An Abstract Machine for Fixed-Order Dynamically Stratified Programs

Konstantinos Sagonas, Terrance Swift, David S. Warren

Department of Computer Science
University at Stony Brook
{kostis,tswift,warren}@cs.sunysb.edu

Abstract. It is known that a fixed computation rule, such as used in most logic programming systems, does not suffice for normal logic programs. For instance, SLS resolution, which evaluates programs according to the well-founded semantics, makes use of an oracle to determine the next literal to select in [5], and of ideal parallelism in [8]. Given these limits, it is natural to define a subclass of normal programs for which a fixed computation rule is adequate. This class, *left-to-right dynamically stratified* programs, properly includes other stratification classes.
Left-to-right dynamically stratified programs are of interest because they are posited to be amenable to efficient implementation. We demonstrate that this is in fact the case with an implementation based on the SLG-WAM of XSB [10]. The SLG-WAM is an extension to the underlying implementation model of Prolog, the WAM, and has been shown to be extremely efficient for definite programs [13]. We show that extending the engine for left-to-right dynamically stratified programs does not slow down execution for definite programs or for Prolog-style SLD evaluation. Indeed, implementation of stratification leads to the technique of *early completion* which can also benefit definite programs.

1 Introduction

SLDNF is the most common resolution method for logic programs with negation. While SLDNF is often a useful programming construct, its semantic limitations are severe. For instance, Datalog programs with negation do not necessarily terminate using SLDNF and their semantics may not be satisfactory. Bottom-up evalutions can address these limitations, and a variety of extensions of magic sets and tabling methods have been devised to evaluate various classes of stratified programs. In this paper we study the efficient implementation of one method, SLG_{strat} [11]. SLG_{strat} is a variant of SLG and inherits many of its properties. In particular it is complete, with polynomial data complexity, for a powerful class of stratified programs. This class, *left-to-right dynamically stratified* (*LRD-stratified*) *programs* characterizes those normal programs which can be evaluated using a fixed computation rule. The class properly includes *left-to-right weakly stratified programs* [4], effectively including all *modularly stratified* [9] programs as well. On LRD-stratified programs, SLG_{strat} restricts the class of possible SLG evaluations to those that use a deterministic, fixed computation rule.

Our implementation is based on the SLG-WAM for definite programs, described in [13]. This engine forms the basis of the freely available XSB system which has been used to implement several large commercial and educational systems, as well as being widely used in research. By basing our implementation of stratified negation on the WAM, we are able to combine the flexibility and programming environment of standard Prolog with the declarative advantages of our bottom-up method. The result is the ability of the programmer (or optimizer) to choose a mixture of SLDNF and SLG_{strat} execution for predicates, depending on which is most appropriate. Our contributions are as follows:

1. We describe fully the design of the SLG-WAM for LRD-stratified programs.
2. We present performance results for this WAM-based engine indicating that support for stratified negation has negligible overhead for SLD or definite SLG evaluation. In fact, implementation of LRD-stratification leads to implementation of a mechanism called *early completion* which can *reduce* time and space needed for definite programs.
3. We compare this overhead to that of Ordered Search [7] and analyze the differences between tabled and magic-style implementations of negation.

After presenting the definitions of LRD-stratification (which was recently defined in [11]), in Section 3 we provide an overview of the operations of SLG_{strat}. In Section 4 the SLG-WAM for LRD-stratified programs is described in detail, and in Section 5 its performance is measured and analyzed.

2 Left-to-Right Dynamic Stratification

We first adapt Przymusinski's definition of the iterated fixed point for the well-founded semantics from [5] to a fixed left-to-right selection strategy. Our single modification is the introduction of the *failing prefix* constraint in Definition 1. Intuitively, the failing prefix restricts false facts from being included in $\mathcal{F}_I(F)$, unless their falsity can be established by a left-to-right examination of literals.

Definition 1. For sets T and F of ground atoms we define

$\mathcal{T}_I(T) = \{A$: there is a clause $B \leftarrow L_1, ..., L_n$ in P and a ground substitution θ such that $A = B\theta$ and for every $1 \leq i \leq n$ either $L_i\theta$ is true in I, or $L_i\theta \in T\}$;

$\mathcal{F}_I(F) = \{A$: for every clause $B \leftarrow L_1, ..., L_n$ in P and ground substitution θ such that $A = B\theta$ (1) there is some i $(1 \leq i \leq n)$, such that $L_i\theta$ is false in I or $L_i\theta \in F$, and (2) there exists a *failing prefix*: for all j $(1 \leq j \leq i-1)$, $L_j\theta$ is true in $I\}$. □

In $\mathcal{T}_I$ and $\mathcal{F}_I$, I represents facts shown to be true or false in a previous fixpoint derivation. These operators serve as primitives upon which inner fixpoint operators T_I and F_I can be built.

Definition 2. Let $I = \langle T; F\rangle$ be an interpretation, we define

$$T_I^{\uparrow 0} = \emptyset \text{ and } F_I^{\downarrow 0} = H_P$$
$$T_I^{\uparrow n+1} = \mathcal{T}_I(T_I^{\uparrow n}) \text{ and } F_I^{\downarrow n+1} = \mathcal{F}_I(F_I^{\downarrow n})$$
$$T_I = \bigcup_{n<\omega} T_I^{\uparrow n} \text{ and } F_I = \bigcap_{n<\omega} F_I^{\downarrow n}.$$

Further, we define $\mathcal{I}(I)$ as: $\mathcal{I}(I) = I \cup \langle T_I; F_I\rangle$. □

The outer (transfinite) fixpoint is based, according to the usual definitions, on the operator $\mathcal{I}$ which extends the interpretation I to $\mathcal{I}(I)$ by adding to I (1) new atomic facts T_I which can be derived from P knowing I, along with (2) negations of atoms in unfounded sets based on the interpretation I.

A LRD-stratified program is defined as one in which the iterated fixpoint using the operator $\mathcal{I}$ produces a two-valued model. The following theorem summarizes results about dynamically stratified programs, and provides substantiation for our claim that LRD-stratified programs characterize those for which a fixed order of evaluation is adequate. As an additional point, we note that since any modularly stratified program can be statically reordered into a left-to-right modularly stratified program [9], LRD-stratified programs effectively include all modularly stratified programs.

Theorem 3. [5, 11]

- *A program has a two-valued well-founded model iff it is dynamically stratified.*
- *The class of LRD-stratified programs properly contains the class of left-to-right weakly stratified programs, which properly contains the class of left-to-right modularly stratified programs.*

A more detailed discussion of LRD stratification and its relationships with other stratification classes can be found in [11] and in the full version of the paper.

3 Tabled evaluation of LRD-stratified programs

A Brief Overview of Tabling Like other tabling methods, SLG [3] evaluates programs by keeping a table of subgoals and their answers, and by resolving repeated subgoals against *answer clauses* from the table rather than against program clauses. By using answer resolution in this manner, rather than repeatedly using program clause resolution as in SLD, SLG avoids looping and terminates for all programs with the bounded term-depth property. In practice, not all predicates need to be tabled for termination or efficiency: predicates may either be declared *tabled* in which case they are evaluated using SLG, or they may be *non-tabled* in which case SLD evaluation is used.

The first time a subgoal S of a tabled predicate is encountered during an evaluation, S is registered in the table, and a new SLG tree is created with root (or *generator*) node S. Program clause resolution is then used to obtain the children for S. Otherwise, if S is not new to the evaluation (S is contained in the table), no new tree is created for S. In either case, the *active* node whose selected literal was S will use answer clause resolution to obtain its children (i.e. answers from the table are resolved against the goal). 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 instances of the subgoal; any subsequent derivations of the same answer are failed. In this manner redundant subcomputations, including loops, are prevented by tabling. Stated informally, the SLG trees and the resulting tables form an *SLG system. SLG systems* can be seen as capturing the *states* of an *SLG evaluation* of a query against a program. In summary, tabled SLG evaluation of definite programs uses the following operations:

New Subgoal creates an active node for a tabled subgoal S and if S is not present in the evaluation, creates a new tree with root S and children created by program clause resolution against S.

New Answer adds an answer to the table if it is not present.

Answer Return returns an answer of a subgoal S to active nodes of S.

Completion marks as *completed* sets of subgoals when it is determined that all possible answers have been derived for them.

In addition, if non-tabled predicates are allowed, a further operation, New Interior, is required to execute non-tabled predicates using program clause resolution (see [13] for more details).

3.1 SLG_{strat}: Tabled evaluation of LRD-stratified programs

Efficiently determining when all possible answers have been derived for a set of subgoals — that is when the subgoals have been *completely evaluated* — is a central issue in designing a tabling engine. To correctly determine this, and hence to implement the completion operation when negation may be present in programs requires exact information about subgoal dependencies. To represent dependencies, we use the notion of a *Subgoal Dependency Graph* (*SDG*) of an SLG system. The *SDG* is a directed graph that characterizes the run-time dependencies between non-completed subgoals in the system.

Definition 4 Subgoal Dependency Graph. Let S be an SLG system. We say that a tabled subgoal $p(\overline{t})$ *directly depends on* a tabled subgoal $q_i(\overline{b_i})$ iff there is a clause

$$p(\overline{a}) :- l_1(\overline{b_1}), \ldots, l_i(\overline{b_i}), \ldots, l_n(\overline{b_n}).$$

Let $l_i = (\neg)t_i(\overline{b_i})$. If $l_i(\overline{b_i})$ is a positive (negative) literal, then we say that subgoal $p(\overline{t})$ *directly depends positively* (*negatively*) *on* subgoal $t_i(\overline{b_i})$. The *Subgoal Dependency Graph* of S, $SDG(S) = (V, E)$, is a directed graph in which V is the set of non-completed subgoals in S and $(s_i, s_j) \in E$ iff subgoal s_i directly depends on subgoal s_j. Edge (s_i, s_j) is labeled *positive* if s_i directly depends positively on s_j, otherwise it is labeled *negative*. □

The recursive components that form the basis of stratification theories [1] can be characterized as *Strongly Connected Components* or *(SCCs)* of the *SDG*.

Three operations for handling clauses whose selected literal is negative are floundering, negation failure, and negation success. In the following let C be the clause containing the selected literal L $(= \neg B)$.

floundering If L is a non-ground negative literal, the evaluation is floundered.

negation failure If L is a ground negative literal $\neg B$, and B currently has an answer, then remove the clause C from the system.

negation success If L is a ground negative literal $\neg B$, where B has no answers and is marked as completely evaluated, then remove this literal from C.

Efficient evaluation of LRD-stratified programs requires an important point of departure from the evaluation technique previously described in [2]. This technique breaks the fixed order of computation for C (by applying the SLG delaying transformation) if L is a ground negative literal whose subgoal (i) has no answers, and (ii) is not completely evaluated. After delaying L, the evaluation may resolve further literals to the right of L potentially expanding the search space of the program. An alternative approach for LRD-stratified programs is to *suspend* the selected literal L, and, as a result, the clause C in which this literal appears. In such a case, we say that C is *suspended on* (the completion of) B, and that subgoal B has *negation suspensions* on it. The abstract operation that performs this suspension is called NEGATION SUSPENSION:

NEGATION SUSPENSION If L is a ground negative literal $\neg B$, and B is not marked as completed, then *suspend* the clause C.

Note that the situation in which the subgoal B is not in the system vacuously satisfies the conditions for the suspension of literal $\neg B$. In such a case, a *generator* node is created for the subgoal, and program clause resolution is used to derive answers for it. The clause C containing $\neg B$ remains *suspended* until one of the conditions that caused its suspension ceases to hold. That is, C remains suspended until an answer is derived for B (at which time C will be *disposed*), or until B is *completed* by invoking a SLG COMPLETION operation (in which case the clause will again become *active*).

So far, no mention has been made of how to determine when a set of subgoals has been completely evaluated. Example 1 motivates Definition 5 below.

Example 1. Let P be the program in Fig. 1 for which the query `?- a` is to be

```
a :- b, ¬ c.

b :- a.          c :- ¬ d.
b :- d.
b.               d :- b, e.
```

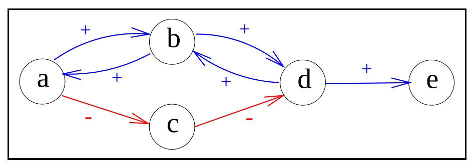

Fig. 1. A LRD-stratified Program and the SDG for the query `?- a`.

evaluated. The execution of this query against P causes cascading suspensions, of `a` on `c` and `c` on `d`, as seen from the SDG. Note that without disregarding the dependencies of subgoal `b` (i.e. removing subgoal `b` from the SDG), the program appears to contain a loop through negation (subgoals `a`, `b`, `c`, and `d` form a strongly connected component). Also notice that simply abolishing negation suspensions of subgoals that have answers does not break the negative loop (all dependencies to and from `b` are positive). ■

Definition 5 Completely Evaluated. Given a state $\mathcal{S}$ of an SLG evaluation, a set Σ of subgoals is *completely evaluated* if either of the following is satisfied:

1. Σ is an *SCC* of *SDG*($\mathcal{S}$) that depends on no other *SCC* (termed an *independent SCC*), no clause of subgoals in Σ is *suspended*, and all applicable SLG operations have been performed for each clause in Σ; or
2. Σ consists of a single subgoal A having an answer that is a variant of A. □

The second condition, which is not a part of previous definitions of SLG, will sometimes be referred to as *early completion*. Note that early completion is necessary to evaluate the program in example 1 using a fixed computation rule. However, as will be demonstrated in Section 5, early completion can be of benefit even when programs do not contain negation. The following theorem, restated from [11], indicates the validity of the approach outlined.

Theorem 6. *The operations* NEW SUBGOAL*,* NEW ANSWER*,* ANSWER RETURN*,* COMPLETION*,* NEGATION SUSPENSION*,* FLOUNDERING*,* NEGATION FAILURE*, and* NEGATION SUCCESS *suffice for the evaluation of ground LRD-stratified programs.*

4 An Engine for LRD-Stratified Programs

Several changes to the WAM are needed to support tabling. We briefly describe extensions to the WAM to provide these functionalities, focusing on features needed for stratified programs. Details of the SLG-WAM's execution for definite programs can be found in [13], and details of table access mechanisms in [6].

4.1 Overview of the SLG-WAM for Definite Programs

Most noticeably, the SLG-WAM adds two new memory areas to the WAM: a *Table Space* where subgoals and their answers are stored, and a *Completion Stack* which is used to detect completely evaluated sets of subgoals (see Section 4.6).

Tabling presents an asynchrony between the generation of answers and their consumption by active subgoals. This asynchrony necessitates deep changes to the WAM in order to retain the execution environment of active subgoals until they are completely evaluated. First, the SLG-WAM must maintain information about a forest of trees rather than a single tree. The SLG-WAM represents a forest of trees as a SLG search tree in which the first active node for a subgoal is directly replaced by a generator node. To ensure that information in the search tree is not lost, WAM stacks are *frozen* and allocation must occur above the *freeze registers* for those stacks. These frozen segments in the stacks can be deallocated only upon completion of the subgoals associated with them. Consider the effect of freezing segments of the choice point stack: choice points in the same branch of computation may not be contiguous. A given choice point may have been allocated above the freeze register so its *parent choice point* may lie on a branch of the search tree arbitrarily deeper in the stack.

Besides extending the WAM data structures, the SLG-WAM also adds a set of new instructions; we present them grouped by the operation they perform. $\boxed{\text{NEW SUBGOAL}}$ This operation occurs when a tabled subgoal is called. In analogy to the `try`, `retry`, and `trust` WAM instructions, the NEW SUBGOAL operation is compiled using `tabletry`, `tableretry`, and `tabletrust` SLG-WAM instructions. In addition to setting up a tabled choice point, the `tabletry` instruction checks whether the subgoal is new and if so, creates a table entry (referred to as a *subgoal frame*), sets up a *generator choice point frame*, and uses program clause resolution to obtain answers for the subgoal. If the subgoal is not new, it sets up an *active subgoal frame* initiating the process of *answer resolution* for this instance of the subgoal. In either case, the `tabletry` instruction must fully traverse the subgoal to check if it is in the table or insert it if not. As it does so, the instruction factors out dereferenced variable occurrences from the subgoal and places them in the generator choice point. Later bindings to these variables will constitute *answer substitutions* which, if new, will be copied into the answer table.

The tabletrust instruction, differs from its WAM counterpart in that it places a completion instruction in the *Next_Clause* cell of the (generator) choice point. Recall that in the WAM the *Next_Clause* cell contains the instruction to be executed upon failure of the current clause; thus, the completion instruction will be executed after all program clause resolution has been performed for the subtree stemming from this generator choice point.

$\boxed{\text{NEW ANSWER}}$ Answers are added to the table via the new_answer instruction. new_answer is compiled as the last instruction of each clause of tabled predicates. Its operation is as follows: when an answer is derived for a particular subgoal, new_answer performs a check to determine whether the answer substitution in the generator choice point has already been entered into the *answer table* associated with the subgoal. If it has, the derivation path fails, a vital step for ensuring termination. If not, the computation continues, and the answer is scheduled for return to the applicable *active* subgoals through an answer_return instruction.

$\boxed{\text{ANSWER RETURN}}$ Returning an answer from execution environment E_1 to a subgoal in execution environment E_2 is a matter of untrailing from E_1 to a common ancestor of E_1 and E_2 and then using values in a *forward trail* to reconstitute the environment of E_2 (the SLG-WAM trail is modified from the WAM trail to perform this environment switch). Because execution may need to return answers to a tabled environment, execution of a tabled predicate requires creation of both a choice point and a local environment.

$\boxed{\text{COMPLETION}}$ In principle, SLG evaluation of a definite program could completely evaluate all subgoals and complete them all as the last step of the evaluation. However for practical evaluation a more incremental completion is vital. The SLG-WAM provides an efficient mechanism to soundly approximate SCCs, and complete them incrementally. We present this mechanism in Section 4.6.

SLG-WAM Changes Necessary for LRD-Stratified Programs In order to evaluate LRD-stratified programs, five main changes are made to the definite engine: addition of a mechanism for early completion, a mechanism for explicit maintenance of subgoal dependencies, general suspend and resume operations, implementation of a stratified negation operation, and a completion instruction with the ability to determine SCCs precisely. We discuss each of these changes in turn.

4.2 Early Completion

In the SLG-WAM framework, early completion is implemented as part of the new_answer instruction. Recall from Section 4.1 that this instruction is laid down as the last instruction of each clause of tabled predicates and that it uses the answer substitution bindings from the generator choice point in the answer check/insert step. To perform early completion the engine must check if the answer is a variant of the call. Note that this condition includes the case of an answer to a ground subgoal. In such a case, the subgoal frame, which is accessible through the generator choice point, is marked as completed, and a completion instruction is placed in the *Next_Clause* cell of the generator choice point, thus bypassing any remaining tableretry and tabletrust instructions for that subgoal.

4.3 Maintenance of the Subgoal Dependency Graph

Recall that in the SLG-WAM, either a generator or an active choice point frame is created for each occurrence of a tabled subgoal that is not yet completed. Thus, vertices of the SDG are present in the SLG-WAM. It turns out that dependency information corresponding to SDG edges is more easily maintainable from body to head than from head to body as specified in Section 3.1. Thus, rather than maintaining the SDG, its transpose (SDG^T) is maintained.

In the WAM, information about the state of the computation is recorded in global registers. We therefore introduce another global register, which we will call *ptcpreg* (short for *parent tabled choice point register*). All tabled or non-tabled choice point frames used for program or answer clause resolution are expanded with an extra field (called *Ptcpreg*) that records the value of *ptcpreg* at the time of choice point creation. This register is updated so that it always points to the generator choice point frame of a tabled subgoal that is the nearest ancestor in the search tree:

- First, *ptcpreg* is modified upon encountering a generator tabled subgoal. This is performed by the `tabletry` and `tabletrysingle` instructions, after the creation of the generator choice point frame, and the value of *ptcpreg* is set to the address of that choice point.
- Secondly, as is done for all other registers of the WAM, *ptcpreg* is restored to its earlier value in the event of backtracking (through the `retry`, `trust`, `tableretry`, and `tabletrust` instructions).
- Finally, *ptcpreg* should be also be restored in forward continuations of tabled predicates. To illustrate, let
$$p(\overline{a}) \;:\!-\; t_1(\overline{b_1}), \ldots, t_i(\overline{b_i}), \ldots, t_n(\overline{b_n}).$$
be the clause under execution, and assume that $t_i(\overline{b_i})$ is a tabled subgoal. Observe that the call to $t_i(\overline{b_i})$ will alter the value of *ptcpreg* if $t_i(\overline{b_i})$ is a generator node using program clause resolution. To compensate for the change, upon the success of $t_i(\overline{b_i})$, the value of *ptcpreg* is restored to the value saved in the *Ptcpreg* field of the choice point frame for $t_i(\overline{b_i})$. This restoration is unnecessary if $t_i(\overline{b_i})$ is a non-tabled subgoal. Therefore the restoration of *ptcpreg* during forward execution should only be performed by the `new_answer`, `answer_return`, and `negation_resume` (Section 4.4) instructions.

4.4 Implementation of $Negation\_Suspend$ and $Negation\_Resume$

In order to implement the NEGATION SUSPENSION and NEGATION SUCCESS operations, mechanisms must be introduced to suspend clauses on negative literals and to resume the computation of these clauses when the suspended subgoal is completed. Note that facilities in the SLG-WAM to support the return of answers to subgoals will support efficient suspend and resume.

The NEGATION SUSPENSION operation is implemented through a builtin, `negation_suspend/1`. This builtin suspends the computation by laying down a *negation suspension frame*. This frame, which resides on the choice point stack, saves the execution environment for the suspended computation in a manner similar to the active subgoal frames used by answer return. Like active subgoal frames, negation suspension frames are chained together and can be accessed through the subgoal frame.

Execution will resume to the environment saved by negation_suspend/1 when the negative subgoal is completed (Section 4.6 provides full details). When completing failed subgoals (those for which a negative call will succeed) the completion instruction creates *negation resume choice points* chained together on the choice point stack. Upon backtracking to these choice points, the engine will use the trail to switch execution environments to the suspended computation (using the negation suspension frame) and continue execution of the suspended clause. The actual implementation of negation_resume resembles that of answer_return except that no actual answer need be returned.

4.5 Implementation of a Predicate for Stratified Negation

The predicate `tnot/1` (Fig. 2) implements stratified negation in the SLG-WAM. A consequence of the SLG operations defined in Section 3 is that any negation suspensions on subgoals that obtain answers are removed by the early completion operation. Thus, upon executing the completion instruction for subgoals that still have negation suspensions on them, the selected literal of the clauses suspended on these subgoals always succeeds and can simply be removed. Therefore, the following invariant is maintained.

Invariant 1 *In the SLG-WAM for LRD-stratified programs, the completion of subgoals initiates only* NEGATION SUCCESS *operations.*

```
tnot(S) :-
    ground(S) →
        ( subgoal_not_in_system(S), call(S), fail
        ; (is_complete(S) → has_no_answers(S)
          ; negation_suspend(S), true )      /* if execution reaches here, S */
        )                                    /* is completed with no answers */
    ; error("Flounder: subgoal S is not ground").
```

Fig. 2. The Prolog implementation of tabled negation for LRD-stratified programs.

Fig. 2 sketches the implementation of `tnot/1` using low-level builtins. Note that ground subgoals with an answer are completely evaluated. `tnot/1` therefore calls negation_suspend/1 only if the subgoal is incomplete and has no answer. Later, the computation resumes (to `true`) only if the completed subgoal has no answers.

4.6 Implementation of Completion

Completion in Definite Programs As mentioned in Section 4.1, the SLG-WAM contains a *Completion Stack* to perform incremental completion efficiently. In the SLG-WAM, as in the WAM, when execution fails over a choice point, all program clause resolution has been performed for that subgoal. However, if the subgoal is tabled and belongs to a larger *SCC*, the subgoal may not have performed all applicable answer resolution. The completion stack maintains a safe over-approximation of each *SCC* (called an *Approximate SCC* or *ASCC*). This approximation allows efficient space reclamation, and as will be shown, combines well with the more precise determination needed for negation.

To complete subgoals, the SLG-WAM maintains a *Depth-First Number* (*DFN*) for each tabled subgoal. Using these DFNs, the completion stack also maintains an (over) approximation of the deepest subgoal (lowest DFN) that each tabled subgoal depends on. Using this approximation a subgoal can be completed, along with its descendants, as long as it does not depend on a subgoal deeper in the completion stack than itself. Such a subgoal is termed a *leader* of an *ASCC*.

In terms of implementation, when a new subgoal is encountered, the `tabletry` instruction pushes a *completion stack frame* for that subgoal onto the completion stack. The frame is later removed when that subgoal is determined to be completely evaluated (through execution of a `completion` instruction). Thus, the following is an invariant of the SLG-WAM for definite programs:

Invariant 2 *The completion stack contains a completion stack frame for a subgoal S only if S is in the system but is not yet marked as completed.*

For definite programs, the format of the completion stack frame can be thought of as follows:

Subgoal	*DFN*	*DirLink*	*MinLink*

In the above data structure, *DirLink* keeps track of the deepest direct subgoal dependency, while *MinLink* keeps an (over) estimate of the subgoal's indirect dependencies by propagating minimum *DirLink* values backwards as goals are checked for completion [2]. Even though this approach to dependency maintenance is highly efficient, using a stack to approximate a subgoal dependency graph can lead to over-approximation of *SCC*s (as shown in [2]).

Completion in LRD-Stratified Programs Selection of a negative literal may cause an *active* clause to *suspend* on another subgoal in the same *ASCC*, complicating the completion algorithm. If the engine is to maintain a fixed computation rule for LRD-stratified programs, it must correctly order the completion of subgoals and the resumption of the suspended computations. The following example illustrates difficulties that are encountered with negative dependencies.

Example 2. Let P be the program in Fig. 3 for which the query `?- p(a)` is to be evaluated. Note that since there is only one predicate `p/1`, P is not modularly stratified for any selection order. It is, however, LRD-stratified. The evaluation

```
:- table p/1.

p(a) :- p(b), ¬p(d).    p(d) :- p(e), p(f).
p(b) :- p(c).           p(e).
p(b).                   p(f) :- ¬p(c), p(d).
p(b) :- ¬p(a).          p(g) :- p(c).
p(c) :- p(b), p(g).     p(g) :- ¬p(e), ¬p(g).
```

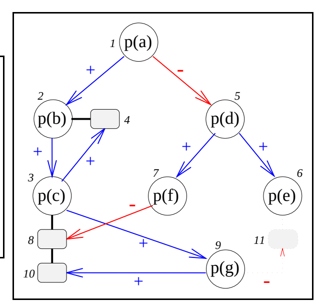

Fig. 3. A LRD-stratified Program and the SDG for the query `?- p(a)`.

of p(a) causes cascading suspensions, as seen from an elaboration of its *SDG* in which circled nodes denote *generator* nodes, while boxes denote corresponding *active* or *suspended* nodes. The *SDG* represents the state when the literal under execution is ¬p(e) (shown underlined in P) when all clauses and literals but those in boxes have been selected. The numbers next to the nodes of the *SDG* depict the order in which these subgoal instances are encountered by the SLG-WAM. The answer derived for p(b) has already been returned to both the p(a) and p(c) clauses by means of the ANSWER RETURN operation. The clauses for p(a) and p(f) are suspended on the completion of p(d) and p(c) respectively. Note that the selected literal ¬p(e), and consequently the second clause for p(g), need not be suspended on the completion of p(e) because p(e) currently has an answer and is completely evaluated due to early completion. Thus a NEGATION FAILURE operation is immediately applicable to the clause. This operation will remove the second clause for p(g) from the *SDG* along with the dotted part of Fig. 3(b). Since there are no more clauses for subgoals p(b) through p(g) that are not suspended, the engine fails back to the choice point for p(b), and schedules a completion instruction. ■

For LRD-stratified programs, each completion stack frame is augmented with two extra fields that are used for the edges of the *SDG* and SDG^T stemming from each subgoal in the stack. In actuality, the SLG-WAM constructs these edges lazily from the choice points whenever the subgoals of the current *ASCC* contain negation suspensions, thus avoiding this construction for definite programs, an important optimization step as will be seen in Section 5. Fig. 4 shows the completion stack for the program of Example 2 before and after the removal of completion stack frames of subgoals through early completion. The edges of *SDG* and SDG^T are restricted to the *ASCC* having p(b) as its leader.

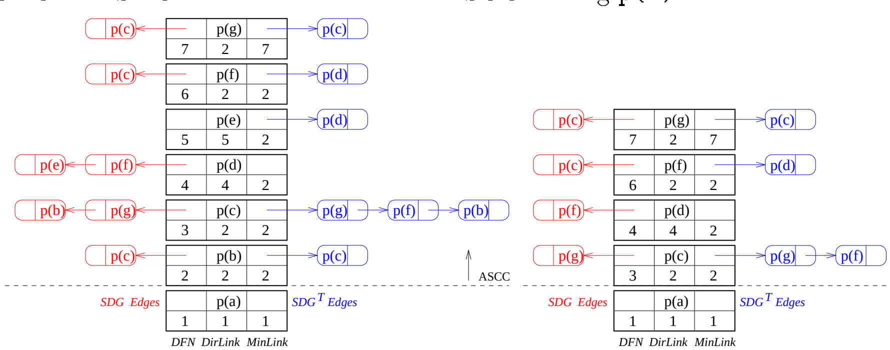

Fig. 4. Completion Stack states when evaluating the program of Example 2.

Recall that rather than eliminating the generator choice point frame, either the tabletrust or the new_answer instruction (in the case of early completion) places a completion instruction in the *Next_Clause* cell of its own choice point. Upon later backtracking to this choice point, the completion instruction will determine whether the *SCC* in which this subgoal participates is completely evaluated. As can be seen from the instruction's pseudocode in Fig. 5, its first action is to get the table entry that is associated with the current generator tabled choice point. A check is then made to determine whether this subgoal is a leader of an $ASCC$ using Definition 7 which ensures that a subgoal will not be considered a leader if it has direct or indirect dependencies lower into the stack.

Definition 7 Leader. A subgoal L is called a *leader* of an $ASCC$ iff the completion stack frame associated with L satisfies the condition:

$$DFN(B) < min(DirLink(L), MinLink(L))$$

where B is the subgoal that is immediately below L in the completion stack. □

If the subgoal is not a leader, the action of the completion instruction is to simply backtrack to the previous choice point. If, however, the subgoal is a leader of an $ASCC$ that does not contain negative links, all subgoals in the $ASCC$ can be completed and their space is reclaimed. If the $ASCC$ does contain a negative link the engine must find an independent SCC. Details of the algorithm can be found in the full version of the paper; we mention is that is done by performing a combination of a topological sort and an SCC computation of the subgraph SDG' of the SDG that has subgoal L as its leader.

```
Instruction completion
1       subgoal := tcp_subgoal(breg);
2       if (subgoal is a leader of an ASCC, A)          /* according to Definition 7 */
2.1       if (there are no negation suspensions on subgoals in A)
2.1.1       Mark as completed all subgoals in A;
2.1.2       Reclaim the stack space of subgoals in A and adjust the freeze registers;
          else
2.2         I := independent_scc(subgoal);
2.3         For each subgoal s ∈ I
2.3.1         Mark subgoal s as completed;
2.3.2         if (there are completion suspensions on s)
              Schedule negation_resume instructions for each completion suspension of s;
2.3.3           if (there exists a subgoal s' ∈ I that is suspended on s)
2.3.4             Warn that the program is not LRD-stratified;
2.4         Let E be the set of subgoals in A that were early completed and let C = E ∪ I;
2.4.1       Compact the completion stack by removing the frames of subgoals in C;
2.4.2       Reclaim the stack space of subgoals in C and adjust the freeze registers;
3       else breg := tcp_previous(breg);
4       fail;
```

Fig. 5. The SLG-WAM completion instruction

The subgoals of the independent SCC are marked as completed and negation_resume instructions are scheduled for all negative literals that were suspended on the completion of these subgoals. This scheduling is done by modifying the *breg* to point to a *negation resume choice point* (Section 4.4) which is responsible for initiating these instructions. In addition, a run-time check on whether the program is LRD-stratified can be performed by the completion instruction. The program-query pair under execution is not LRD-stratified if and only if there exists a negative dependency among the subgoals of an independent SCC. This condition is checked in step 2.3.3. Finally, the completion stack is compacted, the remaining choice points are (re)chained, and, if possible, the stack space of the subgoals that are marked as completely evaluated is reclaimed.

Example 3. Continuing Example 2, the COMPLETION operation for the $ASCC$ led by `p(b)` finds subgoals `p(c)` and `p(g)` to be an independent SCC, and marks them as completely evaluated. The completion frames of these subgoals, as well as those of subgoals `p(b)` and `p(e)` that were early completed, are removed from the completion stack. Also, their completion initiates a NEGATION SUCCESS operation that unsuspends literal ¬`p(c)` in the clause of `p(f)`. When this literal is resumed, computation continues with literal `p(d)` in the body of `p(f)` which modifies the subgoal dependency graph. Computation fails back to the choice point for `p(d)`, and a completion instruction is executed next. Using Definition 7, `p(d)` is found to be a leader of another $ASCC$. Subgoals `p(d)` and `p(f)` are now part of the same independent SCC, and are thus marked as completely evaluated. Finally, literal ¬`p(d)` in the body of `p(a)` is unsuspended and succeeds, which in turn activates the early completion of subgoal `p(a)`. ■

Even though not shown in the above example, in general the approximation of the strongly connected components kept by the completion stack may considerably change as a result of unsuspending computations, since negation_resume can initiate new, untried computation paths.

5 Performance

For definite programs, the SLG-WAM adds minimal overhead to the WAM [13], and performs significantly better than other deductive database systems for in-memory computation of a wide variety of recursive queries. In order to measure the overhead incurred for the increased functionality for LRD-stratified programs, we compared the performance of the following two different types of emulators on standard Prolog benchmarks.[1]

XSB-def Performs SLG evaluation for definite programs only. This emulator has no extra registers/words in choice points, and completes subgoals by approximate SCC detection.

XSB-lrd Performs SLG evaluation for LRD-stratified programs. It contains all additions or changes to the SLG-WAM described in this paper.

As can be seen from Table 1 the overhead of the SLG-WAM changes for dynamically stratified negation to SLD resolution is negligible (usually less than 1%). These times show the effect of adding one register to the SLG-WAM, and having an extra word in all choice points.

	deriv	nreverse	qsort	query	serialise
XSB-def	1	1	1	1	1
XSB-lrd	1.009	1.001	1.011	1.018	1.008

Table 1. Normalized CPU times for executing standard Prolog benchmarks using XSB.

[1] The benchmarking environment was a SPARC 20 with four 50 MHz SuperSPARC processors (although no test used more than one processor), no two-level cache, 128 Mbytes of memory, running Solaris 2.4. A pre-release of XSB version 1.5.0 was used to produce all emulators. The compilation was done with gcc 2.7.0 (using `-O4`).

Next, we measured the overhead of SLG_{strat} on typical benchmark programs that use definite SLG resolution. Table 2 contains normalized times of executing a transitive closure (over a chain, a cycle, and a full binary tree data structure; all of size 8k), and a same generation program (over a randomly generated 24x24x2 cylinder [10]). None of these programs contains negative literals. This set of benchmarks measures the overhead of an extra WAM register, having more words in choice points and completion stack frames, performing early completion, and for checking whether exact SCC detection should be used for the completion of subgoals (which should be avoided in these cases). In this set of benchmarks, an additional emulator was used:

XSB-mod Contains the same changes as *XSB-lrd* with the exception that early completion of subgoals is not performed. It can be proven that *XSB-mod* correctly evaluates the class of (left-to-right) modularly stratified programs.

	`TC-chain`	`TC-cycle`	`TC-tree`	`same generation`
XSB-def	1	1	1	1
XSB-mod	1.008	1.009	1.012	1.010
XSB-lrd	0.947	0.945	0.979	0.961

Table 2. Normalized times for executing standard tabling benchmarks using XSB.

The times in Table 2 indicate that early completion actually *speeds up XSB-lrd* compared to the other two emulators. Furthermore, these numbers indirectly provide a measure of the performance improvement due to early completion; between 4-6% for these benchmarks.

To put the numbers of the previous section in perspective, we also measured the overhead of other evaluation strategies for stratified negation proposed in the literature. One of the more influential of such strategies is Ordered Search [7], a strategy implemented in the CORAL system, which evaluates programs that are left-to-right modularly stratified. Table 3 compares performance of Ordered Search with CORAL's default *Supplementary Magic rewriting*. CORAL provides many annotations that affect the performance of programs; for both evaluation strategies the timings reported are the best that could be obtained by setting these options. In addition to similar benchmarks used for XSB, benchmarks `even` and `congested` are taken from the CORAL examples directory. The results in Table 3 show that Ordered Search is considerably less efficient than ordinary seminaive fixpoint evaluation (from 40% up to an order of magnitude slower).

It should also be mentioned that Ordered Search is restrictive (at least in its current implementation); for example it prohibits the use of certain magic set optimizations like *NRSU factoring* or *SCC-by-SCC* evaluation. More importantly, it is heavily affected by the number of rules in a module. In contrast, our evaluation method for stratified negation does not suffer from such symptoms.

	`TC-chain`	`TC-cycle`	`same generation`	`even`	`congested`
CORAL Semi Naive	1	1	1	1	1
CORAL Ordered Search	1.42	1.45	1.30	1.71	12.33

Table 3. Performance of Ordered Search compared to seminaive evaluation in CORAL.

6 Discussion

The performance numbers of the previous section indicate a minimal overhead for stratified negation compared to Ordered Search. We believe that these results reflect fundamental aspects of the computation strategies involved, rather than accidents of implementation. Part of the overhead of Ordered Search lies in its maintenance of a set of *context nodes* which together serve as an analogue to the SDG. While, as Section 4 indicates, it is a simple matter to use the SLG search tree to maintain SDG dependencies in the SLG-WAM, such a structure does not naturally follow from a semi-naive evaluation such as CORAL's. Rather, it must be built from scratch, leading to the observed overheads. The SLG-WAM's small overhead is especially striking since XSB has been shown to be about an order of magnitude faster than CORAL and LDL++ for definite Datalog queries.

Our engine for LRD-stratified programs allows useful high-performance execution of most deductive-database style queries. However, the engine can also be seen as an efficient component of a WAM-based engine for evaluating the well-founded semantics of all normal programs. This planned extension of the LRD engine should produce an engine capable of executing the well-founded semantics with the speed and robustness of compiled Prolog, and thus be able to serve as a basis for a new generation of systems for non-monotonic reasoning.

References

1. K. Apt, H. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In *Foundations of Deductive Databases and Logic Programming*, pages 89–148, 1988.
2. W. Chen, T. Swift, and D. S. Warren. Efficient top-down computation of queries under the well-founded semantics. *JLP*, 24(3):161–199, 1995.
3. W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *JACM*. To appear.
4. H. Przymusinska and T. C. Przymusinski. Weakly stratified logic programs. *Fundamenta Informaticae*, 13(1):51–65, 1990.
5. T. C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *Proceedings of 8th PODS*, pages 11–21, 1989.
6. I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient tabling mechanisms for logic programs. In *ICLP*, pages 687–711, 1995. MIT Press.
7. R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Controlling the search in bottom-up evaluation. In *Proceedings of JICSLP*, pages 273–287, 1992. MIT Press.
8. K. A. Ross. A procedural semantics for well-founded negation in logic programs. *JLP*, 13(1):1–22, 1992.
9. K. A. Ross. Modular Stratification and Magic Sets for Datalog programs with Negation. *JACM*, 41(6):1216–1266, 1994.
10. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proceedings of the ACM SIGMOD Conference*, pages 442–453, 1994.
11. K. Sagonas, T. Swift, and D. S. Warren. The Limits of Fixed-Order Computation. Submitted for publication, 1996.
12. P. J. Stuckey and S. Sudarshan. Well-Founded Ordered Search. In *Proceedings of 13th Conference on FST-TCS*, LNCS, pages 161–171, 1993. Springer-Verlag.
13. T. Swift and D. S. Warren. An abstract machine for SLG resolution: Definite Programs. In *Proceedings of the 1994 ILPS*, pages 633–652, 1994. MIT Press.