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*

CICLOPS 2008, Udine, Italy, December 2008

Motivation

- 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**.

Motivation

- 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**.

Motivation

- 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 work, we propose a solution that reduces this memory overhead.

Motivation

- 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 work, we propose a solution that reduces this memory overhead.
- We will focus our discussion on a concrete implementation, the **YapTab system**, but our proposal can be generalized and applied to other tabling engines.

Compilation of Tabled Predicates in YapTab

- Tabled predicates defined by a single clause are compiled using the **table_try_single** WAM-like instruction.
- 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.

Compilation of Tabled Predicates in YapTab

- Tabled predicates defined by a single clause are compiled using the **table_try_single** WAM-like instruction.
- 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_single** and **table_try_me** instructions extend 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.

Compilation of Tabled Predicates in YapTab

```
:- table t/1.
t(X) :- ...
```

Compilation of Tabled Predicates in YapTab

```
:- table t/1.
t(X) :- ...

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

Compilation of Tabled Predicates in YapTab

```
:- table t/1.
t(X) :- ...

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

- As **t/1** is a deterministic tabled predicate, the **table_try_single** instruction will be executed for every call to this predicate.

Compilation of Tabled Predicates in YapTab

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

Compilation of Tabled Predicates in YapTab

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

% 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_trust_me
t3_4a: ‘WAM code for clause t(a3,b1,c2) :- ...’
```

Compilation of Tabled Predicates in YapTab

- **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.

Compilation of Tabled Predicates in YapTab

- **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.
- For example, the calls **t(X,Y,c3)** and **t(a3,X,Y)** are deterministic as they only match with a single `t/3` clause.

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

Compilation of Tabled Predicates in YapTab

- **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.
- For example, the calls **t(X,Y,c3)** and **t(a3,X,Y)** are deterministic as they only match with a single `t/3` clause.

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

- For this kind of deterministic calls, YapTab uses the demand-driven indexing mechanism of Yap to dynamically generate **table_try_single** instructions.

Demand-Driven Indexing in YapTab

- Yap implements **demand-driven indexing** (or **just-in-time indexing**) by building an indexing tree using similar building blocks to the WAM but it generates multi-argument indices based on the instantiation on the current goal.

Demand-Driven Indexing in YapTab

- Yap implements **demand-driven indexing** (or **just-in-time indexing**) by building an indexing tree using similar building blocks to the WAM but it generates multi-argument indices based on the instantiation on the current goal.

- Tabled calls matching more than a single clause are dynamically indexed using the **table_try**, **table_retry** and **table_trust** WAM-like instructions in a similar manner to the generic `try/retry/trust` WAM sequence.

```
% indexed code generated by YapTab for call t(X,b1,Y)
table_try   t3_1a
table_retry t3_3a
table_trust t3_4a

t3_1a: 'WAM code for clause t(a1,b1,c1) :- ...'
t3_2a: 'WAM code for clause t(a2,b2,c2) :- ...'
t3_3a: 'WAM code for clause t(a2,b1,c3) :- ...'
t3_4a: 'WAM code for clause t(a3,b1,c2) :- ...'
```

Demand-Driven Indexing in YapTab

- Tabled calls matching a single clause are dynamically indexed using the **table_try_single** instruction.

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

% indexed code generated by YapTab for call t(a3,X,Y)
table_try_single t3_4a

t3_1a: 'WAM code for clause t(a1,b1,c1) :- ...'
t3_2a: 'WAM code for clause t(a2,b2,c2) :- ...'
t3_3a: 'WAM code for clause t(a2,b1,c3) :- ...'
t3_4a: 'WAM code for clause t(a3,b1,c2) :- ...'
```

Demand-Driven Indexing in YapTab

- Tabled calls matching a single clause are dynamically indexed using the **table_try_single** instruction.

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

% indexed code generated by YapTab for call t(a3,X,Y)
table_try_single t3_4a

t3_1a: ‘WAM code for clause t(a1,b1,c1) :- ...’
t3_2a: ‘WAM code for clause t(a2,b2,c2) :- ...’
t3_3a: ‘WAM code for clause t(a2,b1,c3) :- ...’
t3_4a: ‘WAM code for clause t(a3,b1,c2) :- ...’
```

- 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.

Last Matching Clause

- When evaluating a tabled predicate, the **last matching clause** of a call is implemented by one of these instructions:
  - **table_try_single**: when we have a deterministic predicate or a deterministic call optimized by indexing code.
  - **table_trust_me**: when we have a generic call to the predicate (all the arguments of the call are unbound variables).
  - **table_trust**: when we have a more specific call optimized by indexing code (some of the arguments are at least partially instantiated).

Last Matching Clause

- When evaluating a tabled predicate, the **last matching clause** of a call is implemented by one of these instructions:
  - **table_try_single**: when we have a deterministic predicate or a deterministic call optimized by indexing code.
  - **table_trust_me**: when we have a generic call to the predicate (all the arguments of the call are unbound variables).
  - **table_trust**: when we have a more specific call optimized by indexing code (some of the arguments are at least partially instantiated).
- 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 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 and use the same approach to efficiently deal with deterministic tabled calls.

Implementation Details: Generator Nodes

- A generator node is a WAM choice point extended with some extra fields:
  - The top section contains the usual WAM fields needed to restore the computation on backtracking plus two extra fields.
  - The middle section contains the **argument registers** of the call.
  - The bottom section contains the **substitution variables**, i.e., the set of free variables which exist in the terms in the argument registers of the call.

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

Implementation Details: Generator Nodes

- Turning our attention to how generator nodes are handled we find that, with batched scheduling, the computation is never resumed in a deterministic generator node.
- This allow us to remove some fields:
  - The **cp_cp**, **cp_h**, **cp_env** and **cp_dep_fr** fields.
  - The **argument registers**.

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

Implementation Details: Generator Nodes

- The remaining fields are still required because:
  - **cp_b** is needed for failure continuation.
  - **cp_ap** and **cp_tr** are needed when backtracking to the node.
  - **cp_sg_fr** is needed by the new answer and completion operations.
  - The **substitution variables** are needed by the new answer operation.
- In order to avoid extra overheads, we have rearranged all choice points in such a way that the top three fields are now the same as the ones for a deterministic generator node.

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

Implementation Details: Generator Nodes

- 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:

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

- This memory reduction increases when the number of argument registers is bigger and the number of substitution variables is smaller.

Implementation Details: Tabling Operations

- The new representation for deterministic generator nodes required small changes to the tabled subgoal call, new answer and completion operations.

```
table_try_single(TABLED_CALL tc) {
  if (new_tabled_subgoal_call(tc)) {
    store_substitution_variables()
    if (evaluation_mode(tc) == batched_scheduling)       // new
      store_deterministic_generator_cp()
    else {                                   // local scheduling
      store_argument_registers()
      store_generic_generator_cp()
    }
    ...
  }
  ...
}
```

Implementation Details: Tabling Operations

- The new representation for deterministic generator nodes required small changes to the tabled subgoal call, new answer and completion operations.

```
table_trust_me(TABLED_CALL tc) {
  restore_generic_generator_cp()
  if (evaluation_mode(tc) == batched_scheduling &&       // new
      not_frozen(B)) {          // B is the current choice point
    subs_factor = B + sizeof(generic_generator_cp) + arity(tc)
    det_gcp = subs_factor - sizeof(deterministic_generator_cp)
    det_gcp->cp_sg_fr = B->cp_sg_fr
    det_gcp->cp_tr = B->cp_tr
    det_gcp->cp_ap = B->cp_ap
    det_gcp->cp_b  = B->cp_b
    B = det_gcp
  }
  ...
}
```

Implementation Details: Tabling Operations

- The new representation for deterministic generator nodes required small changes to the tabled subgoal call, new answer and completion operations.

```
new_answer(TABLED_CALL tc, ANSWER ans) {
  if (is_deterministic_generator_cp(B)) {                    // new
    det_gp = deterministic_generator_cp(B)
    sg_fr = det_gcp->cp_sg_fr
    subs_factor = det_gcp + sizeof(deterministic_generator_cp)
  } else {                        // generic generator choice point
    gcp = generic_generator_cp(B)
    sg_fr = gcp->cp_sg_fr
    subs_factor = gcp + sizeof(generic_generator_cp) + arity(tc)
  }
  ...
}
```

Implementation Details: Tabling Operations

- The new representation for deterministic generator nodes required small changes to the tabled subgoal call, new answer and completion operations.

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

Experimental Results

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

Memory usage in KBytes and running times in milliseconds for three deterministic tabled predicates (with arities 5, 11 and 17) that call themselves recursively 100,000 times with three different sets of free variables in the arguments.

Experimental Results

Version	Length	YapTab (a) Memory	YapTab (a) Time	YapTab+Det (b) Memory	YapTab+Det (b) Time	Ratio (1–b/a) Memory	Ratio (1–b/a) Time
Orig	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
Transf	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**

Memory usage in KBytes and running times in milliseconds for two versions of the *sequence comparisons* problem (with sequences of length 500, 1000, 1500 and 2000) using the original program and a transformed program that forces all calls to use the **table_try_single** instruction.

Experimental Results

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**

Memory usage in KBytes and running times in milliseconds for a program that computes the transitive closure of a NxN grid (with 30x30, 40x40, 50x50 and 60x60 nodes) using a right recursive algorithm.

Conclusions

- We have presented a proposal for the efficient evaluation of deterministic tabled calls with batched scheduling.
- Our preliminary results are quite promising as they suggest that, for certain class of applications, it is possible not only to reduce the memory usage overhead but also the running time of the evaluation.
- 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.