On the Optimality of Scheduling Strategies in Subsumption based Tabled Resolution*

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

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400
email: prasadr@bellcore.com, {cram, ram}@cs.sunysb.edu

Abstract

Subsumption-based tabled logic programming promotes more aggressive reuse of answer tables over variant-based tabling. However resolving subgoals against answers in tabled logic programming may require accessing incomplete answer tables (*i.e.*, more answers remain to be added). In subsumption-based tabling it is far more efficient to retrieve from completed tables. Scheduling strategies promote more frequent usage of such tables by exercising control over access to incomplete tables. Different choices in the control can lead to different sets of proof trees in the search forest produced by tabled resolution. The net effect is that depending on the scheduling strategy used, tabled logic programs under subsumption can exhibit substantial variations in performance. In this paper we establish that for subsumption-based tabled logic programming an optimal scheduling strategy *does not exist— i.e.*, they are all incomparable in terms of time and space performance.

Subsumption-based tabled resolution under *call abstraction* minimizes the set of proof trees constructed. In the presence of call abstraction, we show that there exists a family of scheduling strategies that minimize the number of calls that consume from incomplete answer tables produced by strictly more general calls.

1 Introduction

Tabled resolution for general logic programs [1, 10, 3] as embodied in the XSB system, introduces a new level of declarativeness over traditional (Prolog-like) logic programming systems. Availability of tabled logic programming systems makes it feasible to develop a larger class of efficient declarative solutions to complex applications than heretofore possible using traditional Prolog-based systems. (See [6] for example.)

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 of a subgoal, proceeds as follows. For nontabled predicates the subgoal is resolved against program clauses. For tabled predicates, if the

*Research partially supported by NSF grants CCR-9404921,9510072,9705998 & 9711386, CDA-9303181 and INT-9314412 & 9600598.

†Current Address: Bellcore, 445 South Street, Morristown, NJ 07960.

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 entered in the table. For both tabled and nontabled predicates, program clause resolution is carried out using SLD. While SLD derivations can be captured as a proof *tree*, derivations using a tabled resolution strategy can be viewed as a *forest* of proof trees, with each proof tree corresponding to an answer table.

Following the original formulation of tabled resolution in [10], we say that a subgoal t_1 is present in the table if there already exists another subgoal t_2 in the table that *subsumes* t_1, *i.e.*, t_1 is an instance of t_2[1]. On the other hand one can also say, as is done in the XSB system, that a subgoal t_1 is present in the table only if the table contains a *variant* of t_1 *i.e.* t_2 is identical to t_1 up to renaming of their variables. Since in a subsumption-based system a new call α is resolved using the answers of an earlier call β *even when* α *is a proper instance of* β, subsumption-based tabling promotes more aggressive reuse of computations than variant-based tabling. Note that, in subsumption-based tabling, when a call α is resolved against the answers of a more general call β made earlier, only a subset of the answers in the answer table of β are relevant for α. Hence, we need mechanisms to quickly index into the answers of β that are relevant for α. We showed, via an implementation [8] that, with judicious choice of data structures for representing tables in a subsumption-based system, the aggressive reuse can result in considerable improvement in performance over any tabling system based solely on variant checks.

From a computational viewpoint, resolving subgoals against answers in tabled logic programming systems may require accessing *incomplete* answer tables, *i.e.*, tables where more answers remain to be added. In a subsumption-based system, when answer clause resolution is performed against incomplete tables, we need to retrieve a *subset* of answers from a dynamically growing set. However, answer clause resolution against *complete* tables— tables with no more answers to be added— selects a subset of a static set. The problem of indexing (selecting a subset of answers from) a dynamic set is inherently more expensive than indexing a static set, making incomplete table accesses considerably more expensive than complete table accesses (sometimes by a few orders of magnitude). Therefore, techniques that promote consumption from complete tables over consumption from incomplete tables are central to efficient implementation of subsumptive tabling.

In a tabling system, *scheduling strategies* determine the order in which proof trees of a forest are selected to be explored. (The search within a proof tree itself is determined by the resolution strategy.) Note that, while exploring a proof tree the corresponding answer table is incomplete. Hence we can use scheduling strategies to control access to incomplete tables, and consequently, promote use of complete tables over incomplete tables. In subsumption-based systems, different scheduling strategies can lead to construction of different sets of trees in the proof forest[2]. The net effect is that depending on the scheduling strategy used, tabled logic programs can exhibit substantial variations in performance. The interesting question is whether there exists a scheduling strategy that constructs the minimum number of proof trees for computing all answers to a query, while at the same time, promoting access to complete tables.

In this paper we first establish that for subsumption-based tabled logic programming an optimal scheduling strategy *does not exist— i.e.*, scheduling strategies are all incomparable in terms of time and space performance. In general, incomparability is primarily caused by lack of control over the number of trees that can be generated by a scheduling strategy.

[1] We say that t_1 strictly subsumes t_2 (t_2 is strictly more general than t_1) iff t_1 subsumes t_2 but is not its variant.

[2] Note that the forest generated by any scheduling strategy is *always* a subset of the forest generated by a variant engine.

An intriguing question then is: if all scheduling strategies produce the same forest, then are there nontrivial cost measures under which scheduling strategies can be compared. We show that this is indeed possible for tabling engines augmented with *call abstraction*. In tabled resolution under call abstraction, whenever a call c is encountered, we make a most general call c' such that c' will be eventually made, and let c consume from the the answer table of c'. Subsumption-based tabled resolution with call abstraction constructs the minimum set of proof trees, independent of scheduling strategies. For resolution under call abstraction, we describe a family of scheduling strategies that *minimize* the number of calls that consume from incomplete answer tables produced by strictly more general calls.

1.1 Overview of Proofs

We show that, given any two different scheduling strategies S_1 and S_2, neither is uniformly better (in terms of running time and table space used) than the other over all programs (see Definition 3.3). The above result is established as follows:

- ABSTRACT EXECUTION MODEL (Section 2): We first present a formal definition of a scheduling strategy, based on the development of an abstract execution model for subsumption-based tabling engines.

- EXECUTION TRACE (Section 3): In order to quantify the effect of a scheduling strategy, we develop the notion of an *execution trace*. The trace generated by a scheduling strategy while evaluating a query is simply the sequence of calls made to tabled predicates (see Definition 3.1). Two strategies S_1 and S_2 are deemed different if there is a program/query pair such that the execution trace generated by the two strategies is different. We show that there exist at least two scheduling strategies and a program/query pair such that these two strategies generate different execution traces (see Lemma 3.1).

- THE INCOMPARABILITY THEOREM (Section 3): Given a program/query pair $\langle P, q \rangle$ such that S_1 and S_2 generate traces T_1 and T_2 respectively, where $T_1 \neq T_2$, we derive P' from P such that evaluation of q in P' using S_2 takes longer time than the corresponding evaluation using S_1 (see Theorem 3.2).

  Specifically, suppose that the traces generated by S_1 and S_2 while evaluating q in P' are T_1' and T_2' respectively. By exploiting the properties of the abstract execution model, P' is so constructed that T_1' contains one additional call over T_1, while T_2' has arbitrarily many additional calls over T_2. By suitably adjusting the weight and number of the additional calls, execution of P' using S_2 can be made slower than the corresponding execution using S_1. Using the same proof strategy we can establish that the table space used is also larger.

  Note that, by symmetry, we can derive another program P'' such that evaluation using S_1 is slower (and uses more table space) than evaluation using S_2. Since there exist strategies with different traces (Lemma 3.1), this completes the proof for nonexistence of optimal scheduling strategies.

- SCHEDULING STRATEGIES UNDER CALL ABSTRACTION (Section 4): We first show that, under call abstraction, all scheduling strategies yield the same subgoal dependency graph for any program/query pair (Lemma 4.2). We introduce the concept of *needed subsumptive consumption from incomplete tables* based on the structure of subgoal dependency graphs. We then define *lazy* scheduling strategies that perform only the needed consumption, thereby minimizing the number of subsumptive consumers of incomplete tables.

Implications of the results of this paper are discussed in Section 5. A detailed illustration of the table access problem and the scheduling problem in a subsumption-based system appears in the appendix.

2 Abstract Model of Tabled Resolution

We now describe an abstract model for tabled resolution of positive logic programs based closely on SLG resolution [3].

Notational Conventions We assume familiarity with the standard logic programming definitions of terms, formulas, predicates, clauses, Horn-clause logic programs, mgu, substitution, unification and subsumption [5]. We use $t \uparrow t'$ to say that t and t' unify. For simplicity of exposition our technical development is based on definite Horn clause programs (*i.e.* no negative literals in clause bodies). We use the notation $\langle a_1, a_2, \ldots, a_n \rangle$ to denote the sequence of terms $a_1, a_2, \ldots, a_n$ and $A \cdot B$ denotes the concatenation of two sequences A and B.

Following Prolog convention we use uppercase letters to begin variable names and lowercase letters to begin names of constants. The symbol "_" stands for an anonymous variable. a:-$a_1; a_2; \ldots; a_n$ stands for the sequence of clauses a :- a_1. a :- a_2. ... a :- a_n.

Tabled Resolution: In tabled resolution, derivations are captured as a proof *forest*, with each tree in the forest corresponding to an answer table. The search for answers within each proof tree is controlled by the resolution strategy (*e.g.*, left-to-right literal selection order), whereas the *interleaving* of the searches among individual trees is controlled by the scheduling strategy. The model we present here abstracts away operational details that are irrelevant to the results of this paper. A more fine-grained abstract operational model of tabled resolution can be found in [9].

Tabled resolution associates a *unique* answer table with each tree in the proof forest. Given a program P and a query q, tabled resolution proceeds by building the proof forest using a sequence of the following four *tabling operations*, starting with a PROGRAM RESOLUTION operation for q.

PROGRAM RESOLUTION**:** A proof tree in the forest is extended by one step using OLD-resolution [10].

NEW SUBGOAL**:** This operation is applicable whenever a tabled subgoal g is encountered in the process of program clause resolution. Given a subgoal g, this operation creates a *consumer* node for g in the current proof tree, and if necessary, creates a new proof tree for computing the answers for g using program clause resolution. In a variant-based engine, a new proof tree is created for g whenever there is no proof tree for a variant of g at the time this operation is performed. In a subsumption-based engine, a new proof tree is created for g only when there is no proof tree for any g' such that g' subsumes g, at the time this operation is done. The root of a newly created proof tree is called the *producer* of the tree (and the associated answer table).

NEW ANSWER**:** Applicable whenever a new answer a has been computed for a tabled subgoal g (*i.e.*, whenever a success leaf is derived in the proof tree for g), this operation places a in the answer table for g.

ANSWER RESOLUTION**:** This operation is parameterized by an answer table t to consume from, a subgoal g, and a set of answers A from t that have been previously used for

resolution of g. The operation becomes applicable whenever there is some answer a in t that is not in A; then a is resolved against g.

The construction of the proof forest terminates when none of the above operations can be applied. The above description of tabled resolution follows the development of the operational semantics of SLG resolution in [3, 2, 4]. Note that COMPLETION operation in tabled resolution does not lead to a growth in the forest of proof trees and hence is treated separately from the above four operations. COMPLETION can be considered simply as a marking scheme implemented in the tabling engine by which all mutually dependent subgoals are marked as complete when none of them have pending PROGRAM RESOLUTION or ANSWER RESOLUTION operations. The operation is formalized by first defining the notion of a subgoal dependency graph.

Definition 2.1 (Subgoal Dependency Graph) *The subgoal dependency graph for evaluating all answers to a query q over program P with a scheduling strategy is $G(q, P, S) = (V, E)$ such that:*

(a) V is the set of all subgoals g such that:

(i) a NEW SUBGOAL *for g was applicable during resolution (g is a* consumer*), or*

(ii) a proof tree with g as the root is constructed in the forest (g is a producer*).*

(b) E is the set of edges (g_1, g_2) satisfying one of the following conditions:

1. *g_1 is a producer subgoal and g_2 is a consumer node in g_1's proof tree (*i.e.*,* NEW SUBGOAL *operation for g_2 became applicable when constructing g_1's proof tree).*
2. *g_1 is a consumer node in some proof tree, g_2 subsumes g_1 and answers to g_1 are computed by performing* ANSWER RESOLUTION *against g_2's answer table (due to subsumptive answer clause resolution).*
3. *g_1 is a consumer node in some proof tree and g_1 is a variant of g_2 such that answers to g_1 are computed by performing* ANSWER RESOLUTION *against g_2's answer table (due to variant answer clause resolution).*

We denote the edges that satisfy conditions (1), (2) and (3) above as type-1, type-2 and type-3 edges respectively.

The subgoal dependency relation *depends* is the path relation on the subgoal dependency graph. We denote the set of all subgoals that g depends on by $dep(g)$. We use $\mu(g)$ to represent the set of all subgoals g' that are mutually dependent on g, *i.e.*, $\mu(g) = \{g' \mid g' \in dep(g) \wedge g \in dep(g')\}$. Note that the set of mutually dependent subgoals form a strongly connected component of the subgoal dependency graph.

A subgoal g is deemed complete if no more answers can be added to its answer table. Formally,

Definition 2.2 (Complete) *An answer table for a subgoal g is* complete *iff there are no* PROGRAM RESOLUTION *or* ANSWER RESOLUTION *steps applicable for the proof tree of g, and all answer tables to subgoals in $dep(g)$ are complete. A producer subgoal is said to be complete if its associated answer table is complete.*

In order for the rest of the system to know whether an answer table is complete (so that future subgoals may use more efficient access algorithms to consume from complete tables), tabling engines implement a completion detection procedure that *marks* tables as complete. For efficiency, implementations of completion detection procedures perform the marking as soon as possible. However, for the results presented in this paper, we require that the completion detection procedure satisfies the following weaker property:

Proposition 2.1 *Suppose there are no non-*PROGRAM RESOLUTION *steps applicable for the proof tree of a tabled subgoal p, and suppose that $dep(p) = \phi$. Then the answer table for p will be* marked *as complete as soon as the last* PROGRAM RESOLUTION *step applied to the proof tree of p fails.*

It should be noted that it is straightforward to design completion detection algorithms that satisfy the above requirement (see, *e.g.*, [9]).

2.1 Scheduling of Tabling Operations

Note that, in the above model to construct proof forests for tabled resolution, more than one resolution operation may be applicable at any step. In a sequential implementation of tabled resolution, the order in which these operations are done needs to be determined. Traditionally [11], a unique "active" tree is chosen, and is "grown" using PROGRAM RESOLUTION steps until no more such steps are applicable. Such a configuration is called a *decision point* of the proof forest construction procedure. At a decision point, another tree is selected as active (possibly after performing a tabling operation— *i.e.*, NEW SUBGOAL, NEW ANSWER or ANSWER RESOLUTION), and grown; these steps are repeated until no further operations are applicable. *Scheduling strategies* select the tabling operation to be performed at each decision point, and determine the order in which trees of the proof forest are grown, as described below.

Since we perform program clause resolution using OLD-resolution, there will be at most one PROGRAM RESOLUTION operation applicable for a given proof tree at any step. Moreover, the sequence of PROGRAM RESOLUTION steps either ends in a tabling operation or ends in failure (when there are no more PROGRAM RESOLUTION steps possible for the subgoal). The tabling operations may lead to switching between proof trees (making some other tree active). When a NEW SUBGOAL operation is selected, the next active tree is determined uniquely by the operation itself. After applying NEW ANSWER, however, we can choose to continue exploring the currently active tree (thereby attempting to generate more answers), or apply an ANSWER RESOLUTION operation (to consume the newly generated answer). When an ANSWER RESOLUTION operation is applicable, we can either perform the resolution or choose to generate more answers for some other subgoal (in an attempt to consume from complete tables as far as possible). However, once we choose to perform an ANSWER RESOLUTION operation, note that the active tree remains unchanged.

Scheduling strategies (deterministically) select the next tabling operation to perform, based only on the set of tabled operations on incomplete tables applicable at the current step, and the dependencies between these tables. We assume that complete tables can be treated the same way as static code (as is done in the XSB system where complete tables are "compiled" into WAM code). Hence scheduling strategies do not distinguish between PROGRAM RESOLUTION operations and ANSWER RESOLUTION operations that consume out of complete tables. Furthermore, scheduling strategies do not postpone a NEW SUBGOAL operation. If the NEW SUBGOAL operation creates a new producer, note that there is no information *before* applying the operation about the producer itself, and hence it is consistent with the view that the scheduling decisions be solely based on the current state of the incomplete tables. We concretize the notion of scheduling strategies below.

Definition 2.3 (Scheduling Strategies) *A scheduling strategy is a deterministic algorithm that selects both the tabling operation to be performed and the next active proof tree. The selection is based only on the set of applicable tabling operations, the corresponding proof trees and the dependencies between them. The selection algorithm is such that*

1. *A* NEW SUBGOAL *operation is performed whenever possible;*

2. *If an* ANSWER RESOLUTION *operation is applicable for the currently active tree, it is performed whenever the operation consumes out of a complete answer table and rule (1) is not applicable;*

3. *If a* NEW ANSWER *operation is selected, then the operation is performed and either*

   (a) *the current proof tree stays active, or*

   (b) *a proof tree with an applicable* ANSWER RESOLUTION *operation is made active;*

4. *If an* ANSWER RESOLUTION *operation is selected, then the corresponding proof tree is made active and the operation is performed.*

2.2 Properties of the Abstract Model

The abstract model presented above captures certain essential properties of tabled resolution that are independent of the scheduling strategy used. We formally state these properties as they will be used in our proofs.

First of all, from the definition of scheduling strategies, observe that there is no way to postpone program clause resolution for producers.

Proposition 2.2 *Program clause resolution for a producer is never postponed.*

Secondly, recall that decisions in scheduling strategy are based solely on the state of incomplete tables. Therefore, invoking a tabled subgoal that fails without performing any other tabling operations does not affect any scheduling decisions in the future. Formally,

Lemma 2.3 *Let r be a tabled predicate in a program P such that r does not depend on any tabled predicate (including itself) and r has no answers. Let θ be the sequence of SLG operations produced by a scheduling strategy S while resolving some query q against P such that θ contains* NEW SUBGOAL *for r. Let P' be a program obtained from P by replacing every occurrence of r in P by* `fail`. *Then the sequence of SLG operations θ' scheduled by S while resolving q against P', is identical to the sequence obtained by deleting the occurrence of* NEW SUBGOAL *for r from θ.*

Proof (Sketch): The sequences θ and θ' are clearly the same until the occurrence of NEW SUBGOAL for r. Let A be the set of applicable operations when NEW SUBGOAL of r was selected by S. Since r derives no new answers, the set of applicable operations after the failure of r is $A - \{$ NEW SUBGOAL for r $\}$, which is identical to the set of applicable operations had fail been called instead of r. From Proposition 2.1, r will be marked as complete upon failure. The rest of the proof follows from the fact that scheduling decisions are based only on the state of incomplete tables. ■

3 Nonexistence of Optimal Scheduling Strategy

Using the properties of the abstract model developed in the previous section we now establish the main results of this paper. We first define the concepts used in the development of our main result.

3.1 Trace Equivalence and Optimality

We use the sequence of program clause resolution steps resulting from the evaluation of tabled predicates to distinguish between scheduling strategies.

```
Program P:
:- table p/2, q/1.
w(X) :- p(X,Y),q(a).
p(a,b).
p(a,c) :- q(b).
q(a).
```

$trace_1 = \langle p(X,Y), q(b), \ldots \rangle$

$trace_2 = \langle p(X,Y), q(a), \ldots \rangle$

Figure 1: Existence of strategies that are not trace equivalent.

Definition 3.1 (Trace) *The trace generated by a scheduling strategy S while evaluating query q using program P, denoted $trace_{\langle S,P,q \rangle}$, is an ordered sequence of producers $\langle p_0, p_1, \ldots, p_n \rangle$ where the call p_i is made before p_j for all i and j such that $i < j$.*

Intuitively, the trace of a program evaluation consists of all and only those subgoal invocations that are producers. We use traces to distinguish between scheduling strategies as follows.

Definition 3.2 (Trace Equivalence) *Let S and S' be two scheduling strategies. S and S' are* trace equivalent *iff for all programs P and for all queries q, $trace_{\langle S,P,q \rangle} = trace_{\langle S',P,q \rangle}$.*

We denote trace equivalence by $=_{trace}$, *i.e.*, $S =_{trace} S'$ means that S and S' are trace equivalent.

Lemma 3.1 *There exist at least two strategies that are not trace equivalent.*

Proof: Consider the evaluation of the program in Figure 1 for the query `w(X)`. A proof tree for `w(X)` is initially constructed and, when doing PROGRAM RESOLUTION for `w(X)`, NEW SUBGOAL for `p(X,Y)` is performed, resulting in a new proof tree for `p(X,Y)`. Now, when answer `p(a,b)` is generated by PROGRAM RESOLUTION of `p(X,Y)`, a NEW ANSWER operation becomes applicable. After placing the answer in `p(X,Y)`'s table we can choose to either (i) continue resolution of `p(X,Y)`, or (ii) perform ANSWER RESOLUTION in the proof tree for `w(X)` that has just become applicable. In case (i), the next NEW SUBGOAL operation will be for `q(b)`, while in case (ii) the next NEW SUBGOAL operation will be for `q(a)`. The two scenarios yield two different traces (see Figure 1). ■

The significance of the above lemma is that we can prove results based on trace equivalence without losing generality.

We shall compare two strategies based on the running time and the table-space used for evaluating all answers to a query. We shall use $time_{\langle S,P,q \rangle}$ to denote the time taken by strategy S for evaluating all answers to a query q using program P.

Definition 3.3 (Uniformly Faster) *We say that S is uniformly faster than S' if and only if for all programs P and queries q, $time_{\langle S,P,q \rangle} < time_{\langle S',P,q \rangle}$*

3.2 Formal Presentation of the Proof

Our proof relies on the properties of a predicate `f` defined in Figure 2. `g` is a non-tabled predicate whose rules are a database of facts. Let n_g denote the number of `g` facts. Let P be a tabled logic program that contains `f` and `g`.

```
:- table f/1.
f(X) :- g(Y),fail.
g(1).  g(2).  g(3).  ...g(n_g)
```

Figure 2: Program defining `f` and `g`

Note that every call to f(X) (X can be bound or free) has no answers. Hence, it follows from Lemma 2.3 that a call to f(X) (X either bound or free) does not affect any of the engine's subsequent scheduling decisions. Secondly, the program clause resolution for such a call completely backtracks through g's database of facts. Let τ be the time taken to backtrack through this entire database of facts. τ is proportional to n_g. We can make τ as large as we want by suitably choosing n_g.

We are now ready to establish our main result. We show that if two strategies are not trace equivalent then neither of them can be uniformly faster than the other.

Theorem 3.2 (Incomparability Theorem for Time) *Let S_1 and S_2 be two scheduling strategies such that $S_1 \neq_{trace} S_2$. Then neither S_1 nor S_2 is uniformly faster than the other.*

Proof: Without loss of generality we prove that S_2 is not uniformly faster than S_1 since the other case is symmetric. Since $S_1 \neq_{trace} S_2$ we can find a program P and query q such that $trace_{\langle S_1,P,q\rangle} \neq trace_{\langle S_2,P,q\rangle}$. Either $time_{\langle S_1,P,q\rangle} < time_{\langle S_2,P,q\rangle}$ or $time_{\langle S_1,P,q\rangle} \geq time_{\langle S_2,P,q\rangle}$. In the former case we have nothing left to prove. Hence let us assume the latter, *i.e.,* $\forall P, q \quad time_{\langle S_1,P,q\rangle} \geq time_{\langle S_2,P,q\rangle}$ and prove the above theorem by arriving at a contradiction.

We proceed as follows: Let α and β be the calls where $trace_{\langle S_1,P,q\rangle}$ and $trace_{\langle S_2,P,q\rangle}$ first differ (α is in the trace generated by S_1 and β is in the trace generated by S_2). Let $A{\cdot}\langle\alpha\rangle{\cdot}B_1$ denote $trace_{\langle S_1,P,q\rangle}$. Similarly let $trace_{\langle S_2,P,q\rangle}$ be denoted by $A{\cdot}\langle\beta\rangle{\cdot}B_2$.

We transform P to P' by adding the rules for f, the database of facts for g and two new rules R_α (based on α) and R_β (based on β) to P such that $trace_{\langle S_1,P',q\rangle} = A{\cdot}\langle\alpha$,f(_)$\rangle{\cdot}B_1$ and $trace_{\langle S_2,P',q\rangle} = A{\cdot}\langle\beta$,f(1), f(2), ...f(k-1)f(_)$\rangle{\cdot}B_2$. Notice that the only changes in the trace are the addition of the calls to f. Intuitively this means that the computation not involving f is left unchanged.

R_α and R_β are so designed that when α is called, either R_α alone is triggered or R_α is triggered before R_β; similarly when β is called, R_β alone is triggered or R_β is triggered before R_α.

Let α' denote α in which all its variables are bound to new constants not in P (*i.e.* a ground instance of α). Similarly let β' denote a ground instance of β in which all its variables have been bound to constants not in P or α'. Let R_α be the clause α' :- f(_) and R_β be the clause β' :- f(1);f(2);...;f(k-1);f(_).

Observe that if α subsumes β then α' will not unify with β, but α unifies with β'. Symmetrically, if β subsumes α then β' will not unify with α but α' will unify with β. We add R_α, R_β and the clauses of f and g defined as in Figure 2 (f and g are assumed to be new symbols that do not occur in P) to the beginning of program P to get P'.

To trigger R_α and R_β in the right order we place R_β before R_α if β subsumes α, otherwise we place R_α before R_β. (Note that if α and β are incomparable under subsumption then the order of insertion does not matter.)

We will prove that

(a) $trace_{\langle S_1,P',q\rangle} = A{\cdot}\langle\alpha$,f(_)$\rangle{\cdot}B_1$

(b) $trace_{\langle S_2,P,q\rangle} = A{\cdot}\langle\beta$,f(1), f(2), ...,f(k-1),f(_)$\rangle{\cdot}B_2$.

(c) τ(the time taken to compute f(_)) can be made the dominating component in $time_{\langle S_1,P',q\rangle}$

(d) $k * \tau$ is the dominating component in $time_{\langle S_2,P',q\rangle}$

- Proof of (a): In S_1 the call α immediately triggers R_α. This results in a producer f(_) which is not postponed (by Proposition 2.2).

  The call f(_) fails without computing any answers and returns. Thus the rest of the proof for (a) follows directly from Lemma 2.3.

- Proof of (b): The details of this proof are similar to the proof of (a) and hence omitted.
- Proof of (c): We can choose n_g, the number of facts in the `g`'s database, to make the evaluation of `f(_)` the dominant part in $time_{\langle S_1, P', q\rangle}$.
- Proof of (d): Observe that `f(1), f(2), ...,f(k-1), f(_)` are producers in $trace_{\langle S_2, P', q\rangle}$. Evaluation of each `f(i)` takes time τ and there are k such calls. Hence, $k * \tau$ is the dominating part of $time_{\langle S_2, P', q\rangle}$.

From (a), (b), (c) and (d) above, it follows that $time_{\langle S_2, P', q\rangle} > time_{\langle S_1, P', q\rangle}$, resulting in a contradiction. ■

We have shown that if two strategies are not trace equivalent then neither is better than the other. Now we can use Lemma 3.1 to show the generality of this result, namely:

Theorem 3.3 (Nonexistence of Time-Optimal Scheduling Strategy) *For evaluating definite logic programs using subsumption-based tabling, there exists no scheduling strategy S such that for all strategies S' , if S and S' are different then evaluation under S is uniformly faster than evaluation under S'.*

Proof: By contradiction. Let us assume there exists a strategy S that is optimal. By Lemma 3.1 we can find S_2 that is not trace equivalent to S. If S is optimal then S is uniformly faster than S_2. However this contradicts Theorem 3.2. ■

We can immediately derive two results from the previous theorems. First of all, since the class of positive programs is a subset of the class of normal logic programs, non-existence of optimal strategies for positive programs clearly implies non-existence for the larger class also. Hence,

Corollary 3.4 *For evaluating normal logic programs using subsumption-based tabling, there exists no scheduling strategy S such that for all strategies S' , if S and S' are different then evaluation under S is uniformly faster than evaluation under S'.*

Secondly, from the proof of Theorem 3.2 it is straightforward to show a similar result for table space. We denote the table space consumed by a strategy S for evaluating all answers to a query q using program P by $space_{\langle S, P, q\rangle}$. Then:

Theorem 3.5 (Nonexistence of Space-Optimal Scheduling Strategy) *For evaluating definite logic programs using subsumption-based tabling, there is no scheduling strategy S such that for all scheduling strategies S', if S and S' are different then $\forall P, q \quad space_{\langle S, P, q\rangle} < space_{\langle S', P, q\rangle}$.*

Proof: We use the transformation in the proof of this theorem to obtain P' from P. Observe that the call table that results when S_2 evaluates query q using program P' contains `f(1), f(2),..., f(k-1)`. These calls are absent when S_1 is used for evaluation instead. ■

4 Minimizing Incomplete Table Accesses under Call Abstraction

Observe from the proof of Theorem 3.2 in Section 3 that incomparability of scheduling strategies in subsumption-based engines stems from lack of control over the sets of trees produced. Now suppose all scheduling strategies generate the same forest. Would it then be possible to compare them with respect to specific cost measures? We explore this question in this section. We show that under such conditions we can indeed devise a scheduling

strategy to minimize the number of calls that consume out of incomplete answer tables produced by strictly more general calls.

The differences between the sets of proof trees constructed by different scheduling strategies in subsumption-based tabled resolution arises from the differences in the order in which specific and more general calls are made. To ensure that the same proof forest is constructed irrespective of the scheduling strategy, we can modify the construction process such that proof trees are built only for the most general calls. When the first NEW SUBGOAL operation for subgoal g is encountered, instead of starting a new proof tree for g and computing its answers by PROGRAM RESOLUTION, we build a proof tree for g' such that g' subsumes g and NEW SUBGOAL for g' will be encountered subsequently; we then compute answers for g by ANSWER RESOLUTION. The general subgoal g' is called an abstraction of g and is represented by $abs(g)$. We call this resolution strategy as *tabled resolution under call abstraction*. Note that in practice, $abs(g)$ may not be computable and can only be estimated, as discussed in Section 5.

Observe that the traces generated by scheduling strategies for any program/query pair in variant-based tabled resolution are permutations of one another. Moreover, the sets of calls in traces produced by any subsumption-based resolution are subsets of the set of calls in traces produced by variant-based resolution. This enables us to define $abs(g)$ as:

Definition 4.1 (Call Abstraction) *Let ω be trace produced by a variant tabling engine for a query q and program P. Let C be the set of calls c' such that c' subsumes c and there is no other call c'' in ω that subsumes c'. The abstraction of the call c in ω, denoted by $abs(c)$, is a call c' in C chosen* a priori*, independent of the scheduling strategy.*

We distinguish between two kinds of consumers from an answer table c depending upon the type of edge connecting the producer and consumer in the subgoal dependency graph. Whenever (c', c) is a type-2 edge, then c' is a *subsumptive consumer* of c; when (c'', c) is a type-3 edge, c'' is a *variant consumer* of c. Recall that our original motivation was to minimize the use of incomplete tables by subsumptive consumers, and hence we choose the set of subsumptive consumers to incomplete tables as our cost measure:

Definition 4.2 *The set of subsumptive consumers of incomplete tables is the set of calls c in a subgoal dependency graph G that consume answers out of incomplete tables c' such that (c, c') is a type-2 edge in G.*

Note that in any subgoal dependency graph, for any node c, either there is a single outgoing edge (c, c') of type-2 or type-3, or one or more outgoing edges of type-1. Furthermore, under call abstraction, program clause resolution is done only for the abstracted calls. Therefore,

Proposition 4.1 *Under call abstraction, the subgoal dependency graph for any given program P, query q and scheduling strategy S has the following property: If $abs(c)$ strictly subsumes c then there is a type-2 edge from c to $abs(c)$.*

From the above property, we can show that for any given program/query pair, all scheduling strategies under call abstraction yield the same subgoal dependency graph.

Lemma 4.2 *For any program P the subgoal dependency graph constructed while evaluating all answers to a query q using subsumption-based tabled resolution under call abstraction is independent of the scheduling strategy.*

Proof (Sketch): Let G_1 and G_2 be the subgoal dependency graphs produced by scheduling strategies S_1 and S_2 respectively. Since the subgoal dependency graphs are such that every node is reachable from initial query q, G_1 and G_2 are identical if for each path of length n from q in G_1, there is an identical path in G_2. We show this by induction on path length as follows.

The base case (for 0-length paths from q) trivially holds since both G_1 and G_2 contain q.

Assume, as induction hypothesis, that for every path of length n from q in G_1, there is an identical path in G_2.

Now let $q, p_2, \ldots, p_n, p_{n+1}$ be a path of length $n+1$ starting at q in G_1. From induction hypothesis, there exists a path $q, p_2, \ldots, p_n$ in G_2. We now have three cases, based on the relationship between p_n and p_{n+1}.

Case 1: (p_n, p_{n+1}) is a type-1 edge in G_1. Since type-1 edges correspond to dependencies due to PROGRAM RESOLUTION operations which are independent of scheduling strategies, there is a type-1 edge (p_n, p_{n+1}) in G_2 also.

Case 2: (p_n, p_{n+1}) is a type-2 edge in G_1. Due to call abstraction, $p_{n+1} = abs(p_n)$, and since p_n is in G_2, so is $abs(p_n)$. Hence, by proposition 4.1, there must be a type-2 edge from p_n and $abs(p_n)$ in G_2.

Case 3: (p_n, p_{n+1}) is a type-3 edge in G_1. Again, due to call abstraction, $p_n = abs(p_n)$, otherwise the edge will not be a type-3 edge. Moreover, p_n is a consumer node in G_1. Since there is a unique producer node for each proof tree, p_{n+1} is uniquely determined given p_n. Since $p_n = abs(p_n)$, there will be a producer corresponding to $abs(p_n)$ in G_2 also. Hence (p_n, p_{n+1}) is a type-3 edge in G_2.

Hence $q, p_2, \ldots, p_n, p_{n+1}$ is a path in G_2. ■

We can now characterize subsumptive consumers that always consume from incomplete tables, irrespective of the scheduling strategy. Observe, from Definition 2.2 that if c_1 and c_2 belong to the same SCC in the subgoal dependency graph, c_2 cannot complete unless c_1 has consumed all its answers. Formally,

Proposition 4.3 (Needed Consumption from Incomplete Tables) *For every type-2 edge (c_1, c_2) in the subgoal dependency graph such that $c_1 \in \mu(c_2)$, c_1 consumes out of c_2's incomplete table under any scheduling strategy.*

The above property leads to the definition of scheduling strategies that schedule only the *needed* subsumptive consumers to perform answer clause resolution against incomplete tables.

Definition 4.3 (Lazy Scheduling Strategies) *A scheduling strategy is* lazy *iff for every type-2 edge (c_1, c_2) in the subgoal dependency graph, c_1 consumes out of c_2's incomplete table only when $c_1 \in \mu(c_2)$.*

An immediate consequence of Proposition 4.3 and Definition 4.3 is:

Theorem 4.4 (Optimality of Lazy Scheduling Strategies) *Lazy scheduling strategies minimize the set of subsumptive consumers from incomplete tables.*

5 Discussion

Subsumption based evaluation of tabled logic programs has the potential to yield superior performance over variant engines since they reuse answers computed for more general calls instead of recomputing them for each specific call. To realize this potential, however, one must reduce the overheads associated with accessing answer tables, especially from incomplete tables. Hence scheduling strategies are devised to improve performance by controlling access to incomplete tables. We showed that for subsumption-based tabled resolution there is no scheduling strategy that is uniformly better than any other.

Our proof of non-existence of optimal scheduling strategies also applies to tabled resolution with backward subsumption. When resolving with backward subsumption, PROGRAM RESOLUTION of subgoal g is terminated when a more general subgoal g' is encountered; the remaining answers to g are then computed by ANSWER RESOLUTION, consuming out of the answer table of g'. Note that, the gadget `f/1` used in our proof is such that the particular calls are completed before the general call is made, and hence the proof is not affected by backward subsumption.

The incomparability of scheduling strategies in subsumption based engines arises due to lack of control over the sets of proof trees produced. By ensuring that all scheduling strategies produce the same forest via tabled resolution under call abstraction, we showed that lazy strategies can in fact minimize the set of subsumptive consumers of incomplete tables. Observe that our optimality result above is predicated on call abstraction, which is not computable in general. One can only hope to implement approximations of tabled resolution under call abstraction by using estimates of $abs(g)$ computed via program analysis. Subsumption-based tabled resolution under call abstraction has other significant applications. For instance, 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. Call abstraction can identify the set of most general calls. Hence, from the viewpoint of optimizing subsumption-based tabling engines, development of program analysis techniques for call abstraction is an important and worthwhile research endeavor.

Although scheduling strategies are in general incomparable, our optimality result for lazy strategies seems to indicate that by using tabled resolution under call abstraction scheduling strategies can be compared with respect to other cost measures also. Identifying such measures is an interesting avenue of research.

Finally some remarks are due about scheduling strategies for a variant tabling engine. Since the difference in cost between accessing complete and incomplete tables is insignificant in a variant engine (see [7] for details), minimizing incomplete table accesses does not appear to be a meaningful cost measure. Observe that, in variant-based tabled resolution, the traces generated by different strategies are permutations of one another. So scheduling strategies only control the order in which the search forest is explored. The operations underlying such a control do not lead to arbitrary slowdowns. Hence we strongly conjecture that scheduling strategies for variant-based tabled resolution differ in performance by at most a constant factor.

References

[1] R. Bol and L. Degerstadt. Tabulated resolution for well-founded semantics. In *Proc. of the Symp. on Logic Programming*, 1993.

[2] W. Chen, T. Swift, and D.S. Warren. Efficient implementation of general logical queries. *J. Logic Programming*, September 1995.

[3] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *JACM*, 43(1), 1996.

[4] J. Freire, T. Swift, and D.S. Warren. Taking I/O seriously: Resolution reconsidered for disk. In *Proc of Intl. Conf. on Logic Programming*, 1997.

[5] J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1984.

[6] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. W. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proc. of the 9th Intl. Conf. on Computer-Aided Verification*, 1997.

[7] I.V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D.S. Warren. Efficient table access mechanisms for logic programs. In *International Conference on Logic Programming*, 1995.

```
:- table a/2, h/2.
a(X,Y) :- p(X,Y).

h(X,Y) :- a(A,X),a(A,Y), X ≠ Y.
p(1,2).  p(2,3).  p(1,4).
```

Figure 3: A Tabled Program

[8] P. Rao, C. R. Ramakrishnan, and I. V. Ramakrishnan. A thread in time saves tabling time. In *Proc. of the Joint Int'l Conf. and Symp. on Logic Programming*, 1996.

[9] Prasad Rao. *Efficient Datastructures for Tabled Resolution*. PhD thesis, SUNY at Stony Brook, 1997.

[10] H. Tamaki and T. Sato. OLDT resolution with tabulation. In *Third Int'l Conf. on Logic Programming*, pages 84–98, 1986.

[11] The XSB Group. The XSB tabled logic system v1.7.2, 1997. Dept. of Computer Science, SUNY at Stony Brook. Available by anonymous ftp from `http://www.cs.sunysb.edu/~sbprolog`.

Appendix

Problem Illustration

Consider evaluation of the call `h(X,Y)` made to the tabled predicate shown in Figure 3. Figures 4(a) and 4(b) show the sequence of calls made (a()'s and p()'s) and answers computed (the integers appearing singly or in pairs) by a tabling system for that call.

Note that each column represents a specific call made at a particular time followed by the sequence of answers computed for that call. For instance, in Figure 4(a) the second column represents the call `a(A,X)` (made at time t_2) and its answers `(1,2), (2,3), (1,4)` that were computed and recorded in the table at times t_3, t_6 and t_9 respectively. Observe that when the call `a(1,Y)` is made at time t_4, it is subsumed by the call `a(A,X)` made earlier. Rather than recomputing the answers for `a(1,Y)` through program clause resolution we now resolve it against the answers computed for the call `a(A,X)` (the second column). The only answer computed so far for `a(A,X)` is (1,2) and so we record the first answer `Y=2` for `a(1,Y)` in its answer table (third column) at time t_5. The rest of the entries in Figure 4(a) are computed similarly.

Observe that the set of answers to a subsumed call is a subsequence of the answers computed for the more general subsuming call. For instance, the answer (`Y=4`) to the call `a(1,Y)` is a subsequence of answers (`(1,2),(1,4)`) for the call `a(A,X)`. Secondly, note that on invoking a subsumed call one can either resolve it immediately against the answers computed for the more general call (*i.e.* the answer tables are *eagerly* consumed) or we can choose to suspend it (resulting in *delayed* consumption). Figures 4(a) and 4(b) illustrate the sequence of calls and answers generated by the former and latter choices respectively.

A consequence of eager consumption is that incomplete answer tables may need to be accessed for performing answer resolution. For example, in Figure 4(a), at time t_3 only the first answer to the call `a(A,X)` has been computed. At time t_4 the call `a(1,Y)` has to access this incomplete table. Accessing incomplete tables is expensive since answers to a subsumed call are to be retrieved from a dynamically growing answer table associated with the general call. On the other hand if we postpone doing answer resolution of a subsumed call until the answer table for the general call is complete then we only need to retrieve

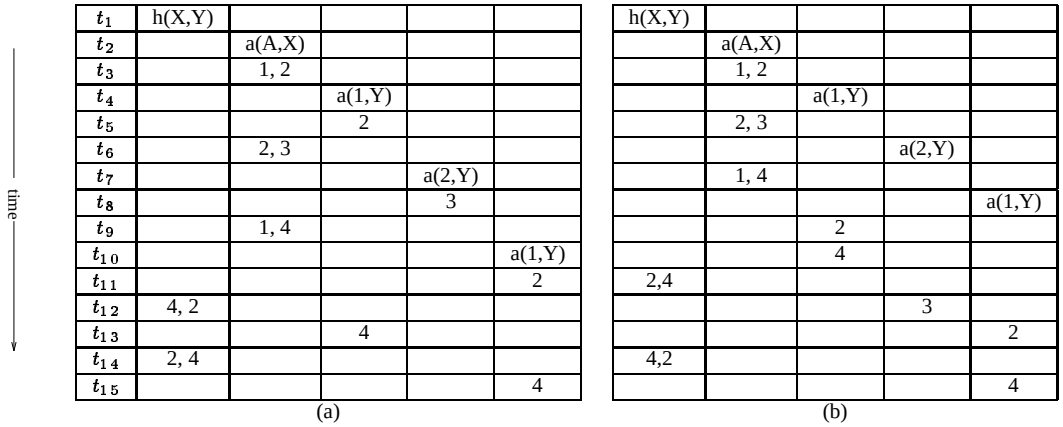

Figure 4: (a)Eager Consumption and (b) Delayed Consumption

```
:- table p/3.
p(A,B,C) :- body of p/3.
q(X,Y) :- p(1,X,Y),
   p(1,X,B),
   p(A1,A2,A3).
q(X,Y) :- p(2,X,Y).
.
.
q(X,Y) :- p(n,X,Y).
```

Figure 5: Illustrative Example

the answers from a static set.(In Figure 4(b) all the calls subsumed by `a(A,X)` consume answers from its answer table only after it is complete.) Hence delayed consumption can result in consuming answers from completed tables and thereby improve efficiency of table access.

Delayed consumption however can introduce new program resolution steps and easily offset these gains. For example, consider the calls resulting from the query `q(X,W),fail` made to the program in Figure 5. Assume that `p(1,X,Y)` succeeds with `X` bound to some constant α. Suppose we defer the consumption of answers until completion of `p(1,X,Y)`. Observe that we will end up exploring alternative paths of computation which will result in calls to `p(2,X,Y)`, `p(3,X,Y)`, ..., `p(n,X,Y)`. All of these calls are solved using program clause resolution. On the other hand by letting `p(1,`α`,B)` eagerly consume answers, `p(A1,A2,A3)` will be called next. All of the calls `p(2,X,Y)`, `p(3,X,Y)`, ..., `p(n,X,Y)` will be made later and will be solved by resolving against answers in the answer table of `p(A1,A2,A3)`. Thus by deferring consumption of answers until completion of the answer table, we have increased the number of program clause resolution steps which could have been avoided by eager consumption. In addition, we also created answer tables for `p(2,X,Y)`, `p(3,X,Y)`, ..., `p(n,X,Y)` and thus increased the *table space* for computing the query `q(X,W)`.