Bridge Transformation for Continuation Call-Based Tabled Execution

Pablo Chico de Guzman[1] Manuel Carro[1] Manuel V. Hermenegildo[1,2]
pchico@clip.dia.fi.upm.es {mcarro,herme}@fi.upm.es

[1] School of Computer Science, Univ. Politecnica de Madrid, Spain
[2] IMDEA Software, Spain

Abstract. The advantages of tabled evaluation regarding program termination and reduction of complexity are well known —as are the significant implementation, portability, and maintenance efforts that some proposals (especially those based on suspension) require. This implementation effort is reduced by program transformation-based continuation call techniques, at some efficiency cost. However, the traditional formulation of this proposal by Ramesh and Cheng limits the interleaving of tabled and non-tabled predicates and thus cannot be used as-is for arbitrary programs. In this paper we present a complete translation for the continuation call technique which, using the runtime support needed for the traditional proposal, solves these problems and makes it possible to execute arbitrary tabled programs. We present performance results which show that CCall offers a useful tradeoff that can be competitive with state-of-the-art implementations.
Keywords: Tabled logic programming, Continuation-call tabling, Implementation, Performance, Program transformation.

1 Introduction

Tabling [19, 4, 18] is a strategy for executing logic programs which uses *memoization* of already processed calls and their answers to improve several of the limitations of SLD resolution. It brings termination for bounded term-size programs and improves efficiency in programs which perform repeated computations and has been successfully applied to deductive databases [14], program analysis [20, 5], reasoning in the semantic Web [23], model checking [13], etc.

However, tabling also has certain drawbacks, including that predicates to be tabled have to be selected carefully[3] in order not to incur in undesired slowdowns and, specially relevant to our discussion, that its efficient implementation is generally complex. In *suspension-based tabling* the computation state of suspended tabled subgoals has to be preserved to avoid backtracking over them. This is done either by *freezing* the stacks, as in XSB [17], by copying to another area, as in CAT [8], or by using an intermediate solution as in CHAT [9]. *Linear tabling* maintains instead a single execution tree without requiring suspension and resumption of sub-computations. The computation of the (local) fixpoint is performed by making subgoals "loop" in their alternatives until no more solutions are found. This may make some computations to be repeated. Examples of

[3] XSB includes an `auto_table` declaration which triggers a conservative analysis to detect which predicates are to be tabled in order to ensure termination. However, more predicates than needed can be selected.

this method are the linear tabling of B-Prolog [22, 21] and the DRA scheme [10]. Suspension-based mechanisms achieve very good performance but, in general, require deeper changes to the underlying implementation. Linear mechanisms, on the other hand, can usually be implemented on top of existing sequential engines without major modifications.

The Continuation Call (`CCall`) approach to tabling [15, 16] tries to combine the best of both worlds: it is a reasonably efficient suspension-based mechanism which requires relatively simple additions to the Prolog implementation / compiler,[4] thus making maintenance and porting much easier. In [6] we proposed a number of optimizations to the `CCall` approach and showed that with such optimizations performance could be competitive with traditional implementations. However, this was only partially satisfactory since the `CCall` tabling approach is restricted to programs with a certain interleaving of tabled and non-tabled predicate calls (see Figure 3 and Section 3.1), and thus cannot execute general tabled programs.

In this paper we present an extension of the `CCall` translation which, using the same runtime support of the traditional proposal, overcomes the problems pointed out above. In Section 5 we present a complexity comparison of the proposed approach with CHAT. Finally, we present performance results from our implementation. These results show that our approach offers a useful tradeoff which can be competitive with state of the art implementations, while keeping implementation efforts relatively low.

2 The Continuation Call Technique

We sketch now how tabled evaluation [4, 17] works from a user point of view and we briefly describe the Continuation Call technique, on which we base our work.

2.1 Tabling Basics

We will use as example the program in Figure 1, whose purpose is to determine the reachability of nodes in a graph. If the graph contains cycles, there will be queries which will make the program loop forever under the standard SLD resolution strategy, regardless of the order of the clauses. Tabling changes the operational semantics for predicates marked with the *:- table* declaration, which forces the compiler and runtime system to distinguish the first occurrence of a tabled goal (the *generator*) and subsequent calls which are identical up to variable renaming (the *consumers*). The generator applies resolution using the program clauses to derive answers for the goal. Consumers *suspend* the current execution path (using implementation-dependent means) and start execution on a different branch corresponding to another clause of the predicate within which the execution was suspended. When such an alternative branch finally succeeds, the answer generated for the initial query (the generator) is inserted in a table associated with that generator. This makes it possible to reactivate consumers and to continue execution at the point where they were stopped. Thus, consumers do not use SLD resolution, but obtain instead the answers

[4] As an example, no modification to the underlying engine is needed.

from the table where they were previously inserted by the generator. Predicates not marked as tabled are executed according to SLD resolution, hopefully with minimal overhead due to the availability of tabling. This can be graphically seen as the ability to suspend execution in a part of the tree which cannot progress (because it enters a loop) and continue it somewhere else, where a solution for the looping goal can be produced.

2.2 `CCall` by Example

`CCall` implements tabling by a combination of program transformation and side effects in the form of insertions into and retrievals from a table which relates calls, answers, and the continuation code to be executed after consumers read answers from the table. We will now sketch how the mechanism works using the `path/2` example (Figure 1). The original code is transformed into the program in Figure 2 which is the one actually executed.

Roughly speaking, the transformation for tabling is as follows: an auxiliary predicate (`slg_path/2`) for `path/2` is introduced so that calls to `path/2` made from regular (SLD) Prolog execution do not need to be aware of the fact that `path/2` is being tabled. The primitive `slg/1` will make sure that its argument is executed to completion and will return, on backtracking, all the solutions found for the tabled predicate. To this end, `slg/1` checks if the call has already been executed. If so, all its answers are returned by backtracking. Otherwise, control is passed to a new predicate (`slg_path/2` in this case).[5] `slg_path/2` receives in its first argument the original call to `path/2` and in the second argument the identifier of its generator, which is used to relate operations on the table with this initial call. Each clause of `slg_path/2` is derived from a clause of the original `path/2` predicate by:

- Adding an `answer/2` primitive at the end of each clause of the original tabled predicate. `answer/2` is responsible for checking for redundant answers and inserting them in the table.
- Instrumenting calls to tabled predicates using the `slgcall/1` primitive. If this tabled call is a consumer, `path_cont/3`, along with its arguments, is recorded as (one of) the continuation(s) of its generator. If the tabled call is a generator, it is associated with a new call identifier and execution follows using the `slg_path/2` program clauses to derive new answers (as done by `slg/1`). Besides, `path_cont/3` will be recorded as a continuation of the generator identified by `Id` if the tabled call cannot be completed (there were dependencies on previous generators). The `path_cont/3` continuation will be called consuming found answers or erased upon completion of its generator.
- Encoding the remaining of the clause body of `path/2` after the recursive call by using `path_cont/3`. It is constructed similarly to `slg_path/2`, i.e., applying the same transformation as for the initial clauses and calling `slgcall/1`.

The second argument of `path_cont/3` is a list of bindings needed to recover the environment of the continuation call. Note that, in the program in Figure 1,

[5] The unique name has been created for simplicity by prepending `slg_` to the predicate name –any safe means of constructing a unique predicate symbol can be used.

```
:- table path/2.

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

Fig. 1. A sample program.

```
path(X, Y):- slg(path(X, Y)).
slg_path(path(X, Y), Id):-
     edge(X, Y),
      slgcall(path_cont(Id, [X], path(Y, Z))).
slg_path(path(X, Y), Id):-
     edge(X, Y),
     answer(Id, path(X, Y)).

path_cont(Id, [X], path(Y, Z)):-
     answer(Id, path(X, Z)).
```

Fig. 2. The program in Figure 1 after being transformed for tabled execution.

an answer to a query such as `?- path(X, Y)` may need to bind variable `X`. This variable does not appear in the recursive call to `path/2`, and hence it does not appear in the `path/2` term passed on to `slgcall/1` either. In order for the body of `path_cont/3` to insert in the table the answer corresponding to the initial query, variable `X` (and, in general, any other necessary variable) has to be passed down to `answer/2`. This is done with the list `[X]`, which is inserted in the table as well and completes the environment needed for the continuation `path_cont/3` to resume the previously suspended call.

A safe approximation of the variables which should appear in this list is the set of variables which appear in the clause before the tabled goal and which are used in the continuation, including the `answer/2` primitive. Variables appearing in the tabled call itself do not need to be included, as they will be passed along anyway. This list of bindings corresponds to the frame of the parent call if the `answer/2` primitive is added to the end of the body being translated.

Key Contribution of CCall: a new predicate name is created for all points where suspension can happen. Suspension is performed by saving this predicate name, a list of bindings, and a generator identifier. Resumption is performed by constructing a Prolog goal with the information saved on suspension plus the answer which raised the resumption. It is clear that this is significantly simpler to implement than other approaches as XSB or CHAT, where changes in the abstract machine have to be introduced. Consequently, porting and maintainability are simpler too, since `CCall` is independent of the compiler and how to create a Prolog term on the heap is the only one low level operation to implement.

3 Mixing Tabled and Non-Tabled Predicates

A continuation is the way `CCall` tabling preserves both the environment and the code of a consumer to be resumed. The list of bindings contains the same variables as the frame of the predicate where the `slgcall/1` primitive is executed, taking into account the `answer/2` primitive added at the end of the clause. However, the `CCall` approach to tabling, as originally proposed, has a problem when Prolog predicates appear between generators and consumers: the environments created by the non-tabled predicates are not taken into account, and they may

```
:- table t/1.

t(A):-
    p(B),
    A is B + 1.
t(0).

p(B):- t(B), B < 1.
```

Fig. 3. A program for which the original `CCall` transformation fails.

```
t(A):- slg(t(A)).
slg_t(t(A), Id):-
    p(B), A is B + 1,
    answer(Id, t(A)).

slg_t(t(0), Id):-
    answer(Id, t(0)).

p(B):- t(B), B < 1.
```

Fig. 4. The program in Figure 3 after being transformed for tabled execution.

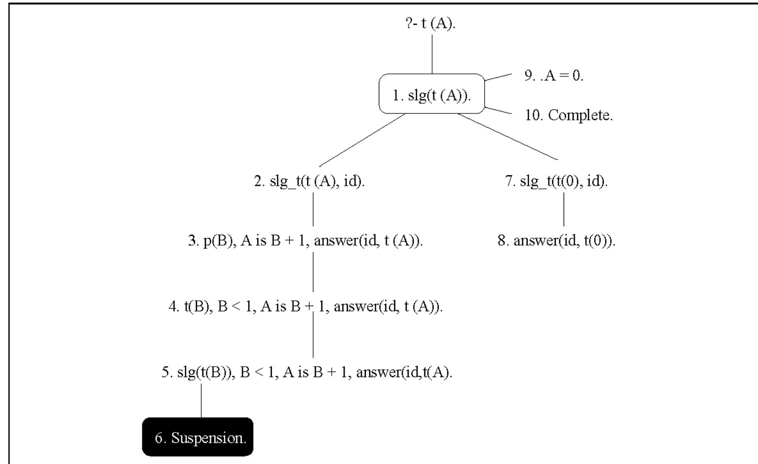

Fig. 5. Tabling execution of example of Figure 1.

be needed to correctly suspend and resume tabled predicates, as the example in the following section shows.

3.1 An Ill-Behaved Transformation

Figure 3 shows an example of a tabled program, where tabled and non-tabled execution (`t/1` and `p/1`) are mixed. The translation of the program is shown in Figure 4, taking into account the rules in Section 2.2.

The execution of the program with the query `t(A)` is shown in Figure 5. The execution is correct until `slg/1` is called again by `p/1`. At that point execution should suspend (and later resume), but `slg/1` does not have any associated continuation, and it does not have any pointer to the code to be executed on resumption (partially in `p/1` and partially in `slg_t/2`): `B < 1, A is B + 1, answer(Id,t(A))` is lost on backtracking and it is not reachable when resuming. Consequently, the second answer to the query, `t(1)`, is lost.

The call to `t(B)` made by `p(B)` could have been translated as if it were in the body of a tabled clause, but in that case the piece of code `A is B + 1` in the first clause of `t/1` would be lost anyway. This is an example of why all the

frames between a consumer and its nearest generator have to be saved when suspending, and it is not enough to save just the last one, as in the original `CCall` proposal [15], which does work, however, when all the calls to the tabled predicates appear in the body of the clause of a tabled predicate. In that case, it is enough to save the last frame with the associated continuation code. Note that all the suspension-based tabling approaches preserve the frames / environments from the consumer until the corresponding generator.

To solve this problem, we have extended the translation to take into account a new kind of predicates, named *bridges*. A bridge predicate is a non-tabled Prolog predicate whose clauses generate frames which have to be saved in the continuation of a consumer. In the example of Figure 3, `p/1` is a bridge predicate.

3.2 Marking Predicates as Bridges

Bridge predicates are all the non-tabled predicates which can appear in the execution tree of a query between a generator and each of its consumers, i.e., the predicates whose environments are in the local stack between the environment of the generator and the environment of each of its consumers. Note that tabled predicates do not need to be included as bridge predicates as their environment will be already saved by the translation. Additionally, only recursive calls which can lead to infinite loops under SLD resolution have to actually be taken into account, because these are the only ones which can suspend and later be resumed. Programs for which tabling merely speeds up already terminating computations are not subject to the problem outlined above, and therefore do not benefit from the improved translation shown herein.

Thus, in order to determine a minimal set of bridge predicates, B_{min}, we need to determine before the minimum set of tabled predicates, T_{min}, which ensures termination. When T_{min} is found, B_{min} is the set of non-tabled predicates which are "in the middle" of two calls to predicates belonging to T_{min}. Since looking for T_{min} is undecidable (because it implies detecting infinite failures), looking for B_{min} is also undecidable and a *safe approximation*, which may mark as bridge some predicates which do not need to be, is needed.

As we will see in Section 4.2, the only disadvantage of such an over-approximation is that some code will be duplicated (to accept a new argument for the case where a bridge predicate is called from a tabled execution), and that bridge predicates, having an extra argument, can be called when this is not needed. The algorithm we have implemented (Figure 6) only looks for tabled predicates which can recursively call themselves. For the examples used for performance evaluation in Section 6, using the safe approximation algorithm produces an average slowdown of only 3% with respect to a perfect characterization of bridge predicates.

4 A General Translation for Tabled Programs

In this section we present program transformation rules which take into account bridge predicates. This transformation assumes that the safe approximation algorithm for bridge predicates has already been run, and all the bridge predicates have been marked by adding a `:- bridge P/N` declaration in the program.

Make a graph G with an edge $(p1/n1,\ p2/n2) \Leftrightarrow p2/n2$ **is** called from $p1/n1$
$Bridges = \emptyset$
FOR each predicate T in TABLED PREDICATES
  $Forward$ = All predicates reached from T in G
  $Backward$ = All predicates from which T **is** reached in G
  $Bridges = Bridges \cup (Forward \cap Backward)$
$Bridges = Bridges -$ TABLED PREDICATES

Fig. 6. Safe approximation to look for bridge predicates.

As seen in Section 2.2, a continuation is the way to save an environment, because the predicate name is the same as the PC counter of the environment and the list of bindings is the same as the variables that a environment saves. Consequently, the goal of the new translation is to associate a continuation with each of the bridge predicates to save their associated environment. Continuations have a new argument (the next continuation to be executed) and they are be pushed onto the local stack in the same way as the environments.

4.1 Translation Rules

The rules for the original translation have three different goals: to maintain the interface with the rest of the code, to manage tabled calls which appear in the body of the clauses of a tabled predicate, and to insert answers at the end of the evaluation of each clause. The same points have to be addressed for bridge clauses, taking into account that a tabled or bridge call has to be translated if it appears in the body of a tabled predicate or a bridge predicate.

The rules for the new translation are shown in Figure 7, where we have used a sugared Prolog-like language. For example, a functional syntax is implicitly assumed where needed, and infix '$\circ$' is a general `append` function which joins either (linear) structures or, when applied to atoms, concatenates them. It may appear in an output head position with the expected semantics.

The `trans/2` predicate receives a clause to be translated and returns the list of clauses resulting from the translation. Its first clause ensures that predicates which are non-tabled and non-bridge are not transformed.[6] The second one is to generate the interface of table predicates with the rest of the code: if there is a tabled declaration, the interface is generated. The third clause translates clauses of tabled predicates, and the fourth one translates clauses of bridge predicates, where the original one is maintained in case it is called outside a tabled call (this is in order to preserve the interface with non-tabled code). They generate the new head of the clause, `Head_tr`, and the code which has to be appended at the end of the body, `End`, before calling `transBody/6` with these arguments. `End` can be the `answers/2` primitive for tabled clauses or `arg(3, Cont, Head), call(Cont)`, which updates the answer found for a bridge predicate in the next continuation to be resumed before calling it.

[6] The predicates `table/1` and `bridge/1` are dynamically generated by the compiler from the corresponding declaration. They check if their argument is a clause of a tabled or bridge predicate, or if their argument is a functor corresponding to a tabled or bridge predicate, respectively.

```
trans(C, C) :- \+ table(C), \+ bridge(C).
trans(( :- table P/N ), ( P(X1..Xn) :- slg(P(X1..Xn)) )).
trans(( Head :- Body ), LC) :-
    table(Head),
    Head_tr =.. ['slg_' ∘ Head, Head, Id],
    End = answer(Id, Head),
    transBody(Head_tr, Body, Id, [], End, LC).
trans(( Head :- Body ), ( Head :- Body ) ∘ LC) :-
    bridge(Head),
    Head_tr =.. [Head ∘ '_bridge', Head, Id, Cont],
    End = (arg(3, Cont, Head), call(Cont)),
    transBody(Head_tr, Body, Id, Cont, End, LC).

transBody([], [], _, _, [], []).
transBody(Head, Body, Id, ContPrev, End, ( Head :- Body_tr ) ∘ RestBody_tr) :-
    following(Body, Pref, Pred, Suff),
    getLBinds(Pref, Suff, LBinds),
    updateBody(Pred, End, Id, Pref, LBinds, ContPrev, Cont, Body_tr),
    transBody(Cont, Suff, Id, ContPrev, End, RestBody_tr).

following(Body, Pref, Pred, Suff) :-
    member(Body, Pred),
    (table(Pred); bridge(Pred)), !,
    Body = Pref ∘ Pred ∘ Suff.

updateBody([], End, _Id, Pref, _LBinds, _ContPrev, [], Pref ∘ End).
updateBody(Pred, _End, Id, Pref, LBinds, ContPrev, Cont, Pref ∘ slgcall(Cont)) :-
    table(Pred),
    getNameCont(NameCont),
    Cont = NameCont(Id, LBinds, Pred, ContPrev).
updateBody(Pred, _End, Id, Pref, LBinds, ContPrev, Cont, Pref ∘ Bridge_call) :-
    bridge(Pred),
    getNameCont(NameCont),
    Cont = NameCont(Id, LBinds, Pred, ContPrev),
    Bridge_call =.. [Pred ∘ '_bridge', Cont] .
```

Fig. 7. The Prolog code of the translation rules.

`transBody/6` generates, in its last argument, the translation of the body of a clause by taking care, in each iteration, of the code until the next tabled or bridge call, or until the end the clause, and appending the translation of the rest of the clause to this partial translation. In other words, it calls `updateBody/8` to translate tabled or bridge calls and continues translating the rest of the body.

The `following/4` splits a clause body in three parts: a prefix, until the first time a tabled or bridge call appears, the tabled or bridge call itself, and a suffix from this call until the end of the clause. `getLBinds/3` obtains the list of variables which have to be saved to recover the environment of the consumer, based on the ideas of Section 2.2.

The `updateBody/8` predicate completes the body prefix until the next tabled or bridge call. Its first six arguments are inputs, the seventh one is the head of

```
t(A) :- slg(t(A)).
slg_t(t(A), Id) :-
    p_bridge(Id, slg_t0(Id, [A], p(B), [])).

slg_t(t(0), Id) :- answer(Id, t(0)).

slg_t0(Id, [A], p(B), []) :-
    A is B + 1,
    answer(Id, t(A)).

p(B) :- t(B), B < 1.

p_bridge(Id, Cont) :-
    slgcall(p_bridge0(Id, [], t(B), Cont)).

p_bridge0(Id, [], t(B), Cont) :-
    B < 1,
    arg(3, Cont, p(B)),
    call(Cont).
```

Fig. 8. The program in Figure 3 after being transformed for tabled execution.

the continuation for the suffix of the body, and the last argument is the new translation for the prefix. The first clause takes care of the base case, when there are no calls to bridge or tabled predicates left, the second clause generates code for a call to a tabled predicate, and the last one does the same with a bridge predicate. That `getNameCont/1` generates a unique name for the continuation.

We will now use the example in Figure 3, adding a `:- bridge p/1` declaration, to exemplify how a translation would take place.

4.2 The Previous Example with the Correct Transformation

The translation of the first clause of `t/1` is done by the third clause of `trans/2`, which makes the head of the translated clause to be `slg_t(t(A), Id)` and states that the final call of that clause has to be `answer(Id, t(A))` —i.e., when the clause successfully finishes, it adds the answer to the table.

`transBody/6` takes care then of the rest of the body, which identifies which environment variables (`A`, in this case) have to be saved and matches `Pref`, `Pred`, and `Suff` with the goals before the call to the tabled predicate (none — and empty conjunction), the call to the tabled predicate (`p(B)`), and the goals after this call (`A is B + 1`). The third clause of `updateBody/8` generates the body of `Head_tr`, to give the first clause of `slg_t/2`. A continuation is generated for the rest of the body; the code of the continuation is a predicate whose head is `slg_t0/3` and its body is generated by the first clause of `updateBody/8`.

The translation of the second clause of `t/1` is simpler, as it only has to add `answer(Id, t(0))` at the end of the body of the new predicate.

The clause for `p/1` is kept to maintain its interface when it is not called from inside a another tabled execution. The translation for the clause of `p/1` is made by the fourth clause of `trans/2` where `Head_tr` is unified with `p_bridge(p(B), Id, Cont)`. In the expression for `End`, the predicate `arg/3` is used to unify the answers found for the bridge predicate with the corresponding argument of the continuation. `transBody/6` finds an empty list of environment variables and unifies `Pref`, `Pred` and `Suff` with `[]`, `t(B)` and `B < 1`, respectively. The second clause of `updateBody/8` generates the body for the new predicate `p_bridge/3`. A continuation is generated to execute the rest of the body, whose head is `p_bridge0/3` and whose body is generated by the first clause of `updateBody/8`. As we can see, bridge predicates are appending continuations and `End` variable is calling them sequentially.

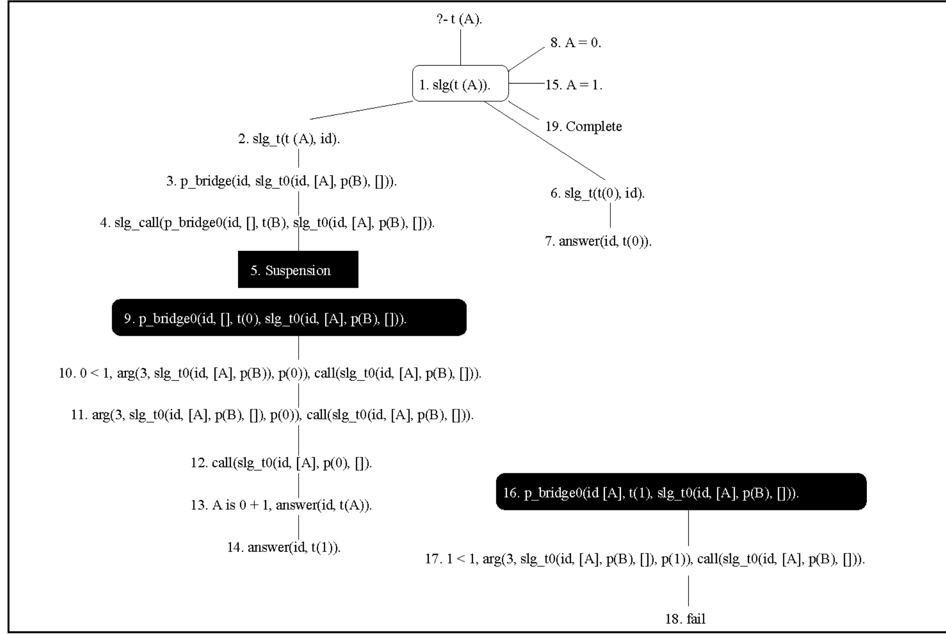

Fig. 9. New `CCall` tabling execution.

4.3 Execution of the Transformed Program

The execution tree of the transformed program is shown in Figure 9. It is similar to that in Figure 5, but a continuation `slg_t0(id, [A], p(B), [])` is passed to the transformed clause of `p/1`. This continuation contains the code to be executed after the execution of `p(B)` and the list `[A]` needed to recover its environment. Consequently, there are two continuations associated with the suspension: one continuation to execute the rest of the code of `p(B)` and another one to execute the rest of the code of `t(A)`.

After the first answer is found, this double continuation is resumed. It is executed as a normal Prolog and the second answer, `t(1)`, is found. The `arg/3` predicate is used to unify the answer of `p(B)` we have just found with the literal stored in the continuation.

5 Θ(CHAT) is not comparable with Θ(CCall)

In this section we present a comparative analysis of the complexity of `CCall` and CHAT, which is an efficient implementation of tabling with a comparatively simple machinery. Since it is known that Θ(CHAT) is Θ(`SLG-WAM`) [7], the comparative analysis applies to the `SLG-WAM` as well.

The complexity analysis focuses on the operations of suspension and resumption. The state of the consumer has to be protected when suspending to reinstall it when resuming. `CCall` achieves that by copying the continuation associated with the consumer in a special memory area to be protected on backtracking. In the original implementation [15] this continuation is copied from the heap to a

separate table (when suspending) and back (when resuming). As proposed in [6], continuations can be saved in a special memory area with the same data format as the heap. This makes it possible to use WAM instructions and additional machinery on them and, when resuming, they can be used as normal Prolog data and code, without being recopied.

On the other hand, CHAT freezes the heap and the frame stack when resuming. The heap and frame stack are frozen by traversing the choice point stack. For all the choice points between the consumer choice point and its generator, the pointer to the end of the heap and frame stack are changed to the values of the consumer choice point values. By doing that, heap and frame stack are protected on backtracking. However, the consumer choice point has to be copied to a special memory area as well as the segment trail (with its associated values) between the consumer and the generator, to reinstall the values of the bound variables at the time of suspension which backtracking will unbind. In consequence, when resuming the trail values have to be reinstalled as well as the consumer choice point.

Each consumer is suspended only once, and it can be resumed several times. The rest of the operations, i.e., checking if a tabled call is a generator or a consumer, are not analyzed, because they are common to both systems. In addition, we will ignore the cost of working at the Prolog level, since this is an orthogonal issue: `CCall` primitives could be compiled to WAM instructions and working at Prolog level does not increase the system complexity.

Θ(`CCall`): when suspending, `CCall` has to copy all the environments until the last generator and the structures in the heap which hang from them. If we name `E` the size of all the environments and `H` the size of the structures in the heap, the time consumption when suspending is: Θ`(E + H)`.

When resuming, `CCall` just has to perform pattern matching of the continuation against its clause. The time taken by the pattern matching depends on the size of the list of bindings, which is known to be Θ`(E)`. Since each consumer can be resumed `N` times, the time consumption of resuming consumers is $\Theta(N \times E)$.

Θ(CHAT): when suspending, CHAT has to traverse the frame and choicepoint stacks, but with the improvements presented in [7], the time this takes can be neglected because a choice point is only traversed once for all the consumers. The trail and the last choice point have to be copied. If we call `T` the size of the trail and `C` the size of the choice point, which is bound by a constant for a given program, the time consumption when suspending is: Θ`(T)`.

When resuming, CHAT has to reinstall the values of the frame and the choice point. Since each consumer can be resumed `N` times, the time consumption of resuming is $\Theta(N \times T)$.

Analyzing the worst cases of both systems: we can conclude $E + H \geq T$, because each variable can only be once in the trail, and then `CCall` is worse than CHAT when suspending. On the other hand, in case that $E < T$, `CCall` is better than CHAT when resuming. Consequently, for a plausible general case, the more resumptions there are, the better `CCall` behaves in comparison with CHAT, and conversely. In any case, the worst and best cases for each implementation are

different, which makes them difficult to compare. For example, if there is a very large structure pointed to from the environments, and none of its elements are pointed to from the trail, `CCall` is slower than CHAT, since it has to copy all the structure in a different memory area when suspending and CHAT does nothing both when suspending and when resuming.

On the other hand, if all the elements of the structure are pointed to from the trail, `CCall` has to copy all the structure on suspension in a different memory area to protect it on backtracking, but it is ready to be resumed without any other operation (just a unification with the pointer to the structure). CHAT has to copy all the structure on suspension too, because all the structure is in the trail. In addition, each time the consumer is resumed, all the elements of the structure have to be reinstalled using the trail, and CHAT has to perform more operations than `CCall`, and then, the more resumptions there are, the worse CHAT would be in comparison with `CCall`. Anyway, as the trail is usually much smaller than the heap, in general cases, CHAT will have an advantage over `CCall`.

6 Performance Evaluation

We have implemented the proposed technique as an extension of the Ciao system [1]. Tabled evaluation is provided to the user as a loadable *package* that implements the new directives and user-level predicates, performs the program transformations, and links in the low-level support for tabling. We have implemented `CCall` tabling with the efficiency improvements presented in [6] and the new translation for general programs explained in this paper.

Table 1 aims at determining how the proposed implementation of tabling compares with state-of-the-art systems —namely, the latest available versions of XSB, YapTab, and B-Prolog, at the time of writing, using the typical benchmarks which appear in other performance evaluations of tabling approaches.[7] In this table we provide, for several benchmarks, the raw time (in milliseconds) taken to execute them using tabling. Measurements have been made with Ciao-1.13, using the standard, unoptimized bytecode-based compilation, and with the `CCall` extensions loaded, as well as in XSB 3.0.1, YapTab 5.1.1, and B-Prolog 7.0. Note that we did not compare with CHAT, which was available as a configuration option in the XSB system and which was removed in recent XSB versions. CHAT can be expected to be at least as fast (if not slightly faster) than XSB.

All the executions were performed using local scheduling and disabling garbage collection; in the end this did not impact execution times very much. We used `gcc 4.1.1` to compile all the systems, and we executed them on a machine with Fedora Core Linux, kernel 2.6.9, and an Intel Xeon DESCHUTES processor.

The first benchmark is `path`, the same as Figure 1, which has been executed with a chain-shaped graph. Since this is a tabling-intensive program with no consumers in its execution, the difference with other systems is mainly due to having large parts of the execution done at Prolog level. The following five benchmarks, until `atr2`, are also tabling intensive. As their associated environments are very

[7] This is in contrast to [6] where, due to the limitations of the `CCall` approach the benchmarks presented did not need the use of bridge predicates.

Program	CCall	XSB	YapTab	BProlog
path	517.92	231.4	151.12	206.26
tcl	96.93	59.91	39.16	51.60
tcr	315.44	106.91	90.13	96.21
tcn	485.77	123.21	85.87	117.70
sgm	3151.8	1733.1	1110.1	1474.0
atr2	689.86	602.03	262.44	320.07
pg	15.240	13.435	8.5482	36.448
kalah	23.152	19.187	13.156	28.333
gabriel	23.500	19.633	12.384	40.753
disj	18.095	15.762	9.2131	29.095
cs_o	34.176	27.644	18.169	85.719
cs_r	66.699	55.087	34.873	170.25
peep	68.757	58.161	37.124	150.14

Table 1. Comparing Ciao+CCall with XSB, YapTab, and B-Prolog.

small, `CCall` is far from its worst case (see Section 5), and the difference with other systems is similar to that in `path` and for a similar reason. The worst case in this set is `tcn` because there are two calls to `slgcall/1` per generator, and the overhead of working at the Prolog level is duplicated.

B-Prolog, which uses a linear tabling approach, suffers if costly predicates have to be recomputed: this is what happens in benchmarks from `pg` until `peep`, where tabled and non-tabled execution is mixed. This is a well-known disadvantage of linear tabling techniques which does not affect suspension-based approaches. It has to be noted, however, that latest versions of B-Prolog implement an optimized variant of its original linear tabling mechanism [21] which tries to avoid reevaluation of looping subgoals.

In order to compare our implementation with XSB and YapTab, we must take into account that the speeds of XSB, and YapTab[8] are different, at least in those cases where the program execution is large enough to be really significant (between 1.8 and 2 times slower in the case of XSB and 1.5 times faster in the case of YapTab).

In non-trivial benchmarks, from `pg` until `peep`, which at least in principle should reflect more accurately what one might expect in larger applications using tabling, execution times are in the end very competitive when comparing with XSB or YapTab. This is probably due to the fact that the raw speed of the basic engine in Ciao is higher than in XSB and closer to YapTab, rather than to factors related to tabling execution, but it also implies that the overhead of the approach to tabling used is reasonable after the proposed optimizations in [6]. In this context it should be noted that in these experiments we have used the baseline, bytecode-based compilation and abstract machine. Turning on global analysis and using optimizing compilers and abstract machines [11, 3, 12] can further improve the speed of the SLD part of the computation.

[8] Note that we are comparing the tabled-enabled version of Yap, which is somewhat slower than the regular Yap.

7 Conclusions

We have presented an extension of the continuation call technique which does not have the limitations of the original continuation call approach regarding the interleaving of tabled and non-tabled predicates. This approach has the advantage of being easier to implement and maintain than other techniques which require non-trivial modifications to low-level machinery. Although there is an overhead imposed by executing at Prolog level, we expect the speed of the source (Prolog) language to gradually improve by using global analysis, optimizing compilers, and better abstract machines. Accordingly, we expect the performance of `CCall` to improve in the future and thus gradually gain ground in the comparisons.

Although a non optimal tabled execution is obviously a disadvantage, it is worth noting that, since our implementation introduces only minimal changes in the WAM and none in the associated Prolog compiler, the speed at which regular Prolog is executed remains unchanged. In addition to this, the modular design of our approach gives better chances of making it easier to port to other systems. In our case, executables which do not need tabling have very little tabling-related code, as the data structures (for tries, etc.) are handled by dynamic libraries loaded on demand, and only stubs are needed in the regular engine. The program transformation is taken care of by a plugin for the Ciao compiler [2] (a "package," in Ciao's terms) which is loaded and active only at compile time, and which does not remain in the final executable.

References

1. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM), 2006. Available at `http://www.ciaohome.org`.
2. D. Cabeza and M. Hermenegildo. The Ciao Modular, Standalone Compiler and Its Generic Program Processing Library. In *Special Issue on Parallelism and Implementation of (C)LP Systems*, volume 30(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.
3. M. Carro, J. Morales, H.L. Muller, G. Puebla, and M. Hermenegildo. High-Level Languages for Small Devices: A Case Study. In Krisztian Flautner and Taewhan Kim, editors, *Compilers, Architecture, and Synthesis for Embedded Systems*, pages 271–281. ACM Press / Sheridan, October 2006.
4. Weidong Chen and David S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.
5. S. Dawson, C.R. Ramakrishnan, and D.S. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems – A Case Study. In *Proceedings of PLDI'96*, pages 117–126, New York, USA, 1996. ACM Press.
6. P. Chico de Guzmán, M. Carro, M. Hermenegildo, Claudio Silva, and Ricardo Rocha. An Improved Continuation Call-Based Implementation of Tabling. In D.S. Warren and P. Hudak, editors, *10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, volume 4902 of *LNCS*, pages 198–213. Springer-Verlag, January 2008.
7. Bart Demoen and K. Sagonas. CHAT is θ(SLG-WAM). In D. Mc. Allester H. Ganzinger and A. Voronkov, editors, *International Conference on Logic for Programming and Automated Reasoning*, volume 1705 of *Lectures Notes in Computer Science*, pages 337–357. Springer, September 1999.

8. Bart Demoen and Konstantinos Sagonas. CAT: The Copying Approach to Tabling. In *Programming Language Implementation and Logic Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 21–35. Springer-Verlag, 1998.
9. Bart Demoen and Konstantinos F. Sagonas. Chat: The copy-hybrid approach to tabling. In *Practical Applications of Declarative Languages*, pages 106–121, 1999.
10. Hai-Feng Guo and Gopal Gupta. A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In *International Conference on Logic Programming*, pages 181–196, 2001.
11. J. Morales, M. Carro, and M. Hermenegildo. Improving the Compilation of Prolog to C Using Moded Types and Determinism Information. In *Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages*, number 3057 in LNCS, pages 86–103, Heidelberg, Germany, June 2004. Springer-Verlag.
12. J. Morales, M. Carro, and M. Hermenegildo. Comparing Tag Scheme Variations Using an Abstract Machine Generator. In *10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 32–43. ACM Press, July 2008.
13. Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, and D.S. Warren. Efficient Model Checking Using Tabled Resolution. In *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 143–154. Springer Verlag, 1997.
14. Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1993.
15. R. Ramesh and Weidong Chen. A Portable Method for Integrating SLG Resolution into Prolog Systems. In Maurice Bruynooghe, editor, *International Symposium on Logic Programming*, pages 618–632. MIT Press, 1994.
16. R. Rocha, C. Silva, and R. Lopes. On Applying Program Transformation to Implement Suspension-Based Tabling in Prolog. In V. Dahl and I. Niemelä, editors, *23rd International Conference on Logic Programming*, number 4670 in LNCS, pages 444–445, Porto, Portugal, September 2007. Springer-Verlag.
17. K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, May 1998.
18. H. Tamaki and M. Sato. OLD resolution with tabulation. In *Third International Conference on Logic Programming*, pages 84–98, London, 1986. Lecture Notes in Computer Science, Springer-Verlag.
19. D.S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, 1992.
20. R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.
21. Neng-Fa Zhou, T. Sato, and Yi-Dong Shen. Linear Tabling Strategies and Optimizations. *Theory and Practice of Logic Programming*, 8(1):81–109, 2008.
22. Neng-Fa Zhou, Yi-Dong Shen, Li-Yan Yuan, and Jia-Huai You. Implementation of a linear tabling mechanism. *Journal of Functional and Logic Programming*, 2001(10), October 2001.
23. Youyong Zou, Tim Finin, and Harry Chen. F-OWL: An Inference Engine for Semantic Web. In *Formal Approaches to Agent-Based Systems*, volume 3228 of *Lecture Notes in Computer Science*, pages 238–248. Springer Verlag, January 2005.