A Thread in Time Saves Tabling Time

Prasad Rao C.R. Ramakrishnan I.V. Ramakrishnan

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
{prasad,cram,ram}@cs.sunysb.edu

Abstract

The use of tabling in logic programming has been recognized as a powerful evaluation technique. Currently available tabling systems are mostly based on variant checks and hence are limited in their ability to recognize reusable subcomputations. Tabling systems based on call subsumption can yield superior performance by recognizing a larger set of reusable computations. However, a straightforward adaptation of the mechanisms used in variant-based systems to reuse these computations can in fact result in considerable performance degradation. In this paper we propose a novel organization of tables using *Dynamic Threaded Sequential Automata* (DTSA) which permits efficient reuse of previously computed results in a subsumptive system. We describe an implementation of the tables using DTSA and the associated access mechanisms. We also report experimental results which show that a subsumptive tabling system implemented by extending the XSB logic programming system with our table access techniques can perform significantly better than the original variant-based system.

1 Introduction

The use of tabling in logic programming is beginning to emerge as a powerful evaluation technique, since it allows bottom-up evaluation to be incorporated within a top-down framework, combining the advantages of both. Although the concept of tabled evaluation of logic programs has been around for a decade (see [8, 9]), practical systems based on tabling are only beginning to appear (*e.g.*, [6, 5, 10]). Early experience with these systems suggest that they are indeed practically viable. In particular the XSB system, based on SLG resolution [1], computes in-memory deductive database queries about an order of magnitude faster than current semi-naive methods, and computes Prolog queries with little loss of efficiency when compared to well known Prolog systems [7].

At a high level, top-down tabling systems evaluate programs by recording subgoals (referred to as *calls*) and their provable instances (referred to as *answers*) in a table. Predicates are marked *a priori* as either *tabled* or *nontabled*. Clause resolution, which is the basic mechanism for program evaluation, proceeds as follows. For nontabled predicates the subgoal is resolved against program clauses. For tabled predicates, if the subgoal is already present in the table, then it is resolved against the answers recorded in the table; otherwise the subgoal is entered in the table, and its answers, computed by resolving the subgoal against program clauses, are also entered in the table. For both tabled and nontabled predicates, program clause resolution is carried out using SLD.

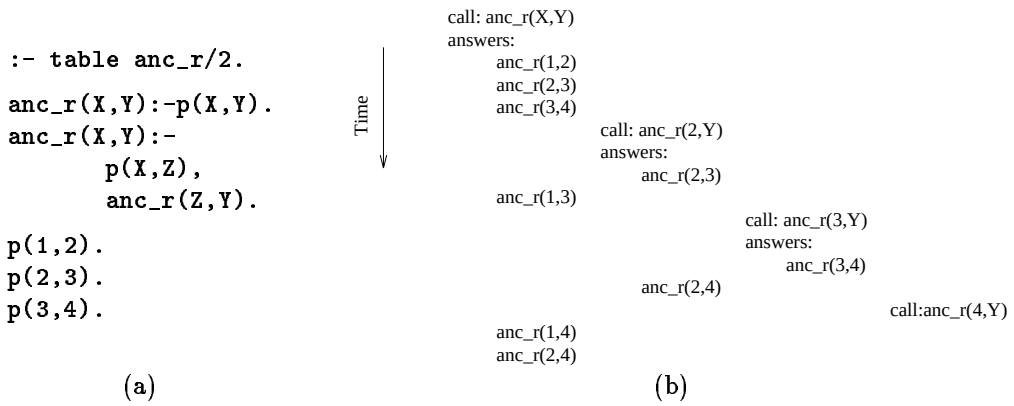


Figure 1: Ancestor program (a) and the sequence of call and answer table entries (b).

Given the relative novelty of table-based logic programming implementations, many promising avenues for substantially improving performance remain to be explored. Efficient organization and manipulation of tables is one such avenue and forms the topic of this paper.

When a tabled subgoal is called, a check for the presence of this subgoal in the table is done first. In currently available tabling systems (*e.g.*, [6, 5]) this is done by checking whether a *variant* of the new goal already exists in the table. We say that two terms t_1 and t_2 are variants of each other if they are identical up to renaming of their variables. Although a variant check is a light-weight operation computationally, tabling systems based on such checks can end up computing answers through expensive program clause resolution steps when they could have as well retrieved them through answer clause resolution. For example, consider evaluation of the call `anc_r(X,Y)` made to the tabled predicate shown in Figure 1a. Figure 1b shows the sequence of calls made and answers computed in a variant based engine for this call.

To begin with the call table is empty; hence `anc_r(X,Y)` is entered in the call table and its answers are computed via program clause resolution. Observe in the figure that the call `anc_r(2,Y)` is made after computing three answers. Since `anc_r(2,Y)` is not a variant of `anc_r(X,Y)`, it is entered in the call table and its answers are computed next using program clause resolution. Resolving against the first clause yields `anc_r(2,3)` as an answer. Note that this answer already exists in the answer table associated with the call `anc_r(X,Y)`. Instead of simply retrieving this answer it is recomputed via program clause resolution. Similarly, the remaining answers for `anc_r(2,Y)` and `anc_r(3,Y)` are also recomputed via program clause resolution.

The key to avoiding answer recomputation is to first verify whether the needed answers have already been computed on behalf of a more general call — *i.e.*, check whether the new goal is *subsumed* by one already existing in the call table. We say that a term t_1 subsumes another term t_2 if t_2 is an instance of t_1. Based on a subsumption check, if it so turns out that the new call (*e.g.*, `anc_r(2,Y)` above) is subsumed by a more general call in the table (*e.g.*, `anc_r(X,Y)`) then the answers to the new call are retrieved from the subsuming call's answer table. By avoiding expensive program clause resolution steps, tabled evaluation based on subsumption checks can, in principle, yield superior performance over one based on variant checks.

But subsumption-based tabling introduces additional overheads that can easily offset the potential gains. First of all, subsumption checks are more expensive than variant checks. Secondly observe in the above example that answers to the subsumed call `anc_r(2,Y)` forms a subset of the

answers computed for the subsuming call `anc_r(X,Y)`, necessitating efficient indexing mechanisms for selecting this subset. Since these subsets have to be extracted from incomplete answer tables[1], an added level of difficulty is imposed on indexing mechanisms. For instance, consider the first time when the call `anc_r(2,Y)` is made in the above example. Observe in Figure 1b that at that time, the answer table for the subsuming call `anc_r(X,Y)` is incomplete and contains only three answers. Therefore, only `anc_r(2,3)` is returned as the answer now. When the answer `anc_r(2,4)` is eventually added to the answer table of the subsuming call, it must also be returned as an answer for `anc_r(2,Y)`.

Selection of subsets and the associated indexing issues do not arise in tabling systems based on variant checks since all of the answers in a call's answer table are relevant to any variant of the call. So organizing the answer table as a simple linked list is adequate for doing efficient retrieval even for incomplete answer tables: as new answers are generated they are simply added to the end of the list. But for subsumptive tabling, where a subset of the added answers is retrieved each time, quickly filtering away irrelevant answers is critical for efficiency. In fact in a preliminary implementation in which we modified XSB's engine to perform subsumption checks but retained the linear list structure of the answer tables, we observed slow downs of more than an order of magnitude over XSB's variant based engine on the example in Figure 1a above.

In summary, while subsumptive tabling holds a lot of promise, the issues raised above reveal that exploiting its potential in practice is a challenging problem. We propose a solution to this problem in this paper. Specifically, we describe a novel organization of tables for answer clause resolution called *Dynamic Threaded Sequential Automata (DTSA)* and provide mechanisms to efficiently manipulate them with minimal overheads (see Section 3). In Section 4 we give an overview of our implementation of a subsumptive engine for the XSB system centered around DTSA. We present performance results in Section 5 and conclude in Section 6.

2 Overview of Tabling Operations

We view top-down tabled evaluation of a program in terms of four abstract table operations. In the following, we describe each of the four abstract operations and how they are used in a tabling engine.

Call-check-insert Given a call c, the *call-check-insert* operation directs the engine to perform answer clause resolution in a variant engine whenever a variant of c is found in the table, and in a subsumptive engine, whenever there is a call c' in the table that subsumes c. Note that while variance is an equivalence relation, subsumption defines only a partial order. Therefore, in variant tabling, only one variant of any call is present in the table. In contrast, in subsumptive tabling, there may be many calls $c_1, c_2, \ldots, c_k$ in the table that subsume a given call c. Although the answer tables for any of the subsuming calls c_i may be used for correctly resolving c, the choice of the answer table can significantly affect the performance of resolution. Furthermore, in contrast to variant tabling, selection of a subsuming call c_i and insertion of c in the table (in the absence of such a call) cannot be done with just one scan of the symbols in c. Hence, *call-check-insert* operation in a subsumptive engine is more expensive than in a variant engine.

[1] We say that the answer table for a call is incomplete whenever all the answers to the call have not yet been computed.

Answer-check-insert Answers are entered in the table using *answer-check-insert* operations. Given an answer a and an answer table T, this operation first checks whether a is already present in T, and inserts a in T otherwise. Note that this operation must also eliminate any duplicates in the table.

Retrieve-answer Answer clause resolution of a goal G against a set of terms $S = \{t_1, t_2, \ldots, t_n\}$ in an answer table produces a set R such that $r \in R \iff r = t_i\theta_i$ for some t_i, where $\theta_i = mgu(G, t_i)$. This resolution is performed using *retrieve-answer* operations. Given a call c and an answer table T, answer resolution proceeds as follows. The operation *retrieve-answer*, when invoked first with the call c and table T, returns an answer from T that unifies with c, and a structure, called an *answer continuation*, that denotes the computation to be done for retrieving the remaining answers. When another answer is demanded, *retrieve-answer* is invoked with the answer continuation, and returns the next answer alongwith the modified answer continuation. Answer continuation $\perp$ denotes that there are no more remaining answers; note that in the presence of incomplete answer tables, $\perp$ cannot be interpreted as the end of answer resolution.

Pending-answers When a *retrieve-answer* operation for a call c on an incomplete table T returns $\perp$, the call c will be *suspended*. The suspended call is later resumed when new answers have been added to T, or when T is determined to be complete. Suspension and resumption of calls are performed by a process called *answer scheduling* in the engine. The answer scheduler invokes *pending-answers* to determine whether a suspended call needs to be resumed. Given an answer continuation, *pending-answers* succeeds if, and only if, there are any more answers represented by the continuation.

3 Dynamic Threaded Sequential Automata

We structure answer tables for subsumptive tabling as *Dynamic Threaded Sequential Automata (DTSA)*. This structure facilitates good indexing and efficient backtracking — the two ingredients necessary for efficiently returning (a subset of) answers to a subsumed call from a dynamically growing answer table. Recall that any technique for retrieving answers from an answer table must guarantee that every answer that unifies with the subsumed call will be returned even when the table is not complete. Abstractly, we can regard all the answers in the answer table as being totally ordered with respect to the time at which they are inserted in the answer table. Ensuring that the order of the returned answers is consistent with this total order enables us to easily mark the remaining answers and facilitates efficient retrieval.

The basic building block for DTSA is a Sequential Factoring Automaton (SFA) — a structure introduced by us in an earlier work on unification factoring [2]. We progressively embellish it with more features culminating with the DTSA in its full detail. We present our design in three steps. In the first two steps (see Sections 3.1 and 3.2) we ignore the dynamic aspect of answers being added to an answer table while retrieval is in progress, and deal with it in the last step (see Section 3.3).

Notations We assume the standard definitions of term, and the notions of substitution and subsumption of terms. A *position* in a term is either the empty string Λ that reaches the root of the term, or $\delta.i$, where δ is a position and i is an integer, that reaches the i^{th} child of the term reached by δ. Positions are totally ordered by a preorder relation $<_{pos}$ such that $\delta <_{pos} \delta'$ iff there is some term such that δ will be visited before δ' in a left-to-right preorder traversal of the term. By $t\,|_\delta$ we denote the subterm at position δ in t. For example, $p(a, f(X))\,|_{2.1} = X$ and

$p(a, f(X))\,|_2 = f(X)$. The symbol at position δ in a term t is denoted by $sym(t, \delta)$. The arity of a symbol α is denoted by $arity(\alpha)$; note that the arity of variable symbols is 0. An anonymous variable is denoted by '$\Box$'. The *prefix* of a term t up to position δ, denoted by $prefix(t, \delta)$, is the set $\{(\delta', \alpha) \mid \delta' \leq_{pos} \delta, sym(t, \delta') = \alpha\}$. The proper extension of a prefix φ, denoted by $ext(\varphi)$, is the set $\varphi \cup \{(\delta.i, \Box) \mid (\delta, \alpha) \in \varphi, (\delta.i, \beta) \notin \varphi, i \leq arity(\alpha)\}$. Note that the proper extension of a prefix denotes a well-formed term. Two prefixes, φ_1 and φ_2 unify iff the term denoted by $ext(\varphi_1)$ unifies with the term denoted by $ext(\varphi_2)$. We assume that all terms entered in tables are *standardized* as follows. If $V_1, V_2, \ldots, V_n$ are the set of variables in a term t such that V_i occurs before V_{i+1} in a left-to-right preorder traversal of t, then V_i is replaced by a standard variable X_i. We represent a totally ordered set $\{\omega_1, \omega_2, \ldots, \omega_n\}$ by the sequence $\langle \omega_1, \omega_2, \ldots, \omega_n \rangle$. A stack is represented by the sequence of elements $\langle \omega_1, \omega_2, \ldots, \omega_n \rangle$ where ω_n is on top of the stack.

3.1 SFA: A Basic Building Block for Answer Tables

The problem addressed in the first step can be cast concretely as follows:

> *Given a static set S of terms, a total ordering $\prec$ on them and any goal term G, retrieve terms in S that unify with G such that the order in which they are retrieved is consistent with $\prec$.*

We use an SFA to solve the above problem. An SFA is an ordered tree-structured automaton, with the root as the start state, and edges, denoting transitions, representing elementary unification operations. Let $S = \{t_1, t_2, \ldots, t_n\}$ be a static set of terms and $\prec$ a total ordering on them such that $\forall i, 1 \leq i < n \quad t_i \prec t_{i+1}$. Every leaf state in an SFA represents a distinct term in S and the transitions on the path from the root to a leaf represent the set of elementary operations needed to unify the goal with the term associated with that leaf. We assign a preorder number $pre(s)$ to every state s in the SFA. An SFA is so constructed that the preorder number of the leaf corresponding to t_i is smaller than that for t_{i+1}. We remark that although SFA's and tries are similar in structure, there are notable differences. Firstly transitions from any state in a trie are all unique. Secondly transitions in tries usually denote elementary match operations. Lastly no ordering is imposed on the terms represented by a trie.

Since transitions in a SFA represent elementary unification operations on the goal, the goal gets progressively instantiated as transitions are made from state to state. Each state s specifies a position, denoted by $\pi(s)$, in this partially instantiated goal where the next unification operation will be performed. Each outgoing transition from state s, denoted by $s \xrightarrow{\alpha} d$, represents a unification operation involving the symbol α and the symbol in the partially instantiated goal at $\pi(s)$.

Definition 3.1 (Partially Instantiated Goal) *Given a goal G and an SFA S, the partially instantiated goal upon reaching state d of S, denoted by $PartInst(G, d)$, is (i) G if d is the root of S, or (ii) $G_s\theta$ if $s \xrightarrow{\alpha} d$ is in S, where $G_s = PartInst(G, s)$ and $\theta = mgu(G_s\,|_{\pi(s)}, \alpha(V_1, V_2, \ldots, V_k))$ such that $V_1, V_2, \ldots, V_k$ are variables that do not occur in G_s or S.*

An example SFA is shown in Figure 2b. In that SFA, the goal `p(a,X,Y)` on reaching s_3 is instantiated as `p(a,a,Y)` and the label on the transition from s_3 to s_4 specifies a unification operation involving the symbol a and the variable at position 3 in this instantiation. Note that $PartInst(G, s)$ is undefined for a state s if the unification operation specified by some transition on the path from root to s fails for the initial goal G. The transitions from a state that represent successful unification operations are called *applicable transitions*, defined as follows.

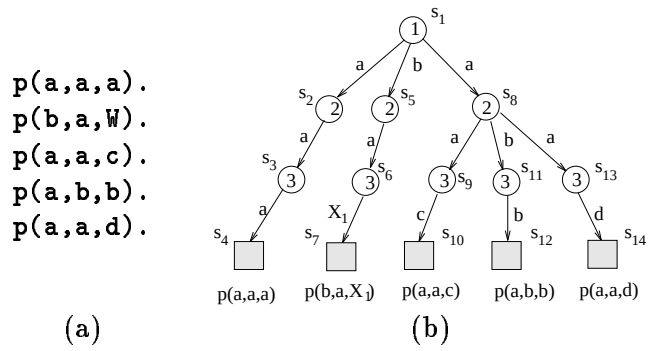


Figure 2: Sequence of answers for call `p(X,Y,Z)` (a) and the corresponding SFA (b).

Definition 3.2 (Applicable Set of Transitions (for SFA)) *The set of all transitions that are applicable on reaching a state s with an initial goal G, denoted by $App(G,s)$, is such that $s \xrightarrow{\alpha} d \in App(G,s)$ iff (i) $s \xrightarrow{\alpha} d$ is a transition in the SFA and (ii) $PartInst(G,d)$ is defined whenever $PartInst(G,s)$ is defined.*

The process of retrieving answers using the SFA proceeds as follows. We traverse the SFA starting from the root state, with an empty *continuation stack*. When a state s is reached, we choose the leftmost applicable transition (if any) in the order specified by the SFA as the transition to be made from s. This choice is made using the function *first*. Before making a transition, we select the next applicable transition, if one exists, and push it on the continuation stack. This selection is made using the function *next*. The pending transitions on the continuation stack form the answer continuation that will be used to retrieve the remaining answers. If there are no applicable transitions at a state, the next transition to be made is picked from the top of the continuation stack. The two functions used to select transitions are defined as:

Definition 3.3 *First and Next for SFAs*

$$\begin{aligned} first(s,G) &= s \xrightarrow{\alpha} d \text{ such that } s \xrightarrow{\alpha} d \in App(G,s) \text{ and} \\ &\quad \forall s \xrightarrow{\alpha} d' \in App(G,s), \quad d \neq d' \Rightarrow pre(d) < pre(d'). \\ next(s \xrightarrow{\alpha} d, G) &= s \xrightarrow{\alpha} d' \text{ such that } s \xrightarrow{\beta} d' \in App(G,s) \text{ and} \\ &\quad \forall s \xrightarrow{\beta} d'' \in App(G,s) \quad pre(d'') > pre(d) \wedge d'' \neq d' \Rightarrow pre(d') < pre(d''). \end{aligned}$$

For the SFA in Figure 2b, and initial goal G =`p(a,a,V)`, $first(s_1, G) = s_1 \xrightarrow{a} s_2$ and $next(s_1 \xrightarrow{a} s_2, G)$ is $s_1 \xrightarrow{a} s_8$. Note that *first* and *next* are not total functions. For example, $next(s_1 \xrightarrow{a} s_8)$ is undefined.

The above intuitive description of answer retrieval from an SFA is realized concretely by Algorithm *Retrieve_Answer* (Figure 3) using the definitions of *first* and *next*. Recall from previous section that we distinguish between the first and subsequent invocations of *Retrieve_Answer*. When *Retrieve_Answer* is invoked with a goal G for the first time, it calls *first_answer* to retrieve the first answer and returns with an answer continuation. For subsequent invocations it takes as input an answer continuation, calls *subsequent_answer* to retrieve the next answer represented by the input continuation, and returns an updated answer continuation.

Using the SFA in Figure 2b, when *Retrieve_Answer* is invoked the first time with the goal `p(a,a,V)`, it is easy to see that the the root-to-leaf path from s_1 to s_4 will be taken and `p(a,a,a)` will be returned as the answer along with the continuation stack $\langle s_1 \xrightarrow{a} s_8 \rangle$. On calling it again with this continuation stack, `p(a,a,c)` will be returned as the next answer. The following soundness result can be readily established.

```
function get_answer(this_trans, cont_stack, G)
/* cont_stack: continuation stack */
begin
    while TRUE do
        if this_trans is not defined then
            fail
        endif
        next_trans := next(this_trans, G)
        if next_trans is defined then
            push(next_trans, cont_stack)
        endif
        let d = dest(this_trans)
        let G_d = PartInst(G, d)
        /* Note: transition is done here */
        if d is a leaf then
            return G_d
        else
            this_trans := first(d, G)
            if this_trans is not defined then
                this_trans := pop(cont_stack)
            endif
        endif
    endwhile
end
```

```
algo first_answer(G)
/* G: Goal */
 begin
    this_trans := first(root, G)
    cont_stack := empty
    answer := get_answer(this_trans,
            cont_stack, G)
    return ⟨ answer, cont_stack ⟩
end
```

```
algo subsequent_answer(cont_stack, G)
/* G: Goal
    cont_stack: continuation stack */
begin
    this_trans := pop(cont_stack)
    answer := get_answer(this_trans,
            cont_stack, G)
    return ⟨ answer, cont_stack ⟩
end
```

Figure 3: Algorithm *Retrieve_Answer*.

Theorem 3.1 *Let $\langle t_1, \ldots, t_n \rangle$ be the sequence of answers represented by an SFA, and let $\langle t_{r_1}, \ldots, t_{r_k} \rangle$ be its subsequence that unify with G. Then (i) Retrieve_Answer returns k answers, and (ii) $\forall j, 1 \leq j \leq k$ the j^{th} answer returned is t_{r_j}.*

Observe that, using an SFA, we did not even attempt to do the unnecessary unification operations involving the symbol b in states s_1 and s_8. In contrast, when selecting from an answer list, lack of indexing would have forced us to attempt them and fail. Although SFA is a clear improvement over an answer list, we can in fact do much better. In the example above, even the two unifications that were done to make the transitions from s_1 to s_8 and from s_8 to s_9 can be eliminated. This is because, for the two selected terms `p(a,a,a)` and `p(a,a,c)`, $prefix(\mathtt{p(a,a,a)}, 2) = prefix(\mathtt{p(a,a,c)}, 2)$. We can use such common prefixes as an index for quickly selecting relevant answers. To exploit such opportunities we enhance our basic SFA structure with threads as follows.

3.2 From SFA to Threaded Sequential Automata

The sequence of positions seen on any root-to-leaf path in an SFA corresponds to some top-down traversal of the term represented by the leaf. Henceforth, we assume that this traversal corresponds to a preorder traversal. We can regard each state of an SFA as representing a prefix of term(s) in the SFA. In Figure 2b, s_6 represents the prefix $\{(\Lambda, p), (1, b), (2, a)\}$ while s_{11} represents $\{(\Lambda, p), (1, a), (2, b)\}$.

Given the goal `p(a,a,V)`, the SFA in Figure 2b retrieves `p(a,a,a)` first by following the path from s_1 to s_4. To retrieve the next answer `p(a,a,c)` it backtracks to the root and then follows the path from s_1 to s_{10}. But observe that whenever states s_2 and s_3 are visited, states s_8 and s_9 are

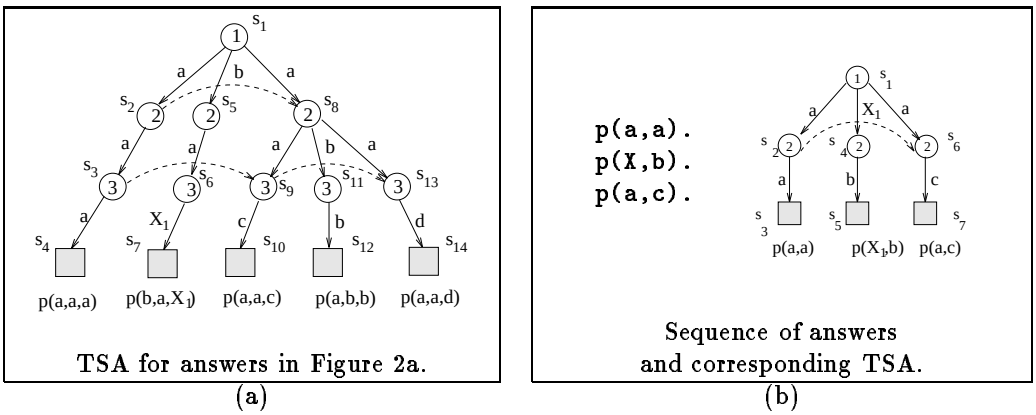


Figure 4: Examples of TSAs.

also guaranteed to be visited. This is because s_2 and s_8 denote the same prefix ($\{(\Lambda, p), (1, a)\}$) and so do s_3 and s_9 ($\{(\Lambda, p), (1, a), (2, a)\}$). This idea is captured by the following notion of equivalent states.

Definition 3.4 (Equivalent States) *A state s is equivalent to state s' (denoted by $s \cong s'$) in an SFA if (i) $s{=}s'$ or (ii) $t \xrightarrow{\alpha} s$ and $t' \xrightarrow{\alpha} s'$ are transitions in the SFA and $t \cong t'$.*

Note that $\cong$ is an equivalence relation. We convert an SFA into a Threaded Sequential Automaton (TSA) by threading states in each equivalence class R induced by $\cong$ using *equivalence links* as follows. Let $\langle d_1, d_2, \ldots, d_k \rangle$ be the sequence of states in R with increasing preorder numbers. Then we place an equivalence link from d_i to d_{i+1} for all $1 \leq i < k$. The TSA obtained from the SFA in Figure 2b is shown in Figure 4a.

Intuitively, following equivalence links amounts to doing indexing. For instance, using the TSA in Figure 4a, after returning the first answer to the goal `p(a,a,V)` upon reaching state s_4, the next answer, `p(a,a,c)`, can be retrieved by backtracking to s_3, following the equivalence link to s_9 and making the transition to s_{10}.

Whereas in an SFA the only possible transitions that could be taken from a state s were all rooted at s, in a TSA it is possible to traverse an equivalence link and make transitions rooted at a different state. For example, in the TSA in Figure 4a, on reaching s_3 if the next symbol is c then one can make transition to s_{10} by first taking an equivalence link from s_3 to s_9. We now expand the set of applicable transitions to include these types of transitions. This is readily done by modifying Definition 3.2 as follows:

Definition 3.5 (Applicable Set of Transitions (for TSA)) *Given a TSA, the set of all transitions that are applicable on reaching a state s with an initial goal G, denoted by $App(G, s)$, is such that $s' \xrightarrow{\alpha} d' \in App(G, s)$ iff (i) $s \cong s'$, (ii) $s' \xrightarrow{\alpha} d'$ is a transition in the TSA and (iii) $PartInst(G, d')$ is defined whenever $PartInst(G, s')$ is defined.*

In an SFA, due to its tree structure, it is straightforward to guarantee that a transition will be taken at most once. But since a TSA is a directed acyclic graph, it is possible to take a transition more than once leading to two undesirable consequences, namely (i) the same answer may be retrieved more than once, and (ii) the order in which answers are returned may not be consistent with the given total order. For example, invoking *Retrieve_Answer* on the goal `p(a,Y)` using the TSA in Figure 4b will return with the answer `p(a,a)` and continuation stack $\langle s_1 \xrightarrow{X_1} s_4, s_6 \xrightarrow{c} s_7 \rangle$. Invoking *Retrieve_Answer* a second time will return `p(a,c)` and the continuation stack $\langle s_1 \xrightarrow{X_1} s_4 \rangle$.

A third invocation will return with `p(a,b)` — which ought to have preceded the previously returned answer — and the continuation stack $\langle s_1 \xrightarrow{a} s_6 \rangle$. In any case we have now retrieved all the answers to the goal. However, the continuation stack still has one more pending transition, whose effect will be to return `p(a,c)` once more when *Retrieve_Answer* is called again.

The above example illustrates that although threads provide the mechanism for indexing, they have to be used with care. Had the transition $s_6 \xrightarrow{c} s_7$ been not regarded as pending then it would not have been pushed onto the continuation stack. We would then have returned all answers once, and in the correct order. The source of this problem stems from our expanded definition of applicable transitions which results in $s_6 \xrightarrow{c} s_7$ to be pushed on the stack. We know on reaching s_2 that answers reachable through the transition $s_6 \xrightarrow{a} s_7$ can also be reached through the transition $s_1 \xrightarrow{a} s_4$ that is already on the stack. Hence we modify the selection of transitions so as to consider only those transitions that reach answers which cannot be reached through the pending transitions on the stack. Such transitions are called *safe* transitions, defined below.

Definition 3.6 (Safe Transitions) *Given a goal G, a continuation stack ts and a state s, the set of safe transitions, denoted by $Safe(G, ts, s)$, is such that $s' \xrightarrow{\alpha} d' \in Safe(G, ts, s)$ iff (i) $s' \xrightarrow{\alpha} d' \in App(G, s)$, and (ii) $\forall s'' \xrightarrow{\beta} d'' \in ts \quad pre(d') < pre(d'')$.*

Simply restricting the traversal of equivalence links without modifying what may appear on the continuation stack will preclude us from taking *any* equivalence link. For example, using the TSA in Figure 4a, *first_answer* for the goal `p(a,a,V)` will pick $s_1 \xrightarrow{a} s_2$, placing $s_1 \xrightarrow{a} s_8$ on the stack. But doing so will disable us from making a transition to s_9 via s_3 for retrieving `p(a,a,c)`. The solution is to ensure that $next(\tau)$ is a transition τ' such that there is some answer reachable from τ' but not from τ. The modified definitions for *first* and *next* for TSAs are as given below.

Definition 3.7 *First and Next for TSA*

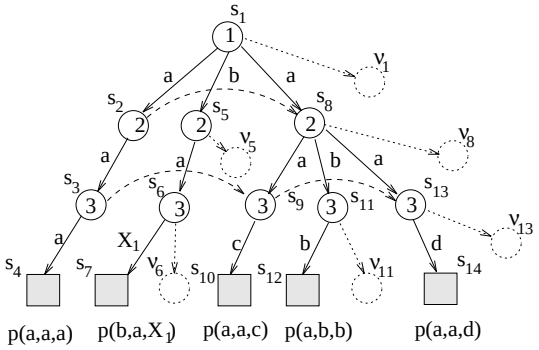
$$\begin{aligned} first(G, ts, s) &= s' \xrightarrow{\alpha} d' \text{ such that } s' \xrightarrow{\alpha} d' \in Safe(G, ts, s) \text{ and} \\ &\quad \forall s'' \xrightarrow{\alpha} d'' \in Safe(G, ts, s) \quad d' \neq d'' \Rightarrow pre(d') < pre(d''). \\ next(G, ts, s \xrightarrow{\alpha} d) &= s' \xrightarrow{\beta} d' \text{ such that } s' \xrightarrow{\beta} d' \in Safe(G, ts, s), \alpha \neq \beta \text{ and} \\ &\quad \forall s'' \xrightarrow{\beta} d'' \in Safe(G, ts, s) \quad pre(d'') > pre(d) \wedge d'' \neq d' \Rightarrow \\ &\quad\quad pre(d') < pre(d''). \end{aligned}$$

Algorithm *Retrieve_Answer* now uses these modified *first* and *next* functions to retrieve answers from a TSA. The soundness theorem for the SFAs (Theorem 3.1) can be established for TSAs when *Retrieve_Answer* is modified as specified above. Moreover, we can establish the following *efficiency* result that exactly characterizes the unifications avoided through the use of threads.

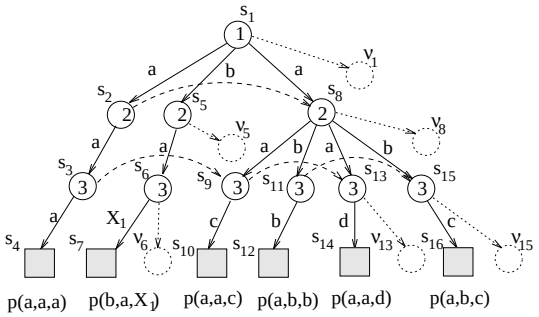
Theorem 3.2 *Let $\langle t_1, \ldots, t_n \rangle$ be the sequence of answers represented by the TSA, and let $\langle t_{r_1}, \ldots, t_{r_k} \rangle$ be its subsequence that unify with G. Then, in the retrieval of t_{r_j} and $t_{r_{j+1}}$ using Retrieve_Answer, the unification operation at a position δ is shared iff (i) $prefix(t_{r_j}, \delta) = prefix(t_{r_{j+1}}, \delta)$, and (ii) there is no $k, r_j < k < r_{j+1}$ such that $prefix(t_k, \delta)$ unifies with $prefix(G, \delta)$ and $prefix(t_k, \delta) \neq prefix(t_{r_j}, \delta)$.*

3.3 From TSA to Dynamic TSA

We now address the problem of retrieving answers from incomplete tables and complete the design of DTSA. To understand the issues here, consider retrieving answers to the goal `p(a,X,c)` using the TSA in Figure 4a. The very first call made by *Retrieve_Answer* will return with `p(a,a,c)`



(a)



(b)

```
algo validate(G, inp_cont_stk, s)
/* (s, inp_cont_stk):
       input answer continuation
   G: Goal */
begin
   let inp_cont_stk = ⟨c_1, ..., c_k⟩
       such that c_i = s_i --□--> ν_i
   if s_k = s then
       n := k
   else
       let s_{k+1} = s
       n := k + 1
   endif
   /* ts: Output cont. stack */
   ts := ⟨ ⟩
   for i := 1 to n − 1
       let τ = next(G, ts, s_i --α_i--> s_{i+1})
       if τ is defined then
           push(τ, ts)
       endif
   endfor
   return ts
end
```

(c)

Figure 5: DTSA corresponding to the TSA of Figure 4a (a), with an added answer(b); Algorithm *Validate* (c).

and the continuation stack $\langle s_8 \xrightarrow{b} s_{11} \rangle$. The next call will fail at s_{13} and will return with no answers and an empty continuation stack. Further invocations of *Retrieve_Answer* will fail to return any answer even when new answers are added to the TSA.

The problem essentially boils down to one of keeping "appropriate" information on the continuation stack to be able to proceed from the point where the last return was made. Observe that due to the total order on the answers, the leaf corresponding to a new answer will have a higher preorder number than the leaf nodes of answers inserted earlier. This observation implies that, in the above example, either s_1, s_8 or s_{13} in Figure 4a are the only states that can be on the paths of any new answer that unifies with the goal `p(a,X,c)`; for instance, when `p(a,b,c)` is added to the TSA in Figure 4a the states s_1 and s_8 will be on its root-to-leaf path.

The idea then is to keep on the continuation stack all and only those states that can be on the paths of new answers that unify with the goal. Operationally this translates to keeping only those states seen on the last root-to-leaf path taken by *Retrieve_Answer* on the stack. To force such states onto the stack we convert a TSA into a DTSA as follows. We associate with every state s_i a special state ν_i with $pre(\nu_i) = \infty$. From every state s_i for which there is no outgoing equivalence link, we create a special transition $s_i \xrightarrow{\square} \nu_i$. (See DTSA in Figure 5a corresponding to TSA in Figure 4a.) The nature of the special transitions is such that although they may be applicable to

any goal, they will never be taken during traversal.

Given the goal `p(a,X,c)`, on invoking *Retrieve_Answer* on the DTSA in Figure 5a, the first answer along with the continuation stack $\langle s_1 \xrightarrow{\square} \nu_1, s_8 \xrightarrow{b} s_{11} \rangle$ is returned. The next call returns with no answer after failing at s_{13} with the continuation stack $\langle s_1 \xrightarrow{\square} \nu_1, s_8 \xrightarrow{\square} \nu_8, s_{13} \xrightarrow{\square} \nu_{13} \rangle$. It is easy to see that whenever we return with no answers the continuation stack will always have a transition $s_i \xrightarrow{\square} \nu_i$ on top.

To retrieve new answers from the information on the stack we proceed as follows. We augment answer continuation to be a pair (s, γ) where s is the state last visited by *Retrieve_Answer* and γ is the continuation stack. Let a DTSA S contain the answer sequence $\{t_1, t_2, \ldots, t_k\}$. Suppose (s, γ) is the answer continuation returned by some invocation of *Retrieve_Answer* over S with $s_i \xrightarrow{\square} \nu_i$ on top of the continuation stack. Assume that answers $\{t_{k+1}, t_{k+2}, \ldots, t_l\}$ have been subsequently added to S, yielding a DTSA S', before the next invocation of *Retrieve_Answer*. To reflect the new answers that have been added we transform γ to γ' such that had we retrieved all the answers, starting with the first one, from S' then γ' would have been the state of the continuation stack on reaching s. For instance, had we retrieved all our answers for `p(a,X,c)` from the DTSA in Figure 5b, on reaching s_{13} the continuation stack will be $\langle s_1 \xrightarrow{\square} \nu_1, s_8 \xrightarrow{b} s_{15} \rangle$. The desired transformation of γ to γ' is performed by Algorithm *Validate* (Figure 5c).

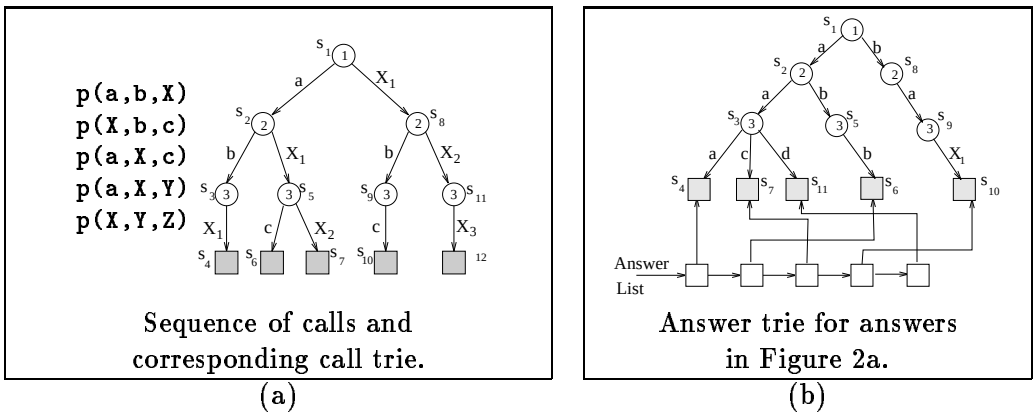
Our modified strategy to retrieve answers from DTSA is as follows. Whenever an invocation of *Retrieve_Answer* returns with no answers and an answer continuation (s, γ), we invoke *Validate* to transform γ to γ'. If no answers are returned using γ' then we assert that there are no new answers to return at the present time. Continuing with our example above, when *Retrieve_Answer* is invoked with the continuation $(s_{13}, \langle s_1 \xrightarrow{\square} \nu_1, s_8 \xrightarrow{\square} \nu_8, s_{13} \xrightarrow{\square} \nu_{13} \rangle)$ on the DTSA in Figure 5b it will fail to return an answer. When *Validate* is called with this continuation it will return the continuation stack $\langle s_1 \xrightarrow{\square} \nu_1, s_8 \xrightarrow{b} s_{15} \rangle$. Now we can retrieve the answer `p(a,b,c)`. It can be established that Theorem 3.1 (soundness) and Theorem 3.2 (efficiency) carry over to DTSA also.

4 Implementation

We now sketch the implementation aspects of tabling operations for a subsumptive engine based on DTSA. Specifically we extend the variant based tabling engine of XSB to support the operations for subsumptive tabling. Our description focuses on operations whose implementation is subtle and interesting while omitting those that are straightforward and routine.

Call-check-insert The call table is organized as a trie (see Figure 6a for an example). Each leaf in the trie represents a call, and the states in a root to leaf path denote positions in a left-to-right preorder traversal of the corresponding call. An edge (s, d) in the trie is labeled with a symbol α such that all calls in the leaves reachable from that edge have α in the position specified by state s. Given a call c, the search for a subsuming call is performed by recursively backtracking through the call trie, trying to match the edge labels with symbols or subterms at the corresponding positions in c. A non-variable label on an edge can match only with an identical symbol in c. If the label on an edge is a variable, say Z, match is done as follows. On the first occurrence of Z, it is bound to the subterm in c, and match succeeds; on subsequent occurrences, the subterm in c must be identical to the term bound to Z for match to succeed. It can be readily established that the above scheme handles non-linear terms correctly.

Note that there may be many calls in the table that subsume a given call c. In such cases,



Sequence of calls and corresponding call trie. (a)

Answer trie for answers in Figure 2a. (b)

Figure 6: Examples of Tries.

for efficient answer resolution, we must find a minimally subsuming call, since if c' subsumes c'', then c' will have at least as many answers as c''. We can find a minimally subsuming call by matching non-variable edges before variable edges at every step. As an immediate consequence, we are guaranteed to find a variant of c whenever one exists. Moreover, if a variant does not exist, the traversal algorithm can be easily extended to insert c in the table without rescanning the longest prefix of c that is already present in the table.

Answer-check-insert An answer table consists of an answer trie and a DTSA. The DTSA is implemented as a threaded tree, with each state s represented by a node, denoted by $node(s)$, in the tree. The tree is implemented using *child* and *sibling* pointers — a standard technique for implementing trees of arbitrary branching factor. The list of children of a node n is called the *child list* of n. Apart from child and sibling fields, $node(s)$ contains fields to store the equivalence link and preorder number of s. The symbol α on a transition $s \xrightarrow{\alpha} d$ is stored in the *atom* field of $node(d)$.

The *answer-check-insert* operation inserts a new answer t into the answer trie as well as the DTSA and eliminates duplicates using the answer trie. Inserting t in the DTSA amounts to adding a rightmost path in the DTSA. To do it efficiently, we maintain the rightmost path of the DTSA in a chain, say r. We traverse t and r simultaneously as long as the the symbols in t and r are identical. Starting from the point of the first mismatch, the symbols in t are inserted into the DTSA and r is updated appropriately.

Although the DTSA itself can be used to perform duplicate checks, using an additional answer trie has several advantages. Observe that each state in the answer trie has a unique prefix, and corresponds to an equivalence class of states in the DTSA. For instance, in Figure 6b the state s_3 in the trie corresponds to the states s_3, s_9 and s_{13} of the DTSA in Figure 5a. As a consequence, duplicate checks can be performed more efficiently using an answer trie. For instance, in the answer trie in Figure 6b, when checking for the existence of an answer `p(a,a,b)`, we will check for an edge with label b from state s_3 of the trie. In contrast, to perform the same check with the DTSA, we will have to check for an outgoing transition with label b in each of the states s_3, s_9 and s_{13}. In addition, new nodes that may be added to the end of equivalence chains during insertion can be added in constant time by maintaining a pointer to the last node in an equivalence chain (of the DTSA) in the corresponding trie node. Finally, the trie, rather than the DTSA, is used to retrieve answers from completed tables (see below). Maintaining a separate trie allows us to reclaim space

on completion of a table by simply deallocating the DTSA.

Retrieve-answer If the answer table is complete then the answer trie is compiled into WAM code and answers are retrieved by backtracking through this compiled trie (see [4] for details). In this case, the WAM choicepoints used for backtracking form the answer continuation. For incomplete tables, if no indexing is needed to retrieve answers from the table (*i.e.*, in the case of variant calls where all answers in the table are relevant), an answer list is used to return answers in sequence. Otherwise, answers are retrieved by traversing the associated DTSA using the technique described in Section 3.

4.1 Comparison with Variant Tabling Implementation

In our implementation we directly borrow the concept of call and answer tries (see Figures 6a,b) from our earlier work on variant tabling for XSB [4]. However, unlike in the variant case, here the *call-check-insert* operation searches for a subsuming call by backtracking through the call trie. More importantly we have replaced the answer list in the answer trie (see Figure 6b) by a DTSA and the corresponding answer retrieval algorithm. Another feature borrowed from our variant tabling is the technique of *substitution factoring* by which we store only the substitutions for variables in a call in the answer table. In our current implementation we apply substitution factoring only to subsuming calls. However it is not difficult to extend the machinery to perform substitution factoring on subsumed calls also and reap the ensuing benefits.

Finally, some remarks are due about the connection between our earlier work on factoring and this paper. We borrowed the idea of SFA from our work in [2]. In that paper SFA was used to develop a compilation technique for efficient sharing of elementary unification operations performed on Prolog clause-heads. But the two critical ingredients of DTSA, namely, indexing and dynamism were absent in that work. Our subsequent work on Clause Resolution Automata [3] described a compile-time transformation technique to share program and answer clause resolution steps based solely on information in program clause-heads. The effect of this technique can be viewed as facilitating sharing among answer tables in a limited sense. The same transformation can also be used as a compile-time optimization for evaluating programs using a subsumptive engine.

5 Performance

We have implemented a subsumption-based tabling system by extending the variant-based tabling operations in XSB (version 1.4.3) with the operations described in previous sections. Below, we first provide experimental results (in Table 1) that measure the effectiveness of our subsumptive engine on program/query pairs that show *subsumptive behavior*: that is, some calls made during evaluation of the query are subsumed by previously made calls. We then report the overheads of the implementation as observed by its performance on programs and queries that do not show subsumptive behavior. We also present an optimization to reduce the overheads of evaluating such programs and show the effectiveness of this optimization (in Table 2).

Effectiveness Table 1 shows the performance of the subsumption-based system relative to that of the native variant-based system on the following programs (that are typically used to stress test deductive database engines): left-, right- and doubly-recursive transitive closure programs that compute the *ancestor* relation (`anc_l`, `anc_r` and `anc_d`), and a program `same_gen` to compute

Program	Query	Database	(size)	Variant	Subsumption	Speedup
anc_r	anc_r(X,Y)	Chain	128	0.73	0.70	1.04
			256	2.59	2.37	1.09
			512	10.42	9.48	1.10
			1024	41.10	35.70	1.15
		Tree	512	0.52	0.58	0.90
			1024	1.36	1.06	1.28
			2048	3.86	2.25	1.72
			4096	12.83	5.85	2.19
anc_d	anc_d(X,Y)	Chain	32	0.36	0.29	1.24
			64	2.42	1.67	1.45
			128	18.18	12.49	1.46
			256	142.51	98.84	1.44
		Tree	512	0.98	1.03	0.95
			1024	2.55	2.56	1.00
			2048	6.51	5.85	1.11
			4096	18.45	13.23	1.39
anc_l	anc_l(1,X), anc_l(2,X)	Chain	512	5.29	0.10	52.90
			1024	21.16	0.22	96.18
			2048	85.92	0.42	204.57
			4096	343.39	0.80	429.24
		Tree	512	3.24	0.10	32.40
			1024	13.03	0.18	72.39
			2048	47.17	0.41	115.05
			4096	185.10	0.81	228.52
same_gen	same_gen(X,Y)	Chain	128	0.34	0.07	4.86
			256	1.20	0.16	7.50
			512	4.63	0.50	9.26
			1024	18.52	1.18	15.75
		Tree	32	0.33	0.22	1.50
			64	2.33	1.41	1.56
			128	16.51	10.19	1.62
			256	125.50	77.56	1.61

Table 1: Performance of subsumption-based tabling.

the *same generation* relation. The queries were evaluated over databases with different structures (Chains and Binary Trees) and sizes[2] The table shows that our implementation of subsumption-based tabling can indeed exploit the subsumptive behavior of these programs effectively. Moreover, observe that the subsumption-based engine often exhibits better time complexity in evaluating programs compared to the variant-based engine. The reason is that while the overheads for subsumption add only a constant factor to the evaluation time, the number of program clause resolution steps saved may, and often does, increase with the size of the program.

The subsumption-based engine is slower than the variant-based engine on two cases in the table: right- and doubly-recursive ancestor programs on trees of size 512. Note that, for a tree-shaped parent database, the atoms at the leaves of the tree cannot be an ancestor of another. In both ancestor programs, half the total number of subgoals of the form `anc(`a`,Z)` (*i.e.*, `anc_r(`a`,Z)` and

[2] All benchmarks were run on a SparcStation 2 using SunOS 4.1.1.

Program	Query	Database (size)		Variant	Subsumption			
					Original		Optimized	
					Time	Speedup	Time	Speedup
anc_l	anc_l(1,Y)	Chain	1024	1.08	1.28	0.84	1.09	0.99
			2048	2.20	2.53	0.87	2.09	1.05
			4096	4.08	4.79	0.85	4.16	0.98
			8192	7.93	9.71	0.82	8.18	0.97
anc_r	anc_r(1,Y)	Chain	128	0.25	0.47	0.53	0.34	0.74
			256	1.06	1.59	0.67	1.08	0.98
			512	4.18	6.25	0.67	4.15	1.01
			1024	16.78	24.31	0.69	16.04	1.05

Table 2: Performance of subsumption-based tabling before and after Lazy Creation optimization.

anc_d(a,Z), where a is some atom drawn from the parent database) fail. Subsumption-based engine treats these calls as dependent on the original goal (anc(X,Y)) and can recognize the failure of these subgoals only when all answers to the original goal have been completed. Hence these programs show a high overhead due to answer scheduling, and exhibit speedups only when the number of program resolution steps avoided — which increases with the size of the parent relation — is sufficient to overcome the overheads. In all other programs, we observe that the overheads for subsumption are low enough to show speedups even when the underlying database is small.

Overheads and Optimization When no two subgoals in a program properly subsume each other, subsumption-based tabling offers no gains to compensate for the overheads due to the more expensive insert operations into call and answer tables. We measure these overheads directly by evaluating programs and queries that do not show subsumptive behavior: the left-recursive ancestor program (anc_l) with the query anc_l(1,Y) and the right-recursive ancestor program (anc_r) with the query anc_r(1,Y). On these queries, we observe a slowdown of 13% to 47% when the subsumption-based engine is used. However, we can eliminate most of these overheads by using the following optimization, called *Lazy Creation*. The central idea of Lazy Creation is to delay the construction of DTSA until the first properly subsumed subgoal is encountered; *i.e.*, a DTSA will be constructed for a subgoal G if and only if another call G' that is properly subsumed by G is made later on. Thus we completely avoid constructing DTSA when no properly subsuming calls are made, thereby enabling the subsumption-based engine to evaluate programs in which only variant calls are made at speeds close to that of the variant-based engine.

Table 2 shows the overheads of performing subsumptive tabling and effect of Lazy Creation optimization on the two sample programs and queries mentioned above. In contrast to the original subsumption engine, the optimized engine does not create DTSAs in either of the two program/query pairs. Hence the only overheads exhibited by the optimized engine are due to the two-phase insert operation used for inserting a subgoal in the call table.

Space Performance Recall that the answer table in our subsumptive engine consists of an answer trie and DTSA. The addition of the latter structure can result in increasing the overall table space by a factor of two when compared to a variant engine. However, answer tables are shared between subsuming and subsumed calls in a subsumptive engine, whereas in a variant engine independent answer tables will be constructed for such calls. Hence sharing of answer tables limits the table space requirments of a subsumptive engine. For instance, we observed that the

answer table space consumed by the subsumptive engine (with Lazy Creation optimization enabled) is 10% to 45% more than that of the variant engine, for the programs in Table 1. Lastly, it should be noted that the call table space requirement of the subsumptive engine never exceeds that of the variant engine.

6 Conclusion

We presented the design and implementation of a novel table organization based on DTSA for subsumption-based tabling. Preliminary experiments indicate that this implementation can effectively exploit the subsumptive behavior of programs and queries, while performing close to the speed of variant engine for programs where only variant calls are made.

The definition of DTSA and the associated retrieval algorithm impose the requirement that the answers retrieved be consistent with the order in which they were inserted in the table. Note that as far as answer resolution is concerned, the order of selection is unimportant; it is imposed merely to ensure that no answer is missed. Whether answers can be efficiently retrieved without imposing such an order remains open.

In a deductive database environment, due to the disk accesses involved, making a few calls that return a large number of answers is clearly more efficient than making a large number of more specific calls. Tabling systems based on subsumption are ideal for such an environment since they reuse answers computed for general calls instead of recomputing them for each specific call. To fully realize this potential in practice, however, new program transformation and query optimization techniques may need to be devised. One such problem is to transform a program so that more general calls are made before specific calls.

Finally, it is interesting to note that TSA can be used for processing clause-heads in Prolog since the selection strategy imposes a top-down order on the clauses. In a similar vein, given the dynamic nature of asserts and retracts, DTSA can in principle be used to handle them. Whether we can gain in performance using TSA and DTSA in these two important applications remains to be investigated.

Acknowledgements

We thank Terrance Swift and David S. Warren for their valuable comments on an early draft of this paper. This work was supported in part by NSF grants CCR-9404921, CCR-9510072, CDA-9303181, CDA-9504275 and INT-9314412.

References

[1] W. Chen and D.S. Warren. Query evaluation under the well-founded semantics. In *ACM Symposium on Principles of Database Systems*. ACM Press, 1993.

[2] S. Dawson, C.R. Ramakrishnan, I.V. Ramakrishnan, K. Sagonas, S. Skiena, T. Swift, and D.S. Warren. Unification factoring for efficient execution of logic programs. In *ACM Symposium on Principles of Programming Languages*, pages 247–258. ACM Press, 1995.

[3] S. Dawson, C.R. Ramakrishnan, I.V. Ramakrishnan, and T. Swift. Optimizing clause resolution: Beyond unification factoring. In *International Logic Programming Symposium*, pages 194–208. MIT Press, 1995.

[4] I.V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D.S. Warren. Efficient tabling mechanisms for logic programs. In *International Conference on Logic Programming*, pages 697–711. MIT Press, 1995.

[5] R. Ramesh and W. Chen. A portable method for integrating SLG resolution into Prolog systems. In *International Logic Programming Symposium*, pages 618–632. MIT Press, 1994.

[6] K. Sagonas, T. Swift, and D.S. Warren. The XSB programmer's manual, Version 1.4.2. Technical report, Department of Computer Science, SUNY, Stony Brook, 1995.

[7] T. Swift and D.S. Warren. Analysis of sequential SLG evaluation. In *International Logic Programming Symposium*, pages 219–235. MIT Press, 1994.

[8] H. Tamaki and T. Sato. OLDT resolution with tabulation. In *International Conference on Logic Programming*, pages 84–98. MIT Press, 1986.

[9] L. Vieille. Recursive query processing: The power of logic. *Theoretical Computer Science*, 69:1–53, 1989.

[10] J. Wunderwald. Memoing evaluation by source-to-source transformation. In *Fifth International Workshop on Logic Programming Synthesis and Transformation*, 1995.