# Some Improvements over the Continuation Call Tabling Implementation Technique

Pablo Chico[1] 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 need significant changes to the underlying Prolog implementation. More portable approaches, generally using program transformation, have been proposed but they often result in lower efficiency. We explore some techniques aimed at combining the best of these worlds, i.e., developing a portable and extensible implementation, with minimal modifications at the abstract machine level, and with reasonably good performance. Our preliminary results indicate promising performance.

## 1 Introduction

Tabling [15, 2, 14] is a resolution strategy which tries to *memorize* previous calls and its 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) and also programs which perform repeated computations can be greatly sped up. Declarativeness of programs is also improved since the order of clauses and goals within a clause is less relevant, if at all.

In all cases this stems from checking whether calls to tabled predicates[4] have been made before. Repeated calls to tabled predicates consume answers from a

[4] A *tabled predicate* is a predicate which has been marked to be evaluated using tabling.

table, they suspend when all answers have been consumed, and they fail when no more answers can be generated.

Tabled evaluation has been successfully applied in many fields, such as deductive databases [10], program analysis [3], reasoning in the semantic Web [17], model checking [8], and others.

Those 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 of selecting which predicates to table in order not to incur in undesired speed-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 [12], by copying to another area, as in CAT [5], or using an intermediate solution as in CHAT [6]. 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 BProlog [16] and the DRA scheme [7].

Suspension-based mechanism have achieved very good performance results if, in general, require deep changes to the underlying Prolog implementation. It implies harder implementation than that of linear mechanisms, which can usually be implemented on top of existing sequential engines without major modifications.

Our objective is to study an implementation (a variation of the continuation call mechanism [11]) which is very portable since does not need deep changes to the underlying standard machinery, but it has worse efficiency than other suspension-based mechanism. We will analyse its bottlenecks to try to improve its efficiency without so much lost of portability.

## 2 Tabling Basics

In this section we will sketch how tabled evaluation works from a user point of view (more details can be found in [2, 12]) and then we will briefly describe the implementation technique proposed in [11] on which we base our work.

### 2.1 Tabling by Example

Let us use as running example the program in Figure 1 (taken from [11]), where we can ignore the declaration `:- tabled path/2` (which instructs the compiler to use tabled execution for the designated predicate) for the moment, and assume that SLD resolution is to be used. The program is aimed at determining reachability of nodes in a graph, and a query such as `?- path(a, N).` will never

```
edge(a, b).
edge(b, c).
edge(b, d).

:- tabled path/2

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

**Fig. 1.** A simple tabled program.

```
path(X, Y):- slg(path(X, Y)).

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

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

**Fig. 2.** Program in Figure 1 transformed for tabled execution.

terminate as there is a left-recursive clause which generates a goal with the same instantiation as the initial call.

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 uses resolution against the program clauses to derive answers for the goal. Consumers *suspend* the current execution path (using implementation-dependent means) and move to 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 and makes it possible to reactivate suspended calls and to continue execution at the point where it was stopped — i.e., consumers do not use SLD resolution, but rather they obtain answers from the table where they had been previously inserted.

Predicates not marked as tabled are executed following SLD resolution, hopefully without any (or minimal) overhead associated to the availability of tabling.

## 2.2 A Concrete Technique: Continuation Call

The *continuation call* technique presented by Ramesh and Chen in [11] implements tabling by a combination of program transformation and side effects in the form of insertions to and reads from an internally-maintained table which relates calls, answers, and the continuation code to be executed after a new answer is inserted in the table. We will sketch how the mechanism works using the `path/2` example (Figure 2).

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 that `path/2` is tabled. `slg/1` will ensure that its argument is evaluated to completion and it will return, on backtracking, all the solutions found for the tabled predicate. `slg/1` generates a new identifier for this call and introduces both the identifier and the call in the answer table as keys for the entry. Control is then passed to a new distinct predicate (in this case, `slg_path/3`) by constructing a term from `path(X, Y)` (which is passed as argument to `slg/1`) and then calling this term, suitably instantiated, from inside the implementation of `slg/1`[5]. The first and second arguments come from `path/2` and the third one is the identifier generated for the parent call, which is used to relate operations on the table with this initial call. Every clause of `slg_path/3` is constructed from a clause of the original `path/2` by:

– Adding an `answer/2` primitive at the end of each clause of `path/2`. `answer/2` is responsible for checking for redundant answers and executing whatever continuations there are associated with that call.
– Instrumenting recursive calls to `path/2` using `slgcall/3`. If the term `path(X, Y)`, passed as an argument, has already been inserted in the table, `slgcall/3` creates a new consumer which reads answers from the table. It is otherwise inserted in the table with a new call identifier and execution follows against the `slg_path/3` program clauses to derive new answers. In both cases `path_cont/2` is associated as (one of) the continuation(s) of `path(X, Y)` (with the remaining clause body of `path/2` after the recursive call), and `slgcall/3` fails. The continuation `path_cont/2` will be activated by `answer/2` upon answer insertion or erased upon completion of the subgoal `path(X, Y)`.
– `path_cont/2` and `slg_path/3` are constructed in a similar way: the continuation is applied the same transformation as the initial clauses and can call `slgcall/3` and `answer/2` at appropriate times.

As this strategy tries to complete subgoals as soon as possible, failing whenever new answers are found, it implements the so called *local scheduling* [12]. This implementation uses the same completion detection algorithm as the SLG-WAM.

Figure 3 shows a variation of the program which requires slight modifications of the translation. Note that an answer to `?- path(X, Y)` needs to give a value to a variable (`X`) which does not appear in the recursive call to `path/2`. Therefore, if we follow the translation in Figure 2, this variable will not be available at the time where the answer is inserted in the table.

The solution adopted in this case is to explicitly carry a set of variables when preparing the call to the continuation. This set is also inserted in the table, and is passed to the continuation call when resumed.

The translation is shown in Figure 4. Note that the call to `slgcall/4` in `path_cont_1` includes a list containing variable `X`. This list is, on resumption, received by `path_cont_1` and used to construct and insert in the table an answer which includes `X`. 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

[5] The new term is created, in this case, by prepending the prefix `slg_` to the argument passed to `slg/1`. Any means of constructing a new unique predicate symbol based on the original one is acceptable.

```
:- tabled path/2.

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

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

**Fig. 3.** A program which needs to keep an environment.

```
path(X, Y):- slg(path(X, Y)).

slg_path(X, Y, Id):-
    edge(X, Y),
    slgcall(Id, [X], path(Y, Z), path_cont_1).
slg_path(X, Y, Id):-
    edge(X, Y),
    answer(Id, path(X, Y)).

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

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

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

**Fig. 5.** Pseudo-code for the `answer/2` primitive

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

**Fig. 6.** Pseudo-code for the `slgcall/4` primitive

appearing in the tabled call itself do not need to be included, as they will be passed along anyway.

The list of bindings is a means to recover the environment existing when a call is suspended. Other approaches recover this environment e.g using lower-level mechanisms, such as the forward trail of SLG-WAM plus freeze registers [12]. The continuation call approach, has, however, the nice property that several of the operations are made at the Prolog level through program transformation, which increases its portability and simplifies implementation. On the other hand, the primitives which insert answers in the table and retrieve them are usually, and for efficiency issues, written using some lower-level language and accessed using a suitable interface.

The pseudo-code for `answer/2` and `slgcall/4` is in Figure 5 and 6, respectively. The pseudo-code for `slg/1` is similar to that of `slgcall/4` but instead of consuming answers and adding a continuation call, it returns each of the answers by backtracking and finally fails when all the answers have been generated.

### 2.3 Issues in the Continuation Call Mechanism

We have identified two performance issues when implementing the technique sketched in the previous section. We will assume in what remains that the new primitives are implemented using a lower-level, imperative language such as C. This is needed, for example, to make table operations as efficient as possible, since the performance of the table is critical the overall performance of the system.

The first issue is rather general and related to the heavy use of the interface from C to Prolog (and back) that the implementation needs to make. Continuation calls (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/3` fail immediately after associating a continuation call with a table call in order to force the program to search 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. When continuations are executed by `answer/2`, it is necessary to reconstruct them as Prolog terms from the data stored in the table to call them as a goal. The adicional overrhead would be in the order of $N * M$ where $N$ is the size (in heap cells) of the continuation call and $M$ is the final number of answers for the goal.

If we would also like to reduce the requirements on the C interface to a minimum, so that porting the code to other systems is as painless as possible, then an additional issue that may be considered is that of the baseline implementation [13] is the capability to backtrack over Prolog predicates called from C.

?????????????????????? Reestructurar, en la realidad la de Claudio no puede hacer eso, por ellos que sólo implementan local scheduling. Quizá el argumento sea la búsqueda de extensibilidad.

This may not be possible in all C interfaces and it may be inefficient in others, so it is also a candidate for simplification.

## 3 An Improvement over the Continuation Call Technique

We have addressed the problems highlighted in the previous section by first identifying some bottlenecks. Two main points were taking most of the time: on one hand, the need to set up Prolog calls from C and, on the other hand, the need to construct and copy terms repeatedly in order to perform consumer resumption. We will go over these two improvements in the following sections.

### 3.1 Using a Lower-Level Interface

The calls C-to-Prolog was initially done using a relatively high-level interface similar to those commonly found in logic programming systems nowadays: operations to create and traverse Prolog terms appear to the programmer as regular C functions. 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.

Since the low-level code which constructs Prolog terms and performs calls from C is the same regardless the program being executed, we decided to skip the programmer interface and call directly macros available in the engine implementation. Given that the complexity of the C code involved is manageable, that was a not difficult task to do and speeds the execution up by a factor of 2.5, on average.

### 3.2 Calling Prolog from C

A major issue when lowering the level of the C-to-Prolog interface is being able to call Prolog goals from C efficiently. This is needed both by `slgcall/3` and `answer/2` in order to invoke continuations of tabled predicates. As mentioned before, we want to design a solution which relies as little as possible on non-widely available characteristics of C-to-Prolog interfaces, and which keeps the efficiency as high as possible. For example, we would like to have the possibility of obtaining on backtracking several answers to a continuation call since this would make it possible to explore more scheduling strategies, thereby improving the extensibility of the system. As mentioned before, C interfaces do not always offer this feature.

The solution we have adopted is to move calls to continuations from the C level to the Prolog level by constructing, in C, the term corresponding to each continuation call (which had to be created anyway) and storing them in a list. `slgcall/3` and `answer/2` need an extra argument to return the terms in that list one at a time on backtracking[6]; these terms are then called by Prolog. Failure happens when this list is empty —i.e., there is no pending continuation call. Resumed consumers which generate additional answers can, upon checking for duplicates, destructively insert these answers at the end of the list in a fashion transparent to Prolog.

In Figure 7 (which shows the translation we propose now for the code in Figure 3), `answer/4`, `read_answers/5`, and `slgcall/4` return in variables `Pred` and `CCall` the goals constructed in C, which are immediately called. This simplifies the parts written in C, which do not need to stack local environments to call Prolog goals, and whose level can be easily lowered (?????????????????????? Explicar lo de evitar entornos de C) by passing the C-style function calls for term handling. The last unused argument of `answer/4` (and `read_answers/5`)

[6] it requires non-deterministic predicates for easier implementation, but we could return a list of continuations and reduce requirements on the C interface to a minimum.

```
path(X,Y) :-
    slgcall(path(X, Y), Sid, true, Pred),
    (
        nonvar(Pred) ->
            (call(Pred); test_complete(Sid))
    ;
        true
    ),
    consume_answer(path(X, Y), Sid).

slg_path(path(X, Y),Sid) :-
    edge(X, Z),
    slgcall(path(Z, Y), NewSid, path_cont_1, Pred),
    (
        nonvar(Pred) ->
            (call(Pred); test_complete(NewSid))
    ;
        true
    ),
    read_answers(Sid, NewSid, [X], CCall, 0),
    call(CCall).

slg_path(path(X, Y), Sid) :-
    edge(X, Y),
    answer(path(X, Y), Sid, CCall, 0),
    call(CCall).

path_cont_1(path(X, Y), Sid, [Z]) :-
    answer(path(Z, Y), Sid, CCall, 0),
    call(CCall).
```

**Fig. 7.** New Program Transformation for right-recursive definition of `path/2`

implements a trick to make the corresponding choicepoint have an extra, unused slot (corresponding to a WAM argument), which will be used to hold a pointer to the list of continuations (or answers) built in C and to be called from Prolog. Having such a slot avoids changing the structure of choicepoints and how they are managed. This pointer is destructively updated every time a continuation call is handed to the Prolog level.

We would like to clarify how some of the primitives used in Figure 7 work for this case (note that the functionality of `slgcall/3` (`slg/1`) has been split acros `slgcall/3`, `test_completion/1` and `read_answer/5` (`consume_answers/2`) in order to being able to perform calls to continuations from Prolog). `slgcall/5`, as in the original definition, checks if a call to a tabled goal is a new one. If so, `Pred` is unified with a goal whose main functor is `slg_path/2` and whose arguments are appropriately instantiated. A free variable is returned otherwise. `test_complete/1` is only useful for its side effects: it tests if the tabled goal identified by `Sid` can be marked as complete, and it gets marked in that case. It always succeeds.

`consume_answer/2` returns the answers stored in the table one at a time on backtracking, and acts as an interface with the regular Prolog execution. Similarly, `read_answers/5` consumes actual answers for the call identified by `NewSid` and then associates a new continuation call to `NewSid`. Its first argument, `Sid` is needed to check for completion.

## 3.3 Freezing continuation calls

In this section we will sketch some proposals to reduce the overhead associated with the way continuation calls were handled in their original formulation.

**The Overhead of Resuming a Consumer** The original continuation call technique saved a binding list to reinstall the environment of consumers instead of copying or freezing the stacks and using a forward trail, as CAT, CHAT, or SLG-WAM. 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 goal identifier (an atomic item), and the third one is a list of bindings (which may be arbitrarily large). If the list of bindings has $N$ elements, constructing the continuation call needs to create $\approx 2N + 4$ 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 all the continuation calls in the heap as a regular Prolog term. This makes calling the continuation a constant time operation, since `answer/4` only has to unify its third argument with the continuation call. As that argument is variable at run time, full unification is not needed. However, the fragment of code which constructs this call performs backtracking as it fails after every success of `answer/4`. This would remove the constructed call from the heap, thereby forcing us to construct it again. Protecting that term would make it possible to construct it only once.

Other systems have to face a similar problem. SLG-WAM uses special *freeze registers* and a *forward trail* to protect and reinstall bindings to recover a previous environment, and CHAT freezes the heap by adjusting heap pointers in choicepoints. Our solution can be seen as a variant of the solution that CHAT takes, but without having to introduce new instructions in the virtual machine.

In order to explain our proposed *freezing* technique we will use the following notation (borrowed from [6]): `H` will denote a pointer to the top of the heap; `B` will be the pointer to the most recent choicepoint. To distinguish different kinds of choicepoints we will use $\mathtt{B}_\mathtt{T}$, where `T` can be `G`, `C` or `P` (standing for generator, consumer, or Prolog). The pointer to the heap stored in a choicepoint will be denoted as $\mathtt{B}_\mathtt{T}\mathtt{[H]}$.

In CHAT the heap pointer is not reset on backtracking (as the WAM does with the assignment $\mathtt{H} := \mathtt{B}_\mathtt{P}\mathtt{[H]}$) by manipulating the heap pointer field $\mathtt{B}_\mathtt{P}\mathtt{[H]}$

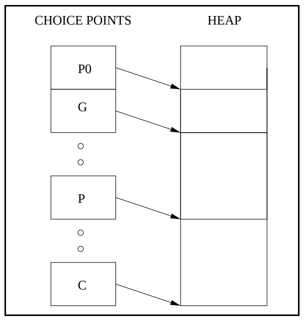

**Fig. 8.** Initial state

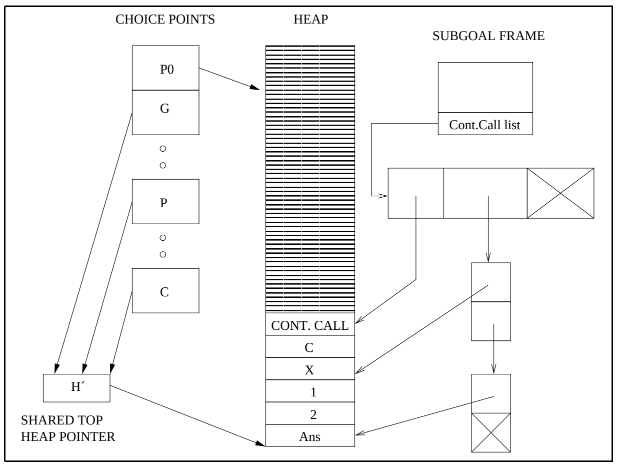

**Fig. 9.** Frozen continuation call

of the Prolog choicepoints between the (newly created) consumer choicepoint and the choicepoint corresponding to its generator so that they all point to the current top of the heap `H`: $\texttt{B}_\texttt{P}\texttt{[H]} := \texttt{B}_\texttt{C}\texttt{[H]}$. Therefore, forward execution will continue building terms on the heap on top of the previous solutions.

This solution can generate garbage in the heap, which is not a serious problem as garbage collection can eventually free it. A more critical problem is the need to traverse an arbitrarily long series of choicepoints, which could make the system efficiency decrease. A solution for this problem has been proposed [4], which for us has the drawback of needing new WAM-level instructions and changing the fields in the choicepoints. As an alternative solution, we update the `B[H]` fields of the choicepoints between the new consumer and its generator so that they point to a pointer `H'` which in turn points to the heap top. Whenever we need to change again the `B[H]` field for these choicepoints, we simply update `H'` plus the choicepoints pushed since the last adjustments. Determining whether `B[H]` points to the heap or to `H'` is very easy by simply deciding whether it falls within the heap limits (it affects to all of the backtracking WAM instructions).

Figure 8 shows the state of the choicepoint stack and heap before freezing a continuation call. On the left of Figure 9 all `B[H]` fields of the choicepoints `G`, `P`, and `C` have changed to a common pointer `H'` to the heap top. Thus, the continuation call `(C, [X,1,2], Ans)` is frozen.

The continuation calls corresponding to a goal $G$ are linked in a (Prolog) list pointed to from the table entry corresponding to $G$. This list is the one accessed by `answer/4` and whose elements are the continuation calls returned to Prolog on backtracking.

**Trail Management to Recover a Continuation Call State** The same term corresponding to a continuation call can be used several times to generate multiple answers to a query. While the heap zone this query has generated is protected and is not overwritten by backtracking, the free variables in the continuation call itself could have been bound. They need to be reset to their original state (i.e., as they were when first stored on the heap) in order to use the continuation call on the heap for the next call.

The approach we have taken to avoid repeatedly copying the same continuation call to the heap is to record in a list the variables which were initially free in the continuation call (list at the right of Figure 9 whose fields point out `X` and `Ans` heap cells). Since we are already copying on the heap, this adds a small overhead. This list is used to reset these variables to unbound every time a continuation call is going to be *reused* [7]. This amounts to traversing this list performing exactly the same operation as an untrailing would do. Note that we do not want to undo all the answers constructed by the continuation call (in fact we want to preserve it), but only the bindings made to the variables which were initially free in the continuation call.

Figure 10 shows some bound variables in the continuation call of Figure 9 before reusing it. Their values are saved in an untrailing list,as shown in Figure 11, and we unbind the original free variables using the list of figure 9. Finally, when the second call is finished, we can make untrailing in the continuation call and restore the state in Figure 10. This trail management is only done when reusing a continuation call, as backtracking would otherwise unbind the initial free variables.

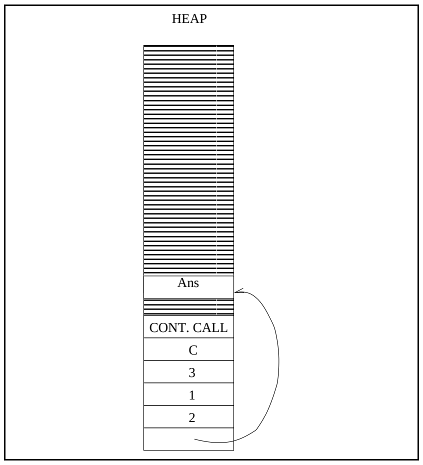

**Fig. 10.** Pre-recall to continuation call

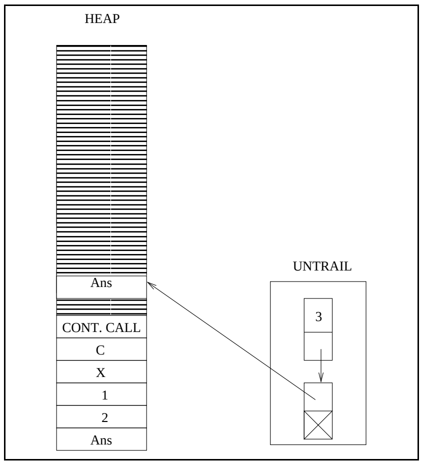

**Fig. 11.** Recall to continuation call

Other systems like CHAT or SLG-WAM do not spend this extra time in preparing a consumer call to be restarted. However, they have an associated

[7] A continuation call G is *reused* when it is being resumed within an execution of a previous resuming of G.

|  |  |
|---|---|
| **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 |
| **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) |

**Table 1.** A terse description of the benchmarks used in the paper

overhead due to the need to record bindings in the forward trail and of reinstalling them (and it is always done, not only for *reused* consumer calls).

### 3.4 Freezing answers

When a consumer is found or when `read_answers/5` is executed a continuation call is created and its 3th variable has to be instantiated using the answers found so far to continue the execution. These answers are, in principle, stored in the table (`answer/4` inserted them), and they have to be constructed on the heap so that the continuation call can access them and proceed with the execution.

The ideas in Section 3.3 can be reused to freeze the answers and avoid the overhead of building them again[8]. 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. When a continuation for some tabled goal is to be executed, the elements of the answer list are unified with the corresponding argument of the continuation call. The list head is, again, accessed through a pointer which is saved in a slot of the corresponding choicepoint and which is updated on backtracking.

In spite of this freezing operation, answers to tabled goals are stored in the table in addition to being linked in a list. 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, and the second one is that the table (which is a trie in our implementation, following [9]) makes checking for duplicate answers faster.

## 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 [13], using the standard, high-level C interface. We

[8] Since there are not *reused* answers trail management is not needed for them.

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 variant incorporates the proposed improvements to the model discussed in Sections 3.3 and 3.4.

We have then evaluated the performance of our proposal using a series of benchmarks which are briefly described in Table 1. The results are shown in Table 2 (in milliseconds). All the measurements have been made using Ciao-1.13 and XSB 3.0.1 compiled with local scheduling and disabling garbage collection in all cases (this in the end did not impact very much the execution times). We used `gcc 4.1.1` to compile both systems, and we executed them on a machine with Fedora Core Linux (kernel 2.6.9).

For reference, we have made an attempt to also compare to the execution times reported in [11]. Due to the difference in technology (Prolog system, C compilers, CPUs, available memory, etc.) it is not possible to compare directly to those execution times. Instead, we took those graph benchmarks which can be executed using SLD resolution and measured their execution times on Ciao-1.13. We then compared these times to those reported in [11] (which were originally executed using SICStus Prolog) and obtained a speed ratio. Finally, we applied this ratio to estimate the execution time that would be obtained for other (tabled) programs by the original implementation in our platform.

These *predicted* times for the original continuation call-based execution (when available) are presented in the second column of Table 2.

The three following columns in the table provide the execution times for the three implementation variants implemented: the direct adaptation the implementation of [13] (our baseline), the results of simplifying the C interface, and the results of implementing the improvements proposed in Sections 3.3 and 3.4. It is reassuring to note that the execution times predicted from those in [11] are within a reasonable range (and with a relatively consistent ratio) when compared to those obtained from our first (baseline) version. We are quite confident, therefore, that they are in general terms comparable, despite the difference in the base system, C compiler technology, implementation of answer tables, etc.

Both lowering the level of the interface to C and improving the transformation for tabling and the way calls are performed have clear impact. It should be also noted that the latter improvement seems to be specially relevant in non-trivial programs which handle data structures[9] 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 our implementation of tabling compares with a state-of-the-art one —namely, the latest version of XSB. In the table we provide, for several benchmarks, the raw time (in milliseconds) taken to execute them using tabling and, when possible, SLD resolution, and the speedup obtained when using tabling, for Ciao and XSB, and the ratio of the execution time of XSB vs. Ciao using SLD resolution and tabling.

[9] The larger data structure are, the more re-copying we avoid.

| **Benchmark** | **Original** | **Ciao Ccal** | **Lower C itf.** | **Copying** |
|---|---|---|---|---|
| lchain 1024 | 8.65 | 7.12 | 2.85 | 2.07 |
| lcycle 1024 | 8.75 | 7.32 | 2.92 | 2.17 |
| rchain 1024 | - | 2620.60 | 1046.10 | 603.44 |
| rcycle 1024 | - | 8613.10 | 2772.60 | 607.68 |
| numbers 5 | - | 1691.00 | 676.40 | 772.10 |
| numbers 5r | - | 3974.90 | 1425.48 | 986.00 |

**Table 2.** Comparison of original implementation and those in Ciao

It should be taken 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 by 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 not present in other Prolog systems (this overhead has been reported to be around a 10% [12]) and also that the priorities of their implementors were understandably more focused on the implementation of tabling.

The speedup obtained when using tabling with respect to SLD resolution (the columns marked $\frac{\text{SLD}}{\text{Tabling}}$) is, in general, favorable to XSB, specially for benchmarks which are tabling-intensive but not resume so much consumers (transitive closure), confirming the advantages of XSB native implementation of tabling. However, and interestingly, the difference in the speedups between XSB and Ciao tends to reduce as the programs get more complex, mix in more SLD execution, XSB forward trail get larger and resume more consumers (specially if answers are large and there is not *reused* continuation calls).

For example, in the `rchain` benchmarks,[10] XSB gets better speedups. However, in the `rcycle N` and `numbers Xr` benchmarks, which are more complex, the difference in speedups between XSB and Ciao is smaller the more complex the execution is (the `numbers Xr` versions generate more consumers than `numbers X`). We attribute this to the forward trailing that XSB uses: when repeatedly generating data structures, XSB needs to keep track of the bindings and reinstall them, while our implementation only performs an initial copy between two memory areas (to have a goal ready to execute) and, since there are not *reused* continuation calls in these programs, it can resume continuations in a constant time. Therefore, answers of `numbers X` and `numbers Xr` are quite large (the full arithmetic expression) and our implementation has them frozen while evaluating a tabled call and XSB implementation does not have.

It is also interesting to note that the final raw speeds (shown in the rightmost column of the table) are in the end somewhat favorable to Ciao in the non-trivial benchmarks, which at least in principle should reflect more accurately what one might expect in larger applications. This is probably due in part to the faster

[10] Which we have to take with a grain of salt, since their executions are in any case quite short.

| Program | Ciao | | | XSB | | | $\frac{\text{XSB}}{\text{Ciao}}$ | |
|---|---|---|---|---|---|---|---|---|
| | SLD | Tabling | $\frac{\text{SLD}}{\text{Tabling}}$ | SLD | Tabling | $\frac{\text{SLD}}{\text{Tabling}}$ | SLD | Tabling |
| rchain 64 | 0.02 | 2.54 | 0.0080 | 0.02 | 0.9 | 0.027 | 1.00 | 0.35 |
| rchain 256 | 0.11 | 37.01 | 0.0027 | 0.11 | 14.4 | 0.008 | 1.00 | 0.39 |
| rchain 1024 | 0.48 | 603.44 | 0.0008 | 0.42 | 216.1 | 0.002 | 0.88 | 0.36 |
| rcycle 64 | - | 2.78 | - | - | 2.1 | - | - | 0.76 |
| rcycle 256 | - | 39.36 | - | - | 35.2 | - | - | 0.90 |
| rcycle 1024 | - | 607.68 | - | - | 650.9 | - | - | 1.07 |
| numbers 3 | 0.56 | 0.63 | 0.88 | 1.0 | 0.7 | 1.43 | 1.79 | 1.11 |
| numbers 4 | 24.89 | 25.39 | 0.98 | 44.4 | 28.7 | 1.55 | 1.78 | 1.13 |
| numbers 5 | 811.08 | 772.10 | 1.05 | 1465.9 | 868.7 | 1.69 | 1.81 | 1.13 |
| numbers 3r | 1.62 | 1.31 | 1.24 | 3.3 | 1.8 | 1.83 | 2.04 | 1.37 |
| numbers 4r | 99.74 | 33.43 | 2.98 | 197.7 | 49.3 | 4.01 | 1.98 | 1.47 |
| numbers 5r | 7702.03 | 986.00 | 7.81 | 15091.0 | 1500.1 | 10.6 | 1.96 | 1.52 |

**Table 3.** Comparing the speed of Ciao and XSB

raw speed of the basic engine in Ciao (which is understandable since the XSB developers have logically concentrated their efforts in the support for tabling) but also implies that the overhead of the approach to tabling used is reasonable after the proposed optimizations. Further work is in any case needed to compare further not only with XSB but also with other systems supporting tabling.

The results are also encouraging to us because they seem to support the "Ciao approach" of starting from a fast and robust, but extensible LP-kernel system and then including 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.

## 5 Conclusions

We have reported on the design and efficiency of some improvements done to the continuation call mechanism of Ramesh and Chen presented in [11]. This mechanism is easier to port to other systems than the SLG-WAM, as it requires minimal changes to the underlying execution engine.

The experimental results show that in general the speedups that the SLG-WAM obtains with respect to SLD execution are better than the ones obtained by our implementation. However, the difference in raw speeds between the systems makes Ciao have sometimes better results in the absolute (and sometimes better convergence results).

To conclude, we think that using an external module implementing tabling is a viable alternative for Prolog systems which want to include tabled evaluation, especially if coupled with the proposed optimizations which we argue not very difficult to implement (all is done in a reusability C library, engine has to be change only to re-interpreter `B[H]` fields).

## 6 Acknowledgements

This work was funded in part by IST programme of the European Commission, FET project FP6 IST-15905 *MOBIUS*, by Ministry of Education and Science (MEC) project TIN2005-09207-C03 *MERIT-COMVERS* and 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 UNM. Ricardo Rocha and Cláudio Silva were partially funded by project Myddas (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC.

## 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. Weidong Chen and David S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.
3. S. Dawson, C.R. Ramakrishnan, and D.S. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems – A Case Study. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 117–126, New York, USA, 1996. ACM Press.
4. Bart Demoen and K. Sagonas. CHAT is $\theta$(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.
5. 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.
6. Bart Demoen and Konstantinos F. Sagonas. Chat: The copy-hybrid approach to tabling. In *Practical Applications of Declarative Languages*, pages 106–121, 1999.
7. 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.
8. 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.
9. I. V. Ramakrishnan, Prasad Rao, K. F. Sagonas, Terrance Swift, and David Scott Warren. Efficient tabling mechanisms for logic programs. In *ICLP*, pages 697–711, 1995.
10. Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1993.
11. 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.

12. 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.
13. C. Silva, R. Rocha, and R. Lopes. An External Module for Implementing Linear Tabling in Prolog. In S. Etalle and M. Truszczyński, editors, *International Conference on Logic Programming*, number 4079 in LNCS, pages 429–430, Seattle, Washington, USA, August 2006. Springer-Verlag.
14. 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.
15. D.S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, 1992.
16. 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.
17. 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.