Dynamic Mixed-Strategy Evaluation of Tabled Logic Programs

Ricardo Rocha[1], Fernando Silva[1], and Vítor Santos Costa[2]

[1] DCC-FC & LIACC
University of Porto, Portugal
{ricroc,fds}@ncc.up.pt
[2] COPPE Systems & LIACC
Federal University of Rio de Janeiro, Brazil
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, several decisions have to be made. These are 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 dynamic mixed-strategy evaluation of the two most successful tabling scheduling strategies: batched scheduling and local scheduling.

1 Introduction

The past years have seen wide efforts at increasing Prolog's declarativeness, expressiveness and performance. One proposal that has gained popularity is the use of *tabling* (also known as *tabulation* or *memoing*). Tabling based models are able to reduce the search space, avoid looping, and have better termination properties than traditional Prolog based models. Several alternative tabling models have been proposed and implemented [1–5]. The most well-known tabling Prolog system is XSB Prolog [6], which proved the viability of tabling technology in application areas such as Natural Language Processing, Knowledge Based Systems, Model Checking, and Program Analysis.

The basic idea behind tabling is straightforward: programs are evaluated by storing answers for current subgoals in an appropriate data space, called the *table space*. Whenever 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 may have 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 a different ordering of solutions to the query goal. Arguably, the two most successful tabling scheduling strategies are batched scheduling and local scheduling [7].

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 as soon as possible. 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.* [7, 8] showed that, regarding the requirements of an application, the choice of the scheduling strategy can affect the memory usage, execution time and disk access patterns. Freire argues [9] 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. Freire and Warren [10] suggested that using multiple strategies within the same evaluation would be most useful. However, to the best of our knowledge, no such implementation has yet been done.

Our main contribution is a novel approach to supporting dynamic mixed-strategy evaluation of tabled logic programs. We have implemented this approach in the YapTab system, as an elegant extension of the original design [2]. YapTab supports the dynamic intermixing of batched and local scheduling at the subgoal level, that is, it allows one to modify at runtime the strategy to be used to resolve the subsequent subgoal calls of a tabled predicate. We show that YapTab's hybrid approach does indeed return very substantial performance gains. Results were impressive both on artificial applications, and on a complex, real-life, application.

The remainder of the paper is organized as follows. First, we briefly introduce the basic tabling definitions and present the differences between batched and local scheduling. Next, we describe the issues involved in providing engine support for integrating both scheduling strategies at the subgoal level. We then discuss some experimental results and outline some conclusions.

2 Basic Tabling Definitions

Tabling is about storing intermediate answers for subgoals so that they can be reused when a repeated call appears. Whenever a tabled subgoal S is first called, a new entry 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 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 models have four main types of operations for definite programs:

1. The *tabled subgoal call* operation is a call to a tabled subgoal. It checks if the subgoal is in the table. If so, it allocates a consumer node and starts consuming the available answers. If not, it adds a new entry to the table, and allocates a new generator node.
2. The *new answer* operation verifies whether a newly found answer is already in the table, and if not, inserts the answer. Otherwise, the operation fails.
3. The *answer resolution* operation is executed every time the computation reaches a consumer node. It verifies whether extra answers are available for the particular consumer node and, if so, consumes the next one. If no answers are available, it *suspends* the current computation, either by freezing the whole stacks [1], or by copying the execution stacks to separate storage [3], 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. 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 the goal's table entry and reclaims stack space. Otherwise, control moves to a consumer 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 the 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* [11] 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.

When we call a variant subgoal that is already completed, we can avoid consumer node allocation and perform instead what is called a *completed table optimization* [1]. 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 [12].

3 Scheduling Strategies

It should be clear that at several points we can choose between continuing forward execution, backtracking to interior nodes, returning answers to consumer nodes, or performing completion. The actual sequence of operations depends on the scheduling strategy. We next discuss in some more detail the batched and local scheduling strategies.

3.1 Batched Scheduling

The batched strategy schedules the program clauses in a depth-first manner as does the WAM. In this strategy, new answers are added to the table space but evaluation continues until it resolves all program clauses for the subgoal in hand. Only when all clauses have been resolved, the newly found answers will be returned to consumer nodes. Hence, 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 the engine level, the *fixpoint check procedure* is controlled by the leader of the SCC. Initially, it searches for the younger consumer with unresolved answers, and as long as there new 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, 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 them are released.

3.2 Local Scheduling

Local scheduling is an alternative tabling scheduling strategy that tries to complete subgoals as soon as possible. In this strategy, evaluation is done one SCC at a time. The key idea is that whenever new answers are found, they are added to the table space as usual but execution fails. Thus, execution explores the whole SCC before returning answers outside the SCC. Hence, answers are only returned when all program clauses for the subgoal in hand were resolved.

Figure 1 shows a small example that clarifies the differences between batched and local evaluation. The top sub-figure illustrates the program code and query goal used in the example. Declaration '`:- table t/1.`' indicates that calls to predicate `t/1` should be tabled. The two sub-figures below depict the evaluation sequence for each strategy and how the table space is filled in. In both cases, the leftmost tree represents the evaluation of the query goal `q(X,Y)`. Nodes are numbered according to the evaluation sequence. Generators are depicted by white oval boxes, and consumers by gray oval boxes. For simplicity of presentation, the computation tree for `t(X)` is represented independently at the right.

Both cases begin by resolving the query goal against the unique clause for predicate `q/2`, thus calling the tabled subgoal `t(X)`. As this is the first call to

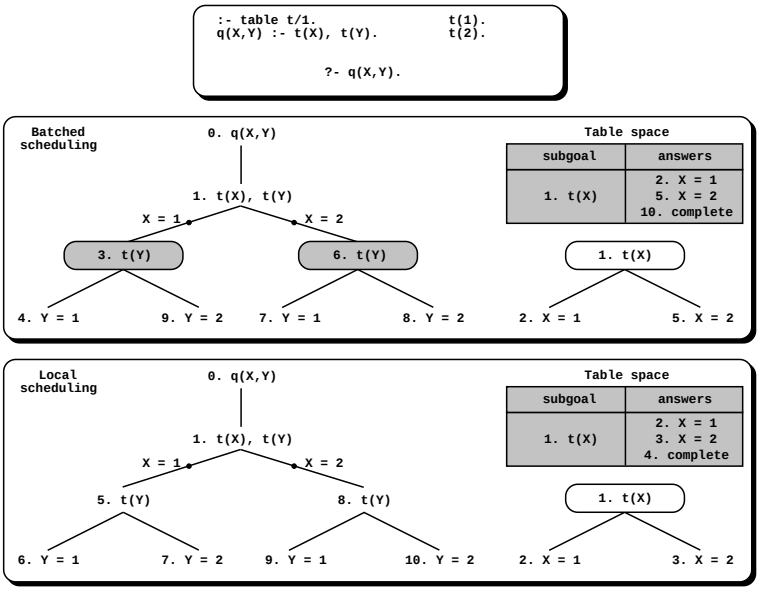

Fig. 1. Batched scheduling versus local scheduling

`t(X)`, we create a generator node and insert a new entry in the table space. The first clause for `t(X)` succeeds immediately, obtaining a first answer for `t(X)` that is stored in the table (step 2). The example is most interesting in how both strategies handle the continuation of `t(X)`.

Evaluation with batched scheduling proceeds as in standard Prolog with a call to the second goal, `t(Y)`, therefore creating consumer node 3. Node 3 is a variant call to `t(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 it, 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 we cannot collect more answers from the table. We then backtrack to node 1 to try the second clause for `t(X)`, and a new answer is found (`X=2`). The continuation again calls `t(Y)`, and a new consumer is created (node 6). Eventually, two new solutions are found for the query goal (steps 7 and 8). We have consumed all answers from the table. Hence, node 6 suspends and the computation backtracks again to node 1. Node 1 is the leader of the SCC, hence at that point, we can check for completion. However, the generator cannot complete because consumer 3 still has unconsumed answers. The computation thus resumes at node 3 and a new solution for the query goal is found (step 9). No new answers are inserted into the table, and all consumers have exhausted the entries in the table. Thus, when we return to the generator node 1, we can finally complete the tabled subgoal call `t(X)` (step 10).

Evaluation with local scheduling differs from early on, namely, after the first answer was found (step 2). To exhaust the current SCC, we should fail, backtrack to node 1, and execute the second clause for `t(X)`. We thus find a second answer

for it (step 3). Then, we fail again to node 1, and the tabled subgoal call `t(X)` can be completed immediately, as we have found no consumers yet (step 4). From now on, answers are consumed by executing compiled code, as we discussed before. The variant calls to `t(X)` at steps 5 and 8 are thus resolved in this way, and no consumer goals are ever needed.

Choosing the best strategy is hard. The main difference between the two is that in batched scheduling, variable bindings are immediately propagated to the calling environment when an answer is found. 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. Freire *et al.* [7] showed that, on average, local scheduling is about 15% slower than batched scheduling in the SLG-WAM [1]. Similar results were also obtained in YapTab [2].

3.3 Defining the Scheduling Strategy

We provide two built-in predicates for defining and controlling the *tabling mode* to be used to evaluate a tabled computation. We extend the standard predicate `yap_flag/2` to define the standard scheduling strategy for the whole computation. Alternatively, we can use the `tabling_mode/2` predicate to define the scheduling strategy of a particular tabled predicate. We next discuss how these predicates can be used to dynamically control the evaluation. Consider, for example, two tabled predicates, `t/1` and `t/2`, and the following query goals:

```
:- t(1).
:- yap_flag(tabling_mode,local), t(2,2).
:- t(3), yap_flag(tabling_mode,default), t(3,3).
:- tabling_mode(t/1,local), t(X), t(X,Y), tabling_mode(t/1,batched), t(Y).
```

In the first example query, `t(1)` evaluates using batched scheduling. This happens because, by default in YapTab, when a predicate is declared as tabled, its initial tabling mode is batched. In the second query, `t(2,2)` evaluates using local scheduling as the call to `yap_flag(tabling_mode,local)` changes the tabling mode of the following computations to local. In the third query, `t(3)` evaluates using local scheduling because the tabling mode for the computation is still local (as a result of the previous `yap_flag/2` declaration in the second query), and `t(3,3)` evaluates using batched. Note that the actual execution tree will have nodes for both strategies: `t(3,3)` might itself call `t(3)`. The call to `yap_flag(tabling_mode,default)` defines that, in what follows, we should use the default strategy of each predicate and the initial tabling mode of `t/2` is batched. Finally, in the fourth query, `t(X)` evaluates using local scheduling and `t(X,Y)` and `t(Y)` evaluates using batched scheduling. The call to `tabling_mode(t/1,local)` initially changes the tabling mode of predicate `t/1` to local and then `tabling_mode(t/1,batched)` changes it back to batched.

4 Implementation

The YapTab design mostly follows the seminal SLG-WAM design [1]: it introduces a new data area to the WAM, 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*. Tables are implemented using tries as proposed in [12]. The differences between the two designs reside in the data structures and algorithms used to control the process of leader detection and the scheduling of unconsumed answers.

Namely, the original SLG-WAM considers that such control should be done at the level of the data structures corresponding to first calls to tabled subgoals, and does so by associating *completion frames* to generator nodes. The SLG-WAM relies on a *completion stack* of generators to detect completion points. 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. Dependency frames store information about the last consumed answer; and information to efficiently check for completion points, and to efficiently move across the consumer nodes with unconsumed answers.

In YapTab, applying batched or local scheduling to an evaluation mainly depends on the way generator nodes are handled. At the engine level, this includes minor changes to the operations tabled subgoal call, new answer and completion. All the other tabling extensions are common across both strategies. We claim that, this makes YapTab highly suitable to efficiently support a dynamic mixed-strategy evaluation.

4.1 Tabled Nodes

By combining the two built-in predicates `yap_flag/2` and `tabling_mode/2` we can dynamically define the scheduling strategy to be used to evaluate each tabled subgoal. Thus, when a tabled subgoal is first called, the tabled subgoal call operation starts by consulting the current tabling mode of the computation/predicate in order to decide the strategy to be used by the corresponding generator node.

In our implementation, generator nodes are WAM choice points extended with two extra fields: `CP_DepFr` is a pointer to the corresponding dependency frame (its use is detailed next) and `CP_SgFr` is a pointer to the associated subgoal frame where answers should be stored. Consumer nodes are WAM choice points extended with the `CP_DepFr` field only. Figure 2 details generator and consumer choice points and their relationship with the table and dependency spaces.

The left sub-figure shows a choice point stack with generator nodes for both strategies and with a consumer node. Remember that the key difference between the two strategies is that local scheduling prevents answers from being returned early by backtracking until getting all answers for the leader generator. At the point all answers have been exhausted, the leader must export them to its environment. To do so, it must act like a consumer: consuming answers

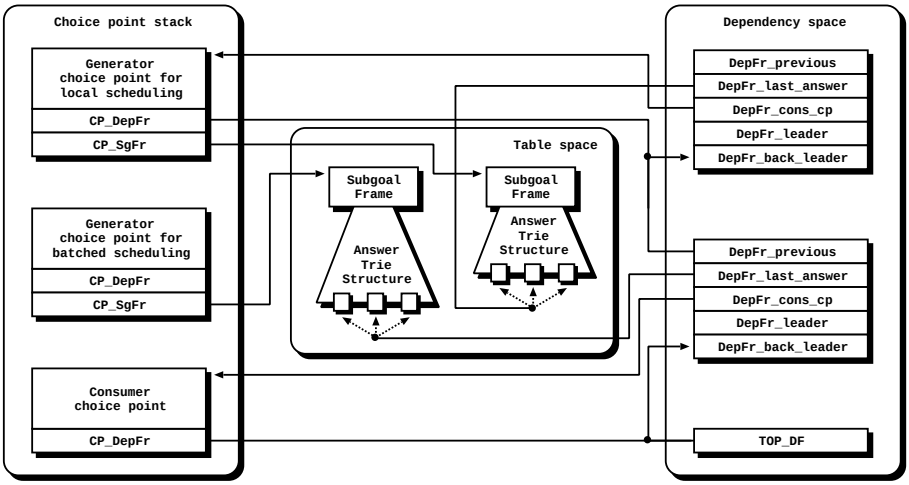

Fig. 2. Generator and consumer choice points in YapTab

and propagating them to caller one by one. In YapTab, this was implemented by having the `CP_DepFr` field in the generators at the same position as for the consumers. This simple extension allows generators being evaluated using local scheduling to easily become consumers. Note that YapTab relies on this field: it is used by the operations new answer and completion as a way to distinguish if a generator is being evaluated using local scheduling (cases where `CP_DepFr` is not `NULL`) or using batched scheduling (cases where `CP_DepFr` is `NULL`).

The right sub-figure shows the dependency frames, the key data structure to synchronize the flow of a tabled evaluation. The `TOP_DF` variable always points to the youngest dependency frame on stack. Frames form a linked list through the `DepFr_previous` field. The `DepFr_last_answer` field points to the last consumed answer in the table space. The `DepFr_cons_cp` field points back to the corresponding consumer choice point. The `DepFr_leader` and the `DepFr_back_leader` fields respectively point to the leader node at creation time and to the leader node where we performed the last unsuccessful completion operation. They are critical in the SCC fixpoint check procedure, that we discuss next.

4.2 Leader Nodes

How does completion change in a mixed environment? Completion 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). The key idea in our original 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. We thus rely on the notion of *leader node*: 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.

Next we show that our algorithm for detecting leader nodes works well in a mixed environment. The algorithm requires computing leader node information whenever creating a new consumer node $\mathcal{C}$. 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 3 shows the procedure that computes the leader node information for a new consumer.

```
compute_leader(consumer node CN) {
  leader_cp = generator_for(CN)    // the generator is the default leader
  df = TOP_DF
  while (DepFr_cons_cp(df) is younger than leader_cp ) {
    if (DepFr_leader(df) is older than leader_cp) {   // older dependency
      leader_cp = DepFr_leader(df)
      break
    }
    df = DepFr_previous(df)
  }
  return leader_cp
}
```

Fig. 3. Pseudo-code for `compute_leader()`

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.

Note that for local scheduling, when we store a generator node $\mathcal{G}$ we also allocate a dependency frame. However, we can avoid calling `compute_leader()` because $\mathcal{G}$ itself is the leader node.

4.3 Completion

Next, we show our implementation of completion. Completion is forced as follows. When a generator choice point tries the last program clause, its `CP_AP` (failure continuation program counter) field is updated to the `completion` instruction. From then on, every time we backtrack to the choice point the operation is executed. Figure 4 shows the pseudo-code for completion in YapTab.

First, the procedure checks out if the generator is the current leader node. If a leader, it checks whether all younger consumer nodes have consumed all their answers. To do so, it traverses the chain of dependency frames looking for a frame that has not yet consumed all the generated answers. If there is such a frame, the computation should be resumed to the corresponding consumer node. Otherwise, it can perform completion. This includes **(i)** marking as complete all the subgoals in the SCC, and **(ii)** deallocating all younger dependency frames. At the end, if the generator was evaluated using local scheduling, we need to

```
completion(generator node GN) {
  if (GN is the current leader node) {
    df = TOP_DF
    while (DepFr_cons_cp(df) is younger than GN) {
      if (unconsumed_answers(DepFr_last_answer(df))) {
        DepFr_back_leader(df) = GN        // mark the leader to return to
        move_to(DepFr_cons_cp(df))
      }
      df = DepFr_previous(df)
    }
    perform_completion()
    if (CP_DepFr(GN) != NULL)                          // local scheduling
      completed_table_optimization()
  }
  if (CP_DepFr(GN) != NULL) {                          // local scheduling
    CP_AP(GN) = answer_resolution
    load_first_available_answer_and_proceed()
  } else backtrack()                                 // batched scheduling
}
```

Fig. 4. Pseudo-code for `completion()`

consume the set of answers that have been found. As the subgoal is already completed, we can execute compiled code directly from the trie data structure associated with the completed subgoal.

On the other hand, if the current node is not the leader, the procedure simply backtracks to the previous node, if in batched mode, or starts acting like a consumer node and consumes the first available answer, if in local mode.

4.4 Answer Resolution

Next, we show that our implementation of answer resolution is independent of strategy. The answer resolution operation executes every time the computation fails back to a consumer. In our implementation, a consumer choice point always points to the `answer_resolution` instruction in its `CP_AP` field. Figure 5 shows the pseudo-code for this instruction in YapTab.

```
answer_resolution(consumer node CN) {
  df = CP_DepFr(CN)                              // dependency frame for CN
  if (unconsumed_answers(DepFr_last_answer(df)))
    load_next_available_answer_and_proceed()
  back_cp = DepFr_back_leader(df)
  if (back_cp == NULL)                                 // first time here
    backtrack()
  df = DepFr_previous(df)
  while (DepFr_cons_cp(df) is younger than back_cp) {
    if (unconsumed_answers(DepFr_last_answer(df))) {
      DepFr_back_leader(df) = back_cp     // mark the leader to return to
      move_to(DepFr_cons_cp(df))
    }
    df = DepFr_previous(df)
  }
  move_to(back_cp)                               // move to last leader node
}
```

Fig. 5. Pseudo-code for `answer_resolution()`

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 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 iterative process of resuming a consumer node, consuming the available set of answers, suspending and then resuming another consumer until a fixpoint is reached, is completely independent of the scheduling strategy being used.

5 Experimental Results

To put the performance results in perspective, we first used a set of common tabled benchmark programs to evaluate the overheads of supporting mixed-strategy evaluation for programs that only require a single-strategy approach. The environment for our experiments was an AMD Athlon XP 2800+ processor with 1 GByte of main memory and running the Linux kernel 2.6.8. YapTab is based on the current development version of Yap, version 4.5.7.

Table 1 shows the running times, in milliseconds, for YapTab supporting a single scheduling strategy (YapTab Single) and supporting the mixed approach (YapTab Mixed). In parentheses, it shows the overhead over YapTab Single. The execution times correspond to the average times obtained in a set of 3 runs. The results indicate that YapTab Mixed introduces insignificant overheads over YapTab Single, both for batched and local scheduling. These overheads are very small. They mainly result from operations that test if a generator is being evaluated using batched or local scheduling.

	Batched Scheduling		**Local Scheduling**	
Program	**YapTab Single**	**YapTab Mixed**	**YapTab Single**	**YapTab Mixed**
mc-iproto	2495	2519 (1.009)	2668	2689 (1.007)
mc-leader	8452	8467 (1.001)	8385	8403 (1.002)
mc-sieve	21568	21325 (0.988)	21797	21217 (0.973)
lgrid	850	870 (1.023)	1012	1031 (1.018)
rgrid	1250	1332 (1.065)	1075	1141 (1.061)
samegen	20	20 (1.000)	21	21 (1.000)
Average		(1.014)		(1.010)

Table 1. Overheads of supporting mixed-strategy evaluation

In the literature, we can find several examples showing that batched scheduling performs better than local scheduling for certain applications and that local scheduling performs better for others [7, 10]. However, usually, these examples

are independent and not part of the same application. To further motivate for the applicability of our mixed-strategy approach, we next present two different examples where we take advantage of YapTab's flexibility.

Our first example is an application in the context of Inductive Logic Programming (ILP) [13]. The fundamental goal of an ILP system is to find a consistent and complete theory (logic program), from a set of examples and prior knowledge, the *background knowledge*, that *explain* all given positive examples, while being consistent with the given negative examples. Since it is not usually obvious which set of hypotheses should be picked as the theory, an ILP system generates many candidate hypotheses (clauses) which have many similarities among them. Usually, these similarities tend to correspond to common prefixes (subgoals) among the hypotheses. Consider, for example, that the system generates an hypothesis '`theory(X):- b1(X),b2(X,Y).`' which obtains a *good coverage quality*, that is, the number of positive examples covered by it is high and the number of negative example is low. Then, it is quite possible that the system will use it to generate more specific clauses like '`theory(X):- b1(X),b2(X,Y),b3(Y).`'.

Computing the coverage of an hypothesis requires, in general, running all positives and negatives examples against the clause. For example, to evaluate if the positive example `theory(p1)` is covered by '`theory(X):- b1(X),b2(X,Y).`', the system executes the goal '`b1(p1),b2(p1,Y)`'. If the same example is then evaluated against the other clause, goal '`b1(p1),b2(p1,Y),b3(Y)`', part of the computation will be repeated. For datasets with a large number of examples, we can arbitrarily do a lot of recomputation. Tabling technology is thus an excellent candidate to significantly reduce the execution time for these kind of problems. Moreover, as we will see, we can benefit from YapTab's mixed-strategy approach to further improve performance.

Assume now that we declared `b2/2` as tabled and that '`b1(p1),b2(p1,Y)`' succeeds. Thus, we can mark `theory(p1)` as covered by the corresponding hypothesis, and we can reclaim space by pruning the search space for the goal in hand. Note that the ILP system is only interested in evaluating the coverage of the examples, and not in finding answers for the subgoals. On the other hand, from the tabling point of view, `b2(p1,Y)` is not completed because it may succeed with other answers for `Y`. A question then arises: should we use batched or local scheduling to table these predicates?

At first, local scheduling seems more attractive because it avoids the pruning problem mentioned above. When the ILP system prunes the search space, the tables are already completed. On the other hand, if the cost of fully generating the complete set of answers is very expensive, then the ILP system may not always benefit from it. Consider, for example a predicate defined by several facts and then by a recursive clause (quite common in some ILP datasets). It can happen that, after completing a subgoal, the subgoal is not called again or when called it succeeds just by using the known facts, thus, turning it useless to compute beforehand the full set of answers.

Note also that, when an example is not covered, all the subgoals in the clause are completed. For example, if in '`b1(p1),b2(p1,Y),b3(Y)`', the subgoal `b3(Y)`

never succeeds then, by backtracking, `b2(p1,Y)` will be completely evaluated. For such cases, batched scheduling is better because variable bindings are automatically propagated. We can also benefit from batched when an example is covered in clauses of the form '`b2(p1,Y),b2(p1,Z)`', with the tabled subgoal appearing repeated. Finally, for subgoals that never succeed or that succeed with a yes answer (all arguments ground), batched and local obtain similar results.

We experimented with using both strategies individually and together. Table 2 shows, the running times, in milliseconds, for the April ILP system [14] running a well-known ILP dataset, the *mutagenesis* dataset.

Predicates	**Running Time**
Without tabling	> 1 day
All batched (11 predicates)	283779
All local (11 predicates)	147937
Some batched (7 predicates), others local (4 predicates)	127388

Table 2. Intermixing batched and local scheduling at the predicate level

We used four different approaches to evaluate the predicates in the background knowledge: **(i)** without tabling; **(ii)** all predicates being evaluated using batched scheduling; **(iii)** all using local scheduling; and **(iv)** some using batched and others using local (for this approach we show the running time for the best result obtained). Note that the running times include the time to run the whole ILP system and not just the time for computing the coverage of the hypotheses. The results show that tabled evaluation can significantly reduce the execution time for these kind of problems. Moreover, they show that, by using mixed-strategy evaluation, we can further speedup the execution. Better performance is still possible if we use YapTab's flexibility to intermix batched and local scheduling at the subgoal level. However, from the programmer point of view, it is very difficult to define the subgoals to table using one or another strategy. Further work is still needed to study how to use this flexibility to, in runtime, automatically adjust the system to the best approach.

We next show a different application where we take full advantage of the dynamic mixed-strategy of YapTab by intermixing batched and local scheduling at the subgoal level. Consider a 30x30 grid, represented by a number of `edge/2` facts, and the following program code:

```
:- table path/2.
path(X,Y) :- path(X,Z), edge(Z,Y).
path(X,Y) :- edge(X,Y).

reachable(X,Y) :- path(F,X), path(F,Y), !.

go_batched :- tabling_mode(path/2,batched).
go_local :- tabling_mode(path/2,local).
```

Now consider the query goal '`path(X,Y),reachable(X,Y)`' that computes the paths in the grid whose extremities are reachable from, at least, another node. We solve this query using three alternative approaches for tabling the

`path/2` predicate: **(i)** only batched scheduling; **(ii)** only local scheduling; and **(iii)** local scheduling for the first query subgoal and batched scheduling for the second. Table 3 shows the running times, in milliseconds, for finding all the solutions for the query above using the three approaches. The execution times correspond to the average times obtained in a set of 3 runs.

Query Goal	Running Time
(i) :- go_batched, path(X,Y), reachable(X,Y), fail.	141962
(ii) :- go_local, path(X,Y), reachable(X,Y), fail.	60471
(iii) :- go_local, path(X,Y), go_batched, reachable(X,Y), fail.	19770

Table 3. Intermixing batched and local scheduling at the subgoal level

The results show that by using local scheduling for computing the first subgoal and batched for the second we are able to significantly reduce the execution time and achieve the best performance. This happens because, by using local scheduling to compute the complete set of answers for `path(X,Y)`, we avoid complex dependencies when executing predicate `reachable(X,Y)` with batched. Note that when we call `path(F,X)` in predicate `reachable/2`, `F` is a free variable. Then, when we use the first clause of `path/2` to solve `path(F,X)`, we get a call to `path(F,Z)` (with both variables free), which is a variant call of the initial query subgoal `path(X,Y)`, and thus we must allocate a consumer node.

On the other hand, if we already have the set of answers for the first query subgoal, it is best if we use batched to solve the calls to the `reachable/2` predicate. If we use local scheduling, we will compute all the answers for each particular call to `path(F,X)`, with `X` ground, and this may lead to unnecessary computation. Note that predicate `reachable/2` succeeds by pruning the search space with a cut operation, which makes batched scheduling more appropriate for this particular example.

6 Conclusions

In this work, we presented the design and implementation of YapTab to support dynamic mixed-strategy evaluation of tabled logic programs. Our approach proposes the ability to combine batched scheduling with local scheduling at the subgoal level with minor changes to the tabling engine. These changes introduced insignificant overheads on YapTab's performance. Moreover, our results show that dynamic mixed-strategies can be extremely important to improve the performance of some applications.

The proposed data structures and algorithms can also be easily extended to support dynamic switching from batched to local scheduling and vice versa, while a generator is still producing answers. In particular, we plan to study how such flexibility can be used to design a more *aggressive* approach for applications that do a lot of pruning over the table space, such as ILP applications. We also plan to further investigate the impact of combining both strategies in other application areas.

Acknowledgments

We are very thankful to Nuno Fonseca for his support with the April ILP System. This work has been partially supported by APRIL (POSI/SRI/40749/2001), Myddas (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC.

References

1. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems **20** (1998) 586–634
2. Rocha, R., Silva, F., Santos Costa, V.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Conference on Tabulation in Parsing and Deduction. (2000) 77–87
3. Demoen, B., Sagonas, K.: CHAT: The Copy-Hybrid Approach to Tabling. Future Generation Computer Systems **16** (2000) 809–830
4. Guo, H.F., Gupta, G.: A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In: International Conference on Logic Programming. Number 2237 in LNCS, Springer-Verlag (2001) 181–196
5. Zhou, N.F., Shen, Y.D., Yuan, L.Y., You, J.H.: Implementation of a Linear Tabling Mechanism. Journal of Functional and Logic Programming **2001** (2001)
6. Sagonas, K., Swift, T., Warren, D.S.: XSB as an Efficient Deductive Database Engine. In: ACM SIGMOD International Conference on the Management of Data, ACM Press (1994) 442–453
7. Freire, J., Swift, T., Warren, D.S.: Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In: International Symposium on Programming Language Implementation and Logic Programming. Number 1140 in LNCS, Springer-Verlag (1996) 243–258
8. Freire, J., Swift, T., Warren, D.S.: Taking I/O seriously: Resolution reconsidered for disk. In: International Conference on Logic Programming. (1997) 198–212
9. Freire, J.: Scheduling Strategies for Evaluation of Recursive Queries over Memory and Disk-Resident Data. PhD thesis, State University of New York (1997)
10. Freire, J., Warren, D.S.: Combining Scheduling Strategies in Tabled Evaluation. In: Workshop on Parallelism and Implementation Technology for Logic Programming. (1997)
11. Chen, W., Swift, T., Warren, D.S.: Efficient Top-Down Computation of Queries under the Well-Founded Semantics. Journal of Logic Programming **24** (1995) 161–199
12. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38** (1999) 31–54
13. Muggleton, S.: Inductive Logic Programming. In: Conference on Algorithmic Learning Theory, Ohmsma (1990) 43–62
14. Fonseca, N., Camacho, R., Silva, F., Santos Costa, V.: Induction with April: A Preliminary Report. Technical Report DCC-2003-02, Department of Computer Science, University of Porto (2003)