An Architecture for a Multi-threaded Tabled Engine
(Extended Abstract)

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 [12] 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 [4, 9] or distributed memory setting [6]. The latter is a most interesting approach as it can make possible for exploiting resources available to multi-computers and it enables the possibility for the execution of tabled programs across geographically distributed environments. The full version of this paper, under preparation, presents an architecture for a multi-threaded TLP engine that can form a basis for the distributed tabling agents of [6], and integrates the concept of a multi-threaded tabled server. It also discusses its correctness and complexity. In this way the architecture allows to exploit a distributed memory platform and it also facilitates a shared memory implementation within the tabled server. The main design principles of our approach 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.

Next, for space efficiency, the separate threads should share information whenever possible.

Principle II A subgoal is created only once, and new consumers will share it’s generator even if it is on a different thread.

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.

The main innovation of our architecture is the design of a completion detection algorithm that spans independent threads.

In this abstract 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.

2 Design Issues in Single-threaded 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.

†Faculdade de Ciências e Tecnologia, Univ. Nova de Lisboa, 2825-114 Caparica, Portugal. rfm@di.fct.unl.pt
†Dept. of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794, U.S.A tswift@cs.sunysb.edu
‡Faculdade de Ciências e Tecnologia, Univ. Nova de Lisboa, 2825-114 Caparica, Portugal. jcc@di.fct.unl.pt
Memory Management  Computing tables 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 [10], 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 providing for copying of trail and choice point stacks and sharing of heap and local stacks - the CHAT memory model of [3].

The weakness of memory sharing for tabled evaluations, then, is that they may trap unneeded information in frozen areas of memory. Of course unneeded information can be freed by using garbage collection algorithms to reclaim trapped choice points, environments and so on, but so far a garbage collector suitable for tabled logic programs has been implemented only for heap information [3]. To date, memory sharing has been used more for tabling systems. The current version of XSB shares all stacks except the choice point stack, while YAP [9] 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 operations are well supported by tries. Trie-based data structures for variant tabling were described in [8].

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 [5]. 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 not $S$ can be resolved away from calling environments; while 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 this approach is computationally expensive. Since the size of the dependency graph is in worst case the size of the program, such an approach would have cost $|\text{Atoms}_P| \times |\text{size}(P)|$ for ground programs where $|\text{Atoms}_P|$ is the number of atoms in a program, and $|\text{size}(P)|$ is the program size.

To address this, [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).
Briefly, each subgoal is given a unique depth-first-number (DFN), and for each subgoal the minimal depthfirst number of all subgoals on which it depends is also maintained. Dependency information, as abstracted through minimal DFNs, is then propagated among subgoals in an efficient manner. Answer scheduling and completion detection are handled in the SLG-WAM through the completion instruction of independent ASCCs. A subgoal \( S_L \) is called a leader if it can be determined that it has no descendants that call any subgoal whose DFN is less than that of \( S_L \). An independent ASCC is defined as the set of subgoals on the completion stack that are younger than the youngest leader. Scheduling of answers is handled by a routine in the completion instruction called check_fspoint [10] which checks, for each subgoal in the ASCC, whether all available answers have been returned to each environment in which that subgoal was selected, and schedules them for return if not. It also ensures that after the answers have been returned to the calling environments, the evaluation will fail back to the completion instruction for the leader, repeating check_fspoint until all answers have been returned. When they have, the evaluation checks the negative contexts in which the subgoals have been called as described below. Once all answers have been returned, and it is determined that none of the subgoals in the ASCC are selected as a negative literal in an active calling environment, it is safe to complete the subgoals to mark them as completely evaluated. In [10] it is shown that the approximation involved in independent ASCCs reflects the allocation of memory in shared stacks. Thus, when an independent ASCC is determined to be completed, shared stack space for that ASCC can be popped off of execution stacks.

**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 [10] (such a check is not needed with local evaluation). Specifically, negation handling uses a dependency check to determine whether an independent ASCC 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 [11].

### 3 Design Issues for a Multi-Threaded Tabling Engine

In keeping with Principle 1, 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. For simplicity, in this section we present a co-routined execution framework for the execution of this threads; however as long as mutual exclusion is ensured for the new_subgoal and completion operations this is irrelevant for the completion algorithm.

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 \( T_1 \) 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, \( T_1 \) 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 three scheduling queues: an I/O queue, an answer queue, and a ready queue. Threads are put onto the I/O when they suspend for I/O; onto the answer queue when waiting for answers (see below). Threads move from the I/O queue to the ready queue when an I/O operation is finished, and from the answer queue to the ready queue as answers are derived for subgoals on the thread. The low level implementation of suspension and resumption for threads is based on the suspend and resume mechanisms of sequential tabled evaluation. Threads can be suspended during predicate call, or during the completion instruction. In the case when suspension occurs during a call instruction, an SLG-WAM resume instruction is placed on the top of the choice point stack which restores
the calling environment when it is failed into. Threads are re-invoked by executing a fail instruction, giving a simple and uniform method to restore registers and execution state.

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 would work for the multi-threaded engine as for the single threaded engine. This simple approach was implemented in a meta-interpreter in [6] 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 keep this approach in a parallel implementation (keeping with principle III.)

Table Access Methods For the most part, the design of table access methods is orthogonal to multi-threading. As long as mutual exclusion is ensured, table access methods do not need to be changed at all.

Scheduling Strategies within a Thread Issues of scheduling for returning answers with 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. However, the advantages of scheduling strategies may differ in a multi-threading environment than in a sequential 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 producing 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, [5] 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 with actual implementations.

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 check is straightforward to implement although it requires synchronization among threads.

Incremental Completion We now sketch our algorithm for incremental completion.

Each thread contains its own completion stack, in shared memory, and all threads share a global DFN counter. Rather than having a single minlink, as in the sequential case, each completion stack frame contains a list of minlink values, one for each thread upon which there is a dependency from this subgoal. Using these minlink values and DFNs global ASCCs and leaders can be defined in essentially the same manner as in the sequential case, so that the ASCC represents an over-approximation of dependencies between subgoals while the leader of an ASCC is the oldest subgoal within the SCC. Given an independent ASCC \( A \), and thread \( T \), the local leader of \( A \) for thread \( T \) is defined as the oldest subgoal frame in \( A \) that is also in \( T \).

As in the sequential engine, minlink values are updated or created upon calling a tabled subgoal. However, when a thread calls an incomplete tagged subgoal that has been started by another thread, a special calling environment frame is set onto the completion stack. To understand the importance of a calling environment frame, consider that regular completion stack frames are associated with subgoals for which
program clause resolution is to be executed. Only when all program clauses have been executed can dependencies be propagated (modulo keeping lists of minlink values rather than a single minlink value; this occurs the same way in the sequential and in the multi-threaded engine) and completion checked. A calling environment frame is associated with a subgoal whose program clause resolution is controlled by a separate thread. Such a frame can be thought of as immediately executing a completion instruction, to ensure that answers are returned in the local thread or to propagate dependency information as the case may be. When executing a completion instruction, a local leader (whether a calling environment frame or a standard frame), checks control flags to see if the global leader has told it to delay selected literals in calling environments, or whether the ASCC can be completed. If no flags are set, it then executes a check_fixpoint routine for each calling environment of the ASCC that is in its thread. The return of answers to calling environments in another thread will be performed when that thread executes a completion instruction for its own local leader.

When a completion instruction is executed for a global leader, it acts first as a local leader by executing a check_fixpoint routine. Next it checks that each of the other threads on which it depends is blocked and waiting for answers and sets a lock on each of these threads to ensure that the thread cannot be woken while the global leader performs a global completion check. The global leader then checks to see that all answers have been returned to all consuming environments in each thread. If not, it releases all locks and fails, but if all answers have been returned, it checks to see whether delaying is needed, or whether the subgoals in each thread can be completed and the space disposed. Both delaying and completion are communicated between threads by control flags in the completion stacks. The global leader finally releases all locks, and when each of the other threads suspends, they check their flags and take the appropriate actions.

4 Discussion

The above sketch of a multi-threaded tabling architecture, based on the SLG-WAM leaves out many details, but several points should become clear. First, in the case of sequential execution, or multi-threaded execution where one thread does not consume answers from an incomplete table, whose generator choice point has been laid by another thread, the incremental completion algorithm is the same as in the present SLG-WAM. Furthermore, if local evaluation is chosen, the only synchronization required between threads occurs during the fixpoint check by a global leader for an ASCC that spans threads, indicating that the model is a suitable basis for a model of an engine in which tables are computed in parallel.

The architecture proposed here is not the first for a multi-threaded or shared-memory parallel engine. An architecture is suggested in [4] that has a single completion stack and a sophisticated completion algorithm, 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 [4], our architecture is optimized for cases that do not have any cross-thread dependencies. Also in [4] there's no provision for multiple tabled subgoals sharing the same stack and thus the overhead of keeping inter-thread dependencies is incurred for any tabled subgoal. This overhead is much larger than in our algorithm, as an explicit dependency graph is maintained for all tabled subgoals.

Our completion algorithm and framework can be used as a basis for table-parallelism (as defined in [4].) 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.

Yet another approach is that pursued by the YAP group [9] in which tabling is performed within an Or-parallel engine. In this approach, a table 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


