Beyond Depth-First Strategies: Improving Tabled Logic Programs through Alternative Scheduling*

Juliana Freire† Terrance Swift David S. Warren

27 April, 1998

Abstract

Tabled evaluation ensures termination for programs with finite models by keeping track of which subgoals have been called. Given several variant subgoals in an evaluation, only the first one encountered will use program-clause resolution; the rest will resolve with the answers generated by the first subgoal. This use of answer resolution prevents infinite looping that sometimes happens in SLD. Because answers that are produced in one path of the computation may be consumed, asynchronously, in others, tabling systems face an important scheduling choice not present in traditional top-down evaluation: when to schedule answer resolution.

This paper investigates alternate scheduling strategies for tabling in a WAM implementation, the SLG-WAM. The original SLG-WAM had a simple mechanism for scheduling answer resolution that was expensive in terms of trailing and choice-point creation. We propose here a more sophisticated scheduling strategy, batched scheduling, which reduces the overheads of these operations and provides dramatic space reduction as well as speedups for many programs. We also propose a second strategy, local scheduling, which has applications to nonmonotonic reasoning, and when combined with answer subsumption, can arbitrarily improve the performance of some programs.

*A preliminary version of this paper appeared in [FSW96].

†This research was performed while the first author was at the Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY.

1 Introduction

Tabling extends the power of logic programming, since it can be used to compute recursive queries at the speed of Prolog but with much better termination properties. This property has led to the use of tabled logic programming for new areas of logic programming. These include not only deductive database-style applications, but other fixpoint-style problems, such as program analysis [DRW96, CDS97], compiler optimization [DRSS96], and model checking [RRR+97]. Ensuring that these new applications run efficiently may require the use of different *scheduling strategies*. The possibility of different useful strategies derives from an intrinsic asynchrony in tabling systems between the generation of answers in one path of a computation and their return to a given consuming tabled subgoal in another path. Depending on how and when the return of answers is scheduled, different strategies—and by implication, searches—can be formulated. Furthermore, these different searches can benefit the research and industrial applications that have begun to emerge.

The efficient evaluation of queries to disk-resident data provides a clear instance of how a scheduling strategy can benefit an application. Efficiently accessing the disk requires a strategy analogous to the semi-naive evaluation of a magic-transformed [BR91] program. A separate paper [FSW97] showed how this could be done using a breadth-first, set-at-a-time tabling strategy for the SLG-WAM [SW94a], the abstract machine of the XSB system.[1] Unlike XSB's original tuple-at-a-time engine, the engine based on the breadth-first strategy showed good performance for disk accesses.

Of course, tabled evaluations must also be efficient for in-memory queries. [SW94b] showed that under several different criteria of measurement, tabled evaluation incurred a minimal execution time overhead compared to Prolog. However, *memory* is also a critical resource for logic programming computations, and memory management for tabled logic programs is complicated by the fact that stack space for a consuming tabled subgoal can be reclaimed only after all answers have been returned to it. Since a scheduling strategy can influence when this condition happens, it can affect the amount of space needed for a computation.

Finally, a number of tabling applications require more than the simple

[1] XSB is a research-oriented logic programming system, and it is freely available at http://www.cs.sunysb.edu/~sbprolog.

recursion needed for pure Horn clause programs. Resolving a call to a negative literal requires completely evaluating the subgoal contained in the literal, along with all other dependent subgoals. In a similar manner, waiting until part of an evaluation has been completely evaluated can also benefit programs that use *answer subsumption* (e.g., [KKTG95]), in which only the most general answers need to be maintained and returned to consuming subgoals. The ability to return only the most general answers out of a table can be useful for program analyses (see e.g., [DRW96, JBD95]), for deductive database queries that use aggregates [vG93], and for answers involving constraints [Tom95].

This paper motivates and describes the design and implementation of two new scheduling strategies for tabled logic programs:

- We describe *batched scheduling* along with an instruction set that has been used to implement this strategy. Batched scheduling is highly efficient for in-memory programs that do not require answer subsumption.
- We describe *local scheduling*, which provides a useful strategy for evaluating both fixed-order stratified programs and programs that use answer subsumption. We also describe the instruction set that is used to implement this strategy.
- We provide detailed results of experiments comparing these two strategies with XSB's original *single-stack scheduling* (described in [SW94b]), showing that:
  - batched scheduling can provide an order of magnitude space reduction over the original strategy, as well as reliably provide a significant reduction in time; and
  - local scheduling can provide large speedups for programs that require answer subsumption, while incurring a relatively small (constant) cost for programs that do not.

2 SLG for Definite Programs: Implementation

2.1 Review of Tabling for Definite Programs

Linear resolution with a selection function for general logic programs (SLG resolution) [CW96] is a tabled evaluation method that is sound and search-space complete with respect to the well-founded partial model for all non-floundering queries. In this section, we review tabling using the notation of SLG resolution reformulated and simplified for definite programs.

As preliminary terminology, subgoals and goals are atoms. Predicates can be annotated as either tabled or nontabled, in which case SLD resolution is used. Evaluations are modeled by a sequence of *systems* or *forests* of *SLG trees*. Figures 1, 4, and 6, which will be discussed in detail below, illustrate forests for a query to a simple recursive program.

Definition 1 (SLG System) *An SLG system consists of a forest of SLG trees. The root nodes of SLG trees have the form:*

$$subgoal \leftarrow subgoal$$

A root node may be marked as completed *when its corresponding tree is* completely evaluated *(Definition 4). In a system, no two trees have the same root nodes; that is, their corresponding subgoals cannot be renaming variants of each other. Nonroot nodes have the form:*

$$answer\_template \leftarrow goal\_list$$

The answer_template is an atom used to represent variable bindings made to the tabled subgoal during the course of resolution, and the goal_list contains a sequence of unresolved atoms.

We assume throughout this paper a left-to-right computation rule, so that the selected literal of a node is always the leftmost literal in the *goal_list*. Two nodes are identical if they are variants when taken as terms. We will slightly abuse notation by allowing resolution of a clause against the selected literal of a node.

An SLG evaluation is defined using SLG systems.

Definition 2 (SLG Evaluation) *Given a definite program P, an SLG evaluation $\mathcal{E}$ for a tabled goal* root *is a sequence of forests of SLG trees, $\mathcal{F}_0$,$\mathcal{F}_1$,...,$\mathcal{F}_n$, such that:*

- *$\mathcal{F}_0$ is the forest containing a single tree, root $\leftarrow$ root, and*
- *for each finite ordinal $n+1$, $\mathcal{F}_{n+1}$ is obtained from $\mathcal{F}_n$ by an application of one of the SLG operations (see Definition 3).*

If no operation is applicable to $\mathcal{F}_n$, $\mathcal{F}_n$ is called a final system *of $\mathcal{E}$.*

New SLG trees are created by the operation SUBGOAL CALL when tabled subgoals that are new to the evaluation become selected literals of nodes. Roots of trees are sometimes called *generator* nodes. The operation PROGRAM CLAUSE RESOLUTION is used to produce the children of generator nodes and of *interior* nodes whose selected literals are nontabled. If the selected literal of a node is tabled, the node is designated as *consuming*, and its children are produced by ANSWER CLAUSE RESOLUTION. An *answer* is a leaf node whose *goal_list* is empty. These operations are summarized in Definition 3.

Definition 3 (SLG Operations)

- SUBGOAL CALL: *Let N be a nonroot node answer_template $\leftarrow S$, goal_list where S is tabled; if S is new to the evaluation, add a new tree with root $S \leftarrow S$.*
- PROGRAM CLAUSE RESOLUTION: *Let N be a root node $S \leftarrow S$, and let C be a program clause head $\leftarrow$ body where head unifies with S with mgu θ. Add $(S \leftarrow body)\theta$ as a child of N, if it is new.*

  Alternatively, let N be a nonroot node answer_template $\leftarrow S$, goal_list with S nontabled. Again, let C be a program clause clause head $\leftarrow$ body where head unifies with S with mgu θ. Add (answer_template $\leftarrow$ body, goal_list)θ as a child of N.
- ANSWER CLAUSE RESOLUTION: *Let N be a nonroot node whose selected literal S is tabled, and Ans be an answer node. Also, let N' be the resolvent of N and Ans on S. Then if N' is not a child of N, add N' as a child of N.*

- COMPLETION: *Given a set $\mathcal{S}$ of subgoals that is completely evaluated (Definition 4), mark all root nodes of the trees for subgoals in $\mathcal{S}$ as completed.*

Example 1 *As an illustration of the operations in Definition 3, consider the following double-recursive transitive closure:*

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

and the query ?- p(1,Y). *An SLG system for this program is shown in Figure 1. The initial SLG system begins with node 1. Nodes 6 and 12 are produced via* SUBGOAL CALL *operations. Nodes 2, 3, 7, 8, 13, and 14 are produced by* PROGRAM CLAUSE RESOLUTION *as applied to root nodes, while nodes 4, 9, and 20 are produced by* PROGRAM CLAUSE RESOLUTION *from interior nodes. Finally, nodes 5, 11, and 18 are produced by* ANSWER CLAUSE RESOLUTION.

In the system depicted in Figure 1, the sets of subgoals {p(3,Y)}, {p(2,Y), p(3,Y)}, *and* {p(1,Y), p(2,Y), p(3,Y)} *are all completely evaluated—the evaluation has produced all possible answers for subgoals in these sets. A* COMPLETION *operation can be applied to any of these sets of subgoals.*

Definition 4 (Completely Evaluated) *Given an SLG system and a set $\mathcal{S}$ of subgoals, $\mathcal{S}$ is* completely evaluated *if at least one of the following conditions is satisfied for each $Subg \in \mathcal{S}$.*

1. *Subg has an answer that is a variant of Subg; or*
2. *for each node with selected literal SL in the tree with root Subg,*
   (a) *SL is completed; or*
   (b) *$SL \in \mathcal{S}$, and there are no applicable* SUBGOAL CALL, PROGRAM CLAUSE RESOLUTION, *or* ANSWER CLAUSE RESOLUTION *operations for SL.*

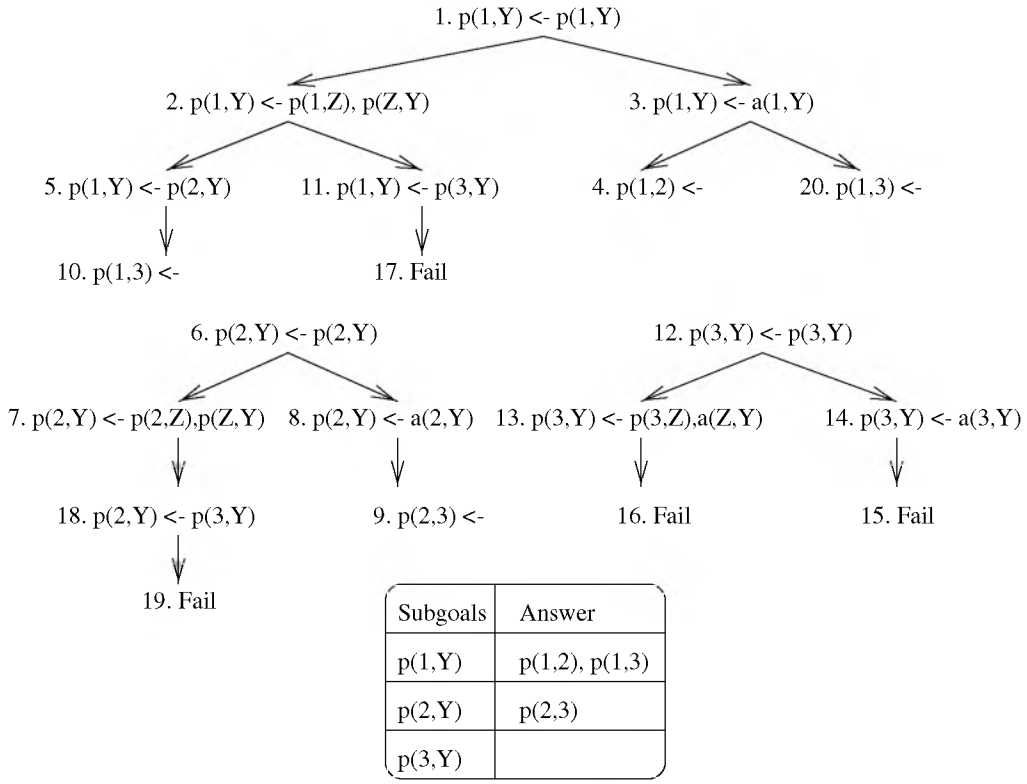


Figure 1: An SLG system

Correctness of SLG was shown in [CW96], along with the correctness of a restriction of SLG for definite programs. To restate this result, we briefly review some terminology. Let $\mathcal{F}$ be a system for an SLG evaluation of a program P and query Q. The partial interpretation of $\mathcal{F}$, $I(\mathcal{F})$, is a set of ground atoms constructed as follows. $A \in I(\mathcal{F})$ if and only if A is a ground instance of some answer in $\mathcal{F}$; *not* $A \in I(\mathcal{F})$ if and only if A is a ground instance of some A', and the SLG tree for A' is completed in $\mathcal{F}$ but does not contain A as an instance of any answer. The following theorem states the correctness of SLG for definite programs by relating the partial interpretations of final systems to the minimal model of P restricted to the set of subgoals $\mathcal{S}$ in $\mathcal{F}$ ($\mathcal{M}_P|_{\mathcal{S}}$).

Theorem 1 ([CW96]) *Let Q be a query to a definite program P. Then an SLG evaluation consisting of the operations* Subgoal Call, Program

CLAUSE RESOLUTION, ANSWER CLAUSE RESOLUTION, *and* COMPLETION *will reach a final system $\mathcal{F}_\alpha$ in which a ground atom A is in $I(\mathcal{F}_\alpha)$ if and only if it is in $\mathcal{M}_P|_\mathcal{S}$, the minimal model of P restricted to the set of subgoals in $\mathcal{F}$.*

2.2 Adding Operational Features to SLG

To make SLG more suitable to implementation, we add two features to the formalism: an explicit table and a mechanism for *incremental completion*. To model the use of an explicit table, we add the NEW ANSWER operation to those of Definition 3. This operation adds an answer of a subgoal to a global table when the answer is new to an evaluation. To understand incremental completion, consider that in Example 1 the set {p(3,Y)} is completely evaluated at any time after nodes 13 and 14 have been created. Given an explicit table, storage for the tree for {p(3,Y)} may be reclaimed, and this ability to reclaim resources used in part of a computation is termed incremental completion. To perform incremental completion, it is necessary to maintain or to approximate a *subgoal dependency graph*.

Definition 5 (Subgoal Dependency Graph) *Let $\mathcal{F}$ be a system. We say that a tabled subgoal S_1 directly depends on a tabled subgoal S_2 in $\mathcal{F}$ if and only if S_1 and S_2 are noncompleted, and S_2 is the selected literal of some node in the tree for S_1. The subgoal dependency graph of $\mathcal{F}$, SDG($\mathcal{F}$), is a directed graph $\{V, E\}$ in which V is the set of root goals for noncompleted trees in $\mathcal{F}$, and $(S_i, S_j) \in E$ if and only if S_i directly depends on S_j.*

We use the relation *depends on* to denote the transitive closure of the relation *directly depends on*. Since *SDG($\mathcal{F}$)* is a directed graph, it can be partitioned into strongly connected components, or SCCs. This leads to an operational definition of sufficient conditions for complete evaluation that will form the basis of algorithms presented in subsequent sections.

Definition 6 (Completely Evaluated: Operational Formulation) *Given an SLG system and a set $\mathcal{S}$ of subgoals, $\mathcal{S}$ is completely evaluated if at least one of the following conditions is satisfied for each subgoal Subg $\in \mathcal{S}$:*

- *Subg has an answer that is a variant of Subg; or*
- *$\mathcal{S}$ is an independent SCC, and there are no applicable* SUBGOAL CALL, PROGRAM CLAUSE RESOLUTION, *or* ANSWER CLAUSE RESOLUTION *operations for Subg.*

2.3 The SLG-WAM: A Virtual Machine for Tabling

The data structures and instruction set used by the SLG-WAM are described in [SW94a]; here we briefly summarize aspects of the SLG-WAM needed to describe scheduling strategies. As mentioned in Section 2.1, there are several types of nodes: generator, consuming, interior, and answer. Interior nodes are those whose selected literal is nontabled, and are represented in the SLG-WAM by Prolog-style (or interior) choice points. An approximation of the SDG is kept on the *completion stack* (explained below). Answer nodes are not kept on the SLG-WAM stacks; instead, they are maintained in an explicit table by the instruction NewAnswer. This instruction checks whether an answer is in the table. If so, the instruction fails; otherwise, the answer is added to the table.

The SLG-WAM represents tables using a trie-like structure [RRS+95]. Tries provide efficient checking and inserting of subgoals and answers, good indexing, and space savings. More specifically, the SLG-WAM uses a *subgoal trie* to represent subgoals present in an evaluation. A subgoal corresponds to a path from the root to the leaf of the subgoal trie; attached to each leaf of the subgoal trie is an *answer trie* containing all answers for that subgoal. To index answers efficiently, the order of leaves does not necessarily correspond to the order of their derivation in an evaluation. It is useful to represent this order so that the SLG-WAM can determine what answers have been returned to a particular consuming node. Accordingly, an *answer list* is also maintained for noncompleted subgoals.

Special choice points are used to represent generator and consuming nodes. Using these choice points, other tabling operations of Definition 3 are reflected more or less directly in SLG-WAM virtual machine instructions, including:

- TableTry (analogous to 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* and a *completion frame* for S.

- RetryConsuming and AnswerReturn (analogous to ANSWER CLAUSE RESOLUTION): RetryConsuming resolves the selected literal of a newly derived consuming node against a set of answers present in a table. AnswerReturn resolves a newly derived answer against the selected literals of a set of consuming nodes that are present in an evaluation.

Label	Instruction	Arg 1	Arg 2
L1	TableTry	2	L3
L2	TableTrust	2	L12
L3	Allocate		
L4	getpbreg	v4	
L5	getpvar	v2	r2
L6	putpvar	v3	r2
L7	Call	5	p/2
L8	putpval	v3	r1
L9	putpval	v2	r2
L10	Call	5	p/2
L11	new_answer	2	r4
L12	Allocate		
L13	getpbreg	v2	
L14	Call	3	a/2
L15	new_answer	2	r2

Table 1: SLG-WAM Code for Predicate p/2 of Example 1.

- CheckComplete (analogous to COMPLETION): Determines when a set of subgoals is completely evaluated.

Table 1 shows SLG-WAM code for the predicate p/2 of Example 1. The actions of these instructions under various scheduling strategies are explained in Section 3.

Two other changes must be made to the WAM to support these instructions. 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 where 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 [LBD+88]). 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, and thus to provide incremental completion. The SLG-WAM adds a new memory area to the WAM: the *completion stack*. The following example illustrates how the completion stack is used.

Example 2 *Consider the following program:*

```
:- table p/1, q/1, r/1.
p(X) :- q(X).
p(X) :- r(X).
r(X) :- p(X).
r(X) :- s(X).
s(1).
```

and the query :-p(X). *Evaluation starts with the initial query,* p(X). *A completion frame is created for it, and its depth-first number (DFN) is set to 1 (see Figure 2a). Next,* q(X) *and subsequently* r(X) *are called; a completion frame with a unique DFN (2 and 3, respectively) is created for each. When* r(X) *calls* p(X), *a* backward *dependency is detected, since* p(X) *is already present in the evaluation. This causes the DFN fields in the completion frames between (and including)* r(X) *and* p(X) *to be set to the DFN of* p(X), *as Figure 2b shows. A sequence of completion frames with the same DFN forms an approximate SCC (ASCC). Note in Figure 2b that* q(X) *is in the same ASCC as* r(X) *and* p(X), *even though it does not depend on either. Later, when* s(X) *is called (Figure 2c), a new frame is created and a new DFN is assigned to it.*

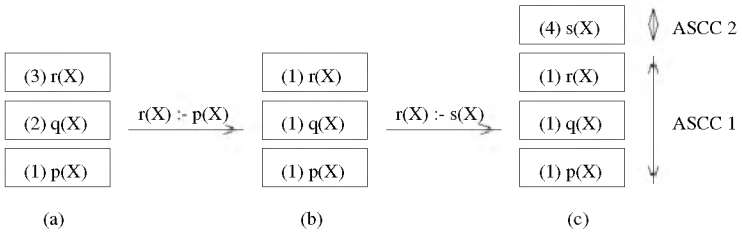


Figure 2: Completion stack sequence for program and query in Example 2

Throughout this paper we will distinguish between the SCCs of an SLG system and their (safe) approximation by the completion stack, or ASCCs.

3 Scheduling Strategies

3.1 Single-Stack Scheduling

The scheduling of program clause resolution in Prolog [AK91, War83] is conceptually simple. The engine performs forward execution for as long as it possibly can. If it cannot—because of failure of resolution, or because all solutions to the initial query are desired—it checks a scheduling stack (the choice-point stack) to determine a *failure continuation* to execute.

The SLG-WAM must also schedule resolution of answers against consuming nodes. A natural way of extending the WAM paradigm to do this is to distinguish between the acts of returning old answers to newly created consuming nodes and returning newly derived answers to old consuming nodes. The first case is simple: when a new consuming node is created, a choice point is set up to backtrack through answers in the table much as if they were unit clauses. To handle the second case, whenever a new answer is derived for which there are existing consuming nodes, an answer-return choice point is placed on the choice-point stack. This choice-point will manage the resolution of the new answer with the appropriate consuming nodes. Forward execution is then continued until failure, at which time the top of the choice-point stack is then used for scheduling. The choice-point stack thus serves as a scheduling stack for both returning answers and resolving program clauses. Accordingly, we call this scheduling strategy *single-stack scheduling*. The operational semantics of this scheduling strategy was described in detail in [Swi94], and forms the basis of the SLG-WAM, as described in [SW94a]. The following example demonstrates how this strategy works.

Example 3 *The node numbers in Figure 1 represent the order in which the actions of single-stack scheduling on the program of Example 1 take place, while Figure 3 represents the choice-point stack at various phases of the program's evaluation. When* p(1,Y) *is called, it is inserted into the table and a* generator choice point *is created (Figure 3a), which corresponds to node 1 in Figure 1. Program-clause resolution is then used to create node 2. Since the selected literal in node 2 is a variant of a tabled subgoal, a* consuming choice-point *frame (corresponding to the consuming node 2) is laid down to serve as*

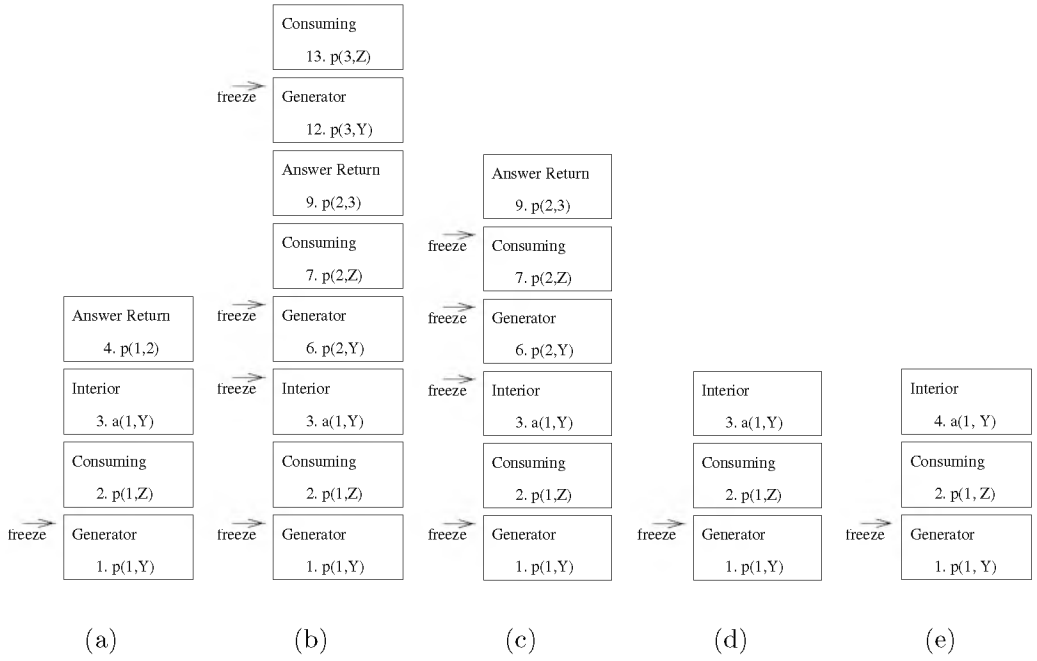


Figure 3: Snapshots of the choice-point stack during the evaluation of the program in Example 3 under single-stack scheduling

an environment through which to return answers, and the stacks are frozen (see Figure 3b), so that backtracking will not overwrite any frames below that point. If there were any answers in the table, the RetryConsuming *instruction would backtrack through them and return each answer to the consuming node. Since there are no answers in the table for* p(1,Y)*, the second clause for* p/2 *is tried. As the selected literal is not a tabled predicate, program-clause resolution is applied, and an* interior choice point *is laid down for* a/2*. The evaluation then gives rise to an answer,* p(1,2) *(in node 4). Since there are no variants of* p(1,2) *associated with* p(1,Y)*, the answer is inserted into the table, and an* answer-return choice point *is laid down. At this stage, the choice-point stack is represented in Figure 3b.*

When the engine backtracks into the answer-return choice point *for* p(1,2) *(see Figure 3c), the* AnswerReturn *instruction freezes the stacks and proceeds to return the answer to consuming nodes. After an* answer-return choice point *returns the answer to the last consuming node, it removes itself from the*

backtracking chain. However, owing to freezing, it may not always be possible to deallocate space for choice points—a situation that would arise if a second consuming node existed for p(1,Y). *In the example,* p(1,2) *will be returned to node 2, which in turn will trigger a call to* p(2,Y). *The evaluation of* p(2,Y) *(see node 6 in Figure 1) is similar to that of node 1: it is inserted into the table and a* generator choice point *is created. The tree for* p(2,Y) *will eventually generate an answer (*p(2,3) *in node 9), which is inserted into the table for* p(2,Y). *In addition, an* answer-return choice point *is laid down, and the bindings for the answer are propagated to node 5, which will then derive the answer* p(1,3). *When this latter answer is returned to node 2, it prompts a call to* p(3,Y) *in node 11.*

As with the other two trees, a generator *and eventually a* consuming choicepoint *are created for* p(3,Y). *After the creation of node 14, the subgoal is completely evaluated and can be completed. Upon completion, the choice points for* p(3,Y) *can be reclaimed—as Figure 3d shows. At this point, the engine backtracks into the* answer-return choice point p(2,3)*, and this answer is returned to the consuming node 7. Node 7 then calls* p(3,Y)*, which is completed and has no answers. The subgoal* p(2,Y) *is now completely evaluated, and space for it can be reclaimed in the stacks (see Figure 3e).*

The evaluation then returns to the choice point for a/2 *(node 3), and the next clause is tried. The answer* p(1,3) *is generated (node 20), but since a variant of this answer is already in the table (from node 10), this answer is disregarded, and the computation path fails. Finally, when the engine backtracks to the generator node for* p(1,Y) *(node 1) and there are no other choices to be tried, the last subgoal in the system can be completed.*

While single-stack scheduling is simple to conceptualize, it contains several drawbacks. The most severe problem is memory usage: to perform answer-clause resolution at different points in the SLG forest, the stacks have to be frozen so that the environment can be correctly reconstructed to later continue resolution at these points. This need to freeze stacks may lead to inefficient space usage by the SLG-WAM, as some frames might get trapped (e.g., the interior choice point in Figure 3c). Also, the addition of new choice points and the need to move around in the SLG forest to return answers means that trailed variables must be continually set and reset to switch binding environments, causing further inefficiencies. Finally, the integration of the action of returning answers into the mechanism of the choice-point stack makes single-stack scheduling not easily adaptable to a

parallel engine [FHSW95].

3.2 Batched Scheduling

Batched scheduling can be seen as an attempt to address the problems with single-stack scheduling mentioned above. Indeed, versions 1.5 and higher of XSB use this new strategy as a default. Batched scheduling reduces the need to freeze and move around the search tree by *batching* the return of answers. That is, if the engine generates answers while evaluating a particular subgoal, the answers are added to the table and the subgoal continues its normal evaluation until it resolves all available program clauses. Only then will it return the answers it generated during the evaluation to consuming nodes. As will be demonstrated in Section 5, this new strategy makes better use of space: by reducing the need to freeze branches, it reduces the number of trapped nodes in the search tree. Along with reducing space, batched scheduling shows significantly better execution times. The following example illustrates some of the differences between single-stack scheduling and batched scheduling.

Example 4 *The execution of the program and query from Example 1 under batched scheduling is depicted through the SLG forest in Figure 4 and the sequence of choice-point stacks in Figure 5. As can be seen from comparing the forests in Figure 1 and Figure 4, the procedures are identical through the first four resolution steps, but differ in the fifth step. Here, batched scheduling resolves a program clause against node 3, while single-stack scheduling returns the answer* p(1,2) *to node 2. This difference reflects the freezing and movement problems mentioned above. First, single-stack scheduling requires an environment switch from node 3 to node 2 to return the answer, and will later require a switch* back *to node 3 to finish program-clause expansion for that node. Furthermore, the unexpanded program clause for node 3 is stored in the engine as a choice point. This choice point not only takes up space itself, but the need to later switch back to it requires the* AnswerReturn *instruction to freeze the stack at that choice point (see Figure 3b). This frozen space cannot be reclaimed until completion of the ASCC in which it lies. Similarly, when the answer* p(2,3) *is returned to the consuming node 6, the return of this answer requires the placement of an explicit choice point and a freeze (c.f., the choice-point stack of Figure 3c). In Figure 5, both of these overheads are avoided.*

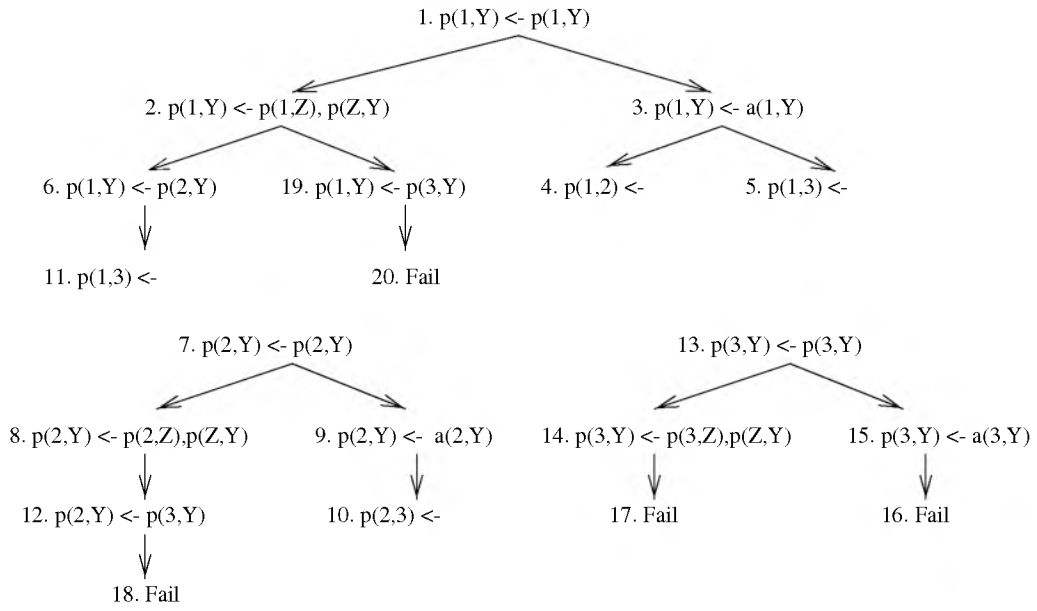


Figure 4: SLG evaluation under batched scheduling

When the generator choice point of a subgoal S has exhausted all program-clause resolution, the SLG-WAM sets the failure continuation for the choice point to a CheckComplete instruction, rather than disposing of the choice point as in the WAM. For batched scheduling, the first action of the CheckComplete instruction is to schedule any unresolved answers to the consuming nodes of S so that each consuming node will backtrack through the unresolved answers (as in the RetryConsuming instruction of single-stack scheduling). Furthermore, the engine resolves answers against each consuming node as long as there are any answers to resolve, and may resolve answers in the same iteration in which they are added. This latter step gives good performance for in-memory queries, but makes the batched scheduling algorithm differ from traditional deductive-database-style evaluations such as the semi-naive evaluation of a Magic-transformed program [BR91]. After all answers are returned, the engine backtracks to the generator choice point of S. If there are no unresolved answers for the consuming nodes of S, the actions of CheckComplete differ, depending on whether the corresponding subgoal is designated as a *leader* of its ASCC or not. (In practice, the oldest subgoal in an ASCC is chosen as leader). If the generator choice point corresponds

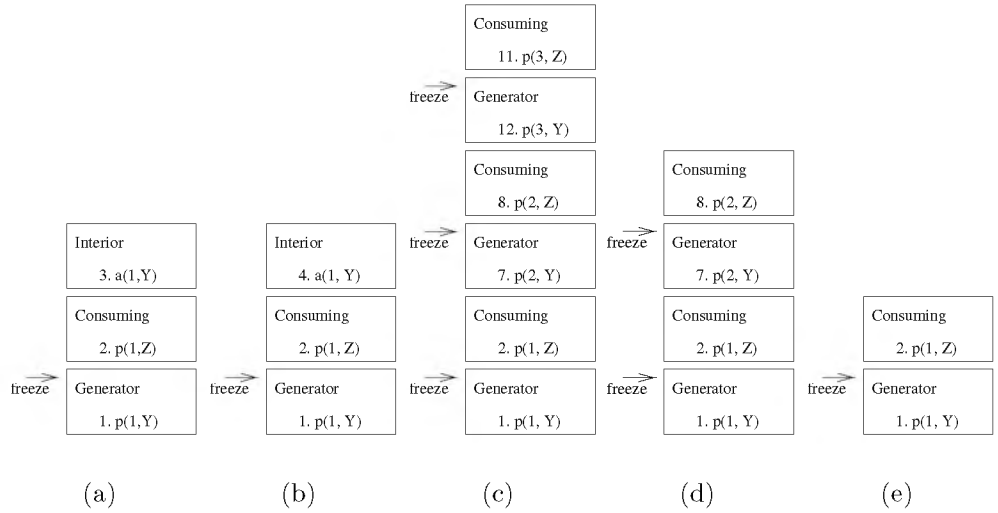


Figure 5: Snapshots of the choice point stack during the evaluation of the program in Example 4 under Batched Scheduling

to a subgoal S that is a not a leader, CheckComplete will simply fail and execute the failure continuation of the generator choice point. Otherwise, if S is a leader, the engine cycles through the subgoals in the ASCC to return unresolved answers to every consuming node whose selected literal is in the ASCC. This process repeats until a fixpoint is reached, and the ASCC can be completed.

3.3 Local Scheduling

For a number of problems, it may be preferable to evaluate a single *exact* SCC at a time, preserving the (dynamic) SCC ordering during the evaluation. We call such an evaluation a *local evaluation*, and define it as follows.

Definition 7 (Locality Property) *Let $\mathcal{F}$ be an SLG system. Resolution of an answer A against a consuming node N occurs in an independent SCC of $\mathcal{F}$ if the root subgoal for N is in an independent SCC in $SDG(\mathcal{F})$. An SLG evaluation has the locality property if any* Answer Resolution *operation applied to a state $\mathcal{F}_n$ occurs in an independent SCC of $\mathcal{F}_n$.*

In other words, in a local evaluation, answers are returned to consuming nodes outside of an SCC only after that SCC is completely evaluated. In

the previous section, we argued that batched scheduling can be more time and memory efficient than single-stack scheduling, so that it is worthwhile to investigate how batched scheduling can be modified to support the locality property, a strategy we call *local scheduling*. Figure 6 illustrates the actions of a local evaluation of the program and query of Example 1. Note that the answer generated for subgoal p(2,Y) in node 10 is only returned to its calling environment (in node 5) after the tree for p(2,Y) is completely evaluated.

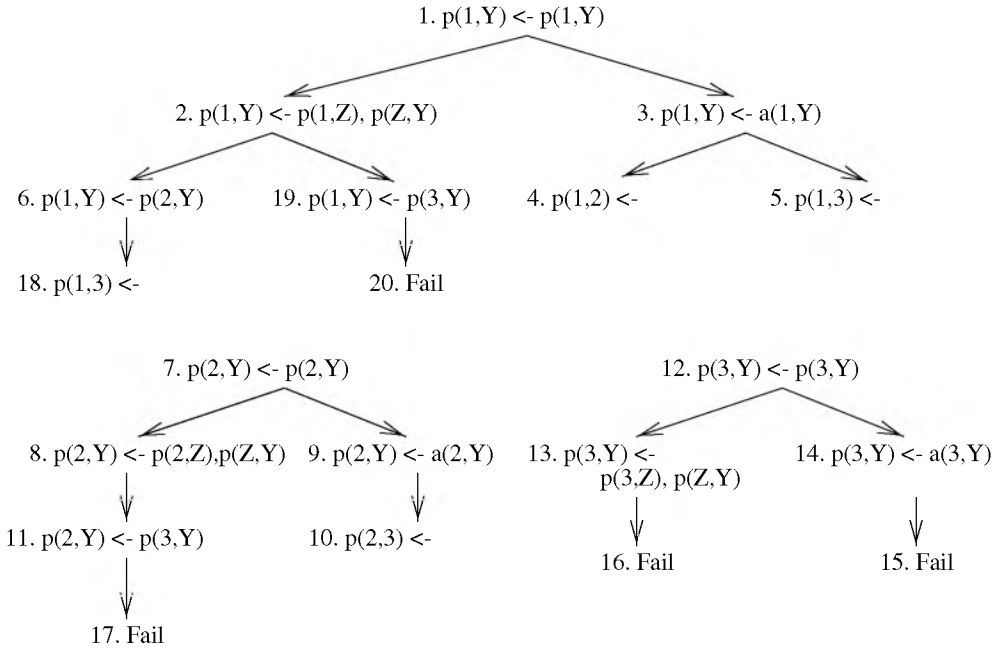


Figure 6: SLG evaluation under local scheduling

It was shown in [Swi94] that the SLG-WAM's completion stack maintains exact dependencies for local evaluations. Maintaining exact dependencies allows the engine to verify whether loops through negation exist, and to delay literals (to break these loops [CW96]) only when it is necessary. Even though negation handling and scheduling strategies are orthogonal issues, some strategies may be more efficient for evaluating normal programs. The following example shows how a local evaluation can benefit the evaluation of programs with negation.

Example 5 *Let P be the stratified program:*

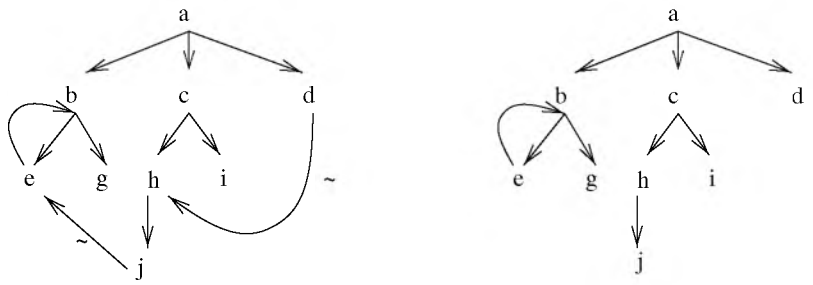
```
:- table a/0,b/0,c/0,d/0,e/0,g/0,h/0,i/0,j/0.

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

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

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

for which the query ?- a *is to be evaluated. If evaluated under either single-stack scheduling or batched scheduling, an SDG will be produced with cascading negative dependencies, as shown in Figure 7a.*



(a) negative dependencies (b) no negative dependencies

Figure 7: Subgoal dependency graphs for the program of Example 5 under different search strategies

Even though there is no cycle through negation, detecting the exact SDG can complicate the evaluation of stratified programs. However if a local evaluation is used, a simpler *SDG is created (as depicted in in Figure 7b). To obtain this latter SDG, each independent SCC has to be completely evaluated before returning any answers to subgoals outside the SCC—making the search depth-first with respect to SCCs. In a local evaluation, 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.*

Local evaluation can also improve the performance of programs that benefit from answer subsumption. Answer subsumption can be performed as a variation of the NEW ANSWER operation. While adding an answer, the engine may check whether that answer is more general than those currently in the table. If it is more general, this new answer is added, and the subsumed answers are removed. Otherwise, the computation path fails. Given that a local evaluation evaluates each SCC completely before returning any answers out of it, we are guaranteed that only the most general answers will be returned out of that SCC. This process is presented in detail in Example 6.

Example 6 *Consider the following HiLog [CKW93] variation of the same-generation program, which finds the smallest distance between two people in the same generation:*

```
sgi(X,Y)(l) :-
        ancestor(X,Z),
        subsumes(min)(sgi(Z,Z1),l1),
        ancestor(Y,Z1),
        l is l1+1.
sgi(X,X)(0).
:- subsumes(min)(sgi(joan,carl),l).
```

where subsumes(min)/2 *is a HiLog tabled predicate that performs answer subsumption by deleting all nonminimal answers every time it adds an answer to the table. Given the facts in Figure 8, there are a number of ways this query can be evaluated. It is well known that for shortest-path-like problems, a breadth-first search can behave exponentially better than a depth-first search; nevertheless, in this example, a breadth-first search is still not* optimal.

The above query seeks to determine how close joan *and* carl *are. Note that they have three common ancestors (*louis, mary*, and* bob*), and thus they are cousins of first, second, and third degree. If evaluated under a breadth-first strategy (the behavior of batched scheduling for this example), all possible subpaths between* joan *and* carl *are considered, and if at some point during the evaluation a subpath is found whose length is less than those so far derived, it is immediately propagated, even though it may not be a minimal subpath. For instance, batched scheduling first finds the distance between two immediate ancestors of* joan *and* carl *to be 2, and concludes the distance between* joan *and* carl *is 3. Then evaluation continues, a new distance between the immediate*

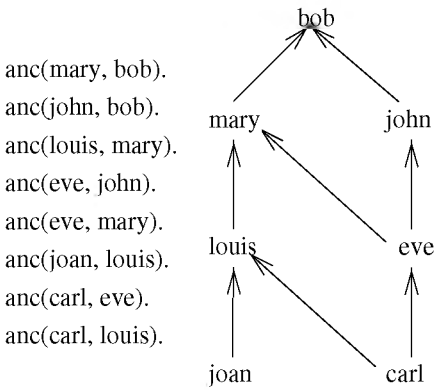


Figure 8: Ancestor relation for Example 6

*ancestors is found to be 1, and a new answer (*l=2*) is generated for the top-level query. Finally, the minimal distance between* joan *and* carl *is found to be 1, and the* correct *answer is returned.*

If a local evaluation is used on the above example, only minimal subpaths are propagated, and the engine is able to prune a number of superfluous choices. This behavior might significantly improve the performance of program analyzers such as those based on Bruynooghe's abstract interpretation framework [Bru91]. In this framework, after all the clauses for a predicate have been analyzed, the abstract substitution for the predicate is computed by taking the most general substitution among the clauses.

Local evaluation can also lead to a reduction in memory usage for some programs, as the following example illustrates.

Example 7 *Given a program of the form:*

```
:- table p/1, q/1, r/1.
p(X,Y)     :- q(X,Z),r(Z,Y).
q(X,Y)     :- q1(X,Y)          r(X,Y) :- r1(X,Y)
q1(X,Y)    :- q2(X,Y)          r1(X,Y) :- r2(X,Y)
  ⋮                               ⋮
qn-1(X,Y) :- qn(X,Y)           rn-1(X,Y) :- rn(X,Y)
qn(1,2).   qn(2,3).            rn(2,4). rn(2,5).
```

and the query p(X,Y). *Under batched scheduling, completion frames are created for each call to* q_i *and* r_i. *The completion stack at the point when* r_m(X,Y)

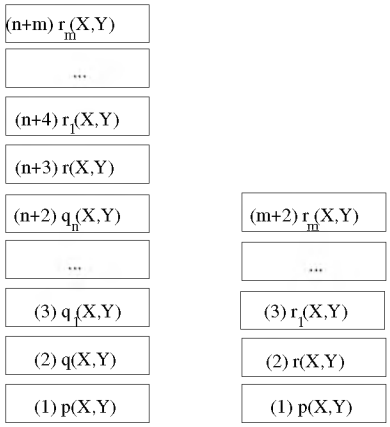


Figure 9: Completion stacks for Example 7 under batched scheduling (a) and local scheduling (b)

is called is depicted in Figure 9a. Under local scheduling, since the SCCs of q(X,Z) *and its descendents are completely evaluated before* r(Z,Y) *is called, all the space allocated for the SCCs can be reclaimed before the call to* r(Z,Y)*, as Figure 9b shows.*

4 Algorithms and WAM-Level Implementation

4.1 Implementation of Batched Scheduling

As mentioned in Section 2, a batched-scheduling evaluation can be thought of as a series of iterations where answers are returned to each of a set of consuming nodes until a fixpoint is reached for that set. We explain the implementation of batched scheduling by contrasting it to single-stack scheduling. When a new consuming node is created under single-stack scheduling, the node backtracks through answers that are already in the table. To do this, a consuming choice point is placed on the choice-point stack. Any new answers are later returned through answer-return choice points, as depicted in

Figure 3. Batched scheduling makes heavy use of consuming choice points, so that it is worthwhile to consider in detail these choice points and the actions they support. In addition to the environment information kept in regular WAM choice points, a consuming choice point keeps a pointer to the previous consuming node for the same subgoal. These pointers maintain a list of all consuming nodes for a subgoal. The consuming choice point also keeps a pointer to the list of answers in the table associated with its subgoal. When the engine backtracks into a consuming choice point, the RetryConsuming instruction is executed. This instruction backtracks through each answer in the answer list, and returns it to the corresponding consuming node. When there are no more answers left, RetryConsuming executes the failure continuation for the consuming choice point.

Batched scheduling does not differ from single-stack scheduling in regard to actions taken for answers derived *before* a given consuming node was created. However, when an answer is derived *after* a given consuming node, the answer will be returned through a later invocation of the consuming choice point, rather than creating an answer-return choice point as in single-stack scheduling. In addition, the consuming choice point is modified to continue backtracking through answers for a given consuming node as long as any answers exist.

The changes in the SLG-WAM to implement batched scheduling are the following:

- The NewAnswer instruction no longer needs to lay down an answer-return choice point when a new answer is derived.
- The consuming choice point for each consuming node will have to keep track of which answers it has consumed, by means of a pointer into the answer list to the last answer it resolved. In addition, the table data structure requires a new pointer to the end of the answer list, since new answers are now inserted at the end of this list.
- The RetryConsuming instruction (Algorithm 1) must fail when there are no more answers to resolve, so that either the action for the previous consuming choice point (RetryConsuming) or for the generator choice point (TableRetry, TableTrust, or CheckComplete) is reinvoked.
- In the CheckComplete instruction (Algorithm 2):

- An extra step is added to the CheckComplete of a subgoal S to handle the return of batched answers. Note that the subgoal S has to schedule any unresolved answers to the consuming nodes of S before the actual completion check for S is issued (see Schedule Answers, Algorithm 3).
- If S is the leader of its ASCC, the engine has to perform a fixpoint check. The instruction scans through all the subgoals in the ASCC for S, and checks whether the consuming nodes for these subgoals still have answers to resolve. If so, the instruction schedules the subgoals on the choice-point stack and arranges the stack so that when their evaluation is finished, another completion check is issued for the leader S. Otherwise, if there is nothing to schedule, a fixpoint has been reached, and the ASCC can be safely completed.

Algorithm 1 (RetryConsuming(Consuming Node *Cons*))

```
Switch environments to Cons;
If Cons has new answers to resolve
    Mark current answer as used;
    Return answer to Cons;
    Set the forward continuation;
Else
    Set the failure continuation;
```

Some general advantages of batched scheduling over single-stack scheduling are claimed in Section 2. We also note in passing that the RetryConsuming instruction requires fewer machine instructions than the AnswerReturn instruction, so that the trade-off of substituting RetryConsumings for AnswerReturns is likely to be beneficial at this level as well. However, batched scheduling also imposes some overheads: the scheduling of answers before the completion check for each subgoal; and the need to perform a fixpoint check for each leader. Section 5 will demonstrate that the advantages of batched scheduling outweigh its overheads.

Algorithm 2 (CheckComplete(Subgoal *S*))

```
If ScheduleAnswers(S)
    Backtrack through each node in the schedule_chain
    to return any unresolved answers;
If S is the leader of its ASCC
```

```
    Fixpoint = true;
    For each subgoal S' in the ASCC of S
      If S' is not completed
        If ScheduleAnswers(S')
          Fixpoint = false;
    If Fixpoint == false backtrack to return answers;
    Mark every subgoal in the ASCC as completed, and
    deallocate stack space
  Else fail;
```

Algorithm 3 (Schedule Answers(Subgoal *S*))

```
  Sched = false;
  While there exists an consuming node Cons whose
  selected literal is S
    If Cons has unresolved answers
      Add Cons to the schedule_chain;
      Sched = true;
  Return sched;
```

4.2 Implementation of Local Scheduling

In Section 3.3 we introduced local scheduling, a strategy that completes subgoals as early as possible by evaluating one SCC completely before returning any answers out of it. We shall now explain how local scheduling is implemented on top of batched scheduling. First, we note that SLG-WAM implementations of nonlocal strategies implement what may be termed *first-call optimization*, which allows a root subgoal S_{root} to share its bindings with the first consuming node $S_{calling}$ that initially called it in the evaluation, and eliminates the need for a consuming choice-point frame for $S_{calling}$. For instance, in Figure 4 (Section 2), nodes 6 and 7 of Figure 4 share the variable Y under first-call optimization. In local scheduling, the SLG-WAM is still able to coalesce the choice point for $S_{calling}$ and S_{root}, but the generator choice point for S_{root} must be able to delay returning answers until it is found not to be the leader of its SCC, or until the completion of the SCC of S_{root}. Accordingly, it is referred to as a generator-consuming choice point in local scheduling. The NewAnswer instruction also requires a modification. In batched scheduling, whenever a new answer is generated, the engine goes

on to execute the continuation for the corresponding tabled subgoal to effectively return the answer in the calling environment. For local scheduling, however, when an answer is generated, it is simply inserted into the table, and the engine then backtracks (to avoid returning the answer). Also, recall from Algorithm 2 that during the batched scheduling completion check, the engine has to: (1) schedule answers, (2) check for fixpoint, and (3) reclaim space for completely evaluated subgoals. Each of these steps is performed in local scheduling, but in slightly different ways to accommodate the extra batching of answers among different SCCs.

The original Schedule Answers routine described in Algorithm 3 simply scheduled any available answer to all applicable consuming nodes of a subgoal. In Algorithm 4, Local Schedule Answers needs to distinguish between consuming and generator-consuming nodes. Local Schedule Answers can schedule unresolved answers except in the following case. If the subgoal being considered is the leader of its SCC, the engine cannot schedule the resolution of its answers through the generator-consuming choice point, since this would return answers out of an SCC before its completion. In Local Find Fixpoint (Algorithm 5), the engine simply scans through the topmost SCC (using the completion stack), checks whether any subgoal still has consuming nodes with unresolved answers, and schedules any such subgoals. However, the engine need not schedule a subgoal if only its generator-consuming node has unresolved answers. If Local Find Fixpoint succeeds, the engine is guaranteed that the topmost SCC is completed. At this point, all subgoals in the SCC can be marked as completed, and space can be reclaimed. Finally, if the leader of the SCC has answers in the table, these should be returned. The completion check for local scheduling is described in Algorithm 6.

Algorithm 4 (Local Schedule Answers(Subgoal *S*))

```
Sched = false;
If there are consuming nodes and answers in the
table for S
   While there exists a consuming node Cons whose selected
   literal is S
      If Cons has unresolved answers
         Add Cons to the schedule_chain;
         Sched = true;
   If S is not the leader of its SCC
      Add the generator-consuming choice point of S to the schedule_chain;
```

```
        Sched = true;
    Return sched;
```

Algorithm 5 (Local Find Fixpoint(Subgoal S))

```
    Fixpoint = true;
    For each subgoal S' in the SCC of S
      If Local Schedule Answers(Subgoal S')
        Fixpoint = false;
    Return fixpoint;
```

Algorithm 6 (Local CheckComplete(Subgoal S))

```
    Local Schedule Answers(S);
    If S is the leader of its SCC
      If Local Find Fixpoint(S)
        Mark all subgoals in the SCC as completed, and
        deallocate stack space;
        If there are answers for S
          Switch environments to the generator-consuming
          choice point for S;
          Backtrack through the completed table to return answers
          to the generator-consuming choice point for S;
      Else fail and backtrack through the schedule_chain to return
      unresolved answers;
    Else fail;
```

5 Experimental Results

In this section we compare both execution time and memory usage of SLG-WAM engines, based on the different scheduling strategies described in the previous sections—XSB version 1.4 uses single-stack scheduling, XSB version 1.5 uses batched scheduling, and Local uses local scheduling—these engines differ only in the scheduling strategy used. For execution time, we considered not only the running time, but also the dynamic count of SLG-WAM instructions and operations. Benches were run on a SPARC2 with 64 MB RAM under SUNOS.

Transitive Closure	reach(X,Y) :- arc(X, Y) reach(X, Y) :- reach(X, Z), arc(Z, Y)
Shortest Path	sp(X, Y)(D) :- arc(X, Y, D) sp(X, Y)(D) :- subsumes(min)(sp(X, Z), D1), arc(Z, Y, D2), D is D1 + D2
Same Generation	sgi(X, Y)(D) :- arc(X, Y) sgi(X, Y)(D) :- arc(X, Z), subsumes(min)(sgi(Z, Z1), D1) arc(Y, Z1), D is D1 + 1

Table 2: Bench programs

The bench programs consisted of variations of transitive closure, same generation, and shortest path on various graphs (the programs are given in Table 2). We experimented with graphs that have well-defined structures, such as linear chains and complete binary trees, as well as less-regular graphs (e.g., variations of Knuth's Words[2]).

5.1 Performance of Batched Scheduling

Let us first examine the differences between single-stack scheduling and batched scheduling for left-recursive transitive closure on a linear chain containing 1,024 nodes, with the query reach(1,X). Under single-stack scheduling, first all facts are used (by backtracking through the facts for arc/2 in the first clause), and when the consuming node is laid down for the subgoal in the second clause, each answer in the table is consumed. Each time a new answer is derived in this process, computation is suspended and the new answer is immediately returned, by freezing the stacks and pushing an answer-return choice point onto the choice-point stack. Under batched scheduling strategy, all answers in the table are returned before any newly derived answer is considered.

Table 3 shows a profile[3] of SLG-WAM execution for the different engines.

[2]The nodes of these graphs are a subset of the 5,757 more common five-letter English words; there is an arc between two words if they differ in a single character [Knu93].

[3]Note that for both Table 3 and 4, we considered only the trailings and untrailings that result from environment switches.

Instructions/Operations	XSB Version 1.4	XSB Version 1.5	Local
RetryConsuming	0	1,023	1,023
RetryGenConsuming	n/a	n/a	1,023
AnswerReturn	1,022	n/a	n/a
CheckComplete	1	1	1
SwitchEnvironments	1,026	1,027	2,050
Freeze	1,022	0	0
Trail	0	0	0
Untrail	2,047	2,047	3,070
Schedule Answers	n/a	1	1
Fixpoint	n/a	1	1
Subgoals	1	1	1
Consuming Nodes	1	1	1

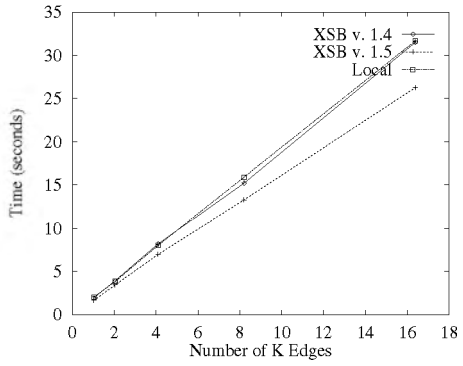
Table 3: SLG-WAM Execution Profile for Left-Recursive Transitive Closure on a Linear Chain with 1,024 Nodes

Instructions/Operations	XSB Version 1.4	XSB Version 1.5	Local
RetryConsuming	1	2,046	2,046
RetryGenConsuming	n/a	n/a	2,046
AnswerReturn	2,044	n/a	n/a
CheckComplete	1	1	1
SwitchEnvironments	3,072	3,073	5,119
Freeze	2,044	0	0
Trail	1,022	0	0
Untrail	5,115	4,093	6,139
Schedule Answers	n/a	1	1
Fixpoint	n/a	1	1
Subgoals	1	1	1
Consuming Nodes	1	1	1

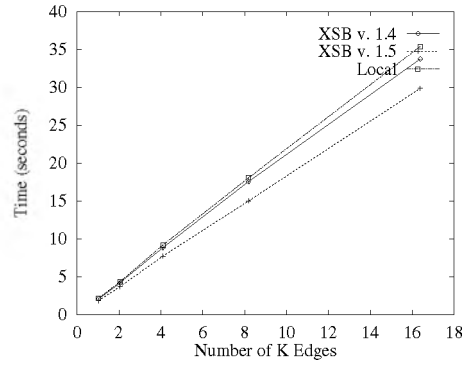
Table 4: SLG-WAM Execution Profile for Left-Recursive Transitive Closure on a Complete Binary Tree of Height 9

Notice that the main difference between XSB versions 1.4 and 1.5 for this example lies in the fact that AnswerReturns are replaced by RetryConsumings. Since RetryConsuming requires about 30% fewer machine instructions than AnswerReturn (Section 4.1), the trade-off is beneficial. The times for the different engines to compute the transitive closure on trees and chains are given in Figure 10. The speedup of XSB version 1.5 over XSB version 1.4 for these examples varies between 11% and 16%.

More significantly, because batched scheduling does not require any stack freezing, it utilizes memory better. Figure 11a gives the total stack space usage (local, global, choice point, trail, and completion stack) for the strategies under consideration (for left-recursive transitive closure on chains of varying lengths). Note that whereas memory consumption grows linearly with the number of facts for XSB version 1.4, the space remains constant for XSB version 1.5 at 2.7 KB (as it does for Local, which is built on XSB version 1.5).



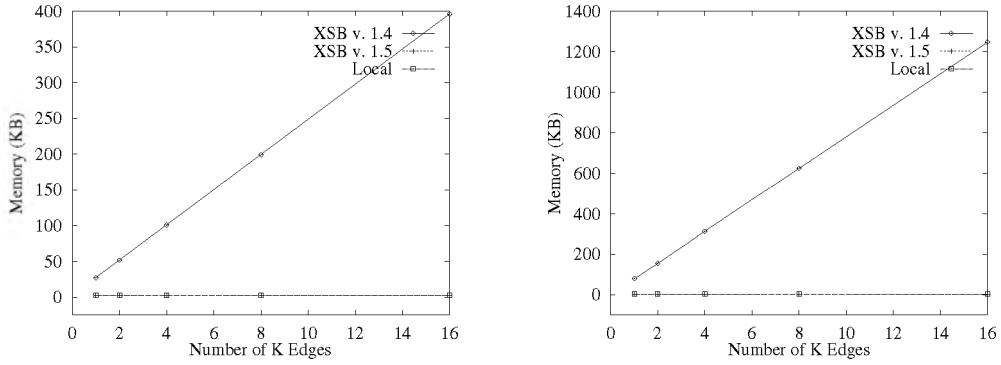
(a) linear chains of varying length



(b) complete binary trees of varying heights

Figure 10: Times for left-recursive transitive closure

Table 4 shows the SLG-WAM instruction/operation count for left-recursive transitive closure on complete binary trees of varying height. The numbers in this table are similar to those of Table 3, but the batching of answer resolution reduces the need for the engine to move around in the SLG forest; thus batched scheduling also reduces the setting/resetting of trailed variables. Memory savings are even bigger than for chains, as Figure 11b



(a) Linear chains of varying length

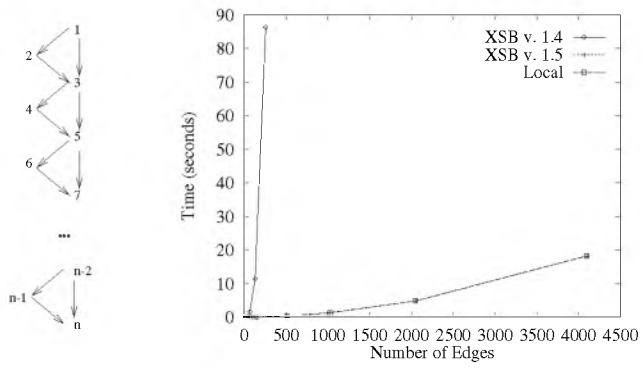
(b) Complete binary trees of varying height

Figure 11: Total memory usage for left-recursive transitive closure

shows (note that for the two new strategies, the space remains constant at 2.88KB).

Single-stack scheduling, as its name implies, uses a stack-based scheduling for answers; when executing transitive closure over, say, a binary tree, it traverses the tree in a depth-first manner. Because batched scheduling effectively uses a queue for returning answers, when executing transitive closure it will traverse the same tree in a breadth-first manner. Accordingly, optimization problems such as shortest path that can (1) be formulated through left-recursive transitive closure, and (2) benefit from a breadth-first search, can be run more efficiently under batched scheduling.[4] To demonstrate this, we first consider the artificial graph described in Figure 12a. If a depth-first search is used to compute the shortest path between nodes 1 and n in this graph, it will run in exponential time. However, the shortest path can be computed in polynomial time if the graph is searched in a breadth-first manner. Figure 12b shows the times XSB version 1.4, XSB version 1.5, and Local take to compute sp(1,n)(Dist) for different values of n. In addition to running slower, XSB version 1.4 ran out of memory on graphs with more than 512

[4]It is worth pointing out that only the underlying data structures are searched in a breadth-first manner. The predominantly depth-first nature of program-clause resolution in the WAM is maintained through all strategies discussed in this paper. However, [FSW97] discusses the SLG-WAM implementation of a general breadth-first search that is also suitable for queries to disk-resident data.

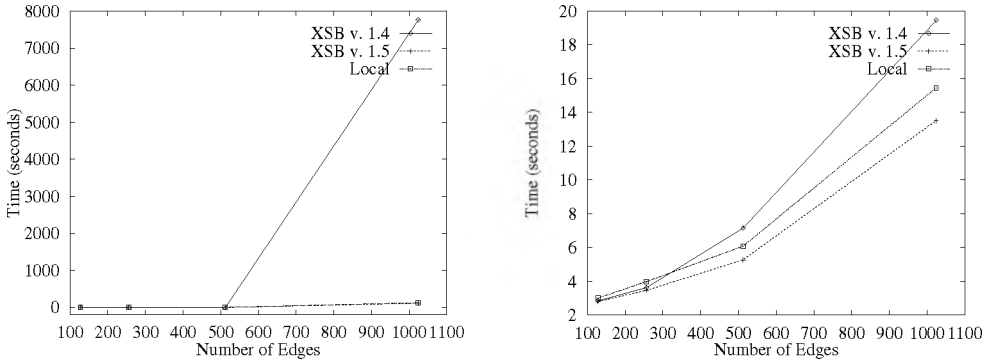


(a) (b)

Figure 12: The time is shown for XSB version 1.4 and XSB version 1.5 to find the shortest path between the two endpoints (1 and n) of a graph of the form depicted at left

nodes. XSB version 1.5 is also consistently faster than XSB version 1.4 for graphs that are less regular, such as variations of Words. Figure 13 shows the times for the engines to compute two shortest-path queries.

In Section 4.1, we mentioned that batched scheduling adds some overhead when compared to single-stack scheduling. The CheckComplete instruction becomes more expensive. At CheckComplete, not only does batched scheduling need to schedule all unresolved answers, but for each leader subgoal it has to check whether all subgoals in the leader's SCC are completely evaluated. Clearly, for programs that do not benefit from batching answers, these extra tasks might result in loss of efficiency. One example where XSB version 1.5 performs worse than XSB version 1.4 is right recursive-transitive closure on a linear chain. In this example, there are as many tabled subgoals as there are nodes in the graph, and each subgoal is the leader of its own SCC. Note that since there are no consuming nodes, answers are not batched. However, at CheckComplete for each subgoal, XSB version 1.5 still has to check whether there are unresolved answers to be scheduled and whether fixpoint has been reached—steps which, in this case, are superfluous.



(a) sp(words,X)(Dist)—500 iterations

(b) sp(there,white)(Dist)—1000 iterations

Figure 13: Timings for shortest-path on Words

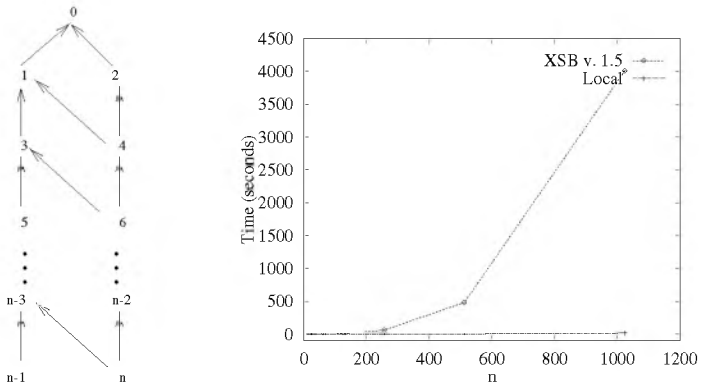
5.2 Performance of Local Scheduling

Rather than sharing the bindings as in batched scheduling and single-stack scheduling, the implementation of local scheduling incurs the cost of explicitly returning an answer of a generator node to its calling environment. This overhead is reflected in Tables 3 and 4—aside from the 1,023 RetryConsumings that are done for XSB version 1.5, Local requires another 1,023 RetryGenConsuming to explicitly return answers to the calling environment. Note that this also incurs a higher number of environment switches: one for each answer returned outside of an SCC (e.g., in Table 3, notice that the number of SwitchEnvironments for Local is 2,050, which is the sum of the number of SwitchEnvironments for XSB version 1.5 and the number of answers). These extra operations add between 10% and 20%, as is evidenced in Figure 10.

As for memory consumption, local scheduling has the same constant behavior as batched scheduling for transitive closure on trees and chains, as evidenced in Figure 11 (notice that the lines for Local and XSB version 1.5 overlap). In Figure 10, we can see that Local adds a roughly constant 15% overhead to XSB version 1.5, and the execution times for the Local engine are comparable to XSB version 1.4. Local also has approximately the same performance as XSB version 1.5 for shortest path (see Figures 12b and 13).

We have stated in Section 4.2 that for programs that use answer subsumption, local scheduling can perform arbitrarily better than batched scheduling.

The graph in Figure 14b substantiates this statement. This experiment measured the times to find the shortest distance between the two *deepest* nodes (n-1 and n) on graphs of the form depicted in Figure 14a, for varying n, using the same-generation program of Example 6. Not only is local scheduling considerably faster, but its execution times increase linearly with the size of the data, whereas for batched scheduling, this growth is exponential.[5]



(a) (b)

Figure 14: The execution time for the query subsumes(min)(sgi(n-1,n),l) is shown on graphs of the form depicted at left for varying n

Local scheduling is an efficient way to find all answers for a given query; however, it is not useful for so-called existential queries—queries that involve an existential variable, or certain forms of ground negation. In these cases, a single answer is required, and local scheduling is inefficient since it fully evaluates SCCs before passing answers back up to consuming nodes. For practical query evaluation, then, one would expect that a mixture of strategies may become useful: batched scheduling for existential queries, and local scheduling for queries that involve answer subsumption, while either may evaluate the remaining class of queries.

[5]In Figure 14b, the times for the Local engine vary from 0.06–15.7, whereas for XSB version 1.5, they range between 0.09–4007.8.

6 Discussion and Future Directions

Batched scheduling bears some resemblance to two independently developed approaches: the ET* algorithm from [FD92] and the AMAI from [JBD95]. However, in [FD92], Fan and Dietrich do not consider strongly connected components in the fixpoint check, and their strategy is fair for answers.[6] The extra check for fairness may result in inefficiencies for in-memory queries such as transitive closure over a chain. In [JBD95], Janssens, Bruynoohge, and Dumortier describe an abstract machine specialized for abstract interpretation, and use a similar scheduling strategy for their fixpoint iterations. Even though they take SCCs into account, these are detected statically. In the SLG-WAM, dynamic detection of SCCs has been proven useful in the evaluation of logic programs with negation (stratified or not). Local scheduling resembles the strategy proposed by Zukowski and Freitag in [ZF96], where program fragments are evaluated by different fixpoints.

The scheduling strategies proposed in this paper can improve the performance—memory usage and execution time—of tabled evaluations. Owing to its performance, batched scheduling is now the default scheduling strategy for XSB. The gains from this strategy are twofold: by eliminating the answer-return choice point and the freezing of stacks done at Answer-Return, memory usage is greatly reduced; and because of the reduction of trailings/untrailings, the execution time decreases.

Local scheduling can perform asymptotically better than batched scheduling when combined with answer subsumption. This can be of use in many different areas, such as aggregate selection and program analysis. In addition, local scheduling may have an important role to play in evaluating programs under the well-founded semantics [vGRS91]. Currently, in the default-scheduling strategy of XSB, the engine may have to construct part of the SDG to check for loops through negation. Since local scheduling maintains exact SCCs, it does not require this step, as was demonstrated by Example 5. Furthermore, when negative literals actually are involved in a loop through negation, SLG uses a DELAY operation to attempt to break the loop. This use of DELAY may create an answer A that is *conditional* on the truth of some unevaluated literal. However, other derivation paths may create an *unconditional* answer for A (for example, all answers considered in this paper are unconditional). Clearly, conditional answers are not needed

[6] That is, answers are not returned in the same fixpoint iteration they are created.

for A if there is a corresponding unconditional answer, and the use of DELAY gives rise to a form of answer subsumption, leading to another advantage of locality. As the well-founded semantics becomes used by practical programs, the advantages of local scheduling may become increasingly necessary for their efficient evaluation.[7]

We have shown that even though local scheduling can achieve near-optimal performance for some applications, for others it may add overheads and even lead to unacceptable inefficiency. This is also true of other scheduling strategies devised for SLG. The ideal would be to use a strategy or set of strategies that results in the best performance for a desired application. Further research is needed to assess the feasibility of combining different scheduling strategies in the same evaluation.

Acknowledgment of support: This work was supported in part by CAPES-Brazil, and NSF grants CDA-9303181 and CCR-9404921.

References

[AK91] H. Aït-Kaci. *WAM: A Tutorial Reconstruction.* Cambridge, MA, 1991, MIT Press.

[BR91] C. Beeri and R. Ramakrishnan. On the power of Magic. *Journal of Logic Programming*, 10(3):255–299, 1991.

[Bru91] M. Bruynooghe. A practical framework for the abstract interpretation of logic programs. *Journal of Logic Programming*, 10(1/2/3&4):91–124, 1991.

[CDS97] M. Codish, B. Demoen, and K. Sagonas. XSB as the natural habitat for general purpose program analysis. In *Proceedings of the International Conference on Logic Programming (ICLP)*, page 416, Cambridge, MA, 1997. MIT Press.

[CKW93] W. Chen, M. Kifer, and D. S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[7] Versions 1.7 and higher of XSB can evaluate the well-founded semantics using either batched scheduling or local scheduling.

[CW96] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[DRSS96] S. Dawson, C. R. Ramakrishnan, S. Skiena, and T. Swift. Principles and practice of unification factoring. *ACM Transactions on Programming Languages and Systems*, 18(5):528–563, September 1996.

[DRW96] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems—a case study. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 117–125, New York, 1996. ACM.

[FD92] C. Fan and S. Dietrich. Extension table built-ins for Prolog. *Software—Practice and Experience*, 22(7):573–597, July 1992.

[FHSW95] J. Freire, R. Hu, T. Swift, and D. S. Warren. Exploiting parallelism in tabled evaluations. In *Proceedings of the 7th International Symposium, PLILP 95*, volume 982 of *Lecture Notes in Computer Science*, pages 115–132, Berlin, 1995. Springer-Verlag.

[FSW96] J. Freire, T. Swift, and D. S. Warren. Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. In *Proceedings of the Eighth International Symposium of Programming Languages, Implementations, Logics and Programs*, pages 243–258, Berlin, March 1996. Springer-Verlag.

[FSW97] 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, Cambridge, MA, 1997. MIT Press.

[JBD95] G. Janssens, M. Bruynooghe, and V. Dumortier. A blueprint for an abstract machine for abstract interpretation of (constraint) logic programs. In *Proceedings of the International Symposium on Logic Programming*, pages 336–350, Cambridge, MA, 1995. MIT Press.

[KKTG95] G. Köstler, W. Kiessling, H. Thöne, and U. Güntzer. Fixpoint iteration with subsumption in deductive databases. *Journal of Intelligent Information Systems*, 4(2):123–148, March 1995.

[Knu93] D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. Reading, MA, 1993. Addison-Wesley.

[LBD+88] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, D. H. D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The Aurora or-parallel Prolog system. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 819–830, 1988. ICOT (Institute for New Generation Computer Technology), Ed. OHMSHA Ltd. Tokyo and Springer-Verlag.

[RRR+97] Y. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. Smolka, T. Swift, and D. Warren. Efficient model checking using tabled resolution. In *Proceedings of Computer Aided Verification*, pages 143–154, Berlin, 1997. Springer-Verlag.

[RRS+95] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient table access mechanisms for logic programs. In *Proceedings of the International Conference on Logic Programming*, pages 697–711, Cambridge, MA, 1995. MIT Press.

[SW94a] T. Swift and D. S. Warren. An abstract machine for SLG resolution: Definite programs. In *Proceedings of the International Symposium on Logic Programming*, pages 633–654, Cambridge, MA, 1994. MIT Press.

[SW94b] T. Swift and D. S. Warren. Analysis of sequential SLG evaluation. In *Proceedings of the International Symposium on Logic Programming*, pages 219–238, Cambridge, MA, 1994. MIT Press.

[Swi94] T. Swift. Efficient evaluation of normal logic programs. PhD Thesis, Department of Computer Science, State University of New York at Stony Brook, 1994.

[Tom95] D. Toman. Top-down beats bottom-up for constraint extensions of datalog. In *Proceedings of the International Logic Program-*

ming Symposium, pages 98–115, Cambridge, MA, 1995. MIT Press.

[vG93] A. van Gelder. Foundations of aggregation in deductive databases. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, pages 13–34, Berlin, 1993. Springer-Verlag.

[vGRS91] A. van Gelder, K. A. Ross, and J. S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[War83] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, Stanford Research Institute, 1983.

[ZF96] U. Zukowski and B. Freitag. Adding flexibility to query evaluation for modularly stratified databases. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 304–318, Cambridge, MA, 1996. MIT Press.