Efficient Evaluation of Deterministic Tabled Calls

Miguel Areias and Ricardo Rocha

DCC-FC & CRACS
University of Porto, Portugal
c0507028@alunos.dcc.fc.up.pt ricroc@dcc.fc.up.pt

Abstract. The execution model in 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. Thus, if tabling is applied to a long deterministic computation, the system may end up consuming a huge amount of memory inadvertently. In this paper, we propose a solution that reduces this memory overhead to a minimum. Our results show that, for deterministic tabled calls with batched scheduling, it is possible not only to reduce the memory usage overhead, but also the running time of the evaluation.

Keywords: Tabling, Deterministic Calls, Implementation.

1 Introduction

Tabling [1, 2] 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 [3], Yap Prolog [4], B-Prolog [5], ALS-Prolog [6], Mercury [7] and more recently Ciao Prolog [8]. Actual implementations differ in the execution rule, in the data structures used to implement tabling, and in the changes to the underlying Prolog engine. Arguably, the SLG-WAM [9] is the most popular execution rule, but even here several issues require careful research, such as engine integration, execution data structures, termination detection, and scheduling support.

The increasing interest in tabling technology 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 [10, 11], scheduling [12] and memory recovery [13]. The discussion we address in this work also results from practical deficiencies that we have found in the execution data structures used to evaluate deterministic tabled calls if applying batched scheduling [14].

The execution model in 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. Thus, if tabling is applied to a long deterministic computation, the system may end up consuming a huge amount of memory inadvertently. In this paper, we propose a solution that reduces this memory 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 [15] to support tabled evaluation for definite programs, but our proposal can be generalized and applied to other tabling engines.

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. We then describe how we have extended YapTab to provide engine support to efficiently deal with deterministic tabled calls. At last, we present some preliminary 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 verify 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 [9]: it extends WAM's execution model [16] 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 is a call to a tabled subgoal. It 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 adds a new entry to the table space, and allocates a new generator node. When the call is deterministic, the tabled subgoal call operation is implemented by the `table_try_single` WAM-like instruction.

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

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

Answer Resolution: this operation verifies 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 complete when no more answers can be generated, that is, when its set of stored answers represent all the conclusions that can be inferred from the set of facts and rules in the program. If the subgoal has been completely evaluated, 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 [14]. YabTab supports both batched scheduling, local scheduling and the dynamic intermixing of batched and local scheduling at the subgoal level [12]. 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 evaluation continues. For some situations, this results in creating dependencies to older subgoals, therefore enlarging the current SCC (*Strongly Connected Component*) and delaying the completion point to an older generator node. By default in YapTab, tabled predicates are evaluated using batched scheduling [12].

3 Deterministic Tabled Calls 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.

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 similar manner 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. This instruction optimizes the `table_try_me` instruction for the case when the tabled predicate is defined by a single clause. Figure 1 shows the 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(X) :- ...

:- table t/3.
t(a1,b1,c1) :- ...
t(a2,b2,c2) :- ...
t(a2,b1,c3) :- ...
t(a2,b3,c1) :- ...
t(a3,b1,c2) :- ...


% 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(a2,b2,c2) :- ...'
t3_3:  table_retry_me t3_4
t3_3a: 'WAM code for clause t(a2,b1,c3) :- ...'
t3_4:  table_retry_me t3_5
t3_4a: 'WAM code for clause t(a2,b3,c1) :- ...'
t3_5:  table_trust_me
t3_5a: 'WAM code for clause t(a3,b1,c2) :- ...'
```

Fig. 1. Compilation of tabled predicates in YapTab

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 this predicate can be deterministic, i.e., defined by a single matching clause. Consider, for example, the previous definition of `t/3` and the calls `t(a3,X,Y)` and `t(X,Y,c3)`. These two calls are deterministic as they only match with a single `t/3` clause, respectively, the 5th and 3rd clause. We next show 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*) [17] 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 on the current goal, and expands indices given different instantiations for the same goal.

This powerful optimization provides that YapTab can execute 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. Figure 2 shows 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(a2,X,Y)
table_try   t3_2a
table_retry t3_3a
table_trust t3_4a

% indexed code generated by YapTab for call t(a3,X,Y)
table_try_single t3_5a

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

Fig. 2. Demand-driven indexing of tabled predicates in YapTab

The call `t(a2,X,Y)` is non-deterministic as it matches the 2nd, 3rd and 4th 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. 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 [16], 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.

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.

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 of YapTab is depicted in Figure 3. 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 [12]: `cp_dep_fr` is a pointer to the corresponding dependency frame, used by local scheduling for fixpoint 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*, i.e., the set of free variables which 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 [18].

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

cp_b	Failure continuation CP
cp_ap	Next unexploit alternative
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
Vm	Substitution Variable m
⋮	⋮
V1	Substitution Variable 1

Fig. 3. Format of a generic generator choice point in YapTab

because, with batched scheduling, the computation is never resumed in a deterministic generator choice point. This allow 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 [12], which is never the case. Figure 4 shows the new format of YapTab's deterministic generator choice point with the strictly necessary fields.

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. In order to avoid extra overheads when manipulating the different kinds of choice points that can coexist in an evaluation, we have rearranged all kinds of choice points in such a way that the top three fields are now the same as the ones for a deterministic generator choice point: the `cp_b`, `cp_ap` and `cp_tr` fields.

The memory reduction obtained with the new representation for deterministic generator choice points increases when the number of argument registers (the arity of the predicate being called) and the number of substitution variables are, respectively, bigger and smaller. Considering that A is the number of arguments registers and that S is the number of substitution variables, the percentage of memory saved with the new representation can be expressed as follows:

cp_b	Failure continuation CP
cp_ap	Next unexploit alternative
cp_tr	Top of trail
cp_sg_fr	Subgoal frame

m	Number of Substitution Vars
Vm	Substitution Variable m
. . .	. . .
V1	Substitution Variable 1

Fig. 4. Format of a deterministic generator choice point in YapTab

$$1 - \frac{4 + 1 + S}{8 + A + 1 + S}$$

4.2 Tabling Operations

In order to deal with the new representation for deterministic generator choice points, this required small changes to the tabled subgoal call, new answer and completion operations. Figures 5, 6, 7 and 8 show in more detail the changes (blocks of code marked with comment '`// new`') made to the `table_try_single`, `table_trust_me`[2], `new_answer` and `completion` instructions. Figure 9 shows the pseudo-code for the auxiliary procedure `is_deterministic_generator_cp()`. We assume that memory addresses grow downwards and that the choice point stack grows upwards.

```
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)
    ...
    goto next_instruction()
  }
  ...
}
```

Fig. 5. Pseudo-code for the `table_try_single` instruction

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

The table_try_single instruction now tests 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.

```
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. 6. Pseudo-code for the table_trust_me instruction

The table_trust_me instruction now tests 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 choice point. 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.

```
new_answer(TABLED_CALL tc, ANSWER ans) {
  if (is_deterministic_generator_cp(B)) {                           // new
    gen_cp = deterministic_generator_cp(B)
    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(B)
    sg_fr = gen_cp->cp_sg_fr
    subs_factor = gen_cp + sizeof(generic_generator_cp) + arity(tc)
  }
  ...
}
```

Fig. 7. Pseudo-code for the new_answer instruction

[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.

```
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 {
    gen_cp = generic_generator_cp(B)
    sg_fr = gen_cp->cp_sg_fr
  }
  complete_subgoal(sg_fr)
  ...
}
```

Fig. 8. Pseudo-code for the `completion` instruction

For the new answer and completion operations, since both generator types have different sizes, we need a way to correctly identify which is the type of the generator in order to correctly access the required fields on each structure. To do that, we use the `is_deterministic_generator_cp()` auxiliary procedure to test if a generator choice point is deterministic or not. Figure 9 shows the pseudo-code for it.

The `is_deterministic_generator_cp()` procedure assumes that, by default, we have a generic generator choice point and we check if the `cp_h` field (which is aligned with the field representing the number of substitution variables in a deterministic generator choice point) is less than the maximum number of allowed substitution variables (`MAX_SUBSTITUTION_VARS`). If this is case, then we know that we have a deterministic generator choice point.

```
is_deterministic_generator_cp(CHOICE_POINT cp) {
  gen_cp = generic_generator_cp(cp)
  if (gen_cp->cp_h <= MAX_SUBSTITUTION_VARS)
    return TRUE
  else
    return FALSE
}
```

Fig. 9. Pseudo-code for the `is_deterministic_generator_cp()` procedure

5 Preliminary Experimental Results

We next present some preliminary experimental results comparing YapTab with and without support for deterministic tabled calls. The environment for our experiments was a AMD Athlon(tm) 64 Processor 3200+ processor with 2 GByte of main memory and running the Linux kernel 2.6.24-19 with YapTab 5.1.3.

To evaluate the impact of our proposal, 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 100,000 in our experiments). In order to have specific combinations of argument registers and substitution variables, we have ran each predicate with three different sets of free variables in the arguments:

```
:- t(100000,A2,A3,A4,A5).
:- t(100000,A2,A3,0,0).
:- t(100000,0,0,0,0).

:- t(100000,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11).
:- t(100000,A2,A3,A4,A5,A6,0,0,0,0,0).
:- t(100000,0,0,0,0,0,0,0,0,0,0).

:- t(100000,A2,A3,A4,A5,A6,A7,A8,A9,A10,A11,A12,A13,A14,A15,A16,A17).
:- t(100000,A2,A3,A4,A5,A6,A7,A8,A9,0,0,0,0,0,0,0,0).
:- t(100000,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0).
```

These experiments are a kind of best-case scenario as they only allocate generator choice points and they do not store permanent variables for *environment frames* [16]. Table 1 shows the memory usage, in KBytes, for the local stack[4] and the running time, in milliseconds, for YapTab without (column ***YapTab***) and with (column ***YapTab+Det***) the new support for deterministic tabled calls. A third column ***Ratio (1–b/a)*** shows the memory and running time ratio between both approaches. For the memory ratio, we show in parentheses the percentage of memory saved if using the formula presented at the end of section 4.1.

The results in Table 1 indicate that YapTab with support for deterministic tabled calls can decrease, on average, memory usage by 48% and running time by 23%. These results also confirm that memory reduction increases when the number of argument registers is bigger and the number of substitution variables is smaller. This is coherent 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 local stack without environment frames.

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

Args	*Subs*	*YapTab (a)*		*YapTab+Det (b)*		*Ratio (1–b/a)*	
		Memory	*Time*	*Memory*	*Time*	*Memory*	*Time*
5	4	9,376	82	5,860	70	0.37 (0.50)	0.15
5	2	8,594	78	5,079	66	0.41 (0.57)	0.15
5	0	7,813	80	4,297	65	0.45 (0.64)	0.19
11	10	14,063	137	8,204	96	0.42 (0.50)	0.30
11	5	12,110	136	6,251	89	0.48 (0.60)	0.35
11	0	10,157	124	4,297	108	0.58 (0.75)	0.13
17	16	18,751	173	10,547	129	0.44 (0.50)	0.25
17	8	15,626	164	7,422	109	0.53 (0.62)	0.34
17	0	12,501	153	4,297	114	0.66 (0.81)	0.25
Average						0.48 (0.61)	0.23

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

Next, we tested our approach with the *sequence comparisons* problem [19]. In this 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 used the original tabled program from [19] and a transformed tabled program that forces all calls to use the `table_try_single` instruction. We experimented these two versions with sequences of length 500, 1000, 1500 and 2000. Table 2 shows the memory usage, in KBytes, for the local stack and the running time, in milliseconds, for YapTab without (column ***YapTab***) and with (column ***YapTab+Det***) the new support for deterministic tabled calls. A third column ***Ratio (1–b/a)*** shows the memory and running time ratio between both approaches.

Version	*Length*	*YapTab (a)*		*YapTab+Det (b)*		*Ratio (1–b/a)*	
		Memory	*Time*	*Memory*	*Time*	*Memory*	*Time*
Original	500	51,774	1,548	44,938	1,264	0.13	0.18
	1000	207,063	13,548	179,719	11,212	0.13	0.17
	1500	465,868	60,475	404,344	50,631	0.13	0.16
	2000	828,188	189,647	718,813	157,213	0.13	0.17
Transformed	500	45,915	1,172	39,051	848	0.15	0.28
	1000	183,625	10,024	156,227	8,460	0.15	0.16
	1500	413,133	45,874	351,528	36,106	0.15	0.21
	2000	734,438	140,068	624,953	113,011	0.15	0.19
Average						0.14	0.19

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

In general, for memory usage, the results in Table 2 are slightly different from the previous results obtained in Table 1. For both version of the *sequence comparisons* program, YapTab with support for deterministic tabled calls can

decrease, on average, memory usage by 14%. This reduction on memory saving, compared with the results on Table 1, happens mainly because of the existence of permanent variables in the body of the clauses in the *sequence comparisons* program. On the other hand, for the running times, the results in Table 2 confirm the previous results obtained in Table 1.

The results in Table 2 also show very similar memory and running time ratios for both versions of the *sequence comparisons* program. This suggests that we can take advantage of our approach by using the last matching clause optimization and not only when a program contains deterministic tabled predicates.

Finally, we tested our approach with a *path* program that computes the transitive closure of a NxN grid using a right recursive algorithm:

```
:- table path/2.

path(X,Z) :- edge(X,Z).
path(X,Z) :- edge(X,Y), path(Y,Z).
```

Regarding the `edge/2` facts, we used four grid configuration with 30x30, 40x40, 50x50 and 60x60 nodes. Table 3 shows the memory usage, in KBytes, for the local stack and the running time, in milliseconds, for YapTab without (column ***YapTab***) and with (column ***YapTab+Det***) the new support for deterministic tabled calls. Again, a third column ***Ratio (1–b/a)*** shows the memory and running time ratio between both approaches.

Grid	***YapTab (a)***		***YapTab+Det (b)***		***Ratio (1–b/a)***	
	Memory	***Time***	***Memory***	***Time***	***Memory***	***Time***
30x30	119	1,304	98	1,464	0.18	-0.12
40x40	211	4,400	175	4,024	0.17	0.09
50x50	330	11,208	273	10,996	0.17	0.02
60x60	476	28,509	393	28,213	0.17	0.01
Average					0.17	0.00

Table 3. Memory usage (in KBytes) and running times (in milliseconds) for YapTab without and with the new support for deterministic tabled calls

The *path* program confirms tendency to memory reduction, this case in 17%, on average. Running time gets sightly worse, thought comparison between both approaches remains in positive territory in three cases. Note however, that our approach was mainly designed to achieve a reduction on memory usage by paying a small cost on running time due to the extra code needed to deal with the new data structures and algorithms. Despite of this fact, on average, our approach showed a very good performance in all experiments.

6 Conclusions and Further Work

We have presented a proposal for the efficient evaluation of deterministic tabled calls with batched scheduling. A well-known aspect of tabling is the overhead in terms of memory usage compared with standard Prolog. This raised us the question of whether it was possible to minimize this overhead when evaluating deterministic tabled computations. Our preliminary results are quite promising, they suggest that, for deterministic tabled calls with batched scheduling, it is possible not only to reduce the memory usage overhead, but also the running time of the evaluation for certain class of 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 research projects STAMPA (PTDC/EIA/67738/2006) and JEDI (PTDC/ EIA/66924/2006) and by Fundação para a Ciência e Tecnologia.

References

1. Tamaki, H., Sato, T.: OLDT Resolution with Tabulation. In: International Conference on Logic Programming. Number 225 in LNCS, Springer-Verlag (1986) 84–98
2. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43**(1) (1996) 20–74
3. Rao, P., Sagonas, K., Swift, T., Warren, D.S., Freire, J.: XSB: A System for Efficiently Computing Well-Founded Semantics. In: International Conference on Logic Programming and Non-Monotonic Reasoning. Number 1265 in LNCS, Springer-Verlag (1997) 431–441
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. Zhou, N.F., Sato, T., Shen, Y.D.: Linear Tabling Strategies and Optimizations. Theory and Practice of Logic Programming **8**(1) (2008) 81–109
6. Guo, H.F., Gupta, G.: A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In: International Conference on Logic Programming. Number 2237 in LNCS, Springer-Verlag (2001) 181–196
7. Somogyi, Z., Sagonas, K.: Tabling in Mercury: Design and Implementation. In: International Symposium on Practical Aspects of Declarative Languages. Number 3819 in LNCS, Springer-Verlag (2006) 150–167
8. Chico, P., Carro, M., Hermenegildo, M.V., Silva, C., Rocha, R.: An Improved Continuation Call-Based Implementation of Tabling. In: International Symposium on Practical Aspects of Declarative Languages. Number 4902 in LNCS, Springer-Verlag (2008) 197–213

9. 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
10. Sagonas, K., Stuckey, P.: Just Enough Tabling. In: ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, ACM Press (2004) 78–89
11. Saha, D., Ramakrishnan, C.R.: Incremental Evaluation of Tabled Logic Programs. In: International Conference on Logic Programming. Number 3668 in LNCS, Springer-Verlag (2005) 235–249
12. 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
13. Rocha, R.: On Improving the Efficiency and Robustness of Table Storage Mechanisms for Tabled Evaluation. In: International Symposium on Practical Aspects of Declarative Languages. Number 4354 in LNCS, Springer-Verlag (2007) 155–169
14. Freire, J., Swift, T., Warren, D.S.: Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In: International Symposium on Programming Language Implementation and Logic Programming. Number 1140 in LNCS, Springer-Verlag (1996) 243–258
15. Santos Costa, V., Damas, L., Reis, R., Azevedo, R.: YAP User's Manual. Available from `http://www.dcc.fc.up.pt/~vsc/Yap`.
16. Aït-Kaci, H.: Warren's Abstract Machine – A Tutorial Reconstruction. The MIT Press (1991)
17. 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
18. 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
19. Warren, D.S.: Programming in Tabled Prolog. Technical report, Department of Computer Science, State University of New York (1999) Available from `http://www.cs.sunysb.edu/~warren/xsbbook/book.html`