A Simple and Efficient Implementation of Concurrent Local Tabling

Rui Marques[1], Terrance Swift[2], and José Cunha[1]

[1] CITI, Dep. Informática – FCT, Universidade Nova de Lisboa
[2] CENTRIA — Universidade Nova de Lisboa

Abstract. Newer Prolog implementations commonly offer support for multi-threading, and have also begun to offer support for tabling. However, most implementations do not yet integrate tabling with multi-threading, and in particular do not support the sharing of a tabled computation among threads. In this paper we present algorithms to share completed tables among threads based on Concurrent Local SLG evaluation (SLG_{CL}). SLG_{CL} is based on the Local scheduling strategy, and is designed to support applications in which threads concurrently share tabled evaluations Version 3.1 of XSB implements SLG_{CL} in the SLG_{CL}WAM, which fully supports well-founded tabled negation, construction of residual programs, tabled constraints and answer subsumption. The implementation of SLG_{CL} requires additions to a single tabling operation only. As a result, SLG_{CL} should be implementable by any tabling system that uses Local evaluation, whether based on the Chat engine, linear tabling, or call continuation.

1 Introduction

A number of Prolgs have become multi-threaded, while at the same time, current or planned versions of several Prologs also support tabling, including XSB, YAP [8], B-Prolog [11], Mercury [10], ALS [4] and Ciao [5]. Although there has been work in combining tabling with parallel Prologs, most notably [8], little work has been done to extend tabling to multi-threaded engines and the types of concurrent applications they support. In this paper, we describe algorithms that allow concurrently executing threads to share tables and that are based on a popular scheduling strategy for tabling called Local evaluation [3]. The general idea behind Local evaluation is to fully evaluate each mutually dependent set of tabled subgoals before returning answers to other subgoals outside of that set. As a result, Local evaluation requires less space than other scheduling strategies for many programs. In addition, since it postpones the return of answers outside of a mutually dependent set of subgoals until that set is completely evaluated, Local evaluation can reduce the amount of delay and simplification operations required for tabled negation. For the same reason, the method is efficient for applications that benefit from *answer ordering*: in which a computation retains only the join of answers over an upper semi-lattice, or only answers that are

maximal for a partial order. Local evaluation is supported by several tabling systems including XSB, YAP, B-Prolog and Ciao.

Our approach is based on an operational semantics called Concurrent Local SLG (SLG_{CL}) [6], which allows concurrently executing threads to share tables while maintaining a Local evaluation. We describe the algorithms to needed to implement SLG_{CL} on XSB's SLG-WAM [9] creating the SLG_{CL}WAM. This engine is supported in the current version of XSB (3.1) and has been fully tested for well-founded tabled negation with residual programs, for tabled constraints, and for answer orderings. Beyond the overhead of a multi-threaded emulator, the implementation of SLG_{CL} imposes no overhead on Prolog execution or on evaluations that use thread-private tabling.

SLG_{CL} is designed primarily for concurrent applications that use shared tables to amortize queries or save space, rather than for parallelism based on tabling, a choice that is determined in part by the nature of Local evaluations. An example is a multi-threaded knowledge server, based on XSB's CDF ontology management system or on the object logic FLORA-2 (both of which make full use of tabled negation). In such a server, shared tables construct T-box or schema information, while thread-private tables support A-box or object level information. The decision to support concurrency over parallelism enables a simplicity of implementation that has helped lead to the robustness indicated above. The extensions for the SLG_{CL}WAM are nearly all made to a single instruction: tabletry (which is executed upon the call of a tabled subgoal) and can nearly all be inserted as a function call. While we describe our implementation in terms of the SLG-WAM, the algorithms are not specific to this engine. In fact, since most tabling methods execute an operation analogous to tabletry when calling a tabled subgoal, SLG_{CL} should not be hard to implement in other tabling systems that support Local evaluation. Section 2 reviews SLG_{CL} and aspects of the SLG-WAM that are most relevant to this paper. Section 3 describes algorithms for the SLG_{CL}WAM, arguing their correctness from theorems of SLG_{CL}, and Section 4 provides an indication of complexity and performance of the SLG_{CL}WAM.

2 Background

Due to space limitations, our presentation assumes a general knowledge of tabled evaluation and of the WAM. In this section, we briefly and sometimes informally review aspects of tabling that pertain to definite programs, and aspects of the SLG-WAM directly relate to the implementation of SLG_{CL}. Discussion of negation in the formalism and implementation is postponed until Section 3.1.

2.1 SLG and Local Evaluation for Definite Programs

Our presentation of SLG [2] and its extensions makes use of a forest of trees model (see e.g. [6]). In this model, an SLG evaluation is a sequence of forests of SLG trees. Each SLG tree is associated with a tabled subgoal encountered in the evaluation (variant subgoals are considered identical), and consists of nodes of the form $Head$:-$Goals$ in which $Head$ carries the bindings found in a partial proof of a tabled subgoal and $Goals$ contains the list of goals remaining for the

partial proof A node with empty *Goals* is termed an *answer*. The literal selection strategy in *Goals* is fixed-order; in this paper we assume it to be left-to-right. If a node in a forest has a selected atom A that is not associated with an SLG tree, a NEW SUBGOAL operation is applicable to allow the creation of a new tree with root A:-A. Children of the root of a tree are produced by the PROGRAM CLAUSE RESOLUTION; children of other nodes are produced by POSITIVE RETURN. For definite programs this operation is equivalent to considering the answer as a fact and resolving it against the selected literal of $Goals$.

The above three operations continue until no more operations are possible for a set of mutually dependent subgoals. To make more precise the notion of dependency, we say that S_1 *directly depends on* a non-completed subgoal S_2 in a forest $\mathcal{F}$ iff S_2 is the selected atom of some node in the tree for S_1 in $\mathcal{F}$. Then, for a given forest $\mathcal{F}$ the *Subgoal Dependency Graph of* $\mathcal{F}$, *SDG(*$\mathcal{F}$*) = (V,E)*, is a directed graph in which $(S_i, S_j) \in E$ iff subgoal S_i directly depends on subgoal S_j, while V is the underlying set of E. The above definition relies on the notion of a subgoal being *completed*. To explain this, we first state that within a finite SLG evaluation, a set $\mathcal{S}$ of subgoals is *completely evaluated* in $\mathcal{F}$ if $\mathcal{S}$ forms a maximal SCC in $SDG(\mathcal{F})$ and all applicable NEW SUBGOAL and resolution operations have been performed on all nodes of every tree in the set. A COMPLETION operation is applicable to a set of completely evaluated subgoals, and explicitly marks each tree with the token *complete*.

For a given forest there may be many applicable SLG operations. Formalisms that restrict the number of applicable SLG operations in a given forest without sacrificing completeness are called *scheduling strategies*. Local evaluation is a scheduling strategy that makes use of the definition of an independent SCC. We call a strongly connected component *SCC independent* if it is maximal and $\forall S \in SCC$, if S depends on some S', then $S' \in SCC$. Informally, a Local Evaluation is one in which for any forest $\mathcal{F}$, NEW SUBGOAL, POSITIVE RETURN and PROGRAM CLAUSE RESOLUTION operations are applied only to trees whose subgoals are in an independent SCC of $\mathcal{F}$, and that COMPLETION operations are applied to all subgoals in an independent SCC at once. Several properties of Local Evaluation are proved in [6]. The implementation in this paper makes direct use of the following theorem:

Theorem 1 ([6]). *Let $\mathcal{E}^L$ be a finite Local SLG evaluation. For each $\mathcal{F}$ in $\mathcal{E}^L$ there is at most one incoming edge for each SCC in $SDG(\mathcal{F})$.*

2.2 SLG_{cl}

SLG$_{CL}$ [6] formalizes the actions of several threads of computation on a set of atomic queries, where each thread performs a Local evaluation. In SLG$_{CL}$ a tree can be marked with a thread identifier (*tid*) in addition to the token *complete*. SLG$_{CL}$ then adds thread compatibility restrictions to those of Local Evaluation. If a NEW SUBGOAL operation is performed by a thread T. the newly created tree is marked with T. Next, an answer A can be returned to a tree marked with T only if the tree in which A occurs is completed or also marked with T. And

finally, a COMPLETION operation is applicable to a set of subgoals only if they are all marked with the same *tid*. When the COMPLETION operation is applied, all completed trees have their *tid* overridden with *complete*. By themselves, the thread compatibility restrictions prevent completeness, as a forest may be deadlocked. A set $\mathcal{S}$ of non-completed subgoals in a forest $\mathcal{F}$ is in *deadlock* if: for each $S \in \mathcal{S}$ there are no applicable SLG_{CL} operations for N. A new SLG_{CL} operation resolves a deadlock for a forest and preserves completeness:

USURPATION: Given a set of subgoals $\mathcal{S}$ in deadlock mark all trees of $\mathcal{S}$ with $marking(S)$ for some $S \in \mathcal{S}$

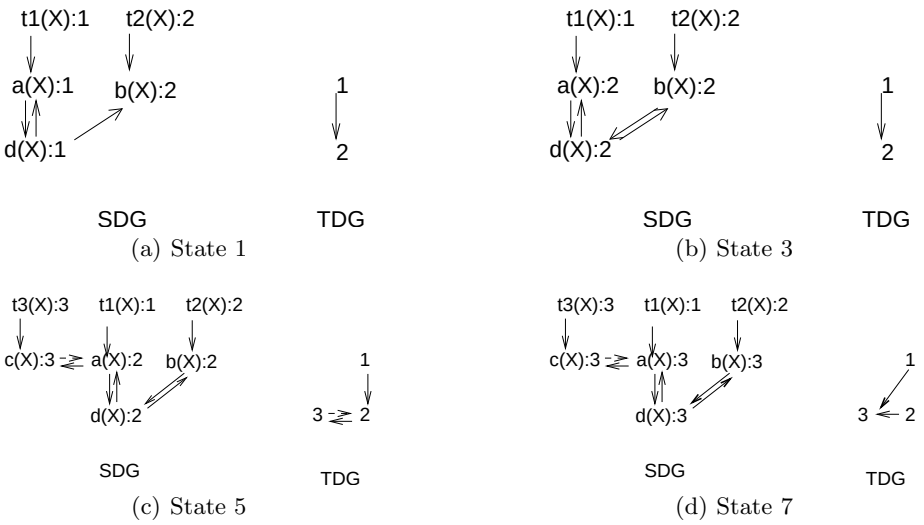
Example 1. To illustrate the USURPATION operation of SLG_{CL} consider program P_1 (Figure 1) and let there be three threads with identifiers 1,2 and 3 executing the initial queries `?- t1(X)`, `?- t2(X)`, `?- t3(X)`, respectively. Assume that thread 1 calls `t1(X)` which calls `a(X)` and then `d(X)`, while meanwhile thread 2 calls `t2(X)` and `b(X)`. Immediately after this sequence thread 1 also calls `b(X)`; because `b(X)` is marked by thread 2, thread 1 has no applicable operations — informally we say thread 1 is *suspended*. The SDG and TDG for the forest at this point are shown in Figure 2a. There is not yet deadlock, because thread 2 can still call `d(X)`. Thread 2 then does call `d(X)`, and determines that there is a deadlock. Thread 2 usurps `d(X)` from thread 1 arriving at the state shown in

```
:- table a/1, b/1, c/1, d/1.

t1(X):- a(X)       t2(X):- b(X).      t3(X):- c(X).

a(X) :- d(X).      b(X) :- d(X).      c(X) :- a(X).      d(X) :- b(X).
a(X) :- c(X).      b(b).              c(y).              d(X) :- a(X).
a(x).                                                    d(d).
```

Fig. 1. Program P_1

Figure 2b. Proceeding onward, thread 2 calls `a(X)`, and again determines that there is a deadlock. The subgoal `a(X)` is usurped from thread 1 and marked as belonging to thread 2. At this point let thread 3 call `t3(X)` and `c(X)`, while just afterward in thread 2, `a(X)` also calls `c(X)`. As `c(X)` belongs to thread 3 and as there is no deadlock, thread 2 suspends. Immediately after, thread 3 calls `a(X)` a state shown in Figure 2c. Now thread 3 detects a deadlock and usurps the SCC that was being computed by thread 2, as shown in Figure 2d. Threads 1 and 2 continue to be suspended. and both depend on thread 3, Eventually thread 3 completes the SCC. Only then are POSITIVE RETURN operations applicable for threads 1 and 2 using answers from the usurped SCC.

With the USURPATION operation, SLG_{CL} can be proved complete, and it can be proved that each thread performs a Local evaluation regardless of whether it suspends or has a subgoal usurped [6]. The algorithm of Section 3 relies on



(a) State 1 (b) State 3

(c) State 5 (d) State 7

Fig. 2. Concurrent execution of P_1

properties about thread dependencies. In a forest F let an *active thread* be a *tid* T such that there exists a tree, $Tr \in \mathcal{F}$, such that $marking(Tr) = T$. Then for two active threads, T_1 and T_2 in a SLG_{cl} forest $\mathcal{F}$, T_1 directly depends on T_2 if there exist a subgoal in T_1 that directly depends on a subgoal in T_2 (according to the definition of $SDG(\mathcal{F})$). The *Thread Dependency Graph TDG(F) = (V,E)* of $\mathcal{F}$ is a directed graph where V is the set of active threads in $\mathcal{F}$ and $(t_i, t_j) \in E$ iff active thread t_i directly depends on active thread t_j. For a forest $\mathcal{F}$ it is not hard to see that $TDG(\mathcal{F})$ is a graph homomorphism of $SDG(\mathcal{F})$, leading to the following theorem, which is used in Section 3.

Theorem 2. *[6] Let $\mathcal{F}$ be a forest in a SLG_{cl} evaluation. Then for each node in $TDG(\mathcal{F})$ there is at most one outgoing edge.*

2.3 Review of Relevant Portions of the SLG-WAM

We briefly review aspects of the SLG-WAM that are affected by or support the addition of concurrency to SLG.

Table Space The SLG-WAM maintains *table space* to store the tabled subgoals with their answers [7]. The relevant data structures include the following. The *Table Information Frame* or *TIF* is the top-level structure for each tabled predicate and contains information about the predicate, information for memory management, and a pointer to a *subgoal trie* which stores all current subgoals for a tabled predicate. When encountering a tabled subgoal, a subgoal_check_insert() function is called, which checks whether a subgoal is present in a trie and inserts it if not. Each leaf node of the subgoal trie corresponds to a subgoal S and points

to a *subgoal frame* which contains information about S. Two fields of the subgoal frame are relevant for our purposes. A *marking* field indicates whether or not S has been completed. An *ansTrieRoot* field points to the root node (if any) for the trie of answers for S. When a derivation produces an answer, answers are inserted into the answer trie if needed by an answer_check_insert() function. Nodes in the answer trie contain executable instructions, so that if S is completed, a call to S branches to the answer trie root to begin returning answers.

The Completion Stack XSB's SLG-WAM keeps a *completion* stack where each frame represents a pointer to a non-completed tabled subgoal in the current forest, along with dependency information used to construct a safe (over-)approximation of the independent SCC of the current SDG [3]. The completion stack is also used for scheduling in Local evaluation, where the oldest subgoal in the independent SCC is called the *leader* of the SCC (cf. [9, 3]).

SLG-WAM instructions The SLG-WAM contains several instructions not in the WAM, the more important of which we briefly cover. The `new_answer` instruction adds answers to the table when an answer derivation succeeds The answer_return instruction corresponds to the SLG Positive Return operation. Finally, the check_complete instruction checks for completion of an SCC and schedules answer_return and other instructions if subgoals in the SCC have not been completely evaluated. The tabletry instruction for a subgoal *Subgoal* has the form tabletry(*Arity*, *Clause*, TIF). As in the WAM, the representation of *Subgoal* is implicit in the argument registers; *Arity* is the number of registers to save and restore upon backtracking; and *Clause* the failure continuation. A new argument, TIF, is used to access the subgoal trie through the table information frame. When tabletry executes, a check/insert operation is performed for *Subgoal* in the subgoal trie for its predicate symbol. If *Subgoal* is in the table and has been marked as completed, tabletry branches directly to the answer trie of *Subgoal* if it has answers (failing if there are no answers). If *Subgoal* is new, a subgoal frame is created for *Subgoal*, along with a *generator choice point* which will be used to perform Program Clause Resolution. If *Subgoal* is not new, a *consumer choice point* is created to return answers to the calling environment. In the SLG-WAM stacks are also frozen so that computations that are waiting for an answer and have been suspended can be resumed when answers are later available.

Extensions for Multi-Threading In multi-threaded XSB, each thread of tabled execution has a structure called a *thread context* in which thread-specific information is maintained. Thus, for instance the **E** register for a thread is accessed as a field of its context structure. When a thread is suspended, any other thread can safely examine, and in some cases change, data in its context. In XSB, tables can be thread-shared or thread-private, although in this paper we restrict our attention to thread-shared tables. For shared tables, there is a lock on the subgoal trie for each (shared) tabled predicate, but as will be seen a lock is not required for answer tries, other than that required by the underlying memory management system.

[3] In Local evaluation, this approximation is exact if Early Completion is not used.

3 Implementing SLG$_{CL}$ in the SLG-WAM

The main addition needed to implement SLG$_{CL}$ in the SLG-WAM is the USURPATION operation: its implementation mainly affects the tabletry instruction, and is summarized in Figure 3. The tabletry instruction for the SLG$_{CL}$WAM differs from that of the SLG-WAM only if the called *subgoal* is not new and is currently marked by another thread (and therefore not marked as completed). In this case deadlock detection is performed and if a deadlock is not found the thread suspends, as it does not have any applicable SLG$_{CL}$ operations; otherwise the thread performs a USURPATION operation. When a thread usurps subgoals in this implementation of SLG$_{CL}$, any partial computations for the usurped subgoals are lost, and will be recomputed by the usurping thread. This design decision allows SLG$_{CL}$ to be added to a tabling engine such as the SLG-WAM in a simple manner, though, as shown below, the abstract complexity of evaluation for the well-founded semantics is not affected.

```
Instruction tabletry(Arity, nextClause, TIF)
    /* subgoal is in argument registers; Arity is arity of subgoal;
       nextClause is failure continuation; TIF points to table information frame */
    Perform the subgoal_check_insert(subgoal) operation in the trie for this predicate
    If subgoal is not new and is marked by another thread
        If waiting for subgoal to complete would produce a deadlock
            /* all other threads in the independent SCC are suspended at deadlock */
            Perform the usurpation operation:
                Mark all subgoals in the independent SCC as usurped
                For each thread T with an usurped subgoal S_T
                    reset T to perform its call to S_T
                    /* T will be awakened when S_T is completed */
        Else suspend the thread until subgoal completes
    Proceed as in the sequential case; if subgoal was usurped, treat it as a new subgoal
```

Fig. 3. Summary of the changes to the tabletry instruction.

Before discussing implementation of the USURPATION operation, we discuss two small changes. First, the *tid* marking for an incomplete tabled subgoal is kept in the *marking* field of the subgoal frame, while for completed tables the field contains the term *complete*. Second, the check_complete instruction is changed to wake any suspended threads waiting on the completing subgoals; as discussed below, this is done through a condition variable associated with each *TIF* frame.

Detecting Deadlock The definition of deadlock used in the SLG$_{CL}$WAM differs from that of Section 2.2 in that the implementation considers an independent SCC to be deadlocked even if there are applicable PROGRAM CLAUSE RESOLUTION or DELAYING operations while the formalism does not. In the SLG$_{CL}$WAM there is no reason to perform these operations since the partial computations of usurped subgoals will be discarded. Checking for deadlock is performed by the check_deadlock() function (Figure 4). check_deadlock() has

```
would_deadlock( subgoal_thread, current_thread )
    /* subgoal_thread marks the subgoal called by current_thread */
    while( subgoal_thread ≠ NULL )
        if( subgoal_thread = current_thread ) return true;
        else subgoal_thread ← subgoal_thread.suspended_on_thread);
    return false;
```

Fig. 4. The check_deadlock function.

a simple form: if the current thread calls a non-completed subgoal marked by another thread, it determines whether adding the dependency from the calling subgoal to the called subgoal would give rise to a deadlock. Dependencies in the TDG are maintained by a new *suspended_on_thread* field in the thread's context.[4] If creating such a dependency would cause the calling thread to depend on itself, then a deadlock is detected and a USURPATION operation will be necessary. Otherwise, if there is no present deadlock, the calling thread can simply suspend, waiting for the called subgoal to be completed. The correctness of check_deadlock() relies on the fact that any thread self-dependencies in the TDG are simple cycles without any subcycles: a corollary of Theorem 2 which states that each thread can depend on at most one other thread.

Gaining Control of Usurped Subgoals The fact that the thread dependencies for deadlocked threads form a simple cycle also underlies the control flow of the usurp() function (Figure 5) which consists of two traversals of the deadlocked TDG cycle. In the first traversal, $T_{usurper}$ updates the TDG, setting the *suspended_on_thread* field of each usurped thread to its own id. Adjusting the TDG must be performed under global mutual exclusion: otherwise two usurping threads concurrently adjusting the TDG might produce an incoherent TDG. Exclusion is enforced by the *usurpation_mutex*, which is set earlier in the tabletry instruction (see Figure 7) and is unset immediately after the TDG is updated in usurp(). In the second traversal, the execution stacks in each usurped thread are examined and manipulated through the function mark_and_reset() (Figure 6). This stack manipulation is safe since each usurped thread is suspended on the completion of a subgoal. In addition to resetting stacks, mark_and_reset() propagates subgoal dependencies among threads. The dependency propagation is based in part on a corollary of Theorem 1 that each thread can depend on at most one subgoal, S_{susp}, in its independent SCC, the value for which is maintained in the new *suspended_on_subgoal* field in the thread context. To characterize S_{susp}, observe that when a thread, $T_{usurped}$, is involved in deadlock, all threads in the deadlock share the same independent SCC, SCC_{dlock} and $T_{usurped}$ should be suspended on the first subgoal in SCC_{dlock} that it encountered during its evaluation. At deadlock, however, $T_{usurped}$ may not know the true value of S_{susp} because it may not know the true extent of SCC_{dlock} as dependencies from other threads may not have been propagated to $T_{usurped}$. In fact, there is

[4] In this presentation, we do not distinguish between a thread's id and its context.

only one dependency that must be propagated to $T_{usurped}$. To see this, recall that because a deadlock is a cycle in the TDG, any $T_{usurped}$ has exactly one thread, T_{dep} depending on it, and by Theorem 1, T_{dep} is suspended on exactly one subgoal in $T_{usurped}$. This dependency is passed into mark_and_reset() which determines the true value of S_{susp} for $T_{usurped}$ in a manner discussed below. When mark_and_reset() succeeds, it returns the true value of S_{susp} to usurp(), which sets the *suspended_on_subgoal* field of $T_{usurped}$. Before doing so, the old suspended subgoal of $T_{usurped}$ is obtained to be propagated to the next thread in the TDG cycle.

```
usurp( dep_SF, first_usurped, T_usurper )
   /* T_usurper called a subgoal with frame dep_SF, marked by first_usurped */
   T_usurped ← first_usurped;
   while( T_usurped ≠ T_usurper )                    /* first reset the TDG */
      T_usurped.next ← T_usurped.suspended_on_thread;
      T_usurped.suspended_on_thread ← T_usurper;
      T_usurped ← T_usurped.next;
   unlock( usurpation_mutex );                       /* locked in tabletry */
   T_usurped ← first_usurped;
   while( T_usurped ≠ T_usurper )                     /* now reset stacks for usurped */
      reset_sf ← mark_and_reset( T_usurper, T_usurped, dep_SF );
      /* reset_sf is true value of S_susp for T_usurped */
      dep_SF ← T_usurped.suspended_on_subgoal;
      /* dep_SF is the dependency to be propagated from T_usurped to T_usurped.next */
      T_usurped.suspended_on_subgoal ← reset_sf;
      T_usurped ← T_usurped.next;
```

Fig. 5. The usurp procedure.

In addition to dependency propagation, mark_and_reset() also marks the subgoal frames for usurped subgoals, and resets the execution stacks for the thread $T_{usurped}$ so that it will no longer compute its usurped subgoal, but rather will return answers once the usurped subgoal has been completed. The details are as follows. The function first checks whether the subgoal frame marked by $T_{usurped}$ has already had its information reset, by checking a new *usurped* field in the subgoal frame. For a previously usurped subgoal, the *marking* field need only be set with the id of $T_{usurper}$ and mark_and_reset() can return immediately. Otherwise if the subgoal has not been previously usurped, the function uses a while loop to traverse the completion stack of $T_{usurped}$ to find its portion of SCC_{dlock} (as mentioned in Section 2.3 an independent SCC is represented by a segment on the top of the completion stack). However, as discussed above, the dependency from T_{dep} to $T_{usurped}$ is not propagated until a thread is actually usurped. Accordingly, in Figure 6, the completion stack is traversed from the top of stack, represented by *usurped.CmplStkReg* to the true leader of the revised SCC. More precisely, the completion stack is traversed until the first leader is found that is at least as

```
mark_and_reset( T_usurper, T_usurped, dep_SF)
    /* The dependency dep_SF is propagated to T_usurped during usurpation*/
    if( dep_SF.marking ≠ T_usurped )      /* usurped was previously usurped */
        return T_usurped.suspended_on_subgoal;
    /* Find the oldest subgoal deadlocking SCC and mark subgoal frames */
    CSF ← T_usurped.CmplStkReg; found_dep_SF ← false;
    while( not (found_dep_SF and is_scc_leader(CSF)) )
        if ( CSF.subg_ptr = dep_SF )      found_dep_SF ← true;
        SF ← CSF.subg_ptr;
        SF.usurped ← true; SF.marking ← T_usurper;
        reset subgoal frame cells in SF having to do with computation state in the stacks
        decrement(CSF);
    /* Finally, reset the stacks of T_usurped */
    T_usurped.ComplStkReg ← CSF;      /* pop the completion stack */
    T_usurped.B ← SF.generator_cp;    /* get the generator cp */
    use the information in the generator cp to reset usurped's stacks;
    T_usurped.P ← T_usurped.B.reset_pcreg;/* set forward continuation */
    T_usurped.B ← T_usurped.B.prevbreg;   /* delete the generator cp */
    return CSF.subg_ptr;
```

Fig. 6. The mark_and_reset() procedure.

old as the new dependency, dep_SF. This is essentially the same propagation as if $T_{usurped}$ itself had called the subgoal represented by dep_SF (see [9] for the actual computation of leaders in the SLG-WAM).

For each completion stack frame traversed in the while loop, the appropriate subgoal frame is obtained, its *usurped* field set, its *marking* field set to $T_{usurper}$, and other fields re-initialized. Once the while loop is exited, CSF is the completion stack frame associated with the proper suspended subgoal, S_{susp}. Next, the state of $T_{usurped}$ is set to call S_{susp}. This is done by obtaining the generator choice point for S_{susp}, which provides information to reset stack and freeze registers of $T_{usurped}$ in a manner analogous to failing. A small difference from failing is that the argument registers of $T_{usurped}$ are reset to their state at the *call* of S_{susp} rather than to a failure continuation. In order to do this, a generator choice point contains a new $reset\_pcreg$ field which points to the original tabletry instruction for S_{susp}, which is used to set the **P** register of $T_{usurped}$. Upon awakening $T_{usurped}$ will re-execute the tabletry instruction, but this time it will determine that S_{susp} has been completed, and will simply return any answers in the completed table for S_{susp}.

The tabletry Instruction The tabletry instruction (Figure 3) detects deadlock, and ensures that if there is deadlock one and only one thread performs usurpation. We describe the actions of a thread T calling a subgoal *subgoal* for a shared tabled predicate $Pred$, ignoring at first concurrency issues. In addition to the $Arity$ and $nextClause$ pointer of a WAM try, tabletry contains a pointer to the predicate-level (*TIF*) (Section 2.3). The *TIF* field contains a pointer to the subgoal trie, but also information on whether $Pred$ is thread shared or thread

private — Figure 3 includes pseudo-code only for shared tables. The function subgoal_check_insert() determines whether *subgoal* resides in the subgoal trie: it returns a pointer to its subgoal frame if so and Null if not. If *subgoal* is new, actions proceed as usual, although the *marking* field of the subgoal frame is set to the executing thread's id, T (case α). Otherwise if *subgoal* is not new (case β), if *subgoal* has been usurped by $T_{usurper}$ (case $\beta.1$), computation of *subgoal* must be started afresh by T so the normal steps for a new tabled subgoal are taken, a step ensured by setting the *new_subgoal* flag. In case $\beta.2$ if *subgoal* is not marked by T (and not completed) a determination must be made whether to suspend T or to perform a USURPATION operation. As discussed above, check_deadlock() is called and if there is a deadlock usurp() is called; afterwards control will jump to the sequential portion of tabletry where the usurped *subgoal* will be treated as new. If there is no deadlock, the *suspended_on_thread* and *suspended_on_subgoal* fields of T's subgoal frame are set, and T suspends on a condition variable associated with the TIF for *subgoal*. The sequential portion of tabletry works as in the SLG-WAM, with the minor exception that a subgoal is treated as new if the *new_subgoal* flag has been set.

Returning to the concurrency issues for tabletry, it must be ensured that one and only one thread will create a new tabled subgoal, and if any deadlock occurs, one and only one thread will usurp the deadlocked SCC — while at the same time allowing as much concurrency as possible. The first issue is to ensure mutual exclusion for the (predicate-level) subgoal trie during the subgoal_check_insert() function. Each subgoal trie (for thread-shared tables) has its own mutex, which is unlocked by tabletry as soon as possible — after the frame is created for *subgoal* if *subgoal* was new. Next, a global usurpation_mutex is locked to ensure exclusion of check_deadlock() and part of the usurp functions; as discussed above this is necessary to prevent two threads from concurrently updating the TDG and possibly making it incoherent. If T suspends, it will wait on a condition variable associated with the predicate-level TIF of *subgoal*. When T is awakened, it must recheck whether *subgoal* was actually completed, as different subgoals may share the same predicate-level condition variable.

Summary of Changes The changes to SLG-WAM data structures include the *suspended_on_thread* and *suspended_on_subgoal* fields of the thread context, which are used to maintain the thread dependency graph and parts of the subgoal dependency graph; a *next* field is also needed for TDG cycle traversal by usurp(). In addition there is the *usurped* field in the subgoal frame and the *reset_pcreg* field in the generator choice point, both of which are required for usurpation. Since any shared table is marked by a single thread, no locks are required for answer tries. Finally, a condition variable is added to each TIF, and a global *usurpation_mutex* is required. At the instruction level, there is a minor change to the check_complete instruction to wake up threads that may be suspended on completing subgoals. The tabletry instruction has the additions described in this section. However, no changes are needed to tabletry code beyond factoring out the subgoal_check_insert() operation and subgoal frame creation as shown in the first few lines of Figure 7.

```
Instruction tabletry(Arity, Clause, TIF)
     /* subgoal is in argument registers; T is executing thread */
     lock(TIF.subgoal_trie_mutex);                    /* Handle shared tables */
     SF ← subgoal_check_insert(subgoal, TIF);
α    if ( SF = NULL )                                 /* subgoal is not in the table */
        SF ← CreateSubgoalFrame(subgoal);
        /* sets SF.usurped ← false; SF.marking ← T; */
        unlock(TIF.subgoal_trie_mutex);
        new_subgoal ← true;
β    else                                             /* subgoal is already in the table */
        unlock(TIF.subgoal_trie_mutex);
β.1     if( SF.marking = T and SF.usurped )           /* subGoal was usurped by T*/
           new_subGoal ← true; SF.usurped ← false;
        else new_subGoal ← false;
β.2     if(SF.marking ≠ T and SF.marking ≠ completed)
           lock(usurpation_mutex);
           while ( SF.marking ≠ completed )
              if( check_deadlock( SF.marking, T) )
                 usurp(SF,SF.marking,T);             /* unlocks usurpation_mutex */
                 new_subgoal ← true; SF.usurped ← false;
                 goto seq_tabletry;
              T.suspended_on_subgoal ← SF;
              T.suspended_on_thread ← SF.marking;
              cond_wait(TIF.cond_var,usurpation_mut);
           unlock(usurpation_mutex);
           T.suspended_on_subgoal ← NULL; T.suspended_on_thread ← NULL;
           T.usurping ← false;
           Branch to instruction in P register;
seq_tabletry:  Execute as in sequential SLG-WAM; treat subgoal as new iff new_subgoal is true
```

Fig. 7. The tabletry instruction for the SLG_{CL}WAM

3.1 Extensions for Negation, Constraints and Answer Orderings

The discussion has so far focussed exclusively on tabled evaluation of definite programs. Because SLG$_{CL}$ differs from SLG essentially only in the USURPATION operation it should not be surprising that the SLG$_{CL}$WAM requires few changes beyond those already indicated in order to implement the well-founded semantics. Consider first the case of stratified programs. In the sequential SLG-WAM, if the underlying (tabled) subgoal S of a selected negative literal is not new and not complete, the computation path "suspends" and resumes only when S has been completed. These operations are the same as the interactions between threads so far described.[5] In the case of non-stratified negation the first new operation to consider is the SLG DELAYING operation. Delaying is handled in the SLG-WAM essentially as with stratified negation. If S is involved in a loop through negation, the resumption mechanism is the same except that a bit in the subgoal frame of S is set to indicate that S was delayed rather than completed. Several cycles of delaying may be needed before S is finally completed, but these may all be handled using the thread suspension and usurpation mechanisms described. When S is completed, any SIMPLIFICATION operations for its SCC are also performed before awakening any threads suspended on S, so that SIMPLIFICATION is not affected by the concurrency mechanisms.

Tabling constraints also carries over to the SLG$_{CL}$WAM in a simple manner. In XSB, constraints are tabled by copying attributed variables into and out of tables for subgoals and answers. The actual mechanism for this copying is encapsulated in the subgoal_check_insert() and answer_check_insert() operations and is therefore unaffected by the changes to tabletry. Similarly, implementations of answer ordering are performed as extensions to the SLG-WAM new_answer operation which is also unaffected by the changes for SLG$_{CL}$.

4 Complexity and Performance

Despite its advantages, the described implementation has to recompute answers for usurped subgoals. To understand this effect on complexity, we denote a SLG$_{CL}$ evaluation that recomputes answers for usurped subgoals as an *SLG$_{CL}$ evaluation with restart*. In [6] it is shown that the maximal number of USURPATION operations in a SLG$_{CL}$ evaluation is linear in $atoms(P)$, the number of atoms in a program P. Now assuming perfect indexing, all SLG operations can be considered to be constant, except for COMPLETION and USURPATION, which have worst-case complexity of $atoms(P)$, and ANSWER COMPLETION which has worst-case complexity of $size(P)$, the size of P. Because completely evaluated SCCs cannot be usurped, and because ANSWER COMPLETION need only be performed on completed tables, USURPATION affects only constant operations, and occurs $\mathcal{O}(atoms(P))$ times, giving rise to the following:

Theorem 3. *Let $\mathcal{E}$ be a finite SLG$_{cl}$ evaluation with restart of a query to a program P. Then $\mathcal{E}$ has worst-case complexity of $\mathcal{O}(atoms(P)size(P))$.*

[5] A minor difference is that threads suspended on negative literals need to be reset to the beginning of `tnot/1`, rather than to the tabletry instruction.

Theorem 3 is significant, since known computation methods of the well-founded semantics have the same complexity for unrestricted normal programs (cf. [1]).

Performance A full performance analysis of the SLG$_{CL}$WAM cannot be presented here, so we focus on illustrating extremal behavior with respect to deadlocking and USURPATION. Table 4 shows scalability on left-recursive transitive closure for randomly generated graphs. Each graph is designated by the notation $Vertices \times Edges\_per\_vertex$: for instance, the first row measures a graph of 256 vertices, each of which have 128 edges per vertex. The columns indicate elapsed time and speedup for N threads to each perform $Vertices/N$ queries of the form `path(bound,free)`. These queries show nearly linear speedup on up to 8 threads, which is not surprising as these evaluations do not require either USURPATION or thread suspension. On the other hand, executing right-recursive

N Threads:	1	2	Speedup	4	Speedup	8	Speedup
256x128	1.46s	0.73s	2.0	0.38s	3.8	0.19s	7.7
512x8	0.60s	0.31s	1.9	0.16s	3.8	0.10s	6.0
2048x2	4.62s	2.38s	1.9	1.27s	3.6	1.03s	4.5
8192x1	1.30s	0.67s	1.9	0.36s	3.6	0.20s	6.5

Table 1. Scalability for left-transitive closure in random graphs using N threads

transitive closure on these graphs provides a situation where USURPATION is expected to occur heavily. Table 4 shows the number of deadlocks for the randomly generated graphs with from 1 to 256 threads on an 8-core machine each evaluating a random right recursive query of the form `path(bound,free)`. Given the number of vertices in the graphs, a relatively high number of threads need to be concurrently invoked to obtain more than than 10 deadlocks or so, which is observed only on the moderately dense graphs. Table 4 shows the times and

N. Threads	1	2	4	8	16	32	64	128	256
256x128	0	0	2	3	7	8	7	12	16
512x8	0	1	0	3	3	9	21	4	8
2048x2	0	1	6	8	20	33	26	16	36
8192x1	0	0	0	0	0	0	0	1	1

Table 2. Number of deadlocks for transitive closure with right recursion for N threads

speedups for right recursion. While there is little speedup for the three more densely connected graphs, the repeated USURPATION operations do not slow the times down, and the evaluations degenerate into behavior similar to a sequential evaluation. Of course, having a large number of threads simultaneously querying small densely connected graphs using right recursion is arguably a "worst"-case situation for the SLG$_{CL}$WAM, but in these cases, the cost of restarting a usurped SCC does not appear to be high.

N Threads:	1	2	Speedup	4	Speedup	8	Speedup
256x128	1.64s	1.64s	1.0	1.62s	1.0	1.65s	1.0
512x8	0.61s	0.58s	1.1	0.57s	1.1	0.56s	1.1
2048x2	3.20s	2.68s	1.2	2.44s	1.3	2.23s	1.4
8192x1	0.65s	0.34s	1.9	0.20s	3.3	0.12s	5.4

Table 3. Scalability for right-transitive closure in random graphs using N threads

5 Summary

While they are conceptually complex at times, the algorithms for deadlock detection and USURPATION are based on a formal semantics for SLG_{CL}, so they can be concisely stated, theiir correctness clearly argued and they support a number of tabling features. In addition, because only actions upon the call of a tabled subgoal are significantly changed, the approach should be adaptable to a variety of engines. Substantiation for this claim is provided by the fact that implementation of SLG_{CL} in XSB required approximately 300 lines of code including code for negation. SLG_{CL} is thus a simple and effective way to extend a tabling engine for concurrency.

References

1. K. Berman, J. Schlipf, and J. Franco. Computing the well-founded semantics faster. pages 113–125. Springer-Verlag, 1995.
2. W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.
3. J. Freire, T. Swift, and D. S. Warren. Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. *JFLP*, 1998(3), 1998.
4. H. Guo and G. Gupta. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternates. In *ICLP*, pages 181–196, 2001.
5. P. Guzman, M. Carro, M. Hermenegildo, C. Silva, and R. Rocha. An improved continuation call-based implementation of tabling. In *PADL*, 2008.
6. R. Marques and T. Swift. Concurrent and local evaluation of normal programs, 2008. Submitted. Available at `http://www.cs.sunysb.edu/~tswift`.
7. I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient access mechanisms for tabled logic programs. *JLP*, 38(1):31–55, January 1999.
8. R. Rocha, F. Silva, and V. S. Costa. On applying or-parallelism and tabling to logic programs. *Theory and Practice of Logic Programming*, 4(6), 2004.
9. K. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM TOPLAS*, 20(3):586 – 635, May 1998.
10. Z. Somogyi and K. Sagonas. Tabling in Mercury: Design and implementation. In *PADL*, 2006.
11. N. Zhou, Y. Shen, L. Yuan, and J. You. Implementation of a linear tabling mechanism. *Journal of Functional and Logic Programming*, 2001(10), 2001.