An Architecture for a Multi-threaded Tabling Engine

Rui Marques    Terrance Swift    José Cunha

1 Introduction

Over the last few years, tabled logic programming (TLP) has been used for large-scale commercial and research programming efforts in data cleaning, diagnosis, ontology management, and model checking (see [14] for a survey of some of these projects). Because of its use in applications, several groups have begun to examine how TLP can be executed within a shared-memory parallel setting [5, 11] or distributed memory setting [7]. The latter approach in particular can exploit resources available to multi-computers and can execute tabled programs across geographically distributed environments. Such an approach is based on a set of distributed agents that communicate with one another to evaluate a (possibly updatable) tabled logic program. A first step in implementing distributed TLP, then, is to create a tabling engine capable of processing multiple requests concurrently. Such an engine is also useful in itself for client-server applications, and can facilitate a shared-memory parallel implementation of TLP. This paper proposes a multi-threaded architecture for such an engine, whose main design principles are as follows.

**Principle I** The multi-threaded architecture should extend that of the SLG-WAM of XSB, and should impose no perceptible overhead for single-threaded evaluation.

This allows us to build and extend on existing XSB technology. Next, for efficiency, the separate threads should share information whenever possible.

**Principle II** A tabled subgoal is created only once, and new invocations of it will share the previously generated information even if made on a different thread.

This principle implies that all threads share common subgoal table information. Finally, effort in creating a multi-threaded engine should make an implementation of table parallelism easier.
Principle III The multi-threaded architecture should be extendable to shared-memory parallel implementation.

This principle implies, as a corollary, that our basic algorithms should not impose unnecessary dependencies to the execution of otherwise independent threads.

In this paper, we first survey design issues in single-threaded TLP engines, and then show how these algorithms are used or modified in the multi-threaded architecture. In particular, our architecture implies a revision of the implementation techniques and scheduling strategies used in the sequential engine, and in particular a new completion algorithm is needed.¹

2 Design Issues in Sequential Tabling Engines

Progress in implementing tabling within a WAM-based framework has led to the identification of issues that need to be addressed by TLP engines along with several proposals for addressing these issues. We discuss major issues in turn.

Memory Management TLP requires the ability to suspend a computation path in order to search for solutions to a tabled subgoal and to resume the computation path to resolve answers once they are derived. The ability to suspend and resume is not present in sequential Prolog but is present in many frameworks for parallel Prologs. As with parallel Prologs, algorithms which implement suspend and resume can be classified into memory-sharing algorithms and memory-copying algorithms. Memory sharing algorithms were proposed in the original implementation of the SLG-WAM [12], which uses freeze registers and a forward trail to support suspending and resuming within WAM stacks. Another class of algorithms copies computation paths into execution space as they are resumed and out of execution space as they are suspended [2]. In terms of tabled evaluation, memory sharing algorithms provide the benefit of simple and efficient switching between environment paths, but may trap unused computation paths in frozen areas of memory, while memory copying algorithms have complementary benefits and weaknesses. Memory management strategies based purely on copying alone proved inefficient due to the cost of copying all information in a computation path into and out of memory. As a result a hybrid was proposed for copying of trail and choice point stacks and sharing of heap and local stacks — the CHAT memory model of [4]. The current default version of XSB shares all stacks except the choice point stack and trail, while YAP[11] shares all stacks during sequential evaluation.

Table Access Methods Tabling systems must support efficient algorithms for maintaining tables. Different algorithms are useful depending on how terms in a table are used during evaluation. For instance, a term (subgoal or answer) may be inferred from a table if it is a variant of a term already in the table, or if it is subsumed by a term in a table (on the lattice of terms or any other lattice). These

¹An extended version of this paper will prove the correctness of the completion algorithm and discuss its complexity.
operations are well supported by tries. Trie-based data structures for variant tabling were described in [10].

**Scheduling Strategies** Scheduling in Prolog essentially involves ordering program clause resolution steps, and is done in a stack-based (depth-first) manner. Scheduling in tabled evaluations includes the decision of when to return answers to tabled subgoals. Several scheduling strategies have been formalized within tabled SLG resolution and implemented in XSB, the most important of which are batched evaluation and local evaluation [6]. From a somewhat simplified perspective, batched evaluation returns answers to consuming subgoals “eagerly”, and reduces the time to derive the first answer to a subgoal. Local evaluation delays the return of an answer for subgoal \( S \) to a consuming environment until \( S \) is completely evaluated — whenever this is possible. Local evaluation can be useful for answer subsumption since all answers but the most general are often discarded before returning answers to a consuming environment.

**Incremental Completion** Determining when a mutually dependent set of subgoals is completely evaluated — when all possible resolution steps have been performed for those subgoals — is of prime importance for practical implementations. Completely evaluated subgoals do not require execution stack space. Furthermore, the use of complete evaluation aids in handling negation efficiently. Given a completely evaluated subgoal \( S \) that has no answers, the literal \( \text{not} \ S \) can be resolved away from calling environments; while \( \text{not} \ S \) would have to be delayed if \( S \) had no answers but was not determined to be completely evaluated. Eagerly detecting that mutually dependent subgoals are completely evaluated is called **incremental completion**.

Performing incremental completion requires knowledge of the dependencies among subgoals. One approach would be to construct (parts) of a suitably defined **subgoal dependency graph** when a subgoal is checked for completion, but experiments have shown that this approach is computationally expensive. To address this problem, [1] provided an algorithm for using a **completion stack** to maintain a safe over-approximation of the dependency graph, called approximate strongly connected components (ASCCs). Using SCCs, stack-based algorithms, such as those described in Section 4, can be implemented to perform incremental completion.

**Negation Handling** The main mechanisms provided by SLG resolution for evaluating negation is to delay a negative literal that may be non-stratified, and to simplify that literal away once its truth value becomes known. Several algorithms are required to support negation. First, in batched evaluation, a check of subgoal dependencies may be useful to minimize the use of the delaying operation and the potentially unnecessary resolution of literals [12] (such a check is not needed with local evaluation, as the interdependent sets of subgoals are computed in a dynamically ordered, bottom-up fashion). Specifically, negation handling uses a dependency check to determine whether an independent SCC contains stratified SCCs as subcomponents. If so, subgoals in these subcomponents are completed, and negative
literals involving these subcomponents are resolved away. If not, all subgoals in the ASCC are delayed. Second, data structures must be created to represent delayed literals in the execution stacks and tables and to support simplification in the tables as discussed in [13].

3 Design Issues for a Multi-Threaded Tabling Engine

In keeping with Principle I, our proposed architecture builds upon the architecture of the SLG-WAM. Broadly, a new execution thread, consisting of SLG-WAM stacks and registers is created when a new subgoal is queried at the top level of the engine. In keeping with Principle III we rely on POSIX threads[8, 9], (also known as pthreads) to support our multi-threaded engine.

For the most part, each thread executes as in the sequential engine and exits when its top-level goal is completely evaluated or when the user aborts the computation. The exception to sequential computation occurs when thread $T_i$ calls a tabled subgoal that has previously been called by a different thread $T_2$ but has not been completed. In keeping with Principle II, $T_i$ consumes answers from the table while $T_2$ produces answers for the table. In this case, the routines for maintaining approximate dependencies for incremental completion must be modified.

Scheduling of different threads is now an issue. We consider four possible scheduling states for a thread: blocked (say on I/O), completing, ready and running — the traditional scheduling states for threads plus the completing state. A thread in the completing state is waiting for completion of an ASCC that spans multiple threads. A completing thread can be resumed either by being fed with new answers, new dependencies that have to be propagated have been found or by the completion of its multi-threaded ASCC.

The multi-threaded engine must also provide some basic operations for the lowest layer of XSB programming to be able to access the multi-threaded functionality. Initially, each primitive directly reflects the semantics of a pthreads primitive. For instance, $xsb\_thread\_create(+Predicate,+ThreadAttributes,-ThreadId)$ creates a new thread to evaluate a given predicate (with arity 0). The $ThreadAttributes$ argument can specify thread attributes such as stack sizes, while the identifier of the thread created is returned in $ThreadId$. Similarly $xsb\_thread\_exit(+Status)$ finishes the execution of the current thread. The primitive $xsb\_thread\_join(-Status)$ allows one thread to wait for the termination of another, returning its exit status. More will eventually be offered, as their need in multi-threaded XSB programs becomes apparent.

Memory Management Since each thread has its own execution stacks, execution of the suspend and resume operations within a thread is essentially as in the single-threaded engine and can be done with any combination of sharing and copying mechanisms. The single difference occurs in freeing stack space (or copied execution environments) when an ASCC spans threads. In this case, a thread may need to
wait to free memory until other subgoals in its ASCC are ready to be completed, as discussed below.

Other algorithmic issues, such as completion detection, table management, and so on work for the multi-threaded engine as for the single threaded engine. This simple approach was implemented in a meta-interpreter in [7] on top of a shared-memory system. However, experiments showed that by switching execution between goals for unrelated threads, many subgoals became trapped in frozen space. Within the current memory management architecture of XSB the problem of trapped execution environments could be addressed by implementing garbage collectors for the shared local stack and trail, and by altering the memory management routines for the copied choice point stack. However it would be difficult to extend this approach to support a parallel implementation (keeping with principle III). As our threads execute in separate stacks, our approach to communication among threads is by sharing the tables, an issue discussed in the following paragraph. While this incurs in more overhead in communication among threads, as compared to the above approaches, it was chosen for its conceptual simplicity.

**Table Access Methods** For the most part, the design of table access methods is orthogonal to multi-threading. Access to the table must be synchronized using locks; we note that only one thread will be the producer for each subgoal; this implies that only one thread will modify the table data structures for one subgoal, while the others, that will host the consumer subgoals, will only need to access the subgoal data structures for reading. Determining the exact granularity of table locks will need experimental results.

**Scheduling Strategies within a Thread** Issues of scheduling for returning answers within a thread are orthogonal to multi-threading in the sense that any scheduling strategy that is correct for a sequential engine is extendable to our proposed multi-threaded architecture. Of course, the advantages of scheduling strategies may vary between a sequential and a multi-threading environment. Batched evaluation has the advantage of reducing the time until a first answer is found, and this advantage may be useful if a multi-threaded engine is to produce answers to a client that must process these answers. However, the dependency check involved in batched evaluation may require a large amount of synchronization and may reduce the potential of the engine for full parallelism. Furthermore, [6] shows that ASCCs tend to be larger for batched evaluation than for local evaluation, so that batched evaluation may require more synchronization overhead of completion detection between threads. Determining which evaluation method may be most useful for the multi-threaded server can probably only be determined by experimentation.

**Negation Handling** As mentioned above, negation handling in batched evaluation requires a dependency check that is performed by traversing the choice point stack, into which the subgoal dependency graph is embedded. Such a dependency

---

2 This involves copying terms in and out of the table and resembles an approach known on the literature as the *closed bindings approach*. 
check is straightforward to implement although it requires synchronization among threads.

4 A Concurrent Algorithm for Incremental Completion

In this section we show the algorithm for completion in the sequential SLG-WAM and propose its extension for the multi-threaded SLG-WAM. For the sake of simplicity we stick to batched scheduling and definite programs. For the concurrent algorithm we assume that all threads can read the completion stacks of each other. This follows from a multi-threaded environment, but we stress that only for negation (as discussed in the previous section) do we need one thread to access another’s choice point stack.

The access to the table data structures is assumed to be protected by locks, whose type and granularity we won’t discuss. To each thread is associated a condition variable, where it is blocked when it assumes the completing state. We base our algorithm in the monitor approach, i.e. all parts of the completion algorithm execute under mutual exclusion, just like if they were procedures of a monitor[8]. While this doesn’t allow for two threads to execute the operations associated with completion simultaneously, thus somehow limiting the opportunities for real parallelism, it works best for single processor systems. For this issue we decided on Principle I over Principle III. A more elaborate, parallel, completion algorithm would provide for more parallelism in the completion operation (see [5]). We believe the current algorithm could be extended for such purposes.

4.1 Completion within the sequential SLG-WAM

Our presentation of the completion algorithm for the sequential SLG-WAM follows that of [12]. Within the sequential SLG-WAM a completion stack is used to keep track of dependencies among incomplete subgoals. The completion stack operates in parallel with the choice point stack. We keep the completion stack separate from the choice point stack to isolate the operations involved in the completion operation as described below. Each frame of the completion stack contains:

- **DFN** - A unique number for each subgoal, representing the chronological order of each subgoal call.

- **DirLink** - Which keeps track of the deepest direct dependency for each subgoal, using its DFN.

For each subgoal, S, we define MinLink(S) to be the least DirLink value for all subgoals on the completion stack whose DFN is greater or equal to that of S.

By propagating DirLink values, as described below, a subgoal whose MinLink equals its DFN value cannot depend on any subgoals whose DFN is smaller. Such a subgoal is called a leader and the segment of the completion stack younger than itself is called an ASCC (Approximate Strongly Connected Component). ASCCs are safe
over-approximations of exact SCCs in that an ASCC may fully contain one or more
SCCs. Completion stack frames are initialized at the New Subgoal operation and
DirLink value are propagated during the completion operation. We now turn to the
description of these operations.

The SLG New Subgoal operation determines if a subgoal, $S$, is new to an
evaluation, in which case program clause resolution should be used or whether that
subgoal has already been encountered in which case answer resolution is used. From
the perspective of dependency propagation if the subgoal is new, a completion stack
frame is created and initialized with a newly generated DFN. The DirLink value is
set equal to the DFN. If $S$ has been called before, the DirLink of the subgoal, whose
frame is at the top of the completion stack, is set to the DFN of $S$.

Within the SLG-WAM, a completion operation is executed for a subgoal $S$
after all program resolution for $S$ has been performed. The completion operation
first determines whether $S$ is a leader by checking its DirLink and MinLink values.
If $S$ is not a leader its DirLink value is propagated to the previous completion stack
frame and the engine backtracks. If $S$ is a leader, each subgoal in its ASCC is
checked for answers to return. If there are no answers to return to any subgoal the
engine completes. Otherwise, the answers are scheduled to be returned and when
the engine backtracks it will find the choice points of the subgoals for which the
answers need to be returned on the backtracking chain. We formalize this operation
in Figure 1.

```c
if MinLink(S) < DFN(S)           /* S is not Leader */
  PropagateDependencies(S);
  FAIL
else /* S is leader */
  if !check_fixpoint(); /* There are answers to return */
    FAIL  /* return the answers */
  else /* complete */
    for each Subgoal S1 on top of S
      reclaim_space(S1)
  top of completion stack = prev(S)
```

Figure 1: Completion operation in the sequential SLG WAM for subgoal $S$

### 4.2 Extending the algorithm for multiple threads

Extending the completion algorithm for multiple threads introduces many difficul-
ties, like synchronization and preservation of the integrity and system invariants.
However the most difficult issue is to deal with dependencies of subgoals among
different threads. In the algorithm we present here the ASCC concept is extended
to span threads; the notion of leader in the sequential engine is now divided into two
categories, the local leader and the global leader. The local leader is just the thread
leader, and controls completion and scheduling in a similar fashion to the sequential
engine leader. The global leader is defined as the last local leader that is involved
in a multi-threaded ASCC to finish its local ASCC.
While sequential ASCCs depend on each other in a purely stack based fashion, Multi-threaded ASCCs (henceforth referred to as MT-ASCCs) depend on each other on a partial order relation. We define an independent MT-ASCC as one that has no dependencies on any other. Only independent MT-ASCCs can be completed. In the purely stack based algorithm an older ASCC is never called to complete until all of the younger ones, which reside on top of it, are done with; in the concurrent algorithm the youngest ASCC of its thread might not belong to an independent MT-ASCC, and so it will be trapped until the subgoals it depends on complete.

When a call is made to a subgoal older than any of the local leaders of the thread, the thread's ASCC (and thus the MT-ASCC) has to be extended. This extension might propagate to ASCCs on other threads. In our algorithm this extension is done lazily; at call time the dependency is propagated between two threads, while at completion time, other recently introduced dependencies are propagated. In this algorithm we have to consider an unique thread identifier (TID) for each thread; for the sake of simplicity we assume that the domain for the value of thread identifiers is infinite (i.e. TID values don't have to be re-used once a thread is finished).

**Data structures for concurrent completion** To model the inter-thread dependencies, each local leader has now to keep a list of the dependencies from its ASC to subgoals on other threads. While this is only needed for the thread leader, it must keep the dependencies of all the subgoals in its ASC, which have to be propagated through the subgoals of the thread’s ASC. For that purpose each completion stack frame for subgoal S now has one more field, a list of triplets \(<TID, Pointer, OldLink>\), known as the **DependencyList**, each representing the deepest dependency of S on a subgoal on a different thread. The values for each of these elements are as follows:

- **TID** - identifier of the thread where the subgoals on which the ASCC depends were created.
- **Pointer** - pointer to the deepest subgoal that the ASCC depends on, in the thread denoted by **TID**.
- **OldLink** - DirLink value for the subgoal of that thread that we depend on when the DirLink field was created. This is used to ensure that the dependency has not been overwritten by new subgoals after the current subgoal has finished evaluating.

From here on we shall refer to elements of the dependency list for completion stack frame A as \(A[i]\), where \(i\) is the value of the thread identifier field of that element. For ease of programming, the **TID** of thread T is also kept on every frame of T's completion stack. We also define a global data structure, the **state table** where information about each thread is stored and which is used to perform synchronization among threads within the completion algorithm. For each thread T there is an entry, referred as **state[i]**, where i is the TID of T. state[i] contains:

---

3This limitation can be overcome adding an incarnation number to each thread identifier. This technique is shown in the complete algorithm at the appendix.
- **live** - a boolean flag, true if $T$ has been started, and still hasn’t evaluated its top goal. Set to true when the completion stack of a thread is created, false just before its released.

- **leader** - the leader of the youngest ASCC of $T$. This is used for other threads to have immediate information on the local leader of the ASCC.

- **completing** - a boolean flag, true if $T$ is waiting for others to complete or return new answers to it.

**Creation of new subgoals** For the MT-ASCeS to be maintained the **NEW SUBGOAL** operation has to be changed. When subgoal $S_a$ in thread $T_1$ calls subgoal $S_b$ in a different thread $T_2$, all the elements in the dependency list have to be updated. If $T_2$ was marked as having no answers to return, it is now marked has having answers to return and it is awakened if it was blocked waiting for completion. This allows for new inter-thread dependencies to be introduced, as a subgoal from a thread calls one that is assigned to another. A completion stack frame is pushed for $S_b$, in the completion stack of $T_1$, so that the **check fixpoint** operation, which traverses the completion stack to return the answers, does not need to be changed for returning answers from other threads (see Figure 2).

```plaintext
if Sa is not yet a consumer of Sb
    t1 = tid(Sa); t2 = tid(Sb);
    if (t1 != t2)
        P = completion_record(Sb)
        for each i in DepList(Sa)
            if P.DFN < A[i].DirLink
                A[i] = <i,P,P.0ldLink>
                if state[t2].completing
                    state[t2].completing = false; WAKE(t2);
                    push_completion_frame(Sb)
```

**Figure 2:** **NEW SUBGOAL** operation: code added for concurrent completion

**The completion operation** The **completion** operation, for subgoal $S$, in thread $T_0$ has to deal with three situations (see Figure 3):

1. It is known that $S$ is not a leader, $MinLink(S) \neq DFN(S)$. All that is needed is to propagate the dependency information, as in the sequential case. However, in this case the **PropagateDependencies** code has to deal with the dependency list, instead of just the DirLink value.

2. $S$ is a local leader. It schedules the answers (using the check fixpoint operation, which now has to adjust the state[$T_0$].leader flag as in the sequential case, with the difference that attention must be paid to the leader attribute for this thread in the state table structure.
if state[T0].completing /* thread was awakened for completion */
  complete_top(T0)
else
  if MinLink(S) < DFN(S) /* S is not local Leader */
    PropagateDependencies(S);
    FAIL
  /* S is local leader */
  if !check_fixpoint() /* There are answers to return */
    FAIL /* return the answers within the thread */
    check_dependencies();
  if NewDependencies /* New inter-thread dependencies added */
    for i in DepList(S)
      state[i].completing = false; WAKE(i);
      BLOCK(T0);
  if !GlobalLeader
    state[T0].completing = true;
    BLOCK(T0); /* wait for other to become global leader */
  for each thread t in DepList(S) /* MT ASCC has completed */
    WAKE(t); /* wake up all its threads */
  complete_top(T0);

Figure 3: Code for concurrent completion instruction

3. S is the global leader, and will initiate the process of termination for the
MT-ASCC.

In the case there are no answers to return to the ASCC, within the current
thread, it may become a global leader. For this to happen every other thread in its
MT-ASCC must be completing and the MT-ASCC must be independent.

At this point, since the last COMPLETION and NEW SUGBOGAL operations, the
global dependency graph may have changed, due to actions of other threads. This
means that before becoming the global leader this thread will have to check for
further dependency propagation, into and out of itself. There might also be depen-
dencies that became irrelevant due to completion of ASCCs in other threads. Last
of all, meanwhile, new answers may have been generated for threads that where
completing. These operations are done by algorithm check dependencies given in
Figure 4, which basically runs the dependency list for the leader sub-goal, and has
to deal with 7 different situations, for each thread that T0 depends on (Ti):

- (1) Ti has finished evaluating its top goal and has exited.
- (2) Ti has completely evaluated its top ASCC (on which the completing thread
  was depending).

In this two cases the dependency can be safely discarded.

- (3) Ti’s top ASCC has been extended and its local leader has changed. The
dependency information on Ti has to be readjusted.
• (4) If $T_i$ is not completing, $T_0$ cannot be a global leader.

• (5) If $T_i$ was waiting for completion and meanwhile got new answers to consume, it has to be awakened, to backtrack for the new answers. In this case $T_0$ can’t become the global leader.

• (6) When a new dependency from $T_i$ into $T_0$ is detected; if this dependency is to an older subgoal than the current leader, the local dependency information has to be updated, and the current ASCC will be extended. In this case the current subgoal will no longer be the local leader.

• (7) When an inter-thread dependency from thread $T_i$ into thread $T_j$ is detected, such that there’s no dependency from $T_0$ to $T_j$, the dependency on $T_j$ must be copied to $T_0$. In this case the threads on which $T_0$ depends on will have to be reawakened for further propagation of these dependencies.

The global leader just signals all the threads that are blocked on completion on this ASCC, so that they complete their top ASCC and are reawakened. Each thread is responsible for the local operations involved in completing its subgoals.

```
GlobalLeader = true ; NewDependencies = false ;
for each i in DepList(S)
(1) if !state[i].live /* Thread i has finished */
   delete S[i] ; continue ;
   ptr = state[i].leader ;
(2) if OldLink(ptr) > S[i].OldLink /* ASCC on thread i completed */
   delete S[i] ; continue ;
(3) if MinLink(ptr) > S[i].DirLink /* Thread i’s leader has changed */
   S[i].DirLink = ptr->DFN ;
   S[i].OldLink = OldLink(ptr);
(4) if !state[i].completing
   GlobalLeader = false ;
(5) else for each subgoal Si in the top ASCC of i
   if Si has answers to consume
      state[i].completing = false ; WAKE(i) ;
      GlobalLeader = false ;
(6) if ptr[i].pointer < completion_record(S)
   OldLink(S) = DFN(ptr[i].pointer) ;
   PropagateDependencies(S) ;
   FAIL ; /* There is a new leader to this Thread */
(7) if <j,P,N> in ptr[i].pointer->DepList and
   not <j,_,_> in S.DepList
   insert <j,P,N> in S.DepList
   NewDependencies = true ;
```

Figure 4: Algorithm check dependencies

When each thread is awakened, two situations might occur: (a) either new answers have been returned to it, or new dependencies have been introduced; or (b) the
global leader has determined a fixpoint for the multi-threaded ASCC. The thread will determine which case has happened by the state of its completion flag. In case (a) the completion operation will be restarted for each thread. In case (b) each thread executes the complete top algorithm (see Figure 5). This algorithm is responsible for releasing the memory of the completed subgoals and updating the table structures. It also has to deal with the trapped MT-ASCC situation; if an MT-ASCC has been trapped by this thread (it is a consumer for some subgoal that has been completed) it has to be reawakened.

for each Sugoal S on top of leader[i]
  for each consumer subgoal (CS) of S
    WAKE(TID(CS))
  top of completion stack = prev(leader[i])
  state[i].completing = false;
  state[i].leader = top of completion stack;

Figure 5: The complete top algorithm for thread i


5 Discussion

While the above sketch of a multi-threaded tabling architecture is quite abstract, several points should become clear. First, in the case of sequential execution as well as in the execution of fully independent threads, the incremental completion algorithm is essentially the same as in the sequential SLG-WAM. The architecture proposed here is not the first for a multi-threaded or shared-memory parallel engine. In [5] an architecture that has a single completion table and a sophisticated completion algorithm is presented, aiming for parallel execution of tabled subgoals as a means to improve the performance of the execution of logic programs. Compared to the architecture of [5], our architecture is optimized for cases that do not have any cross-thread dependencies. Furthermore, the model of [5] maintains an explicit dependency graph for all tabled subgoals. Our completion algorithm and framework can be used as a basis for table-parallelism (as defined in [5]). For that purpose refinements can be made to the completion algorithm to eliminate the need of mutual exclusion among different threads that execute the completion instruction, using techniques similar to the ones described in [5].

Yet another approach is that pursued by the YAP group [11] in which tabling is performed within an Or-parallel engine. In this approach, a tabled evaluation can be performed by different workers at different times, and as a result mutually dependent threads can migrate so that they are performed by a single worker, avoiding the problem of completion detection among different threads. This approach allows a much finer grain parallelism (clause level for any predicate) than table parallelism. The advantage of this approach as compared with the one presented here is still an open question.
References

queries under the well-founded semantics. *J. Logic Programming*, 24(3):161–
199, September 1995.


*Proceedings of ISMM’98: ACM SIGPLAN International Symposium on Memo-

In *Practical Aspects of Declarative Languages: First International Workshop*,
1999.


logic programs through alternative scheduling strategies. *Journal of Functional


[9] Portable Operating System Interface (POSIX) — part 1: System application:

access mechanisms for tabled logic programs. *Journal of Logic Programming*,

In *LNCS 1551, Workshop on Practical Aspects of Declarative Languages

[12] K. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-
