A Tabling Engine Designed to Support Mixed-Strategy Evaluation

Ricardo Rocha Fernando Silva
DCC-FC & LIACC
Universidade do Porto, Portugal
{ricroc,fds}@ncc.up.pt

Vítor Santos Costa
COPPE Systems & LIACC
Universidade do Rio de Janeiro, Brasil
vitor@cos.ufrj.br

Abstract

Tabling is an implementation technique that improves the declarativeness and expressiveness of Prolog by reusing answers to subgoals. During tabled execution, there are several points where different operations can be applied. The decision on which operation to perform is determined by the scheduling strategy. Whereas a strategy can achieve very good performance for certain applications, for others it might add overheads and even lead to unacceptable inefficiency. The ability of using multiple strategies within the same evaluation can be a means of achieving the best possible performance. In this work, we present how the YapTab system was designed to support the two most successful tabling scheduling strategies: batched and local scheduling; and how it can be easily extended to support simultaneous mixed-strategy evaluation.

1 Introduction

The past years have seen wide effort at increasing Prolog's declarativeness and expressiveness. One such proposal that has been gaining in popularity is the use of *tabling* or *tabulation* or *memoing*. Work on SLG resolution [2], as implemented in the XSB logic programming system [1], proved the viability of tabling technology for application areas such as Natural Language Processing, Knowledge Based Systems, Model Checking, and Program Analysis. Tabling based models are able to reduce the search space, avoid looping, and have better termination properties than SLD based models.

The basic idea behind tabling is straightforward: programs are evaluated by storing answers of current subgoals in an appropriate data space, called the *table space*. The method then uses the table to verify whether calls to subgoals are repeated. Whenever such a repeated call is found, the subgoal's answers are recalled from the table instead of being re-evaluated against the program clauses.

During tabled execution, there are several points where we had to choose between continuing forward execution, backtracking, consuming answers from the table, or completing subgoals. The decision on which operation to perform is crucial to system performance and is determined by the *scheduling strategy*. Different strategies may have a significant impact on performance, and may lead to different order of solutions to the query goal. Arguably, the two most successful tabling scheduling strategies are *batched scheduling* and *local scheduling* [6].

Batched scheduling favors forward execution first, backtracking next, and consuming answers or completion last. It thus tries to delay the need to move around the search tree by *batching* the return of answers. When new answers are found for a particular tabled subgoal, they are added to the table space and the evaluation continues. On the other hand, local scheduling tries to complete

subgoals sooner. When new answers are found, they are added to the table space and the evaluation fails. Answers are only returned when all program clauses for the subgoal in hand were resolved.

Empirical work from Freire *et al.* [6, 7] showed that, regarding the requirements of an application, the choice of the scheduling strategy can differently affect the memory usage, execution time and disk access patterns. Freire argues [5] that there is no single best scheduling strategy, and whereas a strategy can achieve very good performance for certain applications, for others it might add overheads and even lead to unacceptable inefficiency. As a means of achieving the best possible performance, Freire and Warren [8] proposed the ability of using multiple strategies within the same evaluation, by supporting mixed-strategy evaluation at the predicate level. However, to the best of our knowledge, no such implementation has yet been done.

In this work, we present how YapTab [10] was designed to support batched and local scheduling independently and how it can be easily extended to support simultaneous mixed-strategy evaluation. YapTab is a sequential tabling engine that extends Yap's execution model [12] to support tabled evaluation for definite programs. YapTab's implementation is largely based on the ground-breaking design of the XSB system [1], which implements the SLG-WAM [11].

The remainder of the paper is organized as follows. First, we briefly introduce the basic tabling definitions and discuss the differences between batched and local scheduling. We then present the support actually implemented in YapTab to deal with both scheduling strategies and discuss and it can be extended to support mixed-strategy evaluation.

2 Basic Tabling Definitions

Tabling is about storing intermediate answers for subgoals so that they can be reused when a repeated subgoal appears. Whenever a tabled subgoal S is first called, an entry for S is allocated in the *table space*. This entry will collect all the answers found for S. Repeated calls to *variants* of S are resolved by consuming the answers already stored in the table. Meanwhile, as new answers are found, they are stored into the table and returned to all variant subgoals. Within this model, the nodes in the search space are classified as either: *generator nodes*, corresponding to first calls to tabled subgoals; *consumer nodes*, corresponding to variant calls to tabled subgoals; or *interior nodes*, corresponding to non-tabled subgoals.

Tabling based evaluation has four main types of operations for definite programs: entering a tabled subgoal; adding a new answer to a generator; exporting an answer from the table; and trying to complete a subgoal. In more detail:

1. The *tabled subgoal call* operation is a call to a tabled subgoal. It checks if the subgoal is in the table, and if not, adds a new entry for it and allocates a new generator node. Otherwise, it allocates a consumer node and starts consuming the available answers.

2. The *new answer* operation returns a new answer to a generator. It verifies whether a newly generated answer is already in the table, and if not, inserts it. Otherwise, it fails.

3. The *answer resolution* operation is executed every time the computation reaches a consumer node. It verifies whether newly found answers are available for the particular consumer node and, if any, consumes the next one. Answers are consumed in the same order they are inserted in the table. Otherwise, it *suspends* the current computation, either by freezing the whole stacks [11], or by copying the execution stacks to separate storage [4], and schedules a possible resolution to continue the execution.

4. The *completion* operation determines whether a tabled subgoal is *completely evaluated*. A subgoal is said to be completely evaluated when all its possible resolutions have been performed, that is, when no more answers can be generated and the variant subgoals have consumed all the available answers. It executes when we backtrack to a generator node and all of its clauses have been tried. If the subgoal has been completely evaluated, the operation closes its table entry and reclaims space. Otherwise, it resumes one of the consumers with unconsumed answers.

Completion is needed in order to recover space and to support negation. We are most interested on space recovery in this work. Arguably, in this case we could delay completion until the very end of execution. Unfortunately, doing so would also mean that we could only recover space for consumers (suspended subgoals) at the very end of the execution. Instead we shall try to achieve *incremental completion* [3] to detect whether a generator node has been fully exploited, and if so to recover space for all its consumers.

Completion is hard because a number of generators may be mutually dependent, thus forming a *Strongly Connected Component* (or *SCC*). Clearly, we can only complete SCCs together. We will usually represent an SCC through the oldest generator. More precisely, the youngest generator node which does not depend on older generators is called the *leader node*. A leader node is also the oldest node for its SCC, and defines the current completion point.

3 Scheduling Strategies

At several points we had to choose between continuing forward execution, backtracking to interior nodes, returning answers to consumer nodes, or performing completion. The actual sequence of operations thus depends on the scheduling strategy. We next discuss in some more detail batched and local scheduling.

3.1 Batched Scheduling

Batched scheduling takes its name because it tries to minimize the need to move around the search tree by *batching* the return of answers. When new answers are found for a particular tabled subgoal, they are added to the table space and the evaluation continues until it resolves all program clauses for the subgoal in hand. Only then the newly found answers will be returned to consumer nodes.

Batched scheduling schedules the program clauses in a depth-first manner as does the WAM. Calls to non-tabled subgoals allocate interior nodes. First calls to tabled subgoals allocate generator nodes and variant calls allocate consumer nodes. However, if we call a variant tabled subgoal, and the correspondent subgoal is already completed, we can avoid consumer node allocation and instead perform what is called a *completed table optimization* [11]. This optimization allocates a node, similar to an interior node, that will consume the set of found answers executing compiled code directly from the table data structures associated with the completed subgoal [9].

When backtracking we may encounter three situations: **(i)** if backtracking to a generator or interior node, we take the next available alternative; **(ii)** if backtracking to a consumer node, we take the next unconsumed answer; **(iii)** if there are no available alternatives or no unconsumed answers, we simply backtrack to the previous node on the current branch. Note however that, if the node without alternatives is a leader generator node, then we must check for completion.

In order to perform completion, we must ensure that all answers have been returned to all consumers in the SCC. The process of resuming a consumer node, consuming the available set of

answers, suspending and then resuming another consumer node can be seen as an iterative process which repeats until a fixpoint is reached. This fixpoint is reached when the SCC is completely evaluated.

At engine level, the *fixpoint check procedure* is controlled by the leader of the SCC. The procedure traverses the consumer nodes in the SCC in a bottom-up manner to determine whether the SCC has been completely evaluated or whether further answers need to be consumed. Initially, it searches for the bottom consumer node with unresolved answers, and as long as there are available answers, it will consume them. After consuming the available set of answers, the consumer suspends and fails into the next consumer with unresolved answers. This process repeats until it reaches the last consumer node, in which case it fails into the leader node in order to allow the re-execution of the fixpoint check procedure. When a fixpoint is reached, all subgoals in the SCC are marked completed and the stack segments belonging to the completed subtree are released.

3.2 Local Scheduling

Local scheduling is an alternative tabling scheduling strategy that tries to complete subgoals sooner. Evaluation is done one SCC at a time, and answers are returned outside of a SCC only after that SCC is completely evaluated. When new answers are found, they are added to the table space and the evaluation fails. Answers are only returned when all program clauses for the subgoal in hand were resolved. We next present in Fig. 1 a small example that clarifies the differences between batched and local evaluation.

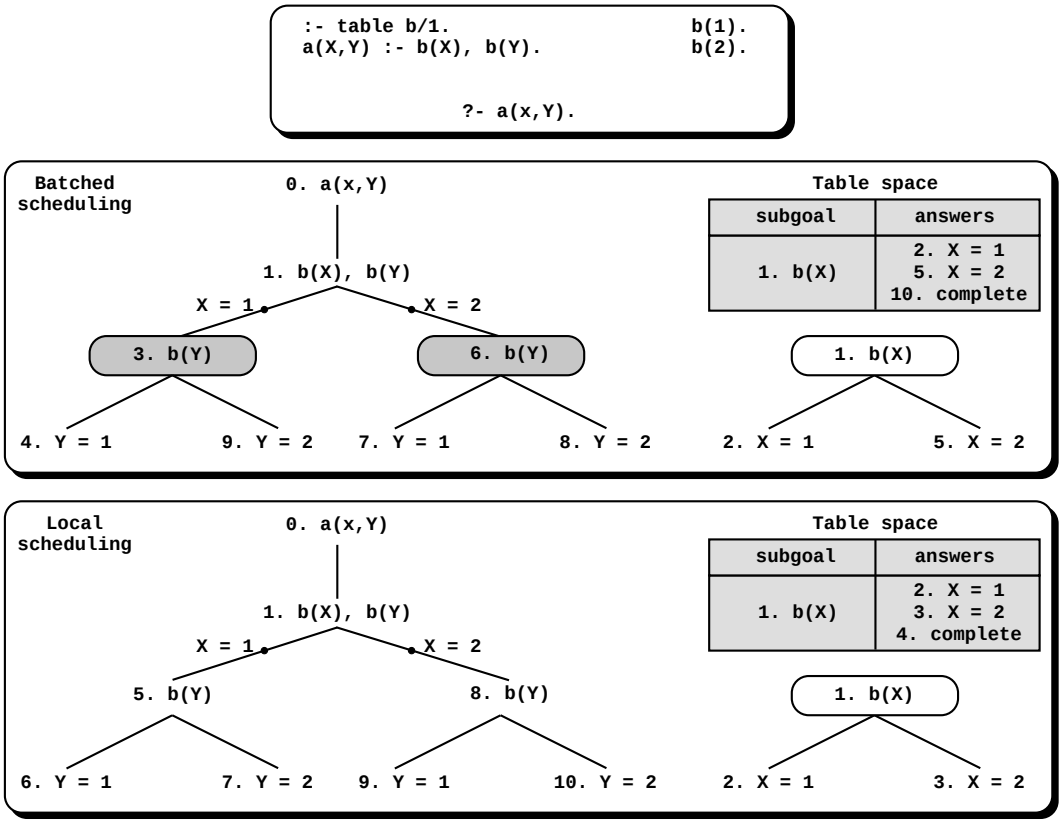

Figure 1: Batched versus local scheduling

At the top, the figure illustrates the program code and the query goal used in both evaluations. Declaration `:- table b/1` in the program code indicates that calls to predicate `b/1` should be tabled. The two sub-figures below depict the evaluation sequence for each scheduling strategy, which includes the resulting table space and forest of trees. The numbering of nodes denotes the evaluation sequence. The leftmost tree represents the original invocation of the query goal `a(X,Y)`. As we shall see, computing `a(X,Y)` requires computing `b(X)`. For simplicity of presentation, the computation tree for `b(X)` is represented independently at the right. We next describe in more detail the two evaluations.

In both cases, the evaluation begins by resolving the query goal against the unique clause for predicate `a/2`, thus calling the tabled subgoal `b(X)`. As this is the first call to `b(X)`, we create a generator node (generators are depicted by white oval boxes) and insert a new entry in the table space for it. The first clause for `b(X)` immediately succeeds, obtaining a first answer for `b(X)` that is stored in the table (step 2). The interesting aspect that results from the figure, is how both strategies handle the continuation of the evaluation of `b(X)`.

For batched scheduling, the evaluation proceeds executing as in standard Prolog with the continuation call `b(Y)`, therefore creating consumer node 3 (consumers are depicted by gray oval boxes). Node 3 is a variant call to `b(X)`, so instead of resolving the call against the program clauses, we consume answers from the table space. As we already have one answer stored in the table for this call (`X=1`), we continue by consuming the available answer, which leads to a first solution for the query goal (`X=1;Y=1`). When returning to node 3, we must suspend the consumer node because there are no more answers for it in the table. We then backtrack to node 1 to try the second clause for `b(X)`, and a new answer is found (`X=2`). In the continuation, a new consumer is created (node 6) and two new solutions are found for the query goal (steps 7 and 8). Node 6 is then suspended and the computation backtracks again to node 1. At that point, we can check for completion. However, the generator cannot complete because consumer 3 has unconsumed answers. The computation is then resumed at node 3 and a new solution for the query goal is found (step 9). When returning to the generator node 1, we can finally complete the tabled subgoal call `b(X)` (step 10).

On the other hand, for local scheduling, the evaluation fails back after the first answer was found (step 2) in order to find the complete set of answers for `b(X)` and therefore complete before returning answers to the calling environment. We thus backtrack to node 1, execute the second clause for `b(X)`, and find a second answer for it (step 3). Then, we fail again to node 1, and the tabled subgoal call `b(X)` can be completed (step 4). The two found answers are consumed next by executing compiled code directly from the table structure associated with the completed subgoal `b(X)`. The variant calls to `b(X)` at steps 5 and 8 are also resolved by executing compiled code from the table.

In batched scheduling, when a new answer is found, variable bindings are automatically propagated to the calling environment. For some situations, this behavior may result in creating complex dependencies between consumers. On the other hand, the clear advantage of local scheduling shown in the example does not always hold. Since local scheduling delays answers, it does not benefit from variable propagation, and instead, when explicitly returning the delayed answers, it incurs an extra overhead for copying them out of the table. Local scheduling does perform arbitrarily better than batched scheduling for applications that benefit from answer subsumption, that is, where we delete non-minimal answers every time a new answer is added to the table. On the other hand, Freire *et al.* [6] showed that, on average, local scheduling is about 15% slower than batched scheduling in the SLG-WAM. Similar results were also obtained for batched and local scheduling in YapTab [10].

4 Implementation

We next give a brief introduction to the implementation of YapTab. Throughout, we focus on the support for the two tabling scheduling strategies.

The YapTab design is very close to the original SLG-WAM [11]: it introduces a new data area, the *table space*; a new set of registers, the *freeze registers*; an extension of the standard trail, the *forward trail*; and four new operations: *tabled subgoal call*, *new answer*, *answer resolution*, and *completion*. The substantial differences between the two designs reside in the data structures and algorithms used to control the process of leader detection and scheduling of unconsumed answers. The SLG-WAM considers that such control should be done at the level of the data structures corresponding to first calls to tabled subgoals, and it does so by associating *completion frames* to generator nodes. It uses a *completion stack* of generators to detect completion points. Essentially, the *completion stack* stores information about the generator nodes and the dependencies between them. Each time a new generator is introduced it becomes the current leader node. Each time a new consumer is introduced one verifies if it is for an older generator node $\mathcal{G}$. If so, $\mathcal{G}$'s leader node becomes the current leader node.

On the other hand, YapTab innovates by considering that the control of leader detection and scheduling of unconsumed answers should be performed through the data structures corresponding to variant calls to tabled subgoals, and it associates a new data structure, the *dependency frame*, to consumer nodes. We believe that managing dependencies at the level of the consumer nodes is a more intuitive approach that we can take advantage of. The new data structure allows us to eliminate the need for a separate completion stack and to slightly improve the fixpoint check procedure. In the SLG-WAM, each step of the fixpoint check procedure is done by traversing the consumers in a SCC by groups, with each group corresponding to consumers for a common variant subgoal. YapTab simplifies by considering the whole set of consumers within a SCC as a single group, and it thus traverses the whole set in a single pass.

4.1 Table Space

The table space can be accessed in different ways: to look up if a subgoal is in the table, and if not insert it; to verify whether a newly found answer is in the table, and if not insert it; to forward answers to consumer nodes; and to mark subgoals as completed. Hence, a correct design of the algorithms to access and manipulate the table is a critical issue to obtain an efficient implementation. Our implementation uses tries as the basis for tables, as proposed by Ramakrishnan *et al.* [9]. Tries provides complete discrimination for terms and permits lookup and possibly insertion to be performed in a single pass through a term.

Figure 2 shows the general table structure for a tabled predicate. Table lookup starts from the *table entry* data structure. Each table predicate has one such structure, which is allocated at compilation time. A pointer to the table entry can thus be included in the compiled code. Calls to the predicate will always access the table starting from this point.

The table entry points to a tree of trie nodes, the *subgoal trie structure*. More precisely, each different call to the tabled predicate in hand corresponds to a unique path through the subgoal trie structure. Such a path always starts from the table entry, follows a sequence of subgoal trie data units, the *subgoal trie nodes*, and terminates at a leaf data structure, the *subgoal frame*.

Each subgoal frame stores information about the subgoal, namely an entry point to its *answer trie structure*. Each unique path through the answer trie data units, the *answer trie nodes*, corresponds to a different answer to the entry subgoal. All answer leave nodes are chained together in insertion time order in a linked list, so that we can recover answers in the same order they were

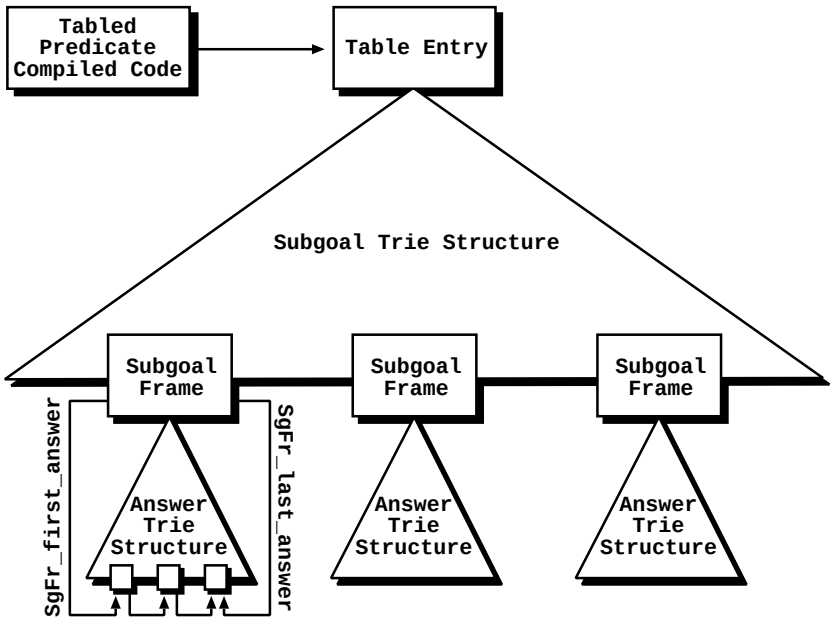

Figure 2: Using tries to organize the table space

inserted. The subgoal frame points at the first and last entry in this list. A consumer node thus needs only to point at the leaf node for its last consumed answer, and consumes more answers just by following the chain of leaves.

4.2 Tabled Nodes

In YapTab, applying batched or local scheduling to a tabled evaluation only depends on the way generators are implemented. All the other tabling extensions are commonly used for both strategies without any modifications. As we shall see, this makes YapTab highly suitable to support mixed-strategy evaluation.

Remember that interior nodes are implemented as WAM choice points [13]: the `CP_TR`, `CP_H`, `CP_B`, `CP_CP`, `CP_AP` and `CP_ENV` choice point fields are used to store at choice point creation, respectively, the top of trail; top of global stack; failure continuation choice point; success continuation program counter; choice point next alternative; and current environment. Generator and consumer nodes are implemented as WAM choice points extended with some extra fields to control tabling execution.

To implement consumer nodes we extended the WAM choice points with the dependency frame data structure. Dependency frames store the last consumed answer for the correspondent consumer node; and information to efficiently check for completion points, and to efficiently move across the consumer nodes with unconsumed answers.

To prevent answers from being returned to the calling environment of a generator node, after a new answer is found for a particular tabled subgoal, local scheduling fails and backtracks in order to search for the complete set of answers. These answers are consumed later when all program clauses for the subgoal in hand were resolved. Therefore, when backtracking to a generator node without alternatives, we must also act like a consumer node to consume the set of found answers. Thus, for local scheduling, generator choice points are also extended with dependency frames. For batched scheduling we only need to access the subgoal frame where answers should be stored, so

generators are implemented as WAM choice points extended with a pointer to the corresponding subgoal frame, the CP_SgFr field. Figure 3 illustrates how consumers and generators are differently handled to support batched and local scheduling.

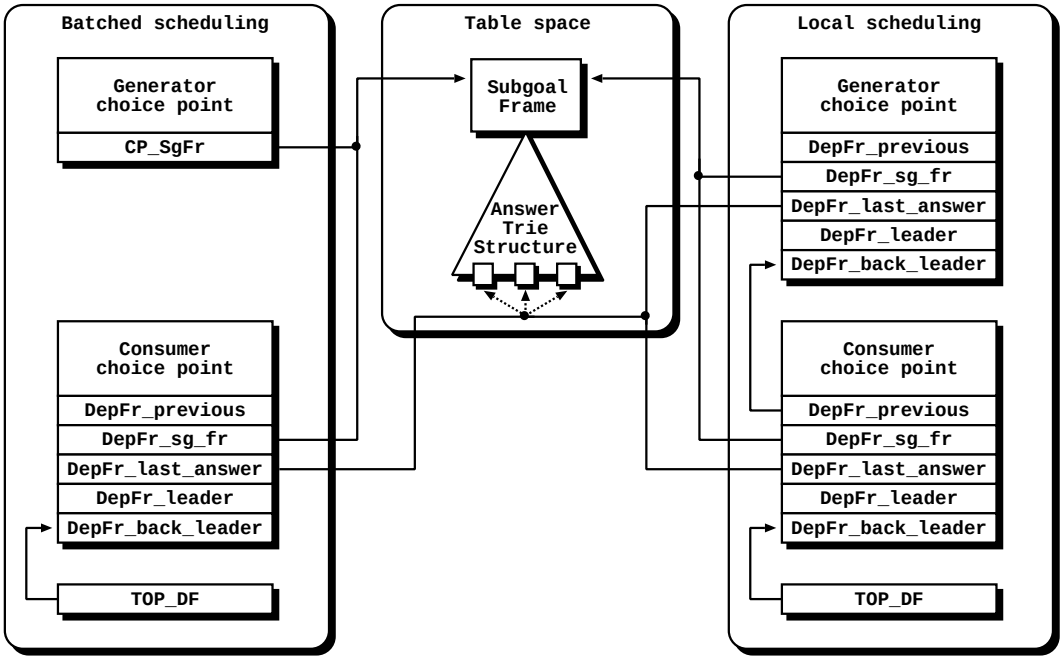

Figure 3: Consumers and generators with batched and local scheduling

Each dependency frame is a five field data structure. The DepFr_previous is a pointer to the previous dependency frame on stack and it allows to form a list of dependency frames on stack. A global TOP_DF variable points to the youngest dependency frame on stack. The DepFr_sg_fr and the DepFr_last_answer are pointers respectively to the correspondent subgoal frame and to the last consumed answer, and they are used to connect choice points with the table space in order to search for and to pick up new answers. Moreover, for local scheduling, we use the DepFr_sg_fr field of the dependency frame to access the correspondent subgoal frame. For batched scheduling, we use the new CP_SgFr choice point field. The DepFr_leader and the DepFr_back_leader are pointers respectively to the leader node at creation time and to the leader node where we perform the most recent unsuccessful completion operation, and they are used to support the fixpoint check procedure. The dependency frame DepFr_leader field is initialized by the compute_leader() procedure, whilst the DepFr_back_leader field is initialized with a NULL value. Their use is detailed next.

4.3 Answer Resolution

The answer resolution operation should be executed every time the computation fails back to a consumer. To achieve this, when a new consumer choice point is allocated, its CP_AP field is made to point to the answer_resolution instruction. Figure 4 shows the pseudo-code for it.

Initially, the procedure checks the table space for unconsumed answers. If there are new answers, it loads the next available answer and proceeds. Otherwise, it schedules for a backtracking node. If this is the first time that backtracking from that consumer node takes place, then it is performed as usual to the previous node. This is the case when the DepFr_back_leader field is NULL. Otherwise, we know that the computation has been resumed from an older leader node $\mathcal{L}$ during an unsuccessful

```
answer_resolution (consumer node CN) {
  DF = dependency_frame_for(CN)
  if (DepFr_last_answer(DF) != SgFr_last_answer(DepFr_sg_fr(DF)))
    load_next_unconsumed_answer_and_proceed()
  back_cp = DepFr_back_leader(DF)
  if (back_cp == NULL)
    backtrack()
  df = DepFr_previous(DF)
  while (consumer_for(df) is younger than back_cp) {
    if (DepFr_last_answer(df) != SgFr_last_answer(DepFr_sg_fr(df))) {
      // move to previous consumer with unconsumed answers
      DepFr_back_leader(df) = back_cp
      move_to(consumer_for(df))
    }
    df = DepFr_previous(df)
  }
  // move to older leader node
  move_to(back_cp)
}
```

Figure 4: Pseudo-code for `answer_resolution()`

completion operation. Therefore, backtracking must be done to the next consumer node that has unconsumed answers and that is younger than $\mathcal{L}$. We do this by restoring bindings and stack pointers. If no such consumer node can be found, backtracking must be done to node $\mathcal{L}$.

The process of resuming a consumer node, consuming the available set of answers, suspending and then resuming another consumer node can be seen as an iterative process which repeats until a fixpoint is reached. This fixpoint is reached when the SCC is completely evaluated.

4.4 Leader Nodes

The completion operation takes place when we backtrack to a generator node that **(i)** has exhausted all its alternatives and that **(ii)** is a leader node (remember that the youngest generator which does not depend on older generators is called a leader node). We designed novel algorithms to quickly determine whether a generator node is a leader node. The key idea in our algorithms is that each dependency frame holds a pointer to the resulting leader node of the SCC that includes the correspondent consumer node. Using the leader node pointer from the dependency frames, a generator can quickly determine whether it is a leader node. More precisely, a generator $\mathcal{L}$ is a leader node when either **(a)** $\mathcal{L}$ is the youngest tabled node, or **(b)** the youngest consumer says that $\mathcal{L}$ is the leader.

Our algorithm thus requires computing leader node information whenever creating a new consumer node $\mathcal{C}$. We proceed as follows. First, we hypothesize that the leader node is $\mathcal{C}$'s generator, say $\mathcal{G}$. Next, for all consumer nodes older than $\mathcal{C}$ and younger than $\mathcal{G}$, we check whether they depend on an older generator node. Consider that there is at least one such node and that the oldest of these nodes is $\mathcal{G}'$. If so then $\mathcal{G}'$ is the leader node. Otherwise, our hypothesis was correct and the leader node is indeed $\mathcal{G}$. Leader node information is implemented as a pointer to the choice point of the newly computed leader node. Figure 5 shows the procedure that computes the leader node information for a new consumer.

The procedure traverses the dependency frames for the consumer nodes between the new consumer and its generator in order to check for older dependencies. As an optimization it only searches until it finds the first dependency frame holding an older reference (the `DepFr_leader` field). The nature of the procedure ensures that the remaining dependency frames cannot hold older references.

For local scheduling, when we store a new generator node $\mathcal{G}$ we also allocate a dependency frame. As an optimization we can avoid calling `compute_leader()` to initialize the `DepFr_leader`

```
compute_leader (consumer node CN) {
  DF = dependency_frame_for(CN)
  leader_cp = generator_for(CN)
  df = TOP_DF
  while (consumer_for(df) is younger than leader_cp ) {
    if (leader_cp is equal or younger than DepFr_leader(df)) {
      // found older dependency
      leader_cp = DepFr_leader(df)
      break
    }
    df = DepFr_previous(df)
  }
  DepFr_leader(DF) = leader_cp
}
```

Figure 5: Pseudo-code for `compute_leader()`

field, because it will always compute $\mathcal{G}$ as the leader node.

4.5 Completion with Batched Scheduling

When a generator choice point tries the last program clause, its `CP_AP` field is updated to the `completion` instruction. Since then, every time we backtrack to the choice point the instruction gets executed. Figure 6 shows the pseudo-code that implements completion for batched scheduling.

```
completion (generator node GN) {
  if (GN is the current leader node) {
    df = TOP_DF
    while (consumer_for(df) is younger than GN)) {
      if (DepFr_last_answer(df) != SgFr_last_answer(DepFr_sg_fr(df))) {
        // move to first consumer with unconsumed answers
        DepFr_back_leader(df) = GN
        move_to(consumer_for(df))
      }
      df = DepFr_previous(df)
    }
    perform_completion()
  }
  backtrack()
}
```

Figure 6: Pseudo-code for `completion()` with batched scheduling

Initially, the procedure finds out if the generator is the current leader node. If not, it simply backtracks to the previous node. Being leader, it checks whether all younger consumer nodes have consumed all their answers. To do so, it walks the chain of dependency frames looking for a frame which has not yet consumed all the generated answers. If there is such a frame, the computation should be resume to the corresponding consumer node. Otherwise, it can perform completion. This includes **(i)** marking as complete all the subgoals in the SCC; **(ii)** deallocating all younger dependency frames; and **(iii)** backtracking to the previous node to continue the execution.

4.6 Completion with Local Scheduling

To implement completion for local scheduling, we only need to slightly change the previous procedure. Figure 7 shows the modified pseudo-code.

```
completion (generator node GN) {
  if (GN is the current leader node) {
    df = TOP_DF
    while (consumer_for(df) is younger than GN) {
      if (DepFr_last_answer(df) != SgFr_last_answer(DepFr_sg_fr(df))) {
        // move to first consumer with unconsumed answers
        DepFr_back_leader(df) = GN
        move_to(consumer_for(df))
      }
      df = DepFr_previous(df)
    }
    perform_completion()
    completed_table_optimization()                              // new
  }
  CP_AP(GN) = answer_resolution                                 // new
  load_first_unconsumed_answer_and_proceed()                    // new
}
```

Figure 7: Pseudo-code for `completion()` with local scheduling

There is a major change to the completion algorithm for local scheduling. As newly found answers cannot be immediately returned, we need to consume them at a later point. If we perform completion successfully, we start consuming the set of answers that have been found by executing compiled code directly from the trie data structure associated with the completed subgoal. Otherwise, we must act like a consumer node and start consuming answers.

5 Discussion

In result of its clear design based on the dependency frame data structure, YapTab already includes all the machinery required to support batched and local scheduling simultaneously. Extending YapTab to use multiple strategies at the predicate level is straightforward. Only two new features have to be addressed: **(i)** support strategy-specific Prolog declarations like '`:- batched path/2.`' in order to allow the user to define the strategy to be used to resolve the subgoals of a given predicate; **(ii)** at compile time generate appropriate tabling instructions, such as `batched_new_answer` or `local_completion`, accordingly to the declared strategy for the predicate. With these two simple compiler extensions we are able to use all the algorithms described and already implemented for batched and for local scheduling without any further modification.

The proposed data structures and algorithms can also be easily extended to support different strategies per predicate, that is, allow the user to define the strategy to be used to resolve each subgoal. Moreover, they can be extended to support dynamic switching from batched to local scheduling, while a generator is still producing new answers. However, further work is still needed to study if there is a use for such flexibility.

In this work we concentrated on the issues concerning the design and implementation of both strategies. Currently, we have already batched and local scheduling functioning separately in YapTab and we are now working on adjusting the system for mixed-strategy evaluation. After having the system implementing mixed-strategy evaluation we plan to use a set of common tabled benchmarks to investigate and study the impact of combining both strategies for tabled evaluation.

Acknowledgments

This work has been partially supported by APRIL (POSI/SRI/40749/2001), $CLoP^n$ (CNPq), PLAG (FAPERJ), and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSI.

References

[1] The XSB Logic Programming System. Available from `http://xsb.sourceforge.net`.

[2] W. Chen, M. Kifer, and D. S. Warren. Hilog: A Foundation for Higher-Order Logic Programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[3] W. Chen, T. Swift, and D. S. Warren. Efficient Top-Down Computation of Queries under the Well-Founded Semantics. *Journal of Logic Programming*, 24(3):161–199, 1995.

[4] B. Demoen and K. Sagonas. CHAT: The Copy-Hybrid Approach to Tabling. *Future Generation Computer Systems*, 16(7):809–830, 2000.

[5] J. Freire. *Scheduling Strategies for Evaluation of Recursive Queries over Memory and Disk-Resident Data*. PhD thesis, Department of Computer Science, State University of New York, 1997.

[6] J. Freire, T. Swift, and D. S. Warren. Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In *International Symposium on Programming Language Implementation and Logic Programming*, number 1140 in Lecture Notes in Computer Science, pages 243–258. Springer-Verlag, 1996.

[7] J. Freire, T. Swift, and D. S. Warren. Taking I/O seriously: Resolution reconsidered for disk. In *International Conference on Logic Programming*, pages 198–212, 1997.

[8] J. Freire and D. S. Warren. Combining Scheduling Strategies in Tabled Evaluation. In *Workshop on Parallelism and Implementation Technology for Logic Programming*, 1997.

[9] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Access Mechanisms for Tabled Logic Programs. *Journal of Logic Programming*, 38(1):31–54, 1999.

[10] R. Rocha, F. Silva, and V. Santos Costa. YapTab: A Tabling Engine Designed to Support Parallelism. In *Conference on Tabulation in Parsing and Deduction*, pages 77–87, 2000.

[11] K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, 1998.

[12] V. Santos Costa. Optimising Bytecode Emulation for Prolog. In *Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 261–267. Springer-Verlag, 1999.

[13] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.