic Tabling Concepts

Tabling consists of storing intermediate answers for subgoals so that they can be reused when a repeated subgoal appears[1]. Whenever a tabled subgoal is first called, a new entry is allocated in an appropriated data space, the *table space*. Table entries are used to collect the answers found for their corresponding subgoals. Moreover, they are also used to check whether calls to subgoals are repeated. Repeated calls to tabled subgoals are not re-evaluated against the program clauses, instead they are resolved by consuming the answers already stored in their table entries. During this process, as further new answers are found, they are stored in their tables and later returned to all repeated calls. Within this model, the nodes in the search space are classified as either: *generator nodes*, corresponding to first calls to tabled subgoals; *consumer nodes*, corresponding to repeated calls to tabled subgoals; or *interior nodes*, corresponding to non-tabled subgoals.

The YapTab design follows the seminal SLG-WAM design [3]: it extends WAM's execution model with a new data area, the *table space*; a new set of registers, the *freeze registers*; an extension of the standard trail, the *forward trail*; and four new operations for definite programs:

Tabled Subgoal Call: this operation checks if the subgoal is in the table space. If so, it allocates a consumer node and starts consuming the available answers. If not, it allocates a generator node and adds a new entry to the table space.

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

New Answer: this operation checks whether a newly found answer is already in the table, and if not, inserts the answer. Otherwise, the operation fails.

Answer Resolution: this operation checks whether extra answers are available for a particular consumer node and, if so, consumes the next one. If no answers are available, it suspends the current computation and schedules a possible resolution to continue the execution.

Completion: this operation determines whether a tabled subgoal is completely evaluated. A subgoal is said to be completely evaluated when the set of stored answers represents all the conclusions that can be inferred from the set of facts and rules in the program. When this is the case, the operation closes the subgoal's table entry and reclaims stack space. Otherwise, control moves to a consumer with unconsumed answers.

During tabled evaluation, at several points, we can choose between continuing forward execution, backtracking to interior nodes, returning answers to consumer nodes, or performing completion. The decision on which operation to perform is determined by the *scheduling strategy*. Different strategies may have a significant impact on performance, and may lead to a different ordering of solutions to the query goal. Arguably, the two most successful tabling scheduling strategies are batched scheduling and local scheduling [2]. YabTab supports both batched scheduling, local scheduling and the dynamic intermixing of batched and local scheduling at the subgoal level [5]. Local scheduling does not have any relevance for this work, so we will not consider it.

Batched scheduling schedules the program clauses in a depth-first manner as does the WAM. It favors forward execution first, backtracking next, and consuming answers or completion last. It thus tries to delay the need to move around the search tree by batching the return of answers. When new answers are found for a particular tabled subgoal, they are added to the table space and the execution continues. For some situations, this results in creating dependencies to older subgoals, therefore enlarging the current SCC (*Strongly Connected Component*) [3] and delaying the completion point to an older generator node. By default in YapTab, tabled predicates are evaluated using batched scheduling [5].

3 Deterministic Tabled Calls and Answers in YapTab

In this section, we discuss how tabled predicates are compiled in YapTab and, in particular, we show how YapTab uses the Yap compiler to generate compiled and indexed code for deterministic tabled calls. We then discuss how deterministic tabled answers for deterministic tabled calls can be handled in YapTab.

3.1 Compilation of Tabled Predicates

Tabled predicates defined by several clauses are compiled using the `table_try_me`, `table_retry_me` and `table_trust_me` WAM-like instructions in a manner similar to the generic `try_me`/`retry_me`/`trust_me` WAM sequence. The `table_try_me` instruction extends the WAM's `try_me` instruction to support the tabled subgoal call operation. The `table_retry_me` and `table_trust_me` differ from the generic WAM instructions in that they restore a generator choice point rather than a standard WAM choice point. Tabled predicates defined by a single clause are compiled using the `table_try_single` WAM-like instruction, a specialized version of the `table_try_me` instruction for deterministic tabled calls. We next show YapTab's compiled code for a tabled predicate `t/1` defined by a single clause and for a tabled predicate `t/3` defined by several clauses.

```
% predicate definitions
:- table t/1, t/3.
t(X) :- ...
t(a1,b1,c1) :- ...
t(a1,b2,c2) :- ...
t(a1,b1,c3) :- ...
t(a2,b2,c4) :- ...

% compiled code generated by YapTab for predicate t/1
t1_1:  table_try_single t1_1a
t1_1a: ‘WAM code for clause t(X) :- ...’

% compiled code generated by YapTab for predicate t/3
t3_1:  table_try_me t3_2
t3_1a: ‘WAM code for clause t(a1,b1,c1) :- ...’
t3_2:  table_retry_me t3_3
t3_2a: ‘WAM code for clause t(a1,b2,c2) :- ...’
t3_3:  table_retry_me t3_4
t3_3a: ‘WAM code for clause t(a1,b1,c3) :- ...’
t3_4:  table_trust_me
t3_4a: ‘WAM code for clause t(a2,b2,c4) :- ...’
```

As `t/1` is a deterministic tabled predicate, the `table_try_single` instruction will be executed for every call to this predicate. On the other hand, `t/3` is a non-deterministic tabled predicate, but some calls to it can be deterministic, i.e., defined by a single matching clause. Consider, for example, the calls `t(a2,X,Y)` and `t(X,Y,c3)`. These two calls are deterministic as each of them matches with a single `t/3` clause, respectively, the 4th and 3rd clause. We next describe how YapTab uses the demand-driven indexing mechanism of Yap to dynamically generate `table_try_single` instructions for this kind of deterministic calls.

3.2 Demand-Driven Indexing

Yap implements *demand-driven indexing* (or *just-in-time indexing*) [6] since version 5. The idea behind it is to generate flexible multi-argument indexing of Prolog clauses during program execution based on actual demand. This feature is implemented for static code, dynamic code and the internal database. All indexing code is generated on demand for all and only for the indices required. This is done by building an indexing tree using similar building blocks to the WAM but it generates indices based on the instantiation of the current goal, and expands indices given different instantiations for the same goal.

This powerful optimization allows YapTab to execute some calls to non-deterministic tabled predicates like deterministic tabled predicates. This happens when Yap's indexing scheme finds that for a particular call to a non-deterministic tabled predicate, there is only a single clause that matches the call. We next show an example illustrating the indexed code generated for a non-deterministic call and two deterministic calls to the previous `t/3` tabled predicate.

```
% indexed code generated by YapTab for call t(a1,X,Y)
table_try   t3_1a
table_retry t3_2a
table_trust t3_3a

% indexed code generated by YapTab for call t(a2,X,Y)
table_try_single t3_4a

% indexed code generated by YapTab for call t(X,Y,c3)
table_try_single t3_3a
```

The call `t(a2,X,Y)` is non-deterministic as it matches the 1st, 2nd and 3rd clauses of `t/3`, so a `table_try`/`table_retry`/`table_trust` sequence is generated. The other two calls, `t(a3,X,Y)` and `t(X,Y,c3)`, are both deterministic as they only match a single `t/3` clause, so a `table_try_single` instruction can be generated. Note however, that there are situations where a call can be deterministic, but Yap's indexing scheme cannot detect it as deterministic in order to generate the appropriate `table_try_single` instruction [6]. In such cases, we cannot benefit directly from our approach, but we can take advantage of the similarities between the `table_try_single` instruction and the *last matching clause* of a non-deterministic tabled call to apply our approach later.

3.3 Last Matching Clause

When evaluating a tabled predicate, the last matching clause of a call to the predicate is implemented either by the `table_trust_me` instruction or by the `table_trust` instruction. The former situation occurs when we have a generic call to the predicate (all the arguments of the call are unbound variables) and the latter situation occurs when we have a more specific call (some of the arguments are at least partially instantiated) optimized by indexing code. In a WAM-based implementation, the last matching clause of a call is implemented by first restoring all the necessary information from the current choice point (usually pointed to by the WAM's `B` register) and then, by discarding the current choice point (by updating `B` to its predecessor). In a tabled implementation, the `table_trust_me` and `table_trust` instructions also restore all the necessary information from the current choice point `B`, but instead of updating `B` to its predecessor, they update the next clause field of `B` to the `completion` instruction. By doing that, they force completion detection when the computation backtracks again to `B`, i.e., whether the clauses for the subgoal call at hand are all exploited.

Hence, the computation state that we have when executing a `table_trust_me` or `table_trust` instruction is similar to that one of a `table_try_single` instruction, that is, in both cases the current clause can be seen as deterministic as it is the last (or single) matching clause for the subgoal call at hand. Thus, we can view the `table_trust_me` and `table_trust` instructions as a special case of the `table_try_single` instruction. This means that the approach used for the `table_try_single` instruction to efficiently deal with deterministic tabled calls can be applied to the `table_trust_me` and `table_trust` instructions. We discuss the implementation details for these instructions in the next section.

3.4 Deterministic Tabled Answers

A tabled answer is deterministic when it is the single matching answer for the corresponding tabled call. Determining when an answer is deterministic is important because the tabled call can be completed early and the corresponding choice point can be removed. Taking into account the formulation discussed above for the last matching clause, we can also extend the definition of being a deterministic tabled answer and say that a tabled answer is deterministic when we can safely conclude that no more answers will be found, i.e., when the current answer is the last (or single) matching answer for the corresponding tabled call.

However, during execution time, it is not always possible to conclude beforehand that no more answers will be found for a particular tabled call. This is a safe conclusion only when the tabled call is deterministic, i.e., the clause being executed for the tabled call at hand is the last (or single) matching clause, and the choice point for the tabled call is the topmost choice point, i.e., no alternative paths exist for evaluating the tabled call at hand. In what follows, we will thus assume that a tabled call is deterministic when the clause being executed for the call is the *last matching clause* and that a tabled answer is deterministic when the answer is the *last matching answer* for the corresponding tabled call.

4 Implementation Details

In this section, we describe in detail how we have extended YapTab to provide engine support to efficiently deal with deterministic tabled calls and answers.

4.1 Generator Nodes

In YapTab, a generator node is implemented as a WAM choice point extended with some extra fields. The format of a generic generator choice point in YapTab is depicted in Fig. 1(a). Fields that are not found in standard WAM choice points are coloured gray. A generator choice point is divided in three sections. The top section contains the usual WAM fields needed to restore the computation on backtracking plus two extra fields [5]: `cp_dep_fr` is a pointer to the corresponding dependency frame, used by local scheduling for fix-point check, and `cp_sg_fr` is a pointer to the associated subgoal frame where answers should be stored. The middle section contains the argument registers of the subgoal and the bottom section contains the *substitution factor* [7], i.e., the set of free variables which

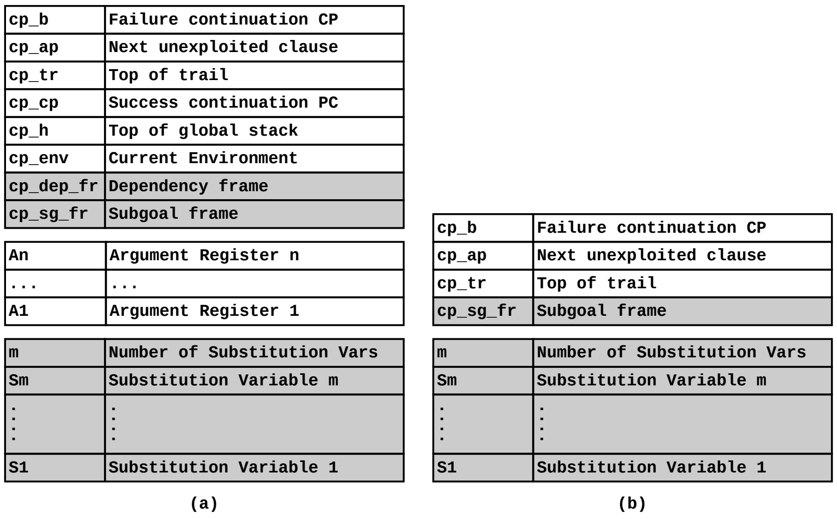
cp_b Failure continuation CP
cp_ap Next unexploited clause
cp_tr Top of trail
cp_cp Success continuation PC
cp_h Top of global stack
cp_env Current Environment
cp_dep_fr Dependency frame
cp_sg_fr Subgoal frame
An Argument Register n
... ...
A1 Argument Register 1
m Number of Substitution Vars
Sm Substitution Variable m
S1 Substitution Variable 1
(a)
cp_b Failure continuation CP
cp_ap Next unexploited clause
cp_tr Top of trail
cp_sg_fr Subgoal frame
m Number of Substitution Vars
Sm Substitution Variable m
S1 Substitution Variable 1
(b)

Fig. 1. (a) Generic and **(b)** deterministic generator choice points in YapTab

exist in the terms in the argument registers. The substitution factor is an optimization that allows the new answer operation to store in the table space only the substitutions for the free variables in the subgoal call.

If we now turn our attention to how generator choice points are handled during evaluation, we find that some of this information is never used when evaluating deterministic tabled calls with batched scheduling. This happens mainly because, with batched scheduling, the computation is never resumed in a deterministic generator choice point. This allows us to remove the argument registers and the standard `cp_cp`, `cp_h` and `cp_env` fields. The `cp_dep_fr` field can also be removed because it is only necessary with local scheduling [5].

Figure 1(b) shows the new format of a deterministic generator choice point in YapTab. The `cp_b` field is needed for failure continuation; the `cp_ap` and `cp_tr` are required when backtracking to the choice point; the `cp_sg_fr` is required by the new answer and completion operations; and the substitution factor fields are required by the new answer operation. This optimization is similar to the *shallow backtracking* optimization as introduced by Carlsson [8]. The main difference to our approach is that, instead of delaying the initialization of some choice point fields, here we can safely ignore and remove them as they are never used.

Considering that N is the number of arguments registers and that M is the number of substitution variables, the ratio of memory usage in the choice point stack between the new and the original representation of deterministic generator choice points can be expressed as follows:

$$\frac{4+1+M}{8+N+1+M}$$

4.2 Tabling Operations

Dealing with deterministic tabled calls and answers required small changes to the tabled subgoal call, completion and new answer operations. Figure 2 shows in more detail the changes (blocks of code marked with a '`// *** new ***`' comment) made to the `table_try_single` and `table_trust_me`[2] instructions.

```
table_try_single(TABLED_CALL tc) {
  sg_fr = subgoal_check_insert(tc)   // sg_fr is the subgoal frame for tc
  if (new_tabled_subgoal_call(sg_fr)) {
    if (evaluation_mode(tc) == batched_scheduling)         // *** new ***
      store_deterministic_generator_node(sg_fr)
    else                                               // local scheduling
      store_generic_generator_node(sg_fr)
    ...
  }
  ...
}


table_trust_me(TABLED_CALL tc) {
  // the B register points to the current choice point
  restore_generic_generator_node(B, COMPLETION)
  if (evaluation_mode(tc) == batched_scheduling &&
      not_in_a_frozen_segment(B) {                         // *** new ***
    subs_factor = B + sizeof(generic_generator_cp) + arity(tc)
    gen_cp = subs_factor - sizeof(deterministic_generator_cp)
    gen_cp->cp_sg_fr = B->cp_sg_fr
    gen_cp->cp_tr = B->cp_tr
    gen_cp->cp_ap = B->cp_ap
    gen_cp->cp_b  = B->cp_b
    B = gen_cp
  }
  ...
}
```

Fig. 2. Pseudo-code for the `table_try_single` and `table_trust_me` instructions

The `table_try_single` instruction now checks whenever the subgoal being called is to be evaluated using batched or local scheduling. If batched, it allocates a deterministic generator choice point. If local, it proceeds as before and allocates a generic generator choice point.

The `table_trust_me` instruction now checks if the current tabled call is being evaluated using batched scheduling and if the current choice point is not in a frozen segment[3]. If these two conditions hold, we can recover some memory space by transforming the current generator choice point into a deterministic generator

[2] The changes made to the `table_trust` instruction are identical.

[3] The YapTab system uses frozen segments to protect the stacks of suspended computations [4]. Thus, if the current choice point is trapped in a frozen segment it is worthless to try to recover memory from it using our approach.

choice point (we assume that the choice point stack grows upwards). To do that, we need to copy the `cp_sg_fr`, `cp_tr`, `cp_ap` and `cp_b` fields in the current choice point to their new position, just above the substitution factor variables.

Deterministic generator choice points can be accessed later when executing the completion and/or new answer operations. Since both generator types have different sizes, we need to distinguish which is the type of the generator in order to correctly access the fields required by each operation. The completion operation needs to access the `cp_sg_fr` field and the new answer operation needs to access the `cp_sg_fr` and substitution factor fields. Figure 3 shows in more detail the changes made to the `completion` and `new_answer` instructions. For

```
completion() {
  ...                                              // fixpoint check loop
  // subgoal completely evaluated
  if (is_deterministic_generator_cp(B)) {                  // *** new ***
    gen_cp = deterministic_generator_cp(B)
    sg_fr = gen_cp->cp_sg_fr
  } else {                              // generic generator choice point
    gen_cp = generic_generator_cp(B)
    sg_fr = gen_cp->cp_sg_fr
  }
  complete_subgoal(sg_fr)
  ...
}


new_answer(TABLED_CALL tc, CHOICE_POINT cp, ANSWER ans) {
  // cp is the generator choice point for the tabled call tc
  if (is_deterministic_generator_cp(cp)) {                 // *** new ***
    gen_cp = deterministic_generator_cp(cp)
    sg_fr = gen_cp->cp_sg_fr
    subs_factor = gen_cp + sizeof(deterministic_generator_cp)
  } else {                              // generic generator choice point
    gen_cp = generic_generator_cp(cp)
    sg_fr = gen_cp->cp_sg_fr
    subs_factor = gen_cp + sizeof(generic_generator_cp) + arity(tc)
  }
  insert = answer_check_insert(sg_fr, subs_factor)
  if (insert == FALSE || evaluation_mode(tc) == local_scheduling) {
    fail()                         // repeated answer or local scheduling
  } else {                           // new answer and batched scheduling
    if (is_deterministic_generator_cp(gen_cp) &&
        topmost_choice_point(gen_cp) {                     // *** new ***
      // the new answer is deterministic, thus the tabled call can be ...
      mark_as_completed(sg_fr)         // ... completed early and the ...
      B = gen_cp->cp_b    // .. corresponding choice point can be removed
    }
    ...
  }
}
```

Fig. 3. Pseudo-code for the `completion` and `new_answer` instructions

the `new_answer` instruction, we also show the pseudo-code that determines when an answer is deterministic. Remember from section 3.4 that we can conclude that an answer is deterministic when the tabled call is deterministic and the choice point for the tabled call is the topmost choice point[4].

5 Experimental Results

We next present some experimental results comparing YapTab with and without the new support for deterministic tabled calls and answers. The environment for our experiments was an Intel(R) Core(TM)2 Quad CPU Q9550 2.83GHz with 2 GBytes of main memory and running the Linux kernel 2.6.24-24 with YapTab 5.1.3. To put the performance results in perspective and have a well-defined starting point comparing the memory ratio between the new and the original representation of deterministic generator choice points, first we have defined three deterministic tabled predicates, respectively with arities 5, 11 and 17, that simply call themselves recursively:

```
:- table t/5, t/11, t/17.

t(N,A2,A3,A4,A5) :- N>0, N1 is N-1, t(N1,A2,A3,A4,A5).

t(N,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11) :-
  N>0, N1 is N-1, t(N1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11).

t(N,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,A16,A17) :- N>0,
  N1 is N-1, t(N1,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,A16,A17).
```

The first argument `N` controls the number of times the predicate is executed. It thus defines the number of generator choice points to be allocated (we used a value of 200,000 in our experiments). In order to have specific combinations of argument registers (column ***Args***) and substitution variables (column ***Subs***), we have run each predicate with three different sets of free variables in the arguments. Table 1 shows the local stack[5] memory usage, in KBytes, and the running time, in milliseconds, for YapTab without (column ***YapTab***) and with (column ***YapTab+Det***) the new support for deterministic tabled calls and answers. A third column shows the memory and running time ratio between both approaches. For the memory ratio, we show in parentheses the ratio given by the formula introduced at the end of section 4.1.

As expected, the results in Table 1 confirm that memory reduction increases when the number of argument registers is bigger and the number of substitution variables is smaller. This is consistent with the formula presented in section 4.1. The small difference between our experiments and the values obtained when using the formula came from the fact that, in the formula, we are considering a

[4] Detecting if a choice point is the topmost choice point is implemented by comparing it against the `B` register (that points to the topmost choice point on the current branch) and the `B_FZ` register (that points to the topmost frozen choice point).

[5] In YapTab, the local stack contains both choice points and environment frames [9]. Other systems, like XSB Prolog, have separate choice point and environment stacks.

Args	***Subs***	***YapTab*** ***Memory***	***Time***	***YapTab+Det*** ***Memory***	***Time***	***YapTab+Det/YapTab*** ***Memory***	***Time***
5	**4**	18,751	224	11,719	160	0.62 (0.50)	0.71
5	**2**	17,188	216	10,157	148	0.59 (0.44)	0.69
5	**0**	15,626	216	8,594	152	0.55 (0.36)	0.70
11	**10**	28,126	332	16,407	240	0.58 (0.50)	0.72
11	**5**	24,219	256	12,501	268	0.52 (0.40)	1.05
11	**0**	20,313	232	8,594	184	0.42 (0.25)	0.79
17	**16**	37,501	444	21,094	340	0.56 (0.50)	0.77
17	**8**	31,251	436	14,844	300	0.47 (0.38)	0.69
17	**0**	25,001	312	8,594	236	0.34 (0.19)	0.76
Average						**0.52 (0.39)**	**0.76**

Table 1. Local stack memory usage (in KBytes) and running times (in milliseconds) for YapTab without and with the new support for deterministic tabled calls and answers

local stack without environment frames and, in the experiments, for each generator choice point we are also allocating an environment frame[6]. The results in Table 1 also suggest that this memory reduction can have an impact in the running time of the execution.

Next, we tested our approach using two well-know problems: the *fibonacci* problem (using lists of integers to represent arbitrary numbers) and the *sequence comparisons* problem[7]. For both problems, we used a standard implementation (tabled predicates `fib/2` and `seq/3`) and a transformed implementation (tabled predicates `fib_t/2` and `seq_t/3`) that forces all tabled calls to use the `table_try_single` instruction. The Prolog code for the main predicates that implement both problems are presented next.

```
:- table fib/2, fib_t/2, seq/3, seq_t/3.

fib(0,[1]).
fib(1,[1]).
fib(N,L) :- N>1, N1 is N-1, N2 is N-2,
  fib(N1,L1), fib(N2,L2), sum_lists(L1,L2,L).

fib_t(N,L):- (N=<1 -> L=[1] ; (N1 is N-1, N2 is N-2,
  fib_t(N1,L1), fib_t(N2,L2), sum_lists(L1,L2,L))).

seq(0,0,0).
seq(0,M,M).
seq(N,0,N).
seq(N,M,C) :- N>0, M>0, N1 is N-1, M1 is M-1,
  seq(N1,M,C1), seq(N,M1,C2), seq(N1,M1,C3), min(N,M,C1,C2,C3,C).

seq_t(N,M,C) :- (N==0 -> M=C ; N>0, (M==0 -> C=M ; N1 is N-1, M1 is M-1,
  seq_t(N1,M,C1), seq_t(N,M1,C2), seq_t(N1,M1,C3), min(N,M,C1,C2,C3,C))).
```

[6] As these experiments do not store permanent variables [9] for environments, this corresponds to adding the size of an environment frame to both parts of our formula.

[7] In the *sequence comparisons* problem, we have two sequences A and B, and we want to determine the minimal number of operations needed to turn A into B.

We experimented the *fibonacci* problem with sizes 1000, 2000 and 4000 and the *sequence comparisons* problem with sequences of length 500, 1000 and 2000. Table 2 shows the local stack memory usage, in KBytes, and the running time, in milliseconds, for YapTab without (column ***YapTab***) and with (column ***YapTab+Det***) the new support for deterministic tabled calls and answers. Again, a third column shows the memory and running time ratio between both approaches.

Program	***Size***	***YapTab***		***YapTab+Det***		***YapTab+Det/YapTab***	
		Memory	***Time***	***Memory***	***Time***	***Memory***	***Time***
fib/2	1000	250	984	203	884	0.8120	0.8984
	2000	375	2,880	305	2,804	0.8133	0.9736
	4000	500	6,492	407	6,420	0.8140	0.9889
seq/3	500	45,914	792	39,079	448	0.8511	0.5657
	1000	183,625	8,108	156,282	3,272	0.8511	0.4036
	2000	734,438	135,580	718,813	117,483	0.9787	0.8665
fib_t/2	1000	250	988	125	368	0.5000	0.3725
	2000	375	3,040	188	1,268	0.5013	0.4171
	4000	500	6,516	250	2,828	0.5000	0.4340
seq_t/3	500	45,914	804	78	252	0.0017	0.3134
	1000	183,625	8,844	157	952	0.0009	0.1076
	2000	734,438	131,904	313	7,048	0.0004	0.0534

Table 2. Local stack memory usage (in KBytes) and running times (in milliseconds) for YapTab without and with the new support for deterministic tabled calls and answers

The results in Table 2 show improvements on the local stack memory usage and in the running time of the execution for all experiments. In particular, for the standard predicates, `fib/2` and `seq/3`, our approach shows, on average, a slightly worse tendency to memory and running time reduction, if compared with the results on Table 1. This happens mainly because of the existence of permanent variables in the body of the tabled clauses. However, on average, our approach is able to improve the performance of the execution for all `fib/2` and `seq/3` experiments. This suggests that it is possible to take advantage of our approach by using the last matching clause optimization when a program do not contains deterministic tabled predicates.

For the transformed predicates, `fib_t/2` and `seq_t/3`, our approach shows very good performance. For the `fib_t/2` experiments, it decreases, on average, memory usage to 50% and running time to 40%. For the `seq_t/3` experiments, it shows impressive gains. In particular for sequences of length 2000, it uses 2500 times less memory on the local stack and executes 19 times faster. This shows that our approach can be quite effective when we have deterministic tabled answers for deterministic tabled calls.

6 Conclusions and Further Work

We have presented a proposal for the efficient evaluation of deterministic tabled calls and answers with batched scheduling. A well-known aspect of tabling is the overhead in terms of memory usage compared with standard Prolog. This suggested to us the question of whether it was possible to minimize this overhead when evaluating deterministic tabled computations. Our initial results are quite promising, they suggest that, for deterministic tabled calls and tabled answers with batched scheduling, it is possible not only to reduce the memory usage overhead, but also the running time of the execution for certain applications.

Further work will include exploring the impact of applying our proposal to more complex problems, seeking real-world experimental results allowing us to improve and expand our current implementation.

Acknowledgements

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

References

1. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43**(1) (1996) 20–74
2. Freire, J., Swift, T., Warren, D.S.: Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In: International Symposium on Programming Language Implementation and Logic Programming. Number 1140 in LNCS, Springer-Verlag (1996) 243–258
3. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems **20**(3) (1998) 586–634
4. Rocha, R., Silva, F., Santos Costa, V.: On applying or-parallelism and tabling to logic programs. Theory and Practice of Logic Programming **5**(1 & 2) (2005) 161–205
5. Rocha, R., Silva, F., Santos Costa, V.: Dynamic Mixed-Strategy Evaluation of Tabled Logic Programs. In: International Conference on Logic Programming. Number 3668 in LNCS, Springer-Verlag (2005) 250–264
6. Santos Costa, V., Sagonas, K., Lopes, R.: Demand-Driven Indexing of Prolog Clauses. In: International Conference on Logic Programming. Number 4670 in LNCS, Springer-Verlag (2007) 395–409
7. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38**(1) (1999) 31–54
8. Carlsson, M.: On the Efficiency of Optimising Shallow Backtracking in Compiled Prolog. In: International Conference on Logic Programming, The MIT Press (1989) 3–16
9. Aït-Kaci, H.: Warren's Abstract Machine – A Tutorial Reconstruction. The MIT Press (1991)