A Tabling Implementation Based on Variables with Multiple Bindings

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

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

Abstract. Suspension-based tabling systems have to save and restore computation states belonging to OR branches. Stack freezing combined with (forward) trailing is among the better-known implementation approaches for this purpose. Resuming a goal using this technique reinstalls the bindings for all the variables in the environment where the goal was suspended. In this paper we explore an alternative approach where variables can keep track of *several* bindings, associated with suspensions. Resuming a goal boils down to determining which suspension has to be resumed, in order to select, when dereferencing, the bindings which were active at the moment of suspending. We present the ideas behind this approach, highlight several advantages over other suspension-based implementations, and perform an experimental evaluation. We also recall the similarity between OR-parallelism and suspension-based implementations of tabling, and briefly discuss similarities with the some implementation techniques for OR-parallelism.
Keywords: Logic Programming, Tabling, Implementation, Performance, OR-Parallelism.

1 Introduction

Tabling [1–3] is a strategy for executing logic programs which *memoizes* already processed calls and their answers to improve several of the limitations of the SLD resolution strategy. It guarantees termination for programs with the bounded term size property, improves efficiency in programs which repeatedly perform some computation, and has been successfully applied to deductive databases [4], program analysis [5, 6], semantic Web reasoning [7], model checking [8], etc.

There are two main approaches for the implementation of tabling: *suspension-based tabling* and *linear tabling*. In *suspension-based tabling* the computation state of suspended tabled subgoals has to be preserved to avoid backtracking over them. This is done either by *freezing* the stacks, as in XSB [9], by copying to another area, as in CAT [10], or by using an intermediate solution as in CHAT [11]. *Linear tabling* maintains instead a single execution tree without requiring suspension and resumption of sub-computations. The computation of the (local) fixpoint is performed by failing on branches which loop and reexecuting them when there is an answer (obtained from some other branch) for the looping

goal, until no more solutions are found. Examples of this method are the linear tabling of B-Prolog [12, 13] and the DRA scheme [14]. Suspension-based mechanisms achieve very good performance but, in general, they need more memory, used to freeze consumer states and to save answers to be reused. Linear mechanisms, on the other hand, do not use memory to freeze computations, but their efficiency is affected by subgoal recomputation.

The most successful suspension-based implementations of tabling are based on trail management. In these, suspension and resumption operations, which allow stopping the execution in a part of the search tree and restarting it in a different node, use the regular trail and/or a forward trail to record the bindings made in the execution path between two nodes (saved as choicepoints in the corresponding stack) and to remember which bindings have to be reinstalled. This technique can perform speculative work if the bindings which are reinstalled are not used.

In order not to incur in the possible overheads stemming from reinstalling bindings which are not going to be used, we propose an implementation based on using variables with multiple bindings (*Multi Value Binding* variables). A global flag indicating which consumer is active at each moment is used as a *key* to retrieve, from a MVB, the value corresponding to that consumer. Therefore, switching to a consumer is a constant-time operation, triggered by giving this global flag the appropriate value. In turn, and in our current implementation, variable access is not constant-time any more.[3] Herein, we present and evaluate an implementation of this idea.

2 Tabling and Variable Management

We start by providing a brief introduction to tabling. Due to space limitations several details of the implementation of tabling based on suspension are not discussed. For a more complete description, the reader is referred to [9, 11, 15].

Tabling Basics: Tabling changes the operational semantics for predicates marked with the `:- table` declaration. The compiler and runtime system distinguish the first occurrence of a goal marked as tabled (the *generator*) and subsequent variant calls (the *consumers*). The generator applies resolution using the program clauses to derive answers for the goal. When a call identical to a previous one is found,[4] the consumer *suspends* the current execution path (using implementation-dependent means) and starts execution on a different branch.

When 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 previously inserted by

[3] We are not taking into account the dereferencing cost here, assuming instead that it is a constant-time operation which was already present in the system.

[4] Which would enter an infinite loop in SLD resolution.

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

Tabling Implementations Based on Trail Management: We use the CHAT [11] approach in order to show how suspension-based techniques which try to fully reinstall consumer environments can incur the costs of speculative work. We have chosen CHAT because of its simplicity and because we think that the improved version of CHAT presented in [16] is among the most efficient tabling implementations based on trail management.

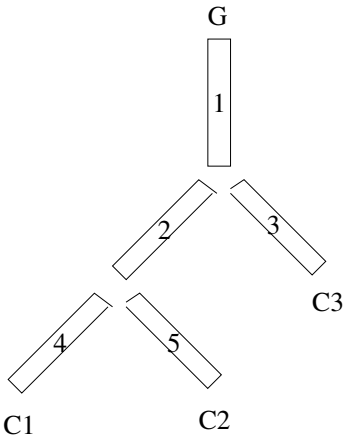

Fig. 1. Sharing trail in CHAT.

CHAT implements suspension by freezing (i.e., protecting by updating pointers) the heap and the local stack and saving the consumer choice points to be reinstalled when the consumer is to restart. Consumers keep track of their conditional bindings (i.e., those bindings appearing in the part of the trail between the consumer and its *leader* generator[5]) to enable them to be reinstalled later on when resuming. Using a tree structure as conditional binding storage (Figure 1), each of these bindings is saved only once, although they are shared between several consumers: the trail of the consumer `C1` is composed of segments 1, 2 and 4, but segment 2 is shared with consumer `C2` and segment 1 is shared also with consumer `C3`. The conclusion is that CHAT, just as SLG-WAM, performs very well with respect to memory usage because it shares all that can be shared and no bindings are saved twice (see [16] for a detailed explanation of CHAT trail management). There is, however, a speculative component of work in CHAT and in all tabling implementations based on trail management. When a consumer restarts, all of its conditional bindings are reinstalled in the stacks. But, in general, not all of these bindings will be necessary in the rest of the execution (some of them might not even be visible at that point). Reinstalling them is, therefore, a waste of work. As an extreme example, let us suppose that a consumer has a (large) number of conditional bindings, which are reinstalled when the consumer is restarted. If

[5] A leader is the generator which marks the following completion point of a consumer. Originally, the leader of a consumer is its generator, but it can change to previous generators if the consumer generator cannot be directly completed because of dependencies.

it fails immediately after that, these bindings will not be accessed and will be immediately untrailed.

3 MVB Tabling

One of the aims of MVB-based tabling is to avoid the speculative work of systems based on trailing. To do that, we have defined a new kind of variable (assigning a new tag, MVB, to it), which can keep several bindings at once. Depending on a global flag, which we will name the `sid` (from *Suspension Identifier*), the correct binding of a variable is accessed. Therefore, dereferencing a variable can, under this approach, be seen as a function:

$$Deref : Variable \times SuspensionId \longrightarrow Value$$

which can retrieve the value a given variable had when a suspension was performed if we provided the identifier of such a suspension.

We now provide more details on how this suspension identifier is managed, and how the multi-value binding variables are accessed and kept up-to-date.

The Management of the Suspension Identifier: the value of the global flag `sid` associated with a normal SLD execution is *zero*. Whenever a consumer appears, a new `sid` is associated with it by incrementing a global counter, `last sid`. When a generator completes, `last sid` and the `sid` global flag are reset to the value they had at the moment in which the generator was created —i.e., a sort of backtracking is performed on the `last sid` when completing generators.

With this scheme, resuming a consumer boils down to changing the value of `sid` to the identifier of the suspension associated with the consumer. Consequently, the bindings accessed through MVB variables will correspond to the bindings existing when the consumer was suspended.

Which variables are MVB? The variables which have to maintain different bindings for each consumer are those appearing in the trail between a consumer and its *leader*, because these bindings would otherwise be lost on backtracking. These bindings are associated with the suspension identifier of the consumer which suspends. For the *zero* suspension identifier the corresponding variables behave as normal WAM variables, and they will be unbound on backtracking.

How is an MVB variable implemented? The implementation of MVB variables is orthogonal to the idea behind them. In our current implementation, MVB variables keep their bindings using a list relating `sid`s with variable values. Traversing this list is necessary in order to determine the value a variable has for some suspension, which makes accessing the value of a variable a non-constant time operation. The (faster) alternative of using an array indexed by suspension identifier may need reserving too much memory in advance. Other possibilities, such as hash tables, are under consideration.

It is important to remark that the suspension identifiers for bindings inserted in the list associated with an MVB are increasing and consecutive, because they

correspond to consumers which have been created on or after the initial creation of the variable. Not all (conditional) variables will necessarily be bound to different values in different suspensions. Additionally, a binding of an MVB can be seen by several consumers, for the same reason that a segment of trail can be shared between different consumers in CHAT. Therefore, we actually compress somewhat the list representation. Instead of a single suspension identifier per node in the list, each binding is associated with a pair of `sid`s which represent a range of suspensions for which the variable had the same value. Thus, each of the elements of the list of an MVB variable is a tuple of the form `<bind,first,last>`, where `bind` is the binding for the variable for the `sid`s between `first` and `last`, both included.

To reduce the impact of non-constant variable access, an MVB variable is equipped with a *cache* where the last accessed value and the range of suspension identifiers for which it is valid are stored. If retrieving from the cache fails (because the `sid` looked up is not available), the binding is searched for in the list and the cache is updated. In addition, the list is kept sorted from the most recent suspensions, with the highest `sid`, to older suspensions, with a lower `sid`. Whenever a new binding is added to the list, its associated `sid` is necessarily greater than all the `sid`s associated with existing bindings, because consumers are generated in a sequential order, and therefore it just has to be added to the front of the list. Keeping the list sorted improves the efficiency of lookups (see later a description of how lookups are done).

If an MVB variable is accessed with a `sid` which does not appear in the MVB list, the cache is updated to point to a new free value, which allows considerable memory savings, since the elements of the list will always be bindings. If that free variable is bound and a suspension is performed later, those bindings will generate a new element of the MVB list.

This MVB representation is illustrated in Figure 6. For example, in heap stack number 5 variable B has created an MVB variable pointing to the cache (which is a free variable associated with `sid` *zero*) and then the value 3 associated with the $\langle 2, 2 \rangle$ range of `sid`s and the value 2 associated with the $\langle 1, 1 \rangle$ range.

Suspensions Nested Inside Resumptions: Assume that `consumer A` is restarted and `consumer B` appears in this restarted execution. There might be MVB variables of `consumer A` which are associated with the `sid` of `consumer A`. If `consumer B` is restarted, some of these variables could be accessed. The right binding for `consumer B` is the same as the binding for `consumer A`, but the `sid` associated with `consumer B` does not belong to that MVB variable and a free variable would be returned instead.

To solve this problem the *dependence* between `consumer A` and `consumer B` has to be remembered. As a consumer can depend directly on at most another consumer, it is enough to have a single field per consumer to record this dependency. This dependency is registered when suspending, and the `sid` of the consumer which suspends is made to depend on the `sid` of the consumer within whose resumption the new consumer has appeared. A `sid` can of course depend on other `sid`s transitively.

```
Term accessMVB(Ref var) {
  int sidAux = sid;
  <value, first, last> = cache(var);
  if (sidAux in [first,last]) return value;
  <value,first,last> = firstElem(var);
  do {
     if (sidAux in [first,last]) {
        if (sid == sidAux) updateCacheMVB(var,<value,first,last>);
        else updateCacheMVB(var,<value,sid,sid>);
        return value;
     }
     if (sidAux > last) {
        sidAux = getDependence(sidAux)
        if (!sidAux) {
           updateCacheMVB(var,<free_var,sid,sid>);
           return free_var;
        }
     }
     else <value,first,last> = nextElem(var);
  } while (hasMoreElem(var));
}
```

Fig. 2. Pseudo-code for MVB access.

As `sid`s are generated in sequential order, a suspension can only depend on another previous suspension, which would have a smaller `sid`. Therefore when the MVB list is accessed, if the `sid S` we are looking for is greater than `last sid` of the element of the MVB list under inspection (which means that it is not going to appear in that list) we can search for the `sid` which `S` depends on, starting at that point in the list. Thanks to the order of `sid`s in the list and the ordering between dependencies, the MVB variable is traversed at most once, even if several dependencies are followed. The code for MVB variable access is shown in Figure 2. Note that we do not advance to the next element if there is a `sid` dependence, because the same element should be inspected again.

When are MVB variables created? MVB variables could be created when suspending a goal, by examining which variables reachable from that goal are conditional and have been recorded in the trail. However, this in the end needs to "simulate" backtracking by traversing the choicepoints and the associated entries in the trail. Since this is going to be done anyway on backtracking, we have decided to actually create them when backtracking from the consumer choicepoints. This saves also work when a binding is shared between several consumers, because regular variables in the shared part are converted into MVB variables only by one of them. This is in some sense similar to trail sharing (see Section 2) in CHAT.

A possible solution is to use a special type of backtracking for the choicepoints associated with tabled execution. When a consumer suspends, the choicepoints between that consumer and the generator have to be marked to reflect the initial

```
if (!TrailYounger(trailPointer, top(MVBstateStack)) {
  createOrUpdateMVB(trailPointer);
  trailPointer = trailPointer − 1;
  top(MVBstateStack) = trailPointer;
  if (top(MVBstateStack) == pre_top(MVBstateStack)) pop(MVBstateStack);
}
else Untrail(trailPointer);
```

Fig. 3. Pseudo-code for MVB untrail.

suspension they belong to. This is done by scanning the choicepoint stack from the topmost choicepoint (i.e., the one corresponding to the suspending consumer) until an already marked choicepoint (which belongs to the execution of another suspended consumer) is found. On backtracking, MVB variables would have to be created and associated with a range of suspensions. The `sid` range associated with these variables is the one which goes from the mark associated with the choicepoint where that variable was trailed to that of the last consumer which suspended. This is, however, difficult to implement in systems which discard choicepoints just before the last alternative is taken (this includes our implementation platform, Ciao [17]): in such cases the bindings of such an alternative would be wrongly associated with the `sid` of the previous choicepoint `sid` mark. Non-trivial changes to the stack management have to be done to avoid this (i.e., not removing a choicepoint when the last alternative is taken), and a memory optimization will be lost.

Our solution is implemented in the untrail operation, and the idea is to use an additional stack (the `MVBstateStack`) which will implement a mechanism similar to the choicepoint-based one, but where the `MVBstateStack` keeps track only of the fields which would otherwise go in the choicepoints. This makes it possible to make these fields survive choicepoint removal without having to fiddle around with the choicepoint management and keeping its "last alternative optimization."

Each time a consumer suspends, the pointer to the top of the trail is pushed onto the `MVBstateStack`, and it is associated with the `sid` of the consumer which suspends. When performing untrailing, if the trail pointer is equal to the value of the topmost element of `MVBstateStack`, we create an MVB variable (or insert a new binding in the MVB list if it was already created) and the current binding is associated with the suspension range from the `sid` corresponding to the topmost element of `MVBstateStack` and the last consumer which suspended, `last sid`. The trail pointer stored at the top of the `MVBstateStack` is then decremented, and if it is equal to the previous element in `MVBstateStack`, it is popped out because the following value to be untrailed is also shared with previous consumers.

The code of the new untrail operation is shown in Figure 3. This can obviously be made more efficient, but we are showing a simple version for clarity.

When Are MVB Variables Removed? When a generator is completed, none of the MVB variables (or, more precisely, none of their bindings) created

within its execution are needed any longer. As we want to interfere as little as possible with the existing backtracking / untrailing mechanism, each time an MVB variable is created (or updated) it is recorded in a list associated with the last generator. On generator completion, the bindings added under the subtree of that generator are removed. If all of the elements of the MVB list are removed, the MVB variable is made a regular and free variable again.

Freezing Stacks: The way choice points, the heap, and the local stack are managed in the MVB approach is just the same as in CHAT with the improvements in [16]. The main difference is the management of the trail. CHAT saves the values pointed from the trail of each consumer to reinstall them when resuming, and MVB tabling uses MVB variables to do that.

Changes to the Prolog Virtual Machine: The changes to be done to a Prolog engine are a special untrail operation (Figure 3), an additional case for the dereferencing routine in order to make it understand MVB variables[6] (Figure 2), and the changes needed to freeze stacks *à la* CHAT. In the case of Ciao, a WAM instruction also has to be modified: `get_first_local_value` gives the first value to a local variable. As it just checks if the local variable was or not initialized to a stack variable, an MVB variable living there could be rewritten. The new `get_first_local_value` instruction checks if the previous value is an MVB variable to make the assignment without destroying the MVB information. The assignment is stored in the cache of the MVB variable.

These changes are in our experience quite local and easy to do, which allows us to conclude that MVB tabling-based implementation is not hard.

4 MVB Tabling Execution

We will try to clarify the MVB tabling approach presented previously using a simple tabled program (Figure 4). The tabled execution tree of this program (not specifically for MVB tabling) is shown in Figure 5, and Figure 6 shows the creation of MVB variables. Figure 7 shows the management of the `MVBstateStack` to create MVB variables to be shared between several consumers.

We start with the query `p(X)`. Execution starts with a global `sid` of *zero*. `A` and `B` are created as unbound variables in Figure 6 (1), and they are unified with `A = 1` and `B = 2` (Figure 6 (2)) before the execution is suspended because a consumer is found (step 3). The suspension identifier associated with this consumer is `sid` = 1, and last `sid` is updated to be 1.

Figure 7 (1) shows the entries in the trail for `A,B` and the record to be inserted in `MVBstateStack`. Each consumer inserts a pair `<trail_pointer, consumer_sid>` in the `MVBstateStack` —in this case, it is the pair `<2,1>`.

Execution fails then and backtracks over the last choice point (2). As the `trail pointer` is equal to the value of the trail stored at the top of the `MVBstateStack`, an MVB variable is created and associated with the range from the `sid` of the

[6] Which in our case are marked with a special tag.

```
:− table p/1.

p(X) :−
    A = 1,
   (B = 2; B = 3),
    p(X),
    A = B.

p(1).
```

Fig. 4. Tabled program.

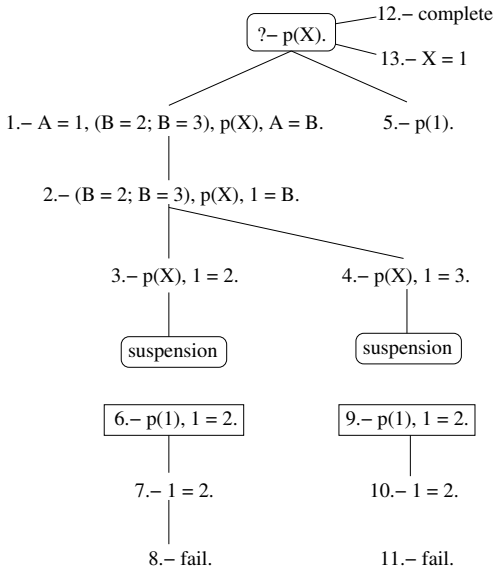

Fig. 5. MVB tabling execution.

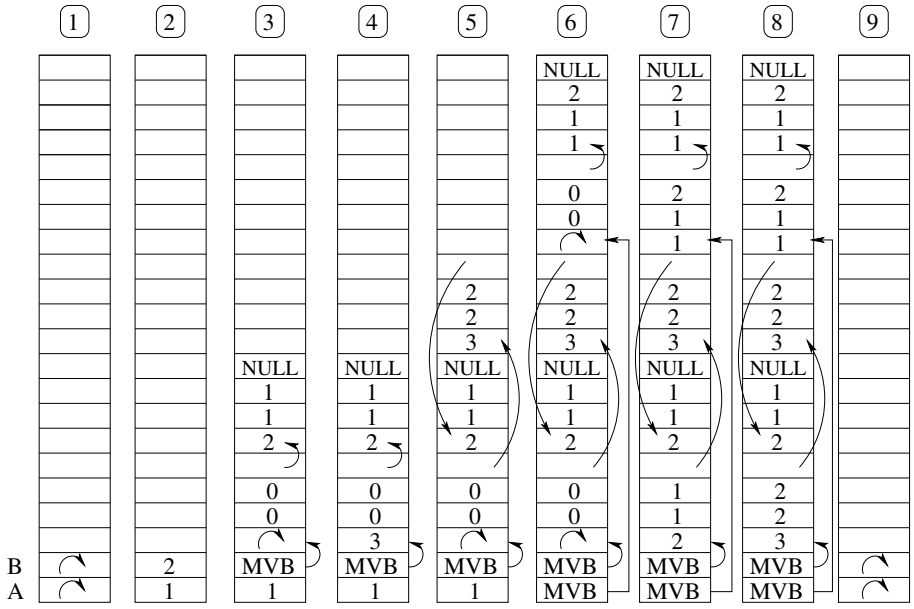

Fig. 6. MVB variables.

topmost element of `MVBstateStack` to the `last sid` variable, which is 1 (Figure 6 (3)). Besides, the value of the trail pointer in the topmost element of `MVBstateStack` is decremented (Figure 7 (2)). Recall that free variables are cre-

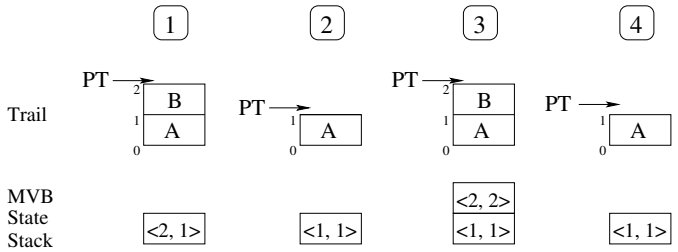

Fig. 7. MVB trail management.

ated in the cache associated with `sid` *zero*. This corresponds to the (unbound) variable which would have been restored on backtracking. The previous binding is maintained as an MVB list element.

After backtracking, `B = 3` (step 4) is executed and the cache of the MVB variable is updated (Figure 6 (4)). A new consumer is found which is associated with the `sid 2`, and `last sid` is updated to be 2. A new element is inserted in `MVBstateStack`, `<2,2>`, which represents the current `trail pointer` and the new consumer `sid` (Figure 7 (3)). Execution fails and backtracks over the second clause of `p/1`. When `B` is untrailed, its MVB variable is updated to the value `B = 3` and it is associated with the `sid`s (2,2) (Figure 6 (5)), inserted in descending order, as explained in Section 3.

To know the `sid`s that **B** is associated with, the topmost item of `MVBstateStack` is used. When the trail value stored there is decremented, it reaches the same value as the previous element in `MVBstateStack`, and the top of the stack is popped out. (Figure 7 (4)). Now, variable `A` is untrailed, and a new MVB variable is created (Figure 6 (6)), but it is associated with `sid`s (1,2) because `last sid` is 2 and the topmost element of `MVBstateStack` represents `sid` 1.

The second clause of `p/1` is executed and the first answer, `p(1)`, is found (step 5). It is inserted in an external table (as all tabling implementations do), and then consumers can be restarted with this answer. In order to do that, the `sid` variable is updated to be the `sid` associated with the consumer being restarted: for example, when consumer 1 is restarted (step 6) and variable `A` is accessed, the cache of its MVB variable is updated with the value associated with `sid` 1, and the same for variable `B` (Figure 6 (7)). When consumer 2 is restarted (step 9), variable `A` is accessed in the cache, but the cache of variable `B` has to be updated (Figure 6 (8)). Finally, the generator `p(X)` can be completed (step 12) and all of the MVB variables created under its execution tree are unbound (Figure 6 (9)). The `sid` and `last sid` variables are updated to the value just before the generator execution (*zero*, in this case) and the answers found and stored in the answer table are returned on backtracking (step 13).

5 Performance Evaluation

In the following two sections we will analyze the performance of our MVB scheme and CHAT, both from a theoretical and an experimental point of view.

5.1 CHAT and MVB — a Conceptual Comparison

In any realization of CHAT and MVB, some essential operations will remain largely untouched regardless of the implementation details. We will focus on these to compare CHAT and MVB at a high abstraction level.

MVB and CHAT freeze the heap and local stacks using the same techniques, and they store consumer choice points and answers using also a similar approach; therefore, their memory consumption should be similar as well. Even the cost of saving the trail of a consumer can be comparable with the cost of creating the MVB variables of such a consumer. Memory consumption should be in the same order too: for each trailed value, CHAT uses 2 slots (value and pointer), and MVB uses 4 slots (value, initial state, last state, and a pointer to the next list element). In our view, the main differences between both approaches are:

- CHAT reinstalls (speculatively) the conditional bindings of consumers.
- MVB variable access is affected by MVB variable indirection.

Although we are using an MVB cache, sometimes the MVB list has to be traversed to find the right MVB bindings for the current `sid`.[7] Consequently, MVB should outperform CHAT when the speculative work of reinstalling the conditional bindings of a consumer is larger than the cost of the overhead due to the MVB variable access, which we will experimentally measure in Section 5.2 for a number of common benchmarks.

This means that *artificial* code can be created where large parts of the trail are saved in a consumer to be later reinstalled and not really used. Therefore MVB could be arbitrarily better than CHAT in this example. On the other hand, code can also be written where MVB variables have a large amount of bindings and they continuously suffer from cache misses. In this case, MVB could perform arbitrarily worse than CHAT. The question therefore remains: in practice, and assuming similarly involved implementation techniques for, e.g., CHAT and MVB, how much does MVB variable handling (including dereferencing), the special untrailing, and the additions to backtracking affect both tabled and regular (SLD) execution.

5.2 Experimental Evaluation

We have implemented the techniques proposed in this paper as an extension of the Ciao system [17]. All of the timings and measurements have been made with Ciao-1.13, using the standard, unoptimized bytecode-based compilation, and with the `MVB` extensions loaded. For XSB we have used XSB 3.1. All the executions were performed using local scheduling and disabling garbage collection. We used `gcc 4.1.1` to compile all the systems, and we executed them on a machine with Debian Linux 5.0, kernel 2.6.18, and an Intel Xeon DESCHUTES processor.

[7] A more efficient mechanism for accessing variable bindings is possible (Section 6), but practical experiments make us doubt about its real usefulness (Section 5.2).

Measure	sgm	atr2	pg	kalah	gabriel	disj	cs_o	cs_r	peep	Avg.
Cache misses	1504	2545	147	100	155	103	33	54	335	—
Cache misses (%)	1%	7%	16.6%	17.8%	9%	9.5%	6.5%	5.3%	5.7%	8.7%
Avg. MVB length	30.6	1	3.5	1.4	1.8	1.3	1.2	1.2	2.1	4.9
Avg. list trav.	15.8	1	11.5	1.3	3.8	2.5	1.8	1.7	5.4	5

Table 1. Some statistics on the dynamic behavior of MVB variables.

Impact of MVB on SLD Execution: A question to ask is to what extent the changes we have introduced in the Prolog machinery (e.g., special trailing, extra cases in dereferencing, changes in one WAM instruction) impact the speed of non-tabled execution. We have measured this using the ECRC set of analytical benchmarks[8] which test different characteristics of Prolog execution using dedicated benchmarks. In our experiments, enabling or disabling the MVB extensions did not have any measurable impact on SLD execution speed.

Impact of MVB on Variable Access We have measured also to what extent having to traverse a list of bindings (even with the improvement of a cache for the most recently accessed value — in the case of a cache hit the value is accessed in constant time) can impact accessing a given value for a consumer. This is difficult to predict as it depends, for each benchmark and variable, on how many conditional bindings for that variable are made by the different consumers.

To measure this, we have instrumented our implementation to count the number of value cache misses, the percentage of cache misses with respect to the total accesses, the average length of the chain of values, and the average number of items traversed in this list for each cache miss.[9] The statistics are shown in Table 1. The main conclusions we can draw are: even if the number of consumers is in principle unknown and can be very large, value chains are usually rather small, which suggests that an implementation with direct indexing may not in the end bring big advantages. Moreover, the benchmark with the longest value chains (**sgm**) has as well the best cache behavior: only 1% of the accessed values were not in the cache. The cache behavior is in general reasonable in the rest of the benchmarks as well.

General Performance Assessment Table 2 aims at determining how the proposed implementation of tabling compares with other tabling implementations. To that end we have implemented a CHAT tabling in Ciao, in order to have a system with a comparable base speed and a similar code maturity. We are also comparing with XSB, arguably the most successful tabling system based on trailing. We have used a set of benchmarks which appear in other performance evaluations of tabling approaches.

In this table we provide, for several benchmarks, the raw time (in milliseconds) taken to execute them using tabling. As these benchmarks do not create

[8] Available as a Ciao Prolog 1.13 library and also at `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/lang/prolog/code/library/`

[9] This can be larger than the average list length because value searches can concentrate on lists with lengths over the average.

large memory structures, the differences between CHAT and MVB are not as large as could be using bigger benchmarks; expanding this assessment is part of our pending work. Besides, the code quality of both implementations (CHAT and MVB) can still be improved, as well as that of the consumer scheduler, MVB representation, etc. However, in general we think that these two implementations are at a similar level of maturity and should be comparable in terms of speed. Of course any improvement in them would bring a competitive advantage with XSB.

The results are in general quite encouraging: speed results are very similar both to CHAT and to XSB, which makes the technique competitive. This provides confidence that an improved implementation (for example, tabling primitives are not yet compiled into WAM code and still have to traverse the Prolog-C interface as in [15]), the internal representation for MVB can be improved, and goal scheduling is still simplistic and does not try to favor our technique by decreasing the probability of cache misses) can make MVB a viable technique for tabling which does not need very complicated stack management and which can compete with state-of-the-art systems.

Program	MVB	CHAT	XSB
sgm	1806	1905	1649
atr2	339.2	353.4	351.0
pg	13.11	13.20	12.03
kalah	19.23	18.82	16.77
gabriel	19.83	19.39	18.42
disj	15.19	15.12	14.02
cs_o	29.18	29.28	25.30
cs_r	58.19	57.80	51.03
peep	60.01	59.20	52.10

Table 2. MVB vs. CHAT vs. XSB.

6 Tabling and Implementation Techniques for Or-Parallelism

The basic problem of Or-parallel systems is "how to represent different bindings of the same variable corresponding to different branches of the search space" [18]. This is of course a concern shared with suspension-based systems for tabling, where suspending and resuming a goal basically has to resort to a representation which makes it possible to save and recover bindings existing at some other part of the search tree. This has been recognized in early work [19]. The similarity between Or-Parallel and tabling using complex trail and stack management (e.g., implementations of the SRI model [20] and SLG-WAM) and those relying on copying (e.g., CHAT and the MUSE [21]) have been mentioned elsewhere [11].

However, to the best of our knowledge, variable access has remained largely untouched in all systems implementing tabling, when, from an abstract point of view, making a variable access different values depending on the environment (e.g., `sid` global flag) which the variable is seeing is a fundamental operation. This has been tackled by installing as a "solid block" all the bindings a consumer has to see, instead of using a *switch* to change the viewpoint of the consumer. This is precisely what systems such as Aurora [22] did —in this case by main-

taining variables as indexes on a binding array with different entries for each processor.

This is not radically different from out approach. In both cases the function in Section 3 is implemented. However, Aurora first discriminates on the second function argument (to select the worker) and then on the first (to select the variable). In our case we take first the variable to dereference, and then the consumer inside whose environment it has to be evaluated; that matches what the Version Vectors model [23] does. While the relation of Or-Parallelism and tabling has been studied before, we think that a deeper connection can be established, which can bring implementation techniques and, e.g., scheduling algorithms developed in the Or-Parallel realm to tabled systems, and also make possible new implementations which seamlessly exploit these relationships.

7 Conclusions

We have presented and evaluated an implementation technique for tabling based on keeping simultaneously several bindings for variables (MVB), corresponding to the environments of the consumers. From the experiments we can conclude that, while theoretically MVB can be arbitrarily better or worse than CHAT, in practice it is a viable way of avoiding some speculative work inherent to trailing-based implementations of tabling. Although our implementation can still be improved in several directions, the performance we obtain is already acceptable and comparable with state-of-the-art systems. Besides, we recall the similarity between OR-parallelism and suspension-based implementations of tabling, and conclude that some amount of cross-fertilization is still possible.

Acknowledgments: This work was funded in part by EU projects FET IST-15905 *MOBIUS*, IST-215483 *SCUBE*, FET IST-231620 *HATS*, and 06042-ESPASS, MICINN projects TIN-2008-05624 *DOVES*, TIN2005-09207-C03 *MERIT-COMVERS*, MIN project FIT-340005-2007-14, and CAM project S-0505/TIC/0407 *PROMESAS*. Pablo Chico is also funded by an FPU Spanish MICINN scholarship.

References

1. Warren, D.S.: Memoing for logic programs. Communications of the ACM **35**(3), pp. 93–111 (1992)
2. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43**(1), pp. 20–74 (January 1996)
3. Tamaki, H., Sato, M.: OLD resolution with tabulation. In: Int'l. Conf. on Logic Programming, pp. 84–98. LNCS, Springer-Verlag (1986)
4. Ramakrishnan, R., Ullman, J.D.: A survey of research on deductive database systems. Journal of Logic Programming **23**(2), pp. 125–149 (1993)
5. Warren, R., Hermenegildo, M., Debray, S.K.: On the Practicality of Global Flow Analysis of Logic Programs. In: Fifth International Conference and Symposium on Logic Programming, pp. 684–699. MIT Press (August 1988)

6. Dawson, S., Ramakrishnan, C.R., Warren, D.S.: Practical Program Analysis Using General Purpose Logic Programming Systems – A Case Study. In: Proceedings of PLDI'96, New York, USA, pp. 117–126. ACM Press (1996)
7. Zou, Y., Finin, T., Chen, H.: F-OWL: An Inference Engine for Semantic Web. In: Formal Approaches to Agent-Based Systems. Volume 3228 of Lecture Notes in Computer Science., pp. 238–248. Springer Verlag (January 2005)
8. Ramakrishna, Y., Ramakrishnan, C., Ramakrishnan, I., Smolka, S., Swift, T., Warren, D.: Efficient Model Checking Using Tabled Resolution. In: Computer Aided Verification. Volume 1254 of LNCS., pp. 143–154. Springer Verlag (1997)
9. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems **20**(3), pp. 586–634 (May 1998)
10. Demoen, B., Sagonas, K.: CAT: The Copying Approach to Tabling. In: Programming Language Implementation and Logic Programming. Volume 1490 of Lecture Notes in Computer Science., pp. 21–35. Springer-Verlag (1998)
11. Demoen, B., Sagonas, K.: CHAT: the copy-hybrid approach to tabling. Future Generation Computer Systems **16**, pp. 809–830 (2000)
12. Zhou, N.F., Shen, Y.D., Yuan, L.Y., You, J.H.: Implementation of a linear tabling mechanism. J. of Functional and Logic Programming **2001**(10), (October 2001)
13. Zhou, N.F., Sato, T., Shen, Y.D.: Linear Tabling Strategies and Optimizations. Theory and Practice of Logic Programming **8**(1), pp. 81–109 (2008)
14. Guo, H.F., Gupta, G.: A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In: International Conference on Logic Programming. pp. 181–196. (2001)
15. de Guzmán, P.C., Carro, M., Hermenegildo, M.: Towards a Complete Scheme for Tabled Execution Based on Program Transformation. In: Int'l. Symp. on Practical Aspects of Declarative Languages (PADL'09). Number 5418 in LNCS, pp. 224–238. Springer-Verlag (January 2009)
16. Demoen, B., Sagonas, K.: CHAT is θ(SLG-WAM). In: Int'l. Conf. on Logic for Programming and Automated Reasoning. Volume 1705 of LNCS., pp. 337–357. Springer (September 1999)
17. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., (Eds.), G.P.: The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM) (2006) Available at `http://www.ciaohome.org`.
18. Warren, D.H.D.: OR-Parallel Execution Models of Prolog. In: Proceedings of TAPSOFT '87. Lecture Notes in Computer Science. Springer-Verlag (March 1987)
19. Warren, D.S.: Efficient Prolog Memory Management for Flexible Control Strategies. In: International Symposium on Logic Programming, Silver Spring, MD, Atlantic City, pp. 198–203. IEEE Computer Society (February 1984)
20. Warren, D.H.D.: The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In: Symp. on Logic Prog. pp. 92–102. (August 1987)
21. Ali, K., Karlsson, R.: The MUSE Approach to Or-Parallel Prolog. International Journal of Parallel Programming **19**(2), pp. 129–162 (1990)
22. Lusk, E., Butler, R., Disz, T., Olson, R., Stevens, R., Warren, D.H.D., Calderwood, A., Szeredi, P., Brand, P., Carlsson, M., Ciepielewski, A., Hausman, B., Haridi, S.: The Aurora Or-parallel Prolog System. New Generation Computing **7**(2/3), pp. 243–271 (1988)
23. Hausman, B., Ciepielewski, A., Haridi, S.: Or-Parallel Prolog Made Efficient on Shared Memory Multiprocessors. In IEEE, ed.: Symposium on Logic Programming, SICS pp. 69–79. (August 1987)