An Improved Continuation Call-Based Implementation of Tabling

Pablo Chico de Guzmán[1,2] Manuel Carro[1] Manuel V. Hermenegildo[1,2]
Cláudio Silva[3] Ricardo Rocha[3]

`pchico@clip.dia.fi.upm.es` `{mcarro,herme}@fi.upm.es`
`herme@cs.unm.edu` `ccaldas@dcc.online.pt` `ricroc@dcc.fc.up.pt`

[1] School of Computer Science, Univ. Politécnica de Madrid, Spain
[2] Depts. of Comp. Science and Electr. and Comp. Eng., Univ. of New Mexico, USA
[3] DCC-FC & LIACC, University of Porto, Portugal,

Abstract. Tabled evaluation has been proved an effective method to improve several aspects of goal-oriented query evaluation, including termination and complexity. Several "native" implementations of tabled evaluation have been developed which offer good performance, but many of them require significant changes to the underlying Prolog implementation, including the compiler and the abstract machine. Approaches based on program transformation, which tend to minimize changes to both the Prolog compiler and the abstract machine, have also been proposed, but they often result in lower efficiency. We explore some techniques aimed at combining the best of these worlds, i.e., developing an extensible implementation which requires minimal modifications to the compiler and the abstract machine, and with reasonably good performance. Our preliminary experiments indicate promising results.
Keywords: Tabled logic programming, Implementation, Performance, Program transformation.

1 Introduction

Tabling [20, 4, 19] is a resolution strategy which tries to *memoize* previous calls and their answers in order to improve several well-known shortcomings found in SLD resolution. It brings some of the advantages of bottom-up evaluation to the top-down, goal-oriented evaluation strategy. In particular, evaluating logic programs under a tabling scheme may achieve termination in cases where SLD resolution does not (because of infinite loops —for example, the tabled evaluation of bounded term-size programs is guaranteed to always terminate). Also, programs which perform repeated computations can be greatly sped up. Program declarativeness is also improved since the order of clauses and goals within a clause is less relevant, if at all. Tabled evaluation has been successfully applied in many fields, such as deductive databases [13], program analysis [21, 5], reasoning in the semantic Web [24], model checking [11], and others.

In all cases the advantages of tabled evaluation stem from checking whether calls to *tabled predicates*, i.e., predicates which have been marked to be evaluated using tabling, have been made before. Repeated calls to tabled predicates consume answers from a table, they suspend when all stored answers have been consumed, and they fail when no more answers can be generated. However, the advantages are not without drawbacks. The main problem is the complexity of some (efficient) implementations of tabled resolution, and a secondary issue is the difficulty in selecting which predicates to table in order not to incur in undesired slow-downs.

Two main categories of tabling mechanisms can be distinguished: *suspension-based* and *linear* tabling mechanisms. In suspension-based mechanisms 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 [16], by copying to another area, as in CAT [7], or by using an intermediate solution as in CHAT [8]. Linear tabling mechanisms maintain a single execution tree where tabled subgoals always extend the current computation without requiring suspension and resumption of sub-computations. The computation of the (local) fixpoint is performed by repeatedly looping subgoals until no more solutions can be found. Examples of this method are the linear tabling of B-Prolog [23, 22] and the DRA scheme [9].

Suspension-based mechanism have achieved very good performance results but, in general, deep changes to the underlying Prolog implementation are required. Linear mechanisms, on the other hand, can usually be implemented on top of existing sequential engines without major modifications but their efficiency is affected by subgoal recomputation. One of our theses is that it should be possible to find a combination of the best of both worlds: a suspension-based mechanism that is reasonably efficient and does not require complex modifications to the compiler or underlying Prolog implementation, thus contributing to its maintainability an making it easier to port it to other Prolog systems. Also, we would like to avoid introducing any overhead that would reduce the execution speed for SLD execution.

Our starting point is the *Continuation Call Mechanism* presented by Ramesh and Chen in [14]. This approach has the advantage that it indeed does not need deep changes to the underlying Prolog machinery. On the other hand it has shown up to now worse efficiency than the more "native" suspension-based implementations. Our aim is to analyze the bottlenecks of this approach, explore variations thereof, and propose solutions in order to improve its efficiency while keeping tabling-related changes clearly separated from the basic WAM implementation. While the approach may not necessarily be significantly simpler than other (native) approaches, we will argue that it does allow a more modular design which reduces and isolates in separate modules the changes made to the underlying WAM. This hopefully will make it easier to maintain the implementation of both tabling and the WAM itself, as well as adapting the tabling scheme and code to other Prolog systems.

In more concrete terms, and in the spirit of [14], the implementation we will propose tries to be non intrusive and change only minimally the initial WAM, moving the low-level tabling data structures either to the Prolog level or to external modules. Other systems, like Mercury [18], also implement tabling using external modules and program transformation, so as not to change the compiler and runtime system. Despite these similarities, the big differences in the base language make the implementation technically very different also.

2 Tabling Basics

We now sketch how tabled evaluation works from a user point of view (more details can be found in [4, 16]) and briefly describe the continuation call mechanism implementation technique proposed in [14], on which we base our work.

2.1 Tabling by Example

We use as running example the program in Figure 1, taken from [14], whose purpose is to determine reachability of nodes in a graph We ignore for now the `:- tabled path/2` declaration (which instructs the compiler to use tabled execution for the designated predicate), and assume that SLD resolution is to be used. Then, a query such as `?- path(a, N).` may never terminate if, for example, `edge/2` represents a cyclic graph.

Adding the `:- tabled` declaration 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. When such an alternative branch finally succeeds, the answer generated for the initial query is inserted in a table associated with the original goal. This makes it possible to reactivate suspended calls and to continue execution at the point where they were stopped. Thus, consumers do not use SLD resolution, but obtain instead the answers from the table where they were inserted previously by the producer. Predicates not marked as tabled are executed following SLD resolution, hopefully with (minimal or no) overhead due to the availability of tabling in the system.

2.2 The Continuation Call Technique

The continuation call technique [14] 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: a bridge predicate for `path/2` is introduced so that calls to `path/2` made from regular Prolog execution do not need to be aware of the fact that `path/2` is being tabled. The call to the `slg/1` primitive will ensure 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` starts by inserting the call in the answer table and

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

generating an identifier for it. Control is then passed to a new, distinct predicate: in this case, `slg_path/2`.[4] `slg_path/2` receives in the first argument the original call to `path/2` and in the second one the identifier generated for the parent call, 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 resulting from a transformation and which is not a *bridge* to call a continuation predicate. `answer/2` is responsible for checking for redundant answers and executing whatever continuations (see the following item) there may be associated with that call identified by its first argument.
– Instrumenting recursive calls to `path/2` using the `slgcall/4` primitive. If the term passed as an argument (i.e., `path(X, Y)`) is already in the table, `slgcall/4` creates a new consumer which consumes answers from the table. Otherwise, the term is inserted in the table with a new call identifier and execution follows using the `slg_path/2` program clauses to derive new answers. In the first case, `path_cont/3` is recorded as (one of) the continuation(s) of `path(X, Y)` and `slgcall/4` fails. In the second case `path_cont/3` is only recorded as a continuation of `path(X, Y)` if the tabled call cannot be completed. The `path_cont/3` continuation will be called from `answer/2` after inserting a new answer or erased upon completion of `path(X, Y)`.
– The body of `path_cont/3` encodes what remains of the clause body of `path/2` after the recursive call. It is constructed in a similar way to `slg_path/2`, i.e., applying the same transformation as for the initial clauses and calling `slgcall/4` and `answer/2` at appropriate times.

The second argument of `slgcall/4` and `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, 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

[4] The distinct 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.

```
answer( callid  Id , term Answer) {
  insert  Answer in answer table
  If  (Answer $\notin$ answer table)
    for each continuation  call  C
          of tabled  call  Id {
       call (C) consuming Answer;
    }
  return FALSE;
}
```

Fig. 3. Pseudo-code for `answer/2`.

```
slgcall ( callid  Parent, term Bindings,
          term Call , term CCall) {
  Id  = insert  Call  into  answer table ;
  if  (Id . state  == READY) {
     Id . state  = EVALUATING;
     call  the transformed  clause  of  Call ;
     check  for  completion;
  }
  consume answers for  Id;
  if  (Id . state  != COMPLETE)
       add a new continuation
         call (CCall,  Bindings)  to  Id;
  return FALSE;
}
```

Fig. 4. Pseudo-code for `slgcall/4`.

hence it does not appear in the `path/2` term passed on to `slgcall/4` 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 if there is one in the continuation —this is the case in our example. Variables appearing in the tabled call itself do not need to be included, as they will be passed along anyway.

Recovering a previous execution environment is an important operation in tabled execution. Other approaches to this end are the use of forward trail and freeze registers of SLG-WAM [16], which involves using lower-level mechanisms. The continuation call approach, which performs several tabling operations at the Prolog level through program transformation and can *a priori* be expected to be somewhat slower, has, however, the nice property that the implementation does not need to change the underlying WAM machinery, which helps its adaptation it to different Prolog systems. On the other hand, the table management is usually, and for efficiency reasons, written using some lower-level language and accessed using a suitable interface.

The pseudo-code for `answer/2` and `slgcall/4` is shown in Figures 3 and 4, respectively. The pseudo-code for `slg/1` is similar to that of `slgcall/4` but, instead of consuming answers, they are returned on backtracking and it finally fails when all the stored answers have been exhausted. The program transformation and primitives try to complete subgoals as soon as possible, failing whenever new answers are found. Thus, they implement the so-called *local scheduling* [16].

Checking for completion: The completion detection algorithm (see [17] for more details) is similar to that in the SLG-WAM. We just provide a sketch here. Completion is checked for in the execution of the `slgcall/4` primitive after exhausting all alternatives for the subgoal call at hand and resuming all of its consumers. To do that, we use two auxiliary fields in the table entry corresponding to every subgoal, `SgFr_dfn` and `SgFr_dep`, to quickly determine whether such a subgoal is a leader node. The `SgFr_dfn` field reflects the order in which the subgoals being evaluated were called. New subgoal frames are numbered incrementally as they are created, adding one to the `SgFr_dfn` of the previous (youngest) subgoal, whose frame is always pointed to by the global variable `SF_TOP`. `SgFr_dep` holds the number of the older call on which it depends, which is initialized with the same number as `SgFr_dfn`, meaning that initially no dependencies exist. If P_1, a tabled subgoal already inserted in the table, is called from the execution of another tabled subgoal, P_2, the `SgFr_dep` field of the table entry of P_2 is updated with the value of `SgFr_dep` field of P_1, meaning P_2 depends on P_1. When checking for completion, and using this information from the table entries, a subgoal can quickly determine whether it is a leader node. If `SgFr_dfn = SgFr_dep`, then we know that during its evaluation no dependencies to older subgoals have appeared and thus the *Strongly Connected Component* (SCC) including the subgoals starting from the table entry referred to by `SF_TOP` up to the current subgoal can be completed. On the other hand, if `SgFr_dep < SgFr_dfn`, we cannot perform completion. Instead, we must propagate the current dependency to C, the subgoal call that continues the evaluation. To do that, the `SgFr_dep` field is copied to `SgFr_dep` field of C, and completion can be performed only when the computation reaches the subgoal that does not depend on older subgoals.

Issues in the Continuation Call Mechanism: We have identified two performance-related issues when implementing the technique sketched in the previous section. The first one is rather general and related to the heavy use of the interface between C and Prolog (in both directions) that the implementation makes, which adds an overhead which cannot be neglected.

The second one is the repeated copying of continuation calls. Continuation calls (which are, in the end, Prolog predicates with an arbitrarily long list of variables as an argument) are completely copied from Prolog memory to the table for every consumer found. Storing a pointer to these structures in memory is not enough, since `slg/1` and `slgcall/4` fail immediately after associating a continuation call with a tabled call in order to force the program to search for more solutions and complete the tabled call. Therefore, the data structures created during forward execution may be removed on backtracking and not be available when needed. Reconstructing continuations as Prolog terms from the data stored in the table when they are resumed to consume previously stored answers is necessary. This can also clearly have a negative impact on performance.

Finally, an issue found with the implementation we started with [15] (which is a version of [14] in Yap Prolog) is that it did not allow backtracking over Prolog predicates called from C, which makes it difficult to implement other scheduling strategies. Since this shortcoming may appear also in other C interfaces, it is a clear candidate for improvement.

3 An Improvement over the Continuation Call Technique

We now propose some improvements to the different limitations of the original design and implementation that we discussed in Section 2.2. In order to measure execution times, we are taking the implementation described in [15] to be close enough to that described in [14] in order to be used as a basis for our developments. It is also an implementation of high quality whose basic components (e.g., tables based on tries, following [12]) are similar to those in use in current tabling systems. This implementation was ported to Ciao, where the rest of the development was performed. In what follows this initial port to Ciao will be termed the "baseline implementation."

3.1 Using a Lower-Level Interface

Calls from C to Prolog were initially performed using a relatively high-level interface similar to those commonly found in current state of the art logic programming systems: operations to create and traverse Prolog terms appear to the programmer as regular C functions, and details of the internal data representation are hidden to the programmer. This interface imposed a noticeable overhead in our implementation, as the calls to C functions had to allocate environments, pass arguments, set up Prolog environments to call Prolog from C, etc.

In order to make our implementation as fast as possible, a possibility is to integrate all the C code into the WAM and try to avoid altogether costly format conversions, etc. However, as mentioned before, we preferred to make as few changes as possible in the WAM. Therefore we chose to use directly lower-level operations and take advantage of facilities (e.g., macros) initially designed to be internally used by the WAM. While this in principle makes porting more involved, the fact is that the facilities provided in C interfaces for Prolog and the internal WAM operations are typically quite related and similar, since they all provide an interface to an underlying architecture and data representation which is common to many Prolog implementations.

Additionally, the code which constructs Prolog terms and performs calls from C is the same regardless of the program being executed and its complexity is certainly manageable. Therefore, we decided to skip the programmer interface and call directly macros available in the engine implementation. That was not a difficult task and it sped the execution up by a factor of 2.5 on average.

3.2 Calling Prolog from C

A relevant issue in the continuation call technique (and, possibly, in other cases) is the use of a C-to-Prolog interface to call Prolog goals from C — e.g., when continuations, which have been internally stored, have to be resumed, as done by `slgcall/4` and `answer/2`. We wanted to design a solution which relied as little as possible on non-widely available characteristics of C-to-Prolog interfaces (to simplify porting the code), but which kept the efficiency as high as possible.

The general solution we have adopted is to move calls to continuations from the C level to the Prolog level by returning them as a term, using an extra argument in our primitives, to be called from Prolog. This is possible since continuations are rewritten as separate, unique predicates which therefore have an entry point accessible from Prolog. If several continuations have to be called, they can be returned and invoked one at a time on backtracking,[5] and fail when there is no pending continuation call. New continuations generated during program execution can be destructively inserted at the end of the list of continuations transparently to Prolog. Additionally, this avoids using up C stack space due to repeated Prolog $\rightarrow$ C $\rightarrow$ Prolog $\rightarrow \ldots$ calls, which may exhaust the C stack. Moreover, the C code is somewhat simplified (e.g., there is no need to set up a Prolog environment to be used from C) which makes using a lower-level, faster interface less of a burden.

3.3 Freezing Continuation Calls

In this section we sketch some proposals to reduce the overhead associated with the way continuation calls are handled in the original continuation call proposal.

Resuming consumers: Our starting point saves a binding list in the table to reinstall the environment of consumers when they have to be resumed. This is a relatively non-intrusive technique, but it requires copying terms back and forth between Prolog and the table where calls are stored. Restarting a consumer needs to construct a term whose first argument is the new answer (which is stored in the heap), the second one is the identifier of the tabled goal (an atomic item), and the third one a list of bindings (which may be arbitrarily large). If the list of bindings has N elements, constructing the continuation call requires creating $\approx 2N + 4$ heap cells. If a continuation call is resumed often and N is high, the efficiency of the system can degrade quickly.

The technique we propose constructs continuation calls on the heap as regular Prolog terms. As these continuations are later recovered through a unique call identifier, and each continuation is unified with a new, fresh variable (`CCall` in `resume_ccalls/4`, Figure 7), full unification or even pattern matching are unnecessary, and resuming a continuation is a constant time operation.

However, the fragment of code which constructs the continuation call performs backtracking to continue exploring pending branches. This will remove the constructed call from the heap. Protecting that term is needed to make it possible to construct it only once and reuse it later. A feasible and simple solution is to freeze continuation calls in a memory area which is not affected by

[5] This exploits being able to write non-deterministic predicates in C. Should this feature not be available, a list of continuations can always be returned instead which will be traversed on backtracking using `member/2`.

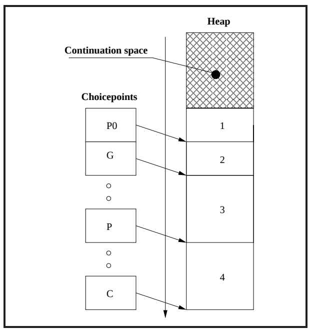

Fig. 5. Initial state.

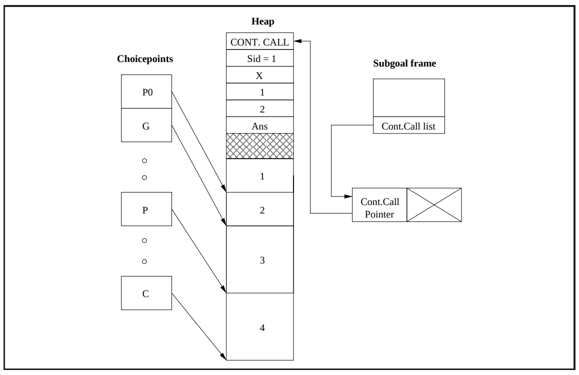

Fig. 6. Frozen continuation call.

backtracking. This will in principle make the aforementioned problem disappear. Selecting a brand new area will, however, bring additional issues as some WAM instructions would have to be changed in order to take it into account: for example, variable binding direction is commonly determined using the addresses of variables (in addition to their tags) so that younger variables point to older variables in order to save trailing. One easy way to reconcile existing WAM machinery with this continuation call area is to reserve part of the heap for it. This makes the usual WAM assumptions to hold and exactly the same WAM instructions can be used to construct and traverse data structures both in the regular heap and in the continuation call area. Therefore, regarding forward execution and backtracking, only minimal changes (e.g., the initialization of the **H** pointer, and selecting the right read/write heap pointer when dealing with the regular heap or the continuation call zone) have to be introduced.

Figure 5 shows the state of the choicepoint stack and heap (both assumed to grow downwards) before freezing a continuation call. Figure 6 shows the continuation call `(C, [X,1,2], Ans)` frozen at the beginning of the heap, where it is unaffected by backtracking as the WAM execution started with the **H** pointer placed just after the continuation call zone. In order to recover the continuation calls, a new field is added to the table pointing to a (Prolog) list whose elements, in turn, point to every continuation found so far for a given tabled goal.

This makes freezing a continuation call require some extra time in order to copy it on the heap. However, resuming a continuation is a constant time operation. Other systems, like CHAT or SLG-WAM, spend some extra time while preparing a consumer to be resumed, as they need to record bindings in a forward trail in order to later reinstall them. In our case, when the continuation is to be executed, the list of bindings carried with it is unified with the variables in its body, implementing essentially the same functionality as the forward trail.

In a previous paper [6] we presented a preliminary version of this technique where the heap was frozen by manipulating the contents of some choicepoints, in what can be seen as a variant of CHAT. The work presented herein works around several drawbacks in that approach.

Memory management for continuation space: As mentioned before, the area for continuations is taken from the same memory zone where the general heap is located, thus making it possible to use the same WAM instructions without any change. In case more memory is needed, reallocating the heap and the continuation area can be done simultaneously, keeping the same placement relation between both. As data inside both areas has the same format, adjusting pointers can be done using memory management routines already existing for the regular WAM implementation, which only have to be updated to take into account the existence of a gap of unused memory between the continuation call and regular heap areas. Additionally, sliding the heap within its zone to make room for more heap or for more continuations amounts only to readjusting pointers by a constant amount.

Frozen continuations are, in principle, only reachable from the table structure, which makes them candidates to be (wrongly) removed in case of garbage collection. A possible solution which needs almost no change to the garbage collector is to link a Prolog list L from some initial, dummy choice point. Each element in L points to the continuation list of a generator, which makes all the continuations reachable by the garbage collector, and therefore protected. When a generator is completed all of its answers are already stored in the trie, and its continuations are no longer needed. Removing the pointer from L to this list of unneeded continuations will make garbage collection reclaim their space. In order to adjust the pointers from table entries to the continuations when these are reallocated after a garbage collection, each element of L includes a pointer back to the corresponding table entry which can be used to quickly locate which pointers have to be updated in the table entries. A new routine has to be added to the garbage collector to perform this step.

Avoiding trail management to recover a continuation call state: The same term T corresponding to a continuation call C can be used several times to generate multiple answers to a query. This is in general not a problem as answers are in any case saved in a safe place (e.g., the answer table), and backtracking would undo the bindings to the free variables in T. There is, however, a particular case which needs special measures. When a continuation call C_1, identical to C, is resumed within the scope of C, and it is going to read a new answer, the state of T has to be reset to its frozen initial state. Since C_1 is using the same heap term T as C, we say that C_1 is a *reusing* call.

The solution we present tries to eliminate the need for treating reusing calls as a special case of a continuation call. Reusing calls appear because our baseline implementation resumes continuations when new answers are found, just when we could be in the scope of an identical continuation call. But resumptions can be delayed until the moment in which we are going to check for completion (in the generator) and then the continuation calls with unconsumed answers can be resumed. Following this approach there are no reusing calls because a new continuation call is never resumed within the scope of another continuation call and we do not need to do any trail management.

New tabling primitives and translation for `path/2`: Figure 7 shows the new program transformation we propose for the `path/2` program in order to take into account the ideas in the previous sections. Variables `Pred`, `CCall`, and `F` will contain goals built in C but called from Prolog (Section 3.2). The third and fourth arguments of `resume_ccalls/4` implement a trick to create a choicepoint with *dummy* slots which will be used to store pointers to the next continuation to execute and to the generator whose continuations we are resuming. Creating such a slot in this way, at the source level, avoids having to change the structure of choicepoints and how they are managed in the abstract machine.

In the clause corresponding to `path/2`, the primitive `slg/1` shown in Figure 2 is now split into `slgcall/3`, `execute_generator/2`, and `consume_answer/2`. `slgcall/3` tests whether we are in a generator position. In that case, it constructs a new goal from the term passed as first argument (the term `slg_path/2` will be constructed in this case). This goal is returned in variable `Pred`, which will be called later. Otherwise, the goal `true` will be returned.

This new goal is always passed to `execute_generator/2` which executes it. If it is `true` it will succeed, and the execution will continue with `consume_answer/2`. However, `slg_path/2` is ensured to ultimately fail (because the solutions to the tabled predicate are generated by storing answers into the table and failing in `answer/2`), so that the "else" part of `execute_generator/2` is taken. There, consumers are resumed before checking for completion and `consume_answer/2` returns, on backtracking, each of the answers found for `path(X, Y)`.

`slg_path/2` is similar to `path/2` but it does not have to return all solutions on backtracking, as `consume_answer/2` does. Instead, it has to generate all possible solutions and save them: `new_ccall/5` inserts a new continuation if the execution of `path(Z,Y)` is not complete. Otherwise, it uses `path_cont_1` as the main functor of a goal whose arguments are answers consumed from the table. This goal is returned in `F` and immediately called. In this particular case the (recursive) call to `path/2` is the last goal in the recursive clause (see Figure 1), and therefore the continuation directly inserts the answer in the table.

Finally, `answer/2` does not resume continuations anymore to avoid reusing calls, since `resume_ccalls/4` resumes all the continuations of the tabled call identified by `Sid` and its dependent generators before checking for completion.

3.4 Freezing Answers

When `resume_ccalls/4` is resuming continuation calls, answers found for the tabled calls so far are used to continue execution. These answers are, in principle, stored in the table (i.e., `answer/2` inserted them), and they have to be constructed on the heap so that the continuation call can access them and proceed with execution.

The ideas in Section 3.3 can be reused to freeze the answers and avoid the overhead of building them again. As done with the continuation calls, a new field is added to the table pointing to a (Prolog) list which holds all the answers found so far for a tabled goal. This list will be traversed for each of the consumers of the corresponding tabled call. In spite of this freezing operation, answers to

```
path(X,Y) :-
    slgcall (path(X, Y), Sid, Pred),
    execute_generator (Pred,Sid ),
    consume_answer(path(X, Y), Sid).

slg_path (path(X, Y),Sid) :-
    edge(X, Z),
    slgcall (path(Z, Y), NSid, Pred),
    execute_generator (Pred,NSid),
    new_ccall (Sid, NSid, [X],
               path_cont_1, F),
    call (F).
```

```
slg_path (path(X, Y), Sid) :-
    edge(X, Y),
    answer(path(X, Y), Sid).

path_cont_1(path(X, Y), Sid, [Z]) :-
    answer(path(Z, Y), Sid).

execute_generator (Pred,Sid) :-
    (
        call (Pred) ->
        true
    ;
        resume_ccalls (Sid, CCall ,0,0),
        call (CCall)
    ).
```

Fig. 7. New program transformation for right-recursive definition of `path/2`.

tabled goals are additionally stored in the table. There are two reasons for this: the first one is that when some tabled goal is completed, all the answers have to be accessible from outside the derivation tree of the goal. The second one is that the table makes checking for duplicate answers faster.

3.5 Repeated continuation calls

Continuation calls could be duplicated in a table entry, which forces an unnecessary recomputation when new answers are found. This problem can also show up in other *suspension-based* tabling implementations and it can degrade the efficiency of the system. As an example, if the program in Figure 7 is executed against a graph with duplicate `edge/2` facts, duplicate continuation calls will be created, as `edge(X, Z)` in the body of `slg_path/2` can match two identical facts and return two identical bindings which will make `new_ccall/4` to insert two identical continuations. Since we traverse the new continuations to copy them in the heap, we can check for duplicates before storing them without having to pay an excessive performance penalty. As done with answers, a trie structure is used to check for duplicates in an efficient manner.

4 Performance Evaluation

We have implemented the proposed techniques as an extension of the Ciao system [1]. Tabled evaluation is provided to the user as a loadable *package* that provides the new directives and user-level predicates, performs the program transformations, and links in the low-level support for tabling. We have implemented and measured three variants: the first one is based on a direct adaptation of the implementation presented in [15], using the standard, high-level C interface. We have also implemented a second variant in which the lower-level and simplified C interface is used, as discussed in Sections 3.1 and 3.2. Finally, a third

lchain X	Left-recursive path program, unidimensional graph.
lcycle X	Left-recursive path program, cyclic graph.
rchain X	Right-recursive path program (this generates more continuation calls), unidimensional graph.
rcycle X	Right-recursive path program, cyclic graph.
rcycleR X	Right-recursive path program, cyclic graph with repeated edges.
rcycleF X	Like rcycle 256, but executing `fib(20,_)` before `edge/2` goals.
numbers X	Find arithmetic expressions which evaluate to some number N using all the numbers in a list L.
numbers Xr	Same as above, but all the numbers in L are all the same (this generates a larger search space).
atr2	A parser for Japanese.

Table 1. Terse description of the benchmarks used.

Benchmark	**Ciao + Ccal (baseline)**	**Lower C interf.**	**Ciao + CC**
lchain 1,024	7.12	2.85	1.89
lcycle 1,024	7.32	2.92	1.96
rchain 1,024	2,620.60	1,046.10	557.92
rcycle 1,024	8,613.10	2,772.60	1,097.26
numbers 5	1,691.00	781.40	772.10
numbers 5r	3,974.90	1,425.48	1,059.93

Table 2. Speed comparison of three Ciao implementations.

variant, which we call `CC` (Callable Continuations), incorporates the proposed improvements to the model discussed in Sections 3.3 and 3.4.

We evaluated the impact of this series of optimizations by using some of the benchmarks in Table 1. The results are shown in Table 2, where times are given in milliseconds. Lowering the level of the C interface and improving the transformation for tabling and the way calls are performed have a clear impact. It should also be noted that the latter improvement seems to be specially relevant in non-trivial programs which handle data structures (the larger the data structures are, the more re-copying we avoid) as opposed to those where little data management is done. On average, we consider the version reported in the rightmost column to be the implementation of choice among those we have developed, and this is the one we will refer to in the rest of the paper.

Table 3 tries to determine 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. In this table we provide, for several benchmarks, the raw time (in milliseconds) taken to execute them using tabling and, when possible, SLD resolution. Measurements have been made on Ciao-1.13, using the standard, unoptimized bytecode-based compilation, and with the CC extensions loaded, as well as in XSB 3.0.1, YapTab 5.1.1, and B-Prolog 7.0. 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 processor.

Analyzing the behavior of the `rcycle X` benchmark, which is an example of almost pure tabling evaluation, we observe that our asymptotic behavior is similar to other tabling approaches. If we multiply `X` by N, the resulting time for all of the systems (except YapTab) is multiplied by approximately $2N$. YapTab does not follow the same behavior, and, while we could not find out exactly the reason, we think it is due to YapTab *on-the-fly* creating an indexing table which selects the right `edge/2` clause in constant time, while other implementations spend more time performing a search.

B-Prolog, which uses a linear tabling approach, is the fastest SLG resolution implementation for `rcycle X`, since there is no recomputation in that benchmark. However, efficiency suffers if a costly predicate has to be recomputed: this is what happens in `rcycleF`, where we added a call to a predicate calculating the 20^{th} Fibonacci number before each of the calls to `edge/2` in the body of `path/2`. This is a (well-known) disadvantage of linear tabling techniques which does not affect suspension-based approaches. It has to be noted, however, that current versions of B-Prolog implement an optimized variant of its original linear tabling mechanism [22] which tries to avoid reevaluation of looping subgoals. The impact of recomputation is, therefore, not as important as it may initially seem. Additionally, in our experience B-Prolog is already a very fast SLD system, and its speed seems to carry on to SLG execution, which makes it, in our experiments, the fastest SLG system in absolute terms, except when unneeded recomputation is performed.

The ideas discussed in Section 3.5 show their effectiveness in the `rcycleR 2048` benchmark, where duplicating the clauses of `edge/2` produces repeated consumers. While B-Prolog is affected by a factor close to 2, and XSB and YapTab by a factor of 1.5, the Ciao+CC implementation is affected only by a factor of a 5% because it does not add repeated consumers to the tabled evaluation.

In order to compare our implementation with XSB, we must take into account that XSB is somewhat slower than Ciao when executing programs using SLD resolution —at least in those cases where the program execution is large enough to be really significant (between 1.8 and 2 times slower for these non-trivial programs). This is partly due to the fact that XSB is, even in the case of SLD execution, prepared for tabled resolution, and thus the SLG-WAM has an additional overhead (reported to be around 10% [16]) not present in other Prolog systems and also presumably that the priorities of their implementors were understandably more focused on the implementation of tabling. However, XSB executes tabling around 1.8 times faster than our current implementation, confirming, as expected, the advantages of the native implementation, since we perform some operations at the Prolog level.

Although this lower efficiency is obviously a disadvantage of our implementation, it is worth noting that, since our approach does not introduce changes neither in the WAM nor in the associated Prolog compiler, the speed at which non-tabled 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 created as dynamic libraries, loaded on demand, and only stubs are needed in the regular

	Ciao+CC		XSB		YapTab		B-Prolog	
Program	SLD	Tabling	SLD	Tabling	SLD	Tabling	SLD	Tabling
rcycle 256	-	70.57	-	36.44	-	59.95	-	26.02
rcycle 512	-	288.14	-	151.26	-	311.47	-	103.16
rcycle 1,024	-	1,097.26	-	683.18	-	1,229.86	-	407.95
rcycle 2,048	-	4,375.93	-	3,664.02	-	2,451.67	-	1,596.06
rcycleR 2,048	-	4,578.50	-	5,473.91	-	3,576.31	-	2,877.60
rcycleF 256	-	1,641.95	-	2,472.61	-	1,023.77	-	2,023.75
numbers 3r	1.62	1.39	3.61	1.91	1.87	1.08	1.46	1.13
numbers 4r	99.74	36.13	211.08	51.72	108.08	29.16	83.89	22.07
numbers 5r	7,702.03	1,059.93	16,248.01	1,653.82	8,620.33	919.88	6,599.75	708.40
atr2	-	703.19	-	581.31	-	278.41	-	272.55

Table 3. Comparing Ciao+CC with XSB, YapTab, and B-Prolog.

engine. The program transformation is taken care of by a *package* (a plugin for the Ciao compiler) [2] which is loaded and active only at compile time.

In non-trivial benchmarks like `numbers Xr`, which at least in principle should reflect more accurately what one might expect in larger applications, execution times are in the end somewhat favorable to Ciao+CC when comparing with XSB. This is probably due to the faster raw speed of the basic engine in Ciao but it also implies that the overhead of the approach to tabling used is reasonable after the proposed optimizations. 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 [10, 3] can further improve the speed of the SLD part of the computation.

The results are also encouraging to us because they appear to be another example supporting the "Ciao approach:" start from a fast and robust, but extensible LP-kernel system and then include additional characteristics by means of pluggable components whose implementation must, of course, be as efficient as possible but which in the end benefit from the initial base speed of the system.

We have not analyzed in detail the memory consumption behavior of the continuation call technique, as we are right now working on improving it. However, since we copy the same part of the heap CAT does, but using a different strategy, and we eventually (as generators are completed) get rid of the data structures corresponding to the frozen continuation calls, we foresee that our memory consumption should currently be in the same range as that of CAT.

5 Conclusions

We have reported on the design and efficiency of some improvements made to the continuation call mechanism of Ramesh and Chen. While, as expected, we cannot achieve using just these techniques the same level of performance during tabled evaluation as the natively implemented approaches our experimental results show that the overhead is essentially a reasonable constant factor, with good scaling and convergence characteristics. We argue that this is a useful result since the proposed mechanism is still easier to add to an existing WAM-based system than implementing other approaches such as the SLG-WAM, as it requires relatively small changes to the underlying execution engine. In fact, almost everything is implemented within a fairly reusable C library and using a Prolog program transformation. Our main conclusion is that using an external module for implementing tabling is a viable alternative for adding tabled evaluation to Prolog systems, especially if coupled with the proposed optimizations. It is also an approach that ties in well with the modular approach to extensions which is an integral part of the design of the Ciao system.

6 Acknowledgments

This work was funded in part by the IST program of the European Commission, FP6 FET project IST-15905 *MOBIUS*, by the Spanish Ministry of Education and Science (MEC) project TIN2005-09207-C03 *MERIT-COMVERS* and by the Government of the Madrid Region (CAM) Project S-0505/TIC/0407 *PROMESAS*. Manuel Hermenegildo is also funded in part by the Prince of Asturias Chair in Information Science and Technology at the U. of New Mexico, USA and the IMDEA-Software Institute, Madrid, Spain. Cláudio Silva and Ricardo Rocha were partially funded by project Myddas (POSC/EIA/59154/2004). Ricardo Rocha was also funded by project STAMPA (PTDC/EIA/67738/2006).

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. Some Improvements over the Continuation Call Tabling Implementation Technique. In *CICLOPS 2007*. ACM Press, September 2007.
7. 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.
8. Bart Demoen and Konstantinos F. Sagonas. Chat: The copy-hybrid approach to tabling. In *Practical Applications of Declarative Languages*, pages 106–121, 1999.

9. 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.
10. 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.
11. 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.
12. I.V. Ramakrishnan, P. Rao, K.F. Sagonas, T. Swift, and D.S. Warren. Efficient tabling mechanisms for logic programs. In *ICLP*, pages 697–711, 1995.
13. Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1993.
14. 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.
15. 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.
16. 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.
17. Cáudio Silva. On Applying Program Transformation to Implement Tabled Evaluation in Prolog. Master's thesis, Faculdade de Ciências, Universidade do Porto, January 2007.
18. Z. Somogyi and K. Sagonas. Tabling in Mercury: Design and Implementation. In *International Symposium on Practical Aspects of Declarative Languages*, number 3819 in LNCS, pages 150–167. Springer-Verlag, 2006.
19. 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.
20. D.S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, 1992.
21. 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.
22. Neng-Fa Zhou, Taisuke Sato, and Yi-Dong Shen. Linear Tabling Strategies and Optimizations. *Theory and Practice of Logic programming*, 2007. Accepted for publication. Available from `http://arxiv.org/abs/0705.3468v1`.
23. 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.
24. 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.