Combining Scheduling Strategies in Tabled Evaluations

Juliana Freire David S. Warren

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

1 Introduction

Tabled evaluations ensure termination for Datalog programs by distinguishing calls to tabled subgoals. Given several variant subgoals in an evaluation, only the first (the generator) will use program clause resolution, the rest (consumers) must perform answer resolution using answers computed by the original invocation. This use of answer resolution prevents the possibility of infinite looping for Datalog programs, which sometimes occurs in SLD. As variant subgoals can be called at different stages of the evaluation, there is an intrinsic asynchronism between the generation and consumption of answers in SLG. Given this asynchrony, implementations of tabled logic programs face an important scheduling choice not present in traditional top-down evaluation: When to return answers to consumer subgoals.

We have experimented with different orders of scheduling the return of answers to consumer nodes as well as the resolution of tabled subgoals, and have derived a number of different scheduling strategies. Each of these strategies has very specific characteristics. *Breadth-First*, for instance, performs a breadth-first (set-at-a-time) search, and in [6] we have shown this strategy is very efficient for evaluating queries which involve relations in external databases while incurring small overheads for in-memory data. Two other strategies were proposed in [5]: Batched Scheduling and Local Scheduling. Batched Scheduling improved on the strategy used in the first implementation of the SLG-WAM [9] both in running time and in memory usage; Local Scheduling has applications to non-monotonic reasoning, and it can arbitrarily improve the performance of some programs that benefit from answer subsumption, such as many aggregate computations [12] and program analyses [2].

Even though a specific strategy can result in considerable speedups for some applications, for others it may add overheads and even lead to unacceptable inefficiency. Since different applications have different requirements, the ability to use multiple strategies in an evaluation is likely to be beneficial. The ideal would be to use a strategy or set of strategies that results in the best performance for a desired application. The importance of providing this kind of flexibility in the evaluation has been identified in deductive databases. In bottom-up systems, the search can be controlled through the use of different rewriting techniques [1]. In Aditi [11], for instance, users may specify at the predicate level not only which transformation to use, but also the evaluation algorithm (e.g., variations of semi-naive [10]).

In this paper, we discuss the issues involved in providing engine support for different scheduling strategies at the predicate level as means of controlling the search in an SLG evaluation. We propose a hybrid strategy that combines Batched Scheduling and Local Scheduling in an SLG evaluation and describe a prototype implementation of this hybrid strategy together with some preliminary performance results.

2 SLG: A Brief Overview

SLG resolution is a partial deduction procedure that is sound and search space complete with respect to the well-founded partial model for all non-floundering queries. This section provides a brief (and informal) overview of SLG; for a more detailed discussion see [4].

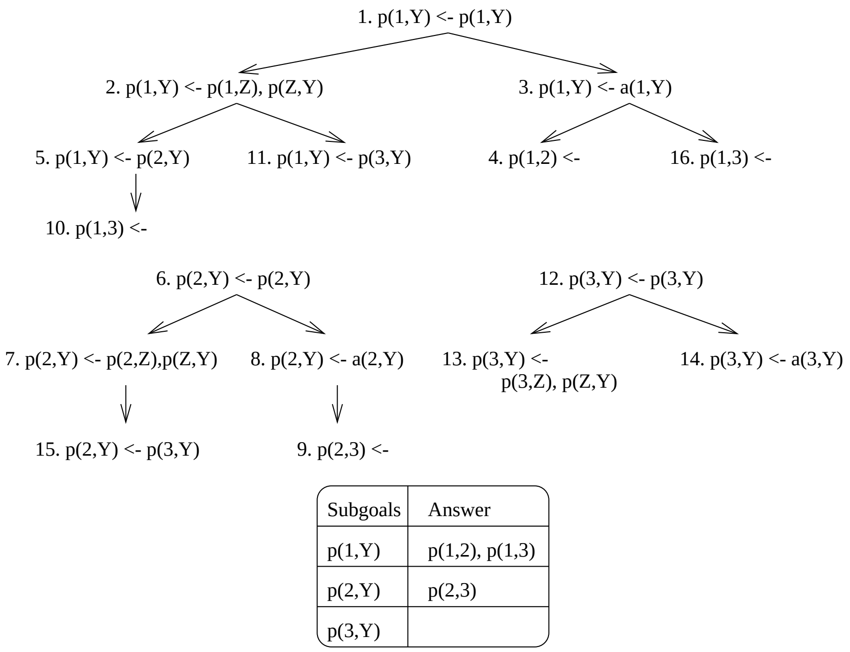


Figure 1: *SLG evaluation*

Example 2.1 Consider the following program

```
:- table p/2.
a(1,2).  a(2,3).  a(1,3).
p(X,Y) :- p(X,Z), p(Z,Y).
p(X,Y) :- a(X,Y).
```

and the query `?- p(1,Y)`, which seeks to find all nodes in the graph defined by the relation `a/2` that are reachable from node `1`. □

SLG evaluations are conveniently modeled as forests of trees as in Figure 1, which represents the transitive closure query of Example 2.1 at the end of its evaluation. The nodes in these trees have the form $AnswerTemplate \leftarrow GoalList$, where $AnswerTemplate$ accumulates the bindings during the evaluation, and $GoalList$ contains the list of literals to be resolved. Consider the operations performed by an SLG evaluation. The first time a subgoal S is encountered during a tabled evaluation, S is registered in the table, and a new tree created with root S. Figure 1 contains trees rooted at nodes `1`, `6` and `12` (we will also refer to these roots as *generator nodes*). Program clause resolution is then used to obtain the immediate children for each root node. In addition, the node calling S becomes a *consuming node*, so named because it will consume answers produced by S's tree. Alternatively, if S is not new to the evaluation (S is contained in the table), no new tree is required for S. However, a *consuming node* is still created for S as in the previous case (in Figure 1 the consuming nodes are 2, 7 and 13). Processing of answers is analogous: the first time an answer to a subgoal is derived during an evaluation it is added to the table and returned to relevant consuming subgoals; any subsequent derivations of the answer are failed. In this manner, redundant subcomputations (including loops) are prevented by tabling. Of course, tabled resolution can be mixed with the program clause resolution of SLD. In this case, we refer to tabled predicates and non-tabled predicates depending on the form of resolution used for each. In an SLG tree, nodes that correspond to non-tabled predicates are termed as *interior nodes*. In Figure 1, nodes 3, 8 and 14 are interior nodes.

In a tabled evaluation, groups of mutually dependent subgoals are called *strongly connected components* or SCCs. When all program and answer clause resolution has been performed for the subgoals in an SCC, the subgoals are termed *completely evaluated* or *completed*. At completion time, all trees for subgoals in the SCC can be disposed since at this point the table contains all pertinent information for the subgoals. The notion of completion is necessary for evaluation of programs with negation, as well as being useful for space reclamation.

To summarize, for definite programs, SLG resolution has four operations:

1. Subgoal Call: creates a consuming node, and if the subgoal is not present in the evaluation a new tree for the subgoal is created.
2. Program Clause Resolution: used for all non-tabled (SLD) subgoals, and for immediate children of the root of each tree.
3. Answer Resolution: resolves the selected literal of a consuming node against an answer from a table.
4. Completion: determines when a set of subgoals is completely evaluated, and disposes of their trees.

In order to handle normal programs, SLG needs the ability to delay ground calls which are involved in negative loops. Accordingly, the notation for nodes in an SLG forest is extended. A non-root node is represented by $AnswerTemplate \leftarrow DelaySet|GoalList$, where $DelaySet$ contains a set of delayed literals. Besides, if the truth value of a delayed literal DL becomes known, the clause where DL appears needs to be simplified. Thus, besides the operations described above, SLG needs two other operations to handle programs with negation:

1. Delay: delays negative literals involved in a negative loop.
2. Simplification: simplifies delayed literals whose truth values are known to be true or false.

The following example illustrates how SLG handles programs with negation.

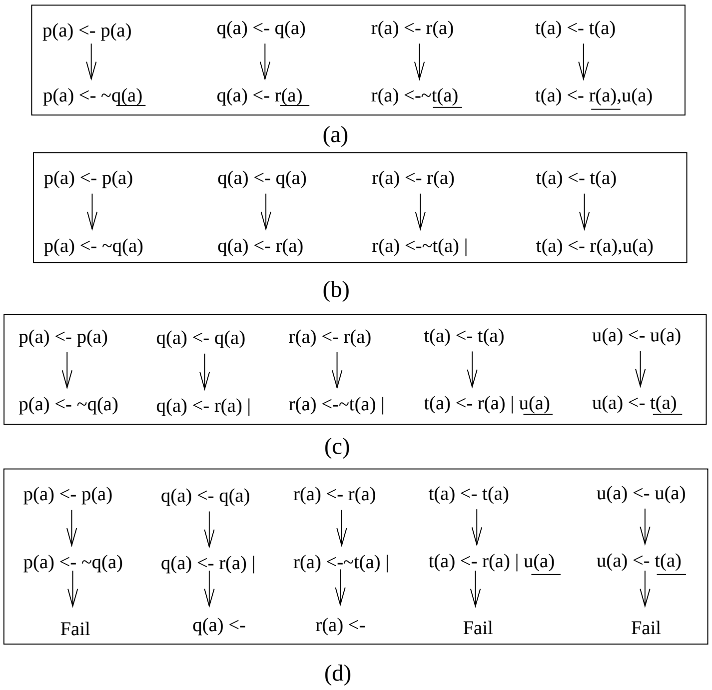


Figure 2: *SLG evaluation of a program with negation*

Example 2.2 Consider the following program

```
:- table_all.
(C1) p(X) :- ~q(X).
(C2) q(X) :- r(X).
(C3) r(X) :- ~t(X).
(C4) t(X) :- r(X), u(X).
(C5) u(X) :- t(X).
```

and the query `?- p(a)`. Figure 2 shows the SLG evaluation of this query.

In Figure 2(a) the leftmost literal of the first four clauses are selected (selected literals are underlined). A negative loop is created between `r(a)` and `t(a)`. The action of SLG is to delay any negative literal involved in a negative loop, and accordingly, `~t(a)` is delayed in the clause `r(a) :- ~t(a) |` (Figure 2(b)). Since there are no more literals to be selected in that clause, this clause becomes a *conditional answer*, that is, `r(a)` is true if `t(a)` is false. This answer can then be returned to the consuming nodes in the clauses for `q(a)` and `t(a)`. When this conditional answer is returned to the clause for `q(a)`, the literal `r(a)` is delayed and a conditional answer `q(a) :- r(a)|` created; and when `r(a)` is returned to the clause for `t(a)`, the literal `r(a)` is delayed and the next literal in the clause is selected (see Figure 2(c)). A call to `u(a)` is made which in turn calls `t(a)`. At this point, there is a positive loop between `u(a)` and `t(a)`, but since both subgoals have been completely evaluated and have no answers, they can be failed. Since the answer for `r(a)` is conditional on `~t(a)`, and `t(a)` is known to be false, this answer can be made unconditional by *simplifying* away the literal `~t(a)`. This step will trigger further simplification: the answer `q(a) :- r(a)|` can be made unconditional (i.e., q(a) succeeds); and as a consequence the answer `p(a)` which is conditional on `~q(a)` fails (Figure 2(d)). □

The SLG-WAM

The data structures and instruction set used by the SLG-WAM are described in [9]; here we briefly summarize aspects of the SLG-WAM needed to describe scheduling strategies.

As mentioned above, there are several types of nodes: *generator*, *consuming*, *interior* and *answer*. Interior nodes are represented in the SLG-WAM by Prolog-style (or interior) choice points. Special choice points are used to represent generator and consuming nodes (information in the consuming choice point will be used to reconstitute the environment in which the subgoal was called, so that answers can be returned to this environment as they are derived). Using these choice points, tabling operations of are reflected more or less directly in SLG-WAM virtual machine instructions.

- TableTry: (implementing SUBGOAL CALL) If a subgoal S is already in the subgoal table, this instruction creates a *consuming choice point*. Otherwise it creates a *generator choice point* for S.
- RetryActive and AnswerReturn: (implementing ANSWER CLAUSE RESOLUTION) RetryActive resolves the selected literal of a consuming node against a set of answers present in a table, whereas AnswerReturn returns a newly created answers to a set of consuming nodes.

Answer nodes are maintained in an explicit table by the instruction NewAnswer:

- NewAnswer: This instruction checks whether an answer is in the table. If so the instruction fails, otherwise the answer is added to the table.

Two other changes must be made to the WAM to support these tabling operations. To see the first change, note that children of a consuming node in one SLG tree may be derived using answers produced by other trees. Indeed, trees may be mutually dependent so that an answer in $tree_1$ is consumed by a node in $tree_2$, which allows the production of a new answer by $tree_2$ to be consumed by $tree_1$. We may thus speak of an asynchronism between the production of answers by one tree and its consumption by nodes in another. To handle this asynchronism, the SLG-WAM must be able to move back and forth between different consuming nodes. The SLG-WAM achieves this by *freezing* the various WAM stacks at the point a new consuming node is created. In fact, the SLG-WAM keeps a *linearized* version of the search space in its stacks (similar to the cactus stacks of OR-parallel implementations such as Aurora [7]). Switching from one environment to another is performed by backtracking to a common ancestor, and then using a forward trail to reconstitute the environments of consuming nodes.

The second change arises from the need to approximate the subgoal dependency graph (SDG), and thus provide incremental completion. The SLG-WAM adds a new memory area to the WAM, the *completion stack*, to keep dependencies among subgoals. Throughout this paper we will distinguish between the SCCs of an SLG system and their (safe) approximation by the completion stack. Notice that an important difference between the WAM and the SLG-WAM is that the trust instruction sets a CheckComplete (implementing COMPLETION) instruction onto the instruction field of the generator choice point, rather than disposing of the choice point as in the WAM. A CheckComplete instruction is thus not invoked until after all program clause resolution is performed for a subgoal. The CheckComplete instruction then uses the completion stack to determine whether a set of subgoals is completely evaluated.

3 Scheduling in SLG

SLG allows an arbitrary computation rule for selecting a literal from a rule body and an arbitrary control (scheduling) strategy for selecting transformations to apply. In our discussion we fix the literal selection (from left to right), and execute clauses in the textual order. The following example illustrates how different control strategies can be devised for SLG.

Example 3.1 Consider the program of Example 2.1 and the corresponding SLG forest in Figure 1. Let us examine the possible actions of SLG for this query. When `?- p(1,Y)` is called, the applicable SLG operations are Program Clause Resolution of the initial query against the two clauses for `p/2`. When the first clause is resolved, node 2 is created with selected literal `p(1,Z)`, a variant of the original query. Node 2 becomes a *consumer*, but since there are no answers available for this subgoal, it is suspended. At this point the only operation left to apply is Program Clause Resolution of node 1 against the second clause of `p/2`, which results in the creation of node 3. Since the selected literal in node 3 is non-tabled, SLD (or program clause) resolution is used. The resolution of the first clause for `a/2` results in a new answer for `p(1,Y)` in node 4 — which makes available Answer Clause Resolution of this new answer against the selected literal of node 2. Note that at this point there is a choice of either returning the newly derived answer in node 4 to the consumer in node 2 (the action taken by Single Stack Scheduling), or resolving the next available clause for `a/2` (the action taken by Batched Scheduling).

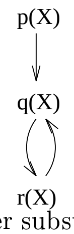
Single Stack Scheduling schedules answers eagerly, as soon as they are created, whereas Batched Scheduling *delays* the return of answers — favoring the execution of Program Clause Resolution and batching the return of answers until no more program clauses are available. Different strategies have different performance behaviors. In [5] we compared implementations of Single Stack Scheduling and Batched Scheduling (for definite programs), and not only Batched Scheduling has proven to be faster, but it also uses significantly less memory than Single Stack Scheduling for a representative set of benchmarks. □

Other strategies are possible. For instance, Local Scheduling can be seen as a variant of Batched Scheduling that tries to completely evaluate subgoals as soon as possible. Evaluation is done one SCC at a time: while an SCC S is not completely evaluated, Local Scheduling prevents answers from being returned to the calling environment of the *leader* of S, the subgoal in S that was first called in the evaluation. By doing this, the exact dependencies among subgoals are preserved during the evaluation. The following example illustrates the actions of Local Scheduling.

Example 3.2 Consider the following program

```
p(X) :- subsumes(min)(q,X),long_computation(X).
q(X) :- r(X).
q(1).
r(X) :- q(X).
r(3).  r(2).
```

SDG



where subsumes(min) is a tabled HiLog [3] predicate that performs answer subsumption and deletes non-minimal answers every time a new answer is added to the table.[1] Given the query `:- p(X)`, the

[1] It is worth pointing out that XSB provides an efficient implementation aggregates using HiLog syntax. For more information on these aggregate predicates, consult the XSB Manual (available at *http://www.cs.sunysb.edu/~sbprolog/manual/manual.html*).

subgoal dependency graph (SDG) for this program is depicted above. Under Local Scheduling, the SCC {`q(X), r(X)`} is completely evaluated before the minimal answer for `q(X)` (`q(1)`) is returned to `p(X)`.

If this query is evaluated under Batched Scheduling, each answer for `q(X)` (`q(3), q(2) and q(1)`) is propagated to `p(X)`, as each of them was minimal at the time it was created. As a result, *long_computation/1* is executed for `X=2` and `X=3` unnecessarily. □

As can be seen from the example above, Local Scheduling can perform arbitrarily better than Batched Scheduling. Evaluating SCC by SCC, and following the exact dependencies of subgoals may be beneficial for a number of applications that use answer subsumption, such as program analyses and aggregate computations.

At the SLG-WAM level, by completing subgoals as early as possible, Local Scheduling may ensue better performance: since accesses to completed tables tend to be more efficient than to non-completed ones, running times can be improved; and by early reclaiming space for completed subgoals, memory usage can be reduced as fewer frames are likely to get trapped on the stacks.

Local Scheduling can also benefit the evaluation of programs with negation. As we have mentioned in Section 2, the SLG-WAM approximates the dependencies between subgoals in the completion stack. As a result, for some programs with negation, false negative loops may be created in the completion stack. In order to avoid unnecessary delaying, the SLG-WAM needs to know the exact dependencies among subgoals so that it only delays negative literals involved in (real) loops through negation. So, during the CheckComplete instruction, if negative dependencies are present, the engine explicitly builds the *exact* SDG of the program to rule out false negative loops [8]. In contrast, an engine based on Local Scheduling can avoid this step: since SCCs are preserved during the evaluation, the completion stack in fact keeps exact dependencies, and negative dependencies are only created if there are actual loops through negation. The following example illustrates how Local Scheduling avoids the creation of extraneous negative dependencies.

Example 3.3 Consider the stratified program

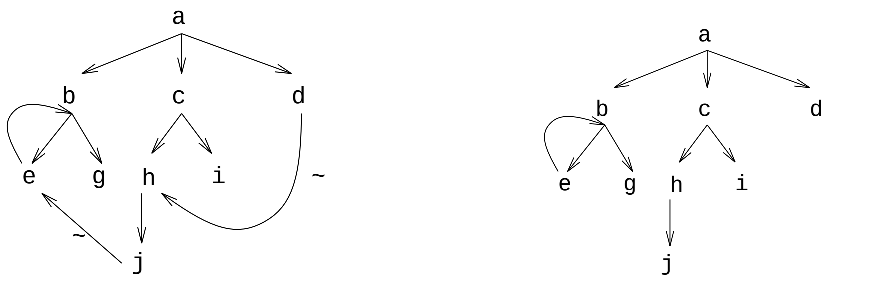
```
:- table a/0,b/0,c/0,d/0,e/0,g/0,h/0,i/0,j/0.

a:-b,c,d.      b:-e.       c:-h.
               b:-g.       c:-i.

d:- ~h.        e:-b.       g.

h:-j.          j:- ~e.     i.
```

for which the query `?- a` is to be evaluated. If evaluated under Batched Scheduling, an SDG will be produced with cascading negative dependencies as shown in Figure 3(a). Even though there is no cycle through negation, the presence of these negative dependencies will result in the explicit construction of the SDG of the program. However if Local Scheduling is used, a *simpler* SDG is created (as depicted in Figure 3(b)). In Local Scheduling, the SCCs {`b, e`} and {`g`} are completely evaluated before `b` returns any answers to `a`. Thus, `e` is completely evaluated when ∼`e` is called and negative dependencies are not created. The negative link from `j` to `e`, and that from `d` to `h` are avoided, since both `e` and `h` are completed by the time they are called negatively. □



(a) Negative dependencies (b) No negative dependencies

Figure 3: *Subgoal dependency graphs for a program under different search strategies*

4 Integrating Different Scheduling Strategies

In the previous section we have mentioned a number of advantages of Local Scheduling over Batched Scheduling:

- As tables are completed as early as possible, memory usage can be reduced; and the more frequent use of completed tables may improve running times;
- programs that benefit from answer subsumption can perform asymptotically better;
- and in programs with negation, extraneous negative dependencies are avoided and evaluation can be simplified; also, better performance might be achieved for non-stratified programs where non-productive computation can be avoided in the presence of superfluous conditional answers.

So, why not always use Local Scheduling? There are a couple of reasons. First of all, even though Local Scheduling may be better for "all-solutions" problems, that might not be the case when all you want is a single answer (e.g., for existential queries). The other reason stems from the cost of delaying the return of answers within an SCC in our particular implementation of the SLG-WAM. In Batched Scheduling variable bindings are shared between the calling environment and the root of an SLG tree: when a new answer is derived in the tree, its bindings are automatically propagated to the calling environment. Since Local Scheduling has to delay answers within an SCC, it cannot allow this optimization, and after the SCC is completed these delayed answers have to be *explicitly* returned, incurring extra costs to copy the answer out of the table, and also requiring extra environment switches. Thus, for programs that do not benefit from answer subsumption, Local Scheduling is likely to perform worse than Batched Scheduling.

As different applications have different requirements, we propose a hybrid strategy that combines Local Scheduling and Batched Scheduling at the predicate level. In what follows we describe how such an evaluation can be formulated for SLG, and implemented as an extension of the SLG-WAM.

Mixed-Strategy SLG-WAM

In a mixed-strategy SLG evaluation, a scheduling strategy (batched or local) is defined for each predicate. In the course of evaluation, depending on the strategy defined for a subgoal, SLG operations might take different actions and/or be scheduled at different times. In what follows we describe a prototype implementation of this hybrid strategy. For more details on the implementation of Batched Scheduling and Local Scheduling the reader is referred to [5].

In the previous section we introduced Local Scheduling, a strategy that evaluates one SCC at a time, following the dynamic dependencies between subgoals. Effectively, Local Scheduling can be seen as variation of Batched Scheduling that adds a barrier at each SCC leader to prevent answers from being returned out of the SCC before it is completely evaluated. Note that non-leader subgoals should propagate their answers to their respective calling environments, so that the SCC they lie in can be completely evaluated.

In order to combine Local Scheduling and Batched Scheduling at the predicate level, the mixed-strategy evaluation needs to enforce the barriers at leader subgoals defined as *local*. Thus, ANSWER RESOLUTION for answers of a *local* leader subgoal LS against its calling environment can only be scheduled after LS's SCC is completed. On the other hand, ANSWER RESOLUTION for answers of any subgoal BS declared as *batched* can be scheduled to the calling environment of BS as soon as the answers are created. This distinction is evident in Algorithm 4.1, which describes the NewAnswer[2] instruction for the mixed-strategy evaluation.

Algorithm 4.1 Integ AnswerReturn($answer$,$subgoal$)

If $answer$ is not in the table for $subgoal$
    Add $answer$ to the table;
    If the scheduling strategy for $subgoal$ is Batched
        Execute the forward continuation to return $answer$ to the calling
        environment of $subgoal$;
    Else if the scheduling strategy for $subgoal$ is Local
        / The return of this answer is delayed until it is known this sub-*
        ** goal is completely evaluated, or is not the leader of an SCC */*
        Fail;
Else fail;

Algorithm 4.2 Integ Find Fixpoint(Subgoal S)

SchedChain = Null;
For each subgoal S' in the ASCC of S
    If S' has a consuming node with unresolved answers
        TmpSchedChain = Integ Schedule Answers(S');
        SchedChain = SchedChain $\cup$ TmpSchedChain;
    If $S' \neq S$
        SchedStrat(S') = Batched;
Return SchedChain;

[2]The SLG-WAM NewAnswer instruction is executed every time an answer is created.

The mixed-strategy evaluation can then be thought of as a variation of Local Scheduling where sets of subgoals (SCCs or approximations of SCCs) are evaluated under a specific strategy — the strategy defined for the leader of each set. Since SCCs are dynamic entities that may change at run-time, an issue to be considered is what actions should be taken when SCCs merge during the evaluation, and some subgoals cease to be leaders. For instance, when from within an SCC SCC_{young} a call is made to an earlier SCC SCC_{old}, these two SCCs as well as all SCCs between them are collapsed into a single SCC SCC_{new}. If an SCC SCC_{local} which was previously being evaluated under Local Scheduling is among the collapsed SCCs, its answers, which are currently being accumulated at the leader LS of SCC_{local}, must be returned to the calling environment of LS, to ensure completeness of the evaluation[3].

Algorithm 4.3 Integ Schedule Answers(Subgoal S)

```
1    SchedChain = Null;
     While there exists a consuming node Cons_S
     whose selected literal is S
        If Cons_S has unresolved answers
5          Add Cons_S to SchedChain;
     If the scheduling strategy of S is Local and S is not the
     leader of its SCC
        Add the generator-active choice point of S to SchedChain;
     Return SchedChain;
```

It is during the CheckComplete operation that the SLG-WAM checks whether fixpoint has been reached, that is, if all answers have been returned to the existing consuming nodes. The fixpoint procedure, which is part of the SLG-WAM CheckComplete instruction, is given as in Algorithm 4.2. During the fixpoint check, the engine iterates through the subgoals in the ASCC SCC_S, whose leader is the subgoal S, and for each subgoal S' in SCC_S it checks whether there is any available Answer Clause Resolution for the consuming nodes of S', and if that is the case, these answers are scheduled (lines 3-4). Under the mixed-strategy, besides unresolved answers for consuming nodes, the engine must schedule any answers for non-leader local subgoals (line 6-8 of the Integ Schedule Answers procedure described in Algorithm 4.3). It is worth pointing out that all the non-leader subgoals, regardless of the strategy originally defined for them, must propagate their answers to their calling environments — effectively their strategy is set to *batched*. Thus, statically defined strategies may be overridden at run-time.[4]

A final issue to consider for the mixed-strategy evaluation is how to handle programs with negation. Recall that in a pure Local Scheduling evaluation, as exact dependencies are followed, negative dependencies are only created if there is a negative loop. When Local Scheduling is combined with Batched Scheduling, this property no longer holds. Therefore, when negative dependencies involve subgoals defined as *batched*, before these subgoals are delayed, checks are needed to verify whether the dependencies are indeed part of a negative loop. Accordingly, the SLG-WAM CheckComplete instruction for the mixed-strategy evaluation is defined in Algorithm 4.4.

[3]Note that no special actions are needed for SCCs whose leaders are defined as *batched*.

[4]As an optimization, the strategy for any non-leader subgoal is set to batched (lines 6-7 of Algorithm 4.2).

Algorithm 4.4 Integ CheckComplete(Subgoal S)

```
If S is the leader of an approximate SCC ASCC_S
    FixpointSchedChain = Find Fixpoint(S)
    If FixpointSchedChain is empty
        If there are negative dependencies within ASCC_S
            If SchedStrat(S) == batched
                Build the SDG for the ASCC of S;
                If the SDG has cyclic negative dependencies
                    Delay subgoals involved in negative loops;
            Else delay subgoals involved in negative loops;
        else
            Mark all subgoals in ASCC as completed;
            Reclaim stack space for the subgoals in ASCC_S;
            If SchedStrat(S) == Local
                return the answers to the generator-consuming of S
    Else backtrack to return unresolved answers;
Else backtrack to previous tabled subgoal;
```

The mixed-strategy evaluation has been implemented on top of the SLG-WAM of XSB. As a first approach, we provide compiler directives so that users can specify for each predicate or set of predicates which scheduling strategy to use, and as we have shown in Algorithms 4.1–4.4, dynamic tests have been added to SLG-WAM operations that take different actions according to the scheduling strategy. Given the dynamic nature of SCCs, some of these checks need to be performed at run-time, but others can be compiled away. For example, the AnswerReturn instruction for a subgoal S could be specialized with respect to the scheduling strategy of S.

5 Experimental Results

In this section we compare the performance of the following engines, which differ only in the scheduling strategy used:[5]

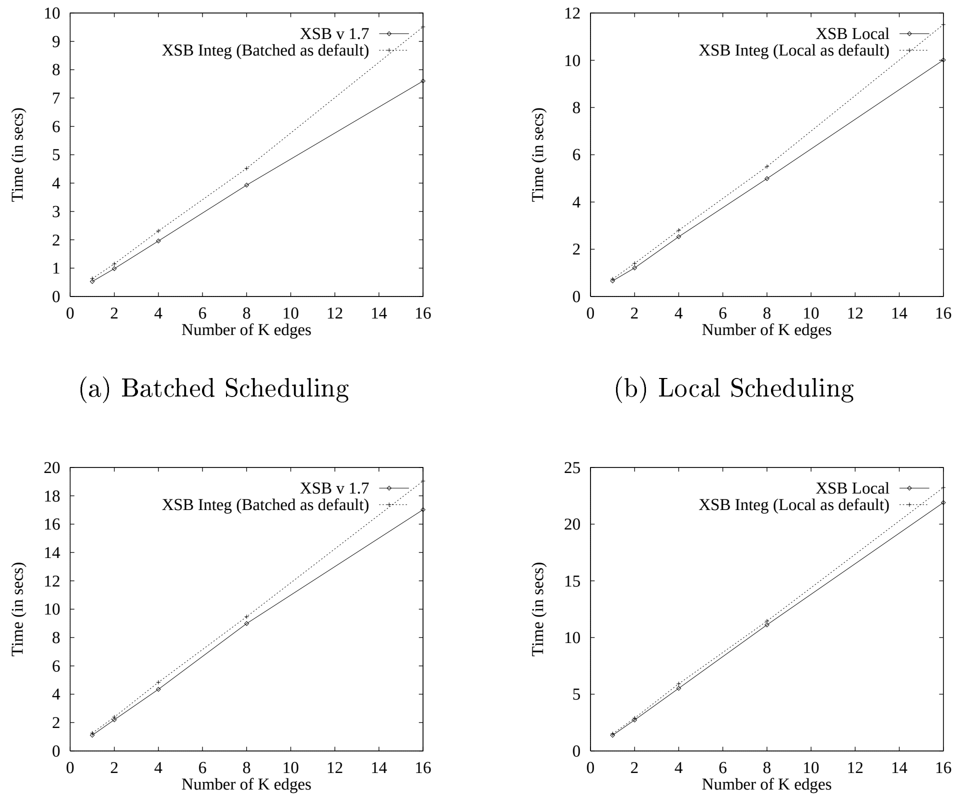
- XSB v. 1.7: uses Batched Scheduling [5].
- XSB-Local: uses Local Scheduling [5].
- XSB-Integ: combines Batched Scheduling and Local Scheduling at the predicate level (Section 4).

We consider both execution time and memory usage of SLG-WAM engines as well as the dynamic count of SLG-WAM instructions and operations. Benches were run on a SPARC2 with 64MB RAM under SUNOS. We will show that, even though the preliminary implementation of XSB-Integ adds some overheads over the single-strategy engines, it can be arbitrarily better for some applications.

[5]XSB v. 1.7 and XSB-Local are freely available at `www.cs.sunysb.edu/~sbprolog`. XSB-Integ is available upon request.

Combining Strategies

The implementation of XSB-Integ described in Section 4 incurs the cost of dynamically checking the strategies of each subgoal at some SLG operations. In order to measure this cost, we compared the running times of XSB-Integ using Batched Scheduling as default against XSB v. 1.7, and XSB-Integ using Local Scheduling as default against XSB-Local. The bench program used was left-recursive transitive on linear chains and complete binary trees. As Figure 4 shows, for these examples the overheads of XSB-Integ range between 5 and 25%.



(a) Batched Scheduling (b) Local Scheduling

(c) Binary trees (Batched) (d) Binary trees (Local)

Figure 4: *Running times for transitive closure on chains and trees of varying size*

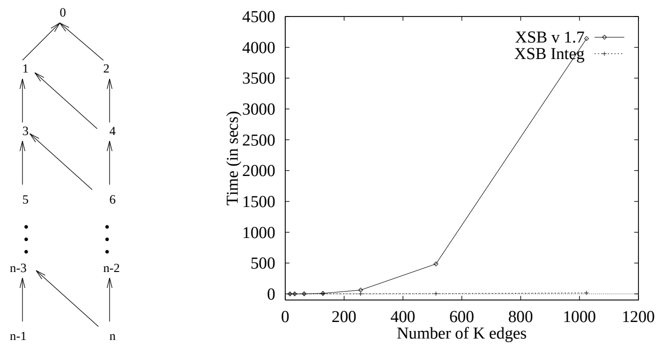
A question then arises: Is it really worth combining these two strategies given these overheads? In Section 4 we mentioned a couple of situations where either Local Scheduling or Batched Scheduling would lead to unacceptable inefficiency, and below we give concrete examples.

Example 5.1 Consider the following variation of the same generation program which finds the smallest distance between two people in the same generation:

```
sgi(X,Y)(I) :-
```

```
        ancestor(X,Z),
        subsumes(min)(sgi(Z,Z1),I1),
        ancestor(Y,Z1),
        I is I1+1.
    sgi(X,X)(0).
```

Figure 5(b) shows the running times of XSB-Integ and XSB v. 1.7 for finding the shortest-path between *n-1* and *n* for varying *n* in graphs such as the one in Figure 5(a). Note that XSB-Integ performs asymptotically better than XSB v. 1.7: the times for XSB v. 1.7 vary between 0.08 and 4142.82 secs, whereas for XSB-Integ they vary between 0.07 and 16.21 secs. □



(a) (b)

Figure 5: *(b) shows the execution time for the query* `subsumes(min)(sgi(n-1,n),I)` *on graphs of the form depicted in (a) for varying* `n`

Example 5.2 Given a graph, suppose we want to find out whether there exists a path between any two nodes (e.g., we issue the query `path(X,Y)`). If we use XSB-Local to answer such a query, it will compute all the existing paths before returning the first answer. Just to give an idea, for a linear chain with 1024 nodes, XSB-Local takes 63.45sec to return the first answer, whereas XSB-Integ returns it *immediately* (in less than a hundredth of a second). □

Now suppose there is an application where both queries above are needed:

```
:- path(X,Y), subsumes(min)(sgi(n-1,n),I).
```

For this query, as the graphs grow larger, XSB-Integ can be arbitrarily better than either single-strategy engine. The times for the different engines to compute the three queries described above are given in Table 1.[6] These times indicate that a mixed-strategy evaluation can indeed be the best alternative for applications that benefit from both Local and Batched Scheduling.

[6]Note that for XSB-Integ `path/2` is declared as batched and `subsumes(min)/2` as local.

Table 1: *Times (ins secs) for different engines*

	XSB-Integ	XSB v. 1.7	XSB-Local
1. `path(X,Y)`	0.001	0.001	61.909
2. `sgi(255,256,D)`	1.1	65.4	1.13
3. `path(X,Y),sgi(255,256,D)`	1.121	65.09	64.94

The current release of XSB (v. 1.7) has two compilation options that allow users to build engines based either on Batched Scheduling or Local Scheduling. The mixed-strategy evaluation will be added as a third option. Ideally, we would like to have the mixed-strategy evaluation as the standard strategy for XSB, but even though the overheads of our implementation are relatively small, they might negatively impact some applications. It is likely that there is room for optimization in our implementation, and we intend to pursue that. Also, even though not all scheduling decisions can be made at compile time — as this integrated evaluation is based on SCCs, which are *dynamic* entities — we would like to explore the possibility of adding strategy-specific SLG-WAM instructions and performing compile-time analysis to further reduce these overheads.

6 Conclusions and Future Directions

Different scheduling strategies have been devised for SLG. Even though these strategies perform well in general, for some applications a specific strategy might add overheads or even lead to unacceptable inefficiency. In order to address this problem, we propose a hybrid strategy that combines Local Scheduling and Batched Scheduling at the predicate level, letting the user control the tabled search in order to obtain the best performance possible. Our preliminary implementation of an engine with combined strategies adds small overheads — mostly due to runtime checks needed to support the mixed-strategy evaluation — but as we have shown in Section 5 it can be arbitrarily faster than either of the single-strategy engines.

There are a number of issues we plan to address in future work: we would like to add Breadth-First [6] as another scheduling option for the mixed-strategy evaluation; we intend to explore the possibility of adding strategy-specific SLG-WAM instructions to reduce the number of run-time checks to support the mixed-strategy evaluation, as well as investigate compile-time analyses to further improve the efficiency of the evaluation. Further research is needed to assess the possibility of automatically inferring for each predicate which strategy might result in the best performance.

References

[1] C. Beeri and R. Ramakrishnan. On the Power of Magic. *JLP*, 10(3):255–299, 1991.

[2] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *JLP*, 10:91–124, 1991.

[3] W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher-order logic programming. *JLP*, 15(3):187–230, 1993.

[4] W. Chen and D.S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *JACM*, 43(1):20–74, 1996.

[5] J. Freire, T. Swift, and D.S. Warren. Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. *Journal of Functional and Logic Programming*, 1997.

[6] J. Freire, T. Swift, and D.S. Warren. Taking I/O seriously: Resolution reconsidered for disk. In *Proceedings of the International Conference on Logic Programming (ICLP)*, pages 198–212, 1997.

[7] E. Lusk et al. The Aurora or-parallel Prolog system. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 819–830. ICOT, 1988.

[8] K. Sagonas, T. Swift, and D.S. Warren. An abstract machine for computing the well-founded semantics. In *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP)*, pages 274–289, 1996.

[9] T. Swift and D.S. Warren. An Abstract Machine for SLG Resolution: Definite Programs. In *Proceedings of the International Symposium on Logic Programming (ILPS)*, pages 633–654, 1994.

[10] J. Ullman. *Principles of Data and Knowledge-base Systems Vol I.* Computer Science Press, 1989.

[11] J. Vaghani, K. Ramamohanarao, D.B. Kemp, Z. Somogyi, P.J. Stuckey, T.S. Leask, and J. Harland. The Aditi deductive database system. *The VLDB Journal*, 3(2):245–288, 1994.

[12] A. van Gelder. Foundations of Aggregation in Deductive Databases. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 13–34, 1993.