A Simple Table Space Design for Retroactive Call Subsumption

Flávio Cruz and Ricardo Rocha

CRACS & INESC TEC, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
{flavioc,ricroc}@dcc.fc.up.pt

Abstract. Tabling is an implementation technique where answers from subgoals are stored in a table space area in order to be reused later by *similar subgoals*. Most tabling engines use *call by variance* to test subgoal similarity by means of simple variable renaming. *Call by subsumption* is a more sophisticated similarity test where subsumed subgoals can use answers from subsuming subgoals, therefore increasing answer reuse and attaining better execution times in general. However, call by subsumption is highly dependent on the call order of the subgoals. A recent strategy, called *Retroactive Call Subsumption* (RCS), supports call by subsumption by allowing full sharing of answers between subsumed/subsuming subgoals, independently on the order in which they are called. For this strategy, we propose a new table space design, the *Single Time Stamped Trie* (STST), that makes answer sharing across subsumed/subsuming subgoals simple and efficient. In this paper, we present the STST design and how it fits within the RCS framework. In experimental results, we observed some overheads in programs that stress the drawbacks of STST when RCS is not applied, but its design's simplicity outweighs these disadvantages when programs take advantage of RCS evaluation.

Keywords: Logic Programing, Tabled Evaluation, Call Subsumption.

1 Introduction

Tabling is an evaluation technique for Prolog systems that has several advantages over traditional SLD resolution: reduction of the search space, elimination of loops, and better termination properties [1]. In a nutshell, tabling works by storing found answers in a memory area called the *table space* and then by reusing those answers in *similar calls* that appear during the resolution process. First calls to tabled subgoals are considered *generators* because they are evaluated as usual and their answers are stored in the table space. Similar calls are named *consumers* because, instead of executing program code, they are evaluated by consuming the answers stored in the table space for the similar generator subgoal. There are two main approaches to determine if two subgoals A and B are similar:

– *Variant-based tabling* (or *tabling by call by variance*): A and B are variants if they are identical up to variable renaming. For example, $p(X, 1, Y)$ is a *variant* of $p(W, 1, Z)$ because both can be made identical to $p(VAR_0, 1, VAR_1)$;

– *Subsumption-based tabling* (or *tabling by call by subsumption*): A is considered similar to B if A is *subsumed* by B (or B *subsumes* A), i.e., if A is more specific than B (or an instance of). For example, subgoal $p(X, 1, f(a, b))$ is subsumed by subgoal $p(Y, 1, Z)$ because there is a substitution $\{Y = X, Z = f(a, b)\}$ that makes $p(X, 1, f(a, b))$ an instance of $p(Y, 1, Z)$. Tabling by call by subsumption is based on the principle that if A is subsumed by B and S_A and S_B are the respective answer sets, therefore $S_A \subseteq S_B$. Please notice that when using some extra-logical features of Prolog, such as the `var/1` predicate, this assumption may not hold and thus call by subsumption should not be used as it can produce wrong results.

Because subsumption-based tabling can detect a larger number of similar subgoals, variant and subsumed subgoals, it allows greater answer reuse and thus better space usage, since the answer sets for the subsumed subgoals are not stored. Moreover, subsumption-based tabling has also the potential to be more efficient than variant-based tabling because the search space tends to be reduced as less code is executed [2]. Despite all these advantages, the more strict semantics of subsumption-based tabling and the challenge of implementing it efficiently makes variant-based tabling more popular among the available tabling systems.

XSB Prolog was the first Prolog system providing support for subsumption-based tabling by introducing a new data structure, the *Dynamic Threaded Sequential Automata (DTSA)* [3], that permits incremental retrieval of answers for subsumed subgoals. However, the DTSA design had one major drawback, namely, the need for two data structures for the same information. A more space efficient design, called *Time-Stamped Trie* (TST) [2, 4], solved this by using only one data structure. Despite the advantages of subsumption-based tabling, the degree of answer reuse might depend heavily on the call order of the subgoals. To take effective advantage of subsumption-based tabling in XSB, the more general subgoals should be called before the specific ones. When this does not happen, answer reuse does not occur and Prolog code is executed for both subgoals.

Recently, a new design called *Retroactive Call Subsumption (RCS)* [5], extended the original TST approach by also allowing sharing of answers when a subsumed subgoal is called before a subsuming subgoal. This extension enables answer reuse independently of the subgoal call order and therefore increases the usefulness of subsumption-based tabling. In a nutshell, RCS works by selectively pruning the evaluation of subsumed subgoals when a more general subgoal is called and then by restarting the evaluation of the subsumed subgoal by turning it into a consumer node of the more general subgoal. To implement RCS the following components are needed: (i) new control mechanisms for retroactive-based evaluation; (ii) an algorithm to efficiently retrieve subsumed subgoals of a subgoal from the table space; and (iii) a new table space organization that facilitates the sharing of answers between subsuming/subsumed subgoals. In this paper, we present a new table space design, named *Single Time-Stamped Trie (STST)*, to support the last requirement. We will focus our discussion on the concrete implementation we have done on the YapTab tabling system [6].

The remainder of the paper is organized as follows. First, we briefly discuss the background concepts behind tabling and the table space and we describe how RCS works through an example. Next, we present the STST design and discuss the main algorithms for answer insertion and retrieval and how the support data structures are laid out. Then, we analyze the table space using several benchmarks to stress some properties of the STST design. Finally, we end by outlining some conclusions.

2 Tabling in YapTab

Tabling is an implementation technique that works by storing answers from first subgoal calls into the table space so that they can be reused when a similar subgoal appears. Within this model, the nodes in the search space are classified as either: *generator nodes*, if they are being called for the first time; *consumer nodes*, if they are similar calls; or *interior nodes*, if they are non-tabled subgoals. In YapTab, we associate a data structure called *subgoal frame* for each generator node and a data structure named *dependency frame* for each consumer node. These two data structures are pushed into two different stacks that are used during tabled evaluation.

2.1 Tabling Operations

In YapTab, programs using tabling are compiled to include *tabling instructions* that enable the tabling engine to properly schedule and extend the SLD resolution process. For both variant-based and subsumption-based tabling, we can synthesize the tabling instruction set into four main operations:

Tabled Subgoal Call: this operation inspects the table space looking for a subgoal S similar to the current subgoal C being called. For call by variance, we check if a variant subgoal exists, and for call by subsumption, we check for subsuming and variant subgoals. If a similar subgoal S is found, C will be resolved using *answer resolution* and for that it allocates a consumer node and starts consuming the set of available answers from S. If no such S exists, C will be resolved using program clause resolution and for that it allocates a generator node and adds a new empty entry to the table space.

New Answer: this operation checks whether a newly found answer A for a generator node C is already in the table space. If A is a repeated answer, the operation fails. Otherwise, A is stored as an answer for C.

Answer Resolution: this operation checks whether a consumer node C has new answers available for consumption. For call by variance, we simply check if new answers are available in the variant subgoal S, but for call by subsumption, we must determine the new relevant answers for C that were stored in the subsuming subgoal S. If no unconsumed answers are found, C *suspends* and execution proceeds according to a specific strategy [7]. Consumers must suspend because new answers may still be found by the corresponding variant/subsuming subgoal S that is executing code.

Completion: this operation determines whether a subgoal S is completely evaluated. If this is not the case, this means that there are still consumers with unconsumed answers and execution must then proceed on those nodes. Otherwise, the operation marks S as completed since all answers were found. Future variant or subsumed subgoal calls to S can then reuse the answers from the table space without the need to suspend.

2.2 Table Space

Due to the nature of the previously described tabling operations, the table space is one of the most important components in a tabling engine, since the lookup, insertion and retrieval of subgoals and answers must be done efficiently. The most successful data structure used to implement the table space is the *trie* [8], a tree-like data structure where common prefixes are represented only once. Figure 1 shows an example of using tries to represent terms. In a tabling setting, tries are used in two levels: the first level is composed of *subgoal tries*, where each trie stores subgoal calls for the corresponding tabled predicate; in the second level, we have *answer tries*, where each trie stores the answers for the corresponding subgoal call.

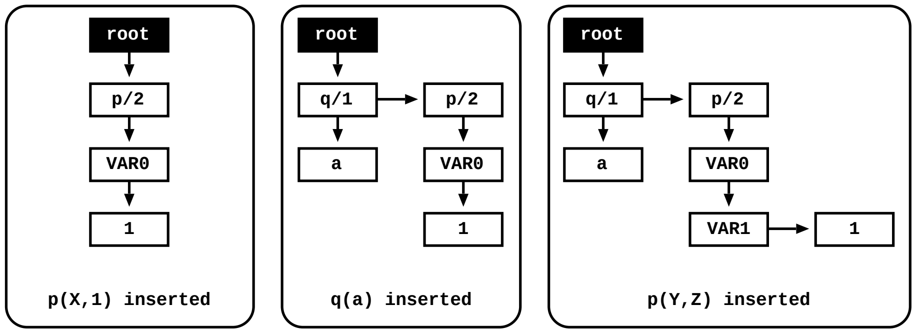

Fig. 1. Using tries to represent terms

In variant-based tabling, each tabled predicate has a *table entry* data structure that contains information about the predicate and a pointer to the subgoal trie. A trie leaf node in the subgoal trie corresponds to a unique subgoal call and points to a data structure called the *subgoal frame*. The subgoal frame contains information about the subgoal, namely, the state of the subgoal and a pointer to the corresponding answer trie. In subsumption-based tabling based on the TST design, we have *subsumptive subgoal frames*, for subgoals that generate answers, and *subsumed subgoal frames*, for subgoals that consume answers from subsumptive subgoals [2].

Subsumptive subgoal frames are similar to variant subgoal frames, but they point to a *time-stamped answer trie* instead, which is an answer trie where each trie node is extended with *timestamp* information. Consider, for example, the subgoal call `p(VAR0,VAR1)` and the time-stamped answer trie in Fig. 2. The trie stores 2 answers, `<f(x),1>` inserted with timestamp 1 and `<10,[]>` inserted with timestamp 2. The root node contains the *predicate timestamp* and is incremented every time a new answer is inserted. Consider now that we insert the answer `<f(y),1>` (see Fig. 3). For this, we increment the predicate timestamp to 3 and then we set the timestamp of each node on the trie path of the new answer also to 3. Notice that if we look at leaf nodes we are able to discern in which order the answers were inserted, because each new answer is numbered incrementally.

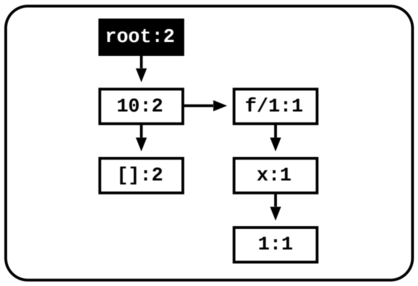

Fig. 2. A time-stamped answer trie

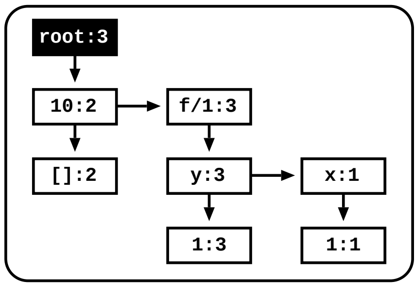

Fig. 3. Inserting the answer `<f(y),1>`

Subsumed subgoal frames store a pointer to the corresponding subsumptive subgoal frame (the more general subgoal) instead of pointing to their own answer tries. The frames have a *answer return list*, a list of pointers to the relevant answers in the subsumptive answer trie and a *consumer timestamp* used for *incremental retrieval* of answers from the subsumptive answer trie. To consume answers, a subsumed subgoal first traverses its answer return list checking for more answers, and then executes a retrieval algorithm in the subsumptive answer trie in order to collect the answers with newer timestamps, which are then added to the answer return list. As an example, consider again the answer trie in Fig. 3 owned by subgoal `p(VAR0,VAR1)` and that we are interested in the incremental retrieval of relevant answers for the consumer subgoal `p(VAR0,1)`. For this, we need to do a depth-first search on the answer trie using the consumer timestamp as a filter to ignore already retrieved answers, as we are only interested in answers that were added after the last retrieval operation. Assuming that the consumer timestamp was 1 (meaning that the answer `<f(x),1>` was already retrieved in a previous step), we would retrieve the answer `<f(y),1>` and add it to the answer return list to be consumed next.

3 Retroactive Call Subsumption

RCS is an extension to subsumption-based evaluation that enables answer reuse independently of the call order of the subgoals. While tabling by call by subsumption only allows sharing of answers when a subsumed subgoal is called after a subsuming subgoal, RCS works around this drawback by selectively pruning the evaluation of subsumed subgoals and by turning them into consumers [5].

Let's consider a subgoal R that is subsumed by a subgoal S. To do retroactive evaluation, we must prune the evaluation of R, first by knowing which parts of the execution stacks are involved in its computation and then by transforming the choice point associated with R into a consumer node, in such a way that it will consume answers from the subsuming subgoal S, instead of continuing the execution as a generator. A vital part in this process is that we need to know the set of answers A_{old}, that were already computed by R, so that, when we transform R into a consumer we only consume the set of answers A_{new}, that will be created by S. In other words, we must ensure that the final set of answers A for R is $A = A_{new} \cup A_{old}$ with $A_{new} \cap A_{old} = \emptyset$. If we do not obey this principle, the evaluation will not be wrong, but several execution branches will be executed more than once, thus eliminating the potential advantage of RCS evaluation.

In RCS, we consider two types of pruning of subgoals. The first type is *external pruning* and occurs when S is an *external subgoal* to the evaluation of R. The second one is *internal pruning* and occurs when S is an *internal subgoal* to the evaluation of R. Both cases are very similar in terms of the challenges and problems that arise when doing pruning. Here, we present an example of external pruning that will be enough to present the challenges in designing a good table space for RCS. Consider thus the query goal '`?- r(1,X), r(Y,Z)`' and the following program.

```
:- use_retrosubsumptive_tabling r/2.
r(1,a).
r(Y,Z) :- ...
```

Execution starts by calling `r(1,X)`, which creates a new generator to execute the program code, and a first answer for `r(1,X)`, `<1,a>`, is found. In the continuation, `r(Y,Z)` is called, which will be a subsumptive subgoal for `r(1,X)`. Thus, `r(1,X)` needs to be pruned and turned into a consumer of `r(Y,Z)`. To prune, first we turn the node of `r(1,X)` into a retroactive node. Later, when backtracking to `r(1,X)`, the retroactive node will be transformed into either a loader[1] or a consumer node according to if, in the meantime, `r(Y,Z)` has completed or not. In both cases, we need to consume only the new answers relevant to `r(1,X)` from `r(Y,Z)` in order to satisfy the constraint shown earlier: $A_{new} \cap A_{old} = \emptyset$, where in this case A_{old} = `<1,a>`.

4 Single Time Stamped Trie

Once a pruned subgoal is reactivated and transformed into a loader or consumer node, it is important to avoid consuming answers that were found as a generator.

[1] A loader node works like a consumer node but without suspending the computation after consuming the available answers, since the corresponding subgoal is completed.

In order to efficiently identify such answers, we designed the *Single Time Stamped Trie (STST)* table space.

In this new organization, each tabled predicate has two tries, the subgoal trie, as usual, and the STST, a time-stamped answer trie common to all subgoal calls for the predicate, while each subgoal frame has an answer return list that references the matching answers from the STST. Figure 4 illustrates an example of the new table space organization for a tabled predicate `p/2` with the subgoals `p(VAR0,1)` and `p(VAR0,VAR1)` and the answers `<f(x),1>`, `<10,[]>` and `<f(y),1>`. This new organization reduces memory usage, since an answer is represented only once, and permits easy sharing of answers between subgoals, as the same answer can be referenced by multiple subgoal frames.

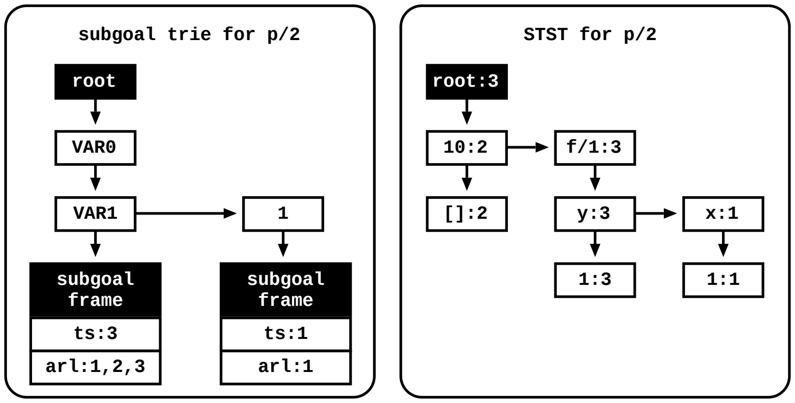

Fig. 4. The new STST table space organization

For the subgoal frames, we have also extended them with a `ts` field that stores the timestamp of the last generated or consumed answer. At any time, the answers in the answer return list (field `arl` in Fig. 4) are thus the matching answers from the STST that have a timestamp between 0 and `ts`. When a subsumed subgoal is pruned, we know its timestamp and we can easily turn it into a consumer, since now instead of inserting answers, the new consumer will now consume them from the STST, like in the TST design, by incrementally retrieving answers from it. Therefore, the cost of such transformation is very low given that both generators and consumers use an answer return list and a timestamp. If we had used the original TST design, the pruned subgoal would have its own answer trie, call it T_1, and we would need to, before consuming answers, check if the answers on T_1 have already appeared on the answer trie of the subsuming subgoal, call it T_2. Such a task is quite complex, since answers in T_1 are instances of the answers in T_2.

Notice that in both variant-based and subsumption-based tabling, only the substitutions for the variables in a subgoal call are stored in the answer tries [8]. For example, for the subgoal `p(VAR0,1)` and the answers `<f(x),1>` and `<f(y),1>`, only the substitutions `<f(x)>` and `<f(y)>` are stored, since during consumption of answers only the substitutions are used for unification. However, in the STST design, we cannot do this, since any subgoal of `p/2` can use the answers stored in the answer trie, therefore we need to store all the subterms of each answer.

4.1 Inserting Answers

The insertion of answers in the STST works like the insertion of answers in standard TSTs, but special care must be taken when updating the `ts` field on the subgoal frames. When only one subgoal is adding answers to the STST, the `ts` field can be incremented each time an answer is inserted. Repeated answers are easily recognized by testing if the answer is new or not by using the `ts` field. The problem arises when several subgoals are inserting answers, as it may be difficult to determine when an answer is new or repeated for a certain subgoal.

Let's consider two subgoals of the same predicate p, S_1 and S_2, and their corresponding timestamps, T_1 and T_2. S_1 has found and inserted the first 3 answers ($T_1 = 3$) in the STST and S_2 then started evaluating and inserted the next 3 answers, answers 4, 5 and 6 ($T_2 = 6$). Now, when execution backtracks to S_1, answer 5 is found and, while it is already on the trie, it must be considered as a new answer for S_1. By default, we could consider answer 5 as new, since T_1 is in the past ($T_1 < 5$). But this can also lead to problems if next we update T_1 to either 6 (the predicate timestamp) or 5 (the timestamp for answer 5). For example, if later, answer 4 is also found for S_1, it will be considered as a repeated answer during its insertion since now $T_1 > 4$. Therefore, we need a more complex mechanism to detect repeated subgoal answers.

In our approach, we use a *pending answer index* for each subgoal frame. This index contains all the answers that are older than the current subgoal frame timestamp field but that have not yet been found by the subgoal. It is built whenever the timestamp of the answer being inserted is younger than the subgoal frame timestamp, by collecting all the relevant answers in the STST with a timestamp younger than the current subgoal frame timestamp. Later, when an answer is found but is already on the trie, and therefore will have an older timestamp than the subgoal frame timestamp, we must lookup on the pending answer index to check if the answer is there. If so, we consider it a new answer and remove it from the index; if not, we consider it a repeated answer.

The pending answer index is implemented as a single linked list, but can be transformed into a hash table if the list reaches a certain threshold. In Fig. 5, we present the code for the `stst_insert_answer()` procedure, which given an answer and a subgoal frame, inserts the answer into the corresponding STST for the subgoal frame. The pseudo-code is organized into four cases:

1. Answers are inserted in order by the same subgoal. This is the most common situation.
2. The answer being inserted is the only answer in the STST that the current subgoal has still not considered. It is trivially marked as a new answer.

3. The timestamp of the answer being inserted is older than the subgoal frame timestamp. The pending answer index must be consulted.
4. The timestamp of the answer being inserted is younger than the subgoal frame timestamp t. We must collect all the relevant answers in the STST with a timestamp younger than t (calling `collect_relevant_answers()`) and add them to the pending answer index, except for the current answer.

```
stst_insert_answer(answer, sg_fr) {
  table_entry = table_entry(sg_fr)
  stst = answer_trie(table_entry)
  old_ts = predicate_timestamp(stst)
  leaf_node = answer_check_insert(answer, stst)
  leaf_ts = timestamp(leaf_node)
  new_ts = predicate_timestamp(stst)

  if (new_ts == old_ts + 1 and ts(sg_fr) == old_ts)
    // case 1: incremental answer by the same subgoal
    ts(sg_fr) = new_ts
    return leaf_node
  else if (new_ts == old_ts == leaf_ts and ts(sg_fr) == new_ts - 1)
    // case 2: only answer still not considered by the current subgoal
    ts(sg_fr) == new_ts
    return leaf_node
  else if (leaf_ts <= ts(sg_fr))
    // case 3: answer with a past timestamp, check pending answer index
    if (is_in_pending_answer_index(leaf_node, sg_fr))
      remove_from_pending_answer_index(leaf_node, sg_fr)
      return leaf_node
    else
      return NULL
  else
    // case 4: answers were inserted by someone else
    ans_tpl = answer_template(sg_fr)
    pending_list = collect_relevant_answers(ts(sg_fr),ans_tpl,stst)
    remove_from_list(leaf_node, pending_list)
    add_to_pending_answer_index(pending_list, sg_fr)
    ts(sg_fr) = new_ts
    return leaf_node
}
```

Fig. 5. Pseudo-code for procedure `stst_insert_answer()`

Note that when a generator subgoal frame is transformed into a consumer subgoal frame, we remove all the answers from the pending answer index and we safely insert them on the answer return list. With this, all the consumer mechanisms can be used as usual.

4.2 Reusing Answers

The STST approach also allows reusing answers when a new subgoal is called. As an example, consider that two unrelated (no subsumption involved) subgoals S_1 and S_2 are fully evaluated. If a subgoal S is then called, it is possible that some of the answers on the STST match S even if S neither subsumes S_1 nor S_2. Hence, instead of eagerly running the predicate clauses, we can start by loading the matching answers already on the STST, which can be enough if, for example, S is pruned by a cut. This is a similar approach to the *incomplete tabling* technique for variant-based tabling [9].

While the reuse of answers has some advantages, it can also lead to redundant computations. This happens when the evaluation of S generates more general answers than the ones initially stored on the STST. For an example, consider the retroactive tabled predicate `p/2` with only one fact, `p(X,a)`. If subgoal `p(1,Y)` is first called, the answer represented as `<1,a>` is added to the STST for `p/2` and execution would succeed. If the subgoal `p(X,Y)` is then called, we would search the STST for relevant answers and the first answer would be `<1,a>`. If we ask for more answers, the system would return a new answer, `<VAR0,a>`, and add it to the STST. On the other hand, if we called `p(X,Y)` with an empty STST, only the answer `<VAR0,a>` would be returned.

4.3 Answer Templates

The *answer template* is a data structure that is built on the choice point stack when a new subgoal, generator or consumer, is called. The contents of the answer template are the terms from the subgoal call that must be accessed when inserting a new answer, if a generator, or the terms from the subgoal call that must be unified when consuming answers, if a consumer.

For variant-based tabling, the answer template is just the set of variables in the subgoal call, since we only store variable substitutions on the answer trie. For call by subsumption based on the TST design, where we use an answer trie per subgoal, the answer template for each consumer subgoal is built according to its generator subgoal. For example, if the subsumptive subgoal is `p(1,f(X),Y)` and the subsumed subgoal is `p(1,f([A,B]),a(C))`, the answer template for the subsumed subgoal will be `<[A,B],a(C)>`. With RCS, we need the full answer template because the answers stored on the STST contain all the predicate arguments, hence the unification of matching answers must be seen as unifying against the most general subgoal. The answer template is thus built by simply copying the full set of argument registers from the generator or consumer call.

4.4 Compiled Tries

Compiled tries are a well-known implementation mechanism in which we decorate a trie with WAM instructions when a subgoal completes in such a way that, instead of consuming answers one by one in a bottom-up fashion, we execute the trie instructions in order to consume answers incrementally in a top-down fashion, thus taking advantage of the nature of tries [8].

Our approach only compiles the STST when the most general subgoal is completed. This avoids problems when a subgoal is executing compiled code and another subgoal is inserting answers, leading to the loss of answers as hash tables can be dynamically created and expanded. With this optimization, we can throw away the subgoal trie and the subgoal frames when the most general subgoal completes and the STST is compiled. Later, when a new subgoal call is made, we just build the answer template by copying the argument registers and then we execute the compiled trie, thus bypassing all the mechanisms of locating the subgoal on the subgoal trie, leading to memory and speedup gains.

5 Experimental Results

As shown in Table 1, in previous experiments using the STST design for comparing RCS with variant-based (**V/RCS**) and subsumption-based tabling (**S/RCS**), we got good speedups when executing programs that take advantage of RCS [10].

Table 1. Average speedups ratios for programs taking advantage of RCS

Program	V/RCS	S/RCS
double_first	**1.07**	**1.09**
double_last	**1.05**	**1.10**
reach_first	**2.54**	**1.76**
reach_last	**3.22**	**1.87**
fib	**1.95**	**2.02**
flora	**3.17**	**1.17**
big	**13.26**	**13.66**

In this paper, we will focus in measuring the space and time impact of having to store the complete answers on the STST, instead of storing only the variable substitutions, since it is more expensive to insert/load terms to/from the STST.

The environment for our experiments was a PC with a 2.66 GHz Intel Core(TM) 2 Quad CPU and 8 GBytes of memory running the Linux kernel 2.6.38 with Yap 6.03. For benchmarking, we used six different versions of the well-known `path/2` program, that computes the reachability between nodes in a graph, with several dataset configurations: `chain`, `cycle`, `grid`, `pyramid` and `tree`. We also consider two versions of the `path/2` program: the **Original** version; and the **Transformed** version, where the subgoals were transformed to use functor terms in each argument. For example, the transformed version of the **left_first** (left recursion with the recursive clause first) `path/2` program is:

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

We experimented the six versions of the `path/2` program with different graph sizes for the datasets using two queries: `path(X,Y)`, for the **Original** version, and `path(f(X),f(Y))`, for the **Transformed** version. Note that in these benchmarks we do not take advantage of RCS evaluation, i.e., more general subgoals are never called after specific ones, since we are only interested in measuring the impact of having a table space organization based on the STST design.

5.1 Execution Times

In Table 2, we present the execution times, in milliseconds, for RCS evaluation (columns **RCS**) and the respective overheads for variant-based (columns **RCS/V**) and subsumption-based tabling (columns **RCS/S**) for the **Original** and **Transformed** versions. Each execution time is the average of 3 runs.

From these results, we can observe that, on total average for these set of benchmarks, the **Transformed** `path/2` program has an overhead of 25% and 37% when compared with variant-based and subsumption-based tabling, respectively. The insertion of new answers into the table space and the consumption of answers from the table space are the primary causes for these overheads, since the STST stores all the arguments of an answer in the trie and not only the answer substitutions. For the **Original** version, the average overhead is much smaller and is mainly due to the new mechanisms to control RCS evaluation.

For **Transformed**, the programs with the worst overheads are **double_first** and **double_last**, with 46% and 67% of overhead against subsumption-based tabling. These programs also create the higher number of consumers, both variant consumers and subsumed consumers, than any other benchmark in these experiments. The **right_first** and **right_last** only create subsumed consumers, and they have an overhead of 37% and 15%, respectively. In the **left_first** and **left_last** programs, only one variant consumer is allocated, however their performance is very similar to the **right** programs.

We thus argue that the number of consumer nodes can greatly reduce the applicability and performance of the STST design when the operation of loading answers from the trie is more expensive. While this situation seems disadvantageous, execution time can be reduced if a subsuming subgoal call appears (for example, `path(X,Y)` in the **Transformed** version) where it is possible to reuse the answers from the table before executing the predicate clauses.

5.2 Memory Usage

We executed the previous benchmarks and measured the number of answer trie nodes stored for each program using the **Transformed** version. Table 3 presents such numbers for RCS evaluation and the relative numbers for variant (column **V/RCS**) and subsumption-based tabling (column **S/RCS**).

From these results we can observe that, on total average for this set of benchmarks, the variant-based table design requires 1.89 times more memory space

Table 2. Execution times, in milliseconds, for RCS evaluation and the respective overheads for variant-based and subsumption-based tabling for the **Original** and **Transformed** programs

Program/Dataset		**Original**			**Transformed**		
		RCS	**RCS/V**	**RCS/S**	**RCS**	**RCS/V**	**RCS/S**
left_first	**chain (2048)**	1,368	1.13	1.09	1,540	1.04	1.19
	cycle (4096)	17,614	1.14	1.10	22,452	1.42	1.43
	grid (64)	23,529	1.31	1.19	25,923	1.28	1.01
	pyramid (4096)	34,191	1.02	1.14	41,581	1.28	1.15
	tree (32768)	204	1.22	1.30	327	1.34	1.20
	Average		1.17	1.16		1.27	1.19
left_last	**chain (4096)**	6,506	1.06	1.02	7,965	1.20	1.19
	cycle (4096)	17,760	1.15	1.02	22,584	1.42	1.27
	grid (64)	23,797	1.37	1.08	30,913	1.53	1.41
	pyramid (2048)	6,363	1.20	1.04	7,910	1.38	1.37
	tree (32768)	204	1.28	1.02	335	1.67	1.59
	Average		1.21	1.04		1.44	1.37
right_first	**chain (4096)**	3,893	**0.56**	**0.95**	5,899	**0.74**	1.43
	cycle (4096)	9,686	**0.66**	1.02	8,467	**0.57**	1.10
	grid (64)	21,495	**0.97**	1.08	26,734	1.06	1.36
	pyramid (4096)	15,706	**0.56**	**0.96**	19,592	**0.78**	1.57
	tree (32768)	285	**0.99**	1.22	355	1.24	1.39
	Average		**0.75**	1.04		0.88	1.37
right_last	**chain (4096)**	3,921	**0.49**	**0.77**	4,587	**0.58**	1.14
	cycle (4096)	8,675	**0.58**	1.17	11,359	**0.67**	1.20
	grid (64)	18,806	**0.62**	1.08	25,090	**0.72**	1.40
	pyramid (4096)	16,214	**0.58**	**0.98**	19,905	**0.63**	1.19
	tree (65536)	1,863	3.19	**0.99**	1,569	2.41	**0.80**
	Average		1.09	1.00		1.00	1.15
double_first	**chain (512)**	3,116	**0.75**	**0.81**	6,342	1.36	1.58
	cycle (256)	2,277	**0.86**	1.09	3,488	**0.99**	1.74
	grid (16)	2,333	**0.90**	1.08	3,879	1.12	1.17
	pyramid (256)	1,667	**0.95**	1.09	4,309	1.86	1.61
	tree (16384)	631	1.57	1.29	624	1.54	1.18
	Average		1.01	1.07		1.37	1.46
double_last	**chain (512)**	3,737	1.07	1.31	8,403	1.79	1.75
	cycle (256)	2,376	**0.83**	1.14	3,471	**0.98**	1.74
	grid (16)	2,328	**0.90**	1.08	5,205	1.53	2.09
	pyramid (256)	1,697	**0.73**	1.13	4,273	1.90	1.60
	tree (16384)	505	1.41	**0.98**	624	1.55	1.19
	Average		**0.99**	1.13		1.55	1.67
	Total Average		1.04	1.07		1.25	1.37

than the STST table space organization. In particular, for the **double_first** program, these differences are higher because in the variant design there are more generator subgoal calls and thus more answer tries are created.

Table 3. Number of stored answer trie nodes for RCS evaluation and the relative numbers for variant-based and subsumption-based tabling for the **Transformed** programs

Program/Dataset		**Transformed**		
		#RCS	**V/RCS**	**S/RCS**
left_first	**chain (2048)**	2,100,233	0.999	0.999
	cycle (2048)	4,200,450	0.999	0.999
	grid (64)	16,789,506	0.999	0.999
	pyramid (1024)	1,576,457	0.998	0.998
	tree (65536)	983,056	0.966	0.966
	Average		0.992	0.992
right_first	**chain (4096)**	8,398,847	1.997	0.999
	cycle (4096)	16,789,506	1.999	0.999
	grid (32)	1,051,650	1.996	0.998
	pyramid (2048)	6,302,719	1.996	0.998
	tree (32768)	491,520	1.766	0.900
	Average		1.951	0.979
double_first	**chain (256)**	26,387	2.483	0.874
	cycle (256)	36,844	3.571	0.893
	grid (16)	59,028	2.229	0.825
	pyramid (256)	56,638	3.475	0.951
	tree (16384)	213,008	1.884	0.961
	Average		2.728	0.901
	Total Average		1.890	0.957

When comparing RCS to the subsumption-based engine, the latter only stores, on total average for this set of benchmarks, around 4% less trie nodes than RCS evaluation, even if the `f/1` functor terms need to be stored in the STST. This is easily understandable because the first `f/1` functor term is only represented once, at the top of the STST, and then there is one second `f/1` functor for each node in the graph, therefore, the total number of functors stored in the STST is insignificant when compared to the total number of terms stored in the trie. Also note that, for the **double_first** benchmarks, the datasets used are small if compared to the datasets used for the other benchmarks, but the space overhead is more significant (18% in the worst case). We thus argue that the cost of the extra space needed to store terms in the STST is less significant as more terms are stored in the tries.

6 Conclusions

We presented a new table space organization that is particularly well suited to be used with Retroactive Call Subsumption. Our proposal uses ideas from the original TST design and innovates by having only a single answer trie per predicate, making it easier to share answers across subgoals for the same predicate. We presented the challenges when using a single answer trie and how they have been solved, for example, with the use of pending answer indices. Moreover, we think that the new design should not be very difficult to port to other tabling engines, since it uses the trie data structure extended with timestamp information.

Our previous experiments with RCS and the STST showed promising results when used with programs that take advantages of the new mechanisms. In this paper, we benchmarked and discussed the overhead in terms of time and space when storing and loading complete answers, instead of using variable substitutions, for programs that do not take advantage of RCS evaluation. Our results show that, for some programs, the time overhead can be considerable, however, in terms of space, the number of extra trie nodes appears to be relatively small.

Acknowledgments

This work has been partially supported by the FCT research projects HORUS (PTDC/EIA-EIA/100897/2008) and LEAP (PTDC/EIA-CCO/112158/2009).

References

1. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43**(1) (1996) 20–74
2. Johnson, E., Ramakrishnan, C.R., Ramakrishnan, I.V., Rao, P.: A Space Efficient Engine for Subsumption-Based Tabled Evaluation of Logic Programs. In: Fuji International Symposium on Functional and Logic Programming. Number 1722 in LNCS, Springer-Verlag (1999) 284–300
3. Rao, P., Ramakrishnan, C.R., Ramakrishnan, I.V.: A Thread in Time Saves Tabling Time. In: Joint International Conference and Symposium on Logic Programming, The MIT Press (1996) 112–126
4. Johnson, E.: Interfacing a Tabled-WAM Engine to a Tabling Subsystem Supporting Both Variant and Subsumption Checks. In: Conference on Tabulation in Parsing and Deduction. (2000) 155–162
5. Cruz, F., Rocha, R.: Retroactive Subsumption-Based Tabled Evaluation of Logic Programs. In: European Conference on Logics in Artificial Intelligence. Number 6341 in LNAI, Springer-Verlag (2010) 130–142
6. Rocha, R., Silva, F., Santos Costa, V.: On applying or-parallelism and tabling to logic programs. Theory and Practice of Logic Programming **5**(1 & 2) (2005) 161–205
7. Freire, J., Swift, T., Warren, D.S.: Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In: International Symposium on Programming Language Implementation and Logic Programming. Number 1140 in LNCS, Springer-Verlag (1996) 243–258
8. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38**(1) (1999) 31–54
9. Rocha, R.: Handling Incomplete and Complete Tables in Tabled Logic Programs. In: International Conference on Logic Programming. Number 4079 in LNCS, Springer-Verlag (2006) 427–428
10. Cruz, F.: Call Subsumption Mechanisms for Tabled Logic Programs. Master's thesis, Department of Informatics Engineering, University of Porto (2010)