A Tabling Engine for the Yap Prolog System

Ricardo Rocha Fernando Silva Vítor Santos Costa

Abstract

This paper addresses the design and implementation of YapTab, a tabling engine that extends the Yap Prolog system to support sequential tabling. The tabling implementation is largely based on the XSB engine, the SLG-WAM, however substantial differences exist since our final goal is to support parallel tabling execution. We discuss the major contributions in YapTab and outline the main differences of our design in terms of data structures and algorithms. Finally, we present some initial performance results for YapTab and compare with those for XSB.

Keywords: Logic Programming, Tabling, Implementation.

1 Introduction

Logic programming systems provide a high-level, declarative approach to programming. Arguably, Prolog is the most popular logic programming language. Prolog programs are written in a subset of First-Order Logic, Horn clauses, that has an intuitive interpretation as positive facts and rules. Following Kowalski's [10] motto:

$$algorithm = logic + control$$

Ideally, one would want Prolog programs to be written as logical statements first, and for control to be tackled as a separate issue. In practice, the operational semantics of Prolog is given by SLD-resolution with depth-first search, a refutation strategy particularly simple and that matches current stack-based machines particularly well. Unfortunately, the limitations of SLD-resolution means that Prolog programmers must be very aware of the Prolog computation rule throughout program development. For instance, it is in fact quite possible that logically correct programs will enter infinite loops.

Several proposals have been put forth to improve the declarativeness and expressiveness of Prolog. One such proposal that has been gaining in popularity is the

Ricardo Rocha and Fernando Silva are with DCC-FC & LIACC, University of Porto, Portugal. E-mails: {ricroc,fds}@ncc.up.pt

Vítor Santos Costa is with COPPE Systems, Federal University of Rio de Janeiro, Brazil. E-mail: vitor@cos.ufrj.br

Work partially supported by the projects Melodia (JNICT PBIC/C/TIT/2495/95), Dolphin (PRAXIS/2/2.1/TIT/1577/95), CLoP (CNPq), PLAG (FAPERJ) and by Fundação para a Ciência e Tecnologia.

use of *tabling* or *memoing*. In a nutshell, tabling consists of storing intermediate solutions to a query so that they can be reused during the query execution process. It can be shown that tabling-based computational rules can have better termination properties than SLD-based models, and indeed termination can be guaranteed for all programs with the *bounded term-size property* [3].

Work on SLG-resolution [2], as implemented in the XSB System [16], proved the viability of tabling technology for applications such as natural language processing, knowledge-base systems and data-cleaning, model-checking, and program-analysis. Tabling also facilitates the implementation of several extensions to Prolog, including support for non-definite clauses [16] that allows for non-monotonic reasoning.

Although tabling can work for both deterministic [18] and non-deterministic programs, quite a few interesting applications of tabling are by nature non-deterministic. This rises the question of whether further efficiency would be possible by running several branches of the search tree in parallel. Freire and colleagues were the first to propose that tabled goals could be a source of parallelism [7]. In previous work [12] we showed that the same mechanism can be used to exploit or-parallelism from both tabled and non-tabled goals. We also presented two computational models, the *Or-Parallelism within Tabling (OPT)* and *Tabling within Or-Parallelism (TOP)* models, that combine tabling with or-parallelism. We have since decided to implement the OPT model [13] over the YapOr system [14], based on the high-performance Yap Prolog compiler [4].

This paper addresses the implementation of YapTab, a sequential tabling engine designed to support or-parallelism. The implementation is based on the ground-breaking work for the XSB system, and specifically on the SLG-WAM [17]. We innovated by considering the novel issues arising with the introduction of parallelism. In terms of the basic engine, the original XSB design was therefore changed when restoring computations, determining leader nodes and completing subgoals.

The remainder of the paper is organized as follows. First, we briefly introduce the tabling concepts and the SLG-WAM. Next, we present the OPT computational model and discuss the major contributions in YapTab. We then present the main data areas, data structures and algorithms to extend the Yap Prolog system to support tabling. Last, we present some early performance data and terminate by outlining some conclusions and further work.

2 Tabling Concepts and the SLG-WAM

Tabling is about storing and reusing intermediate answers for goals. Whenever a tabled subgoal S is called for the first time, an entry for S is allocated in the *table space*. This entry will collect all the answers generated for S. Repeated calls to *variants* of S are resolved by consuming the answers already stored in the table. Meanwhile, as new answers are generated for S, they are inserted 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*, that consume answers from the table space, and *interior nodes*, that are evaluated by standard SLD-resolution.

Space for a subgoal can be reclaimed when the subgoal has been *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. Note that a number of subgoals may be mutually dependent, forming a *strongly connected component* (or *SCC*) [16], and therefore can only be completed together. The completion operation is thus performed by the *leader* of the SCC, that is, by the oldest subgoal in the SCC, when all possible resolutions have been made for all subgoals in the SCC. Hence, in order to efficiently evaluate programs one needs an efficient and dynamic detection scheme to determine when all the subgoals in a SCC have been completely evaluated.

For definite programs, tabling based evaluation has four main types of operations: *Tabled Subgoal Call* creates a generator node; *New Answer* verifies whether a newly generated answer is already in the table, and if not, inserts it; *Answer Resolution* consumes an answer from the table; and *Completion* determines whether an SCC is completely evaluated, and if not, schedules a resolution to continue the execution.

In XSB, the implementation of tabling was attained by extending the WAM [19] into the SLG-WAM, with minimal overhead. In short, the SLG-WAM introduces special instructions to deal with the operations above and two new memory areas: a *table space*, used to save the answers for tabled subgoals; and a *completion stack*, used to detect when a set of subgoals is completely evaluated.

Further, whenever a consumer node gets to a point in which it has consumed all available answers, but the correspondent tabled subgoal has not yet completed and new answers may still be generated, the computation must be suspended. In the SLG-WAM the suspension mechanism is implemented through a new set of registers, the *freeze registers*, which freeze the WAM stacks at the suspension point and prevent all data belonging to the suspended branch from being erased. To later resume a suspended branch, the bindings belonging to the branch must be restored. SLG-WAM achieves this by using an extension of the standard trail, the *forward trail*, to keep track of the bindings values. An alternative to the SLG-WAM is to keep on working on the current stacks but store the frozen stacks away on an external space, as is done in CAT [5] (the same principle was used to suspend or-parallel work in Muse [1]). The CHAT model [6] significantly reduces the overheads of copying and is currently the default scheme in XSB-Prolog. In more recent work some authors have proposed a linear tabling mechanism [20] and a reordering based tabling mechanism [9]. Both of these schemes avoid freezing by recomputing goals until a fixed point is reached.

3 Tabling and Parallelism

In previous work [12] we proposed two computational models to combine tabling with or-parallelism, the OPT and the TOP approaches. We have decided to implement the OPT approach [13]. The OPT approach generalizes Warren's multi-sequential engine framework for or-parallelism. The or-parallelism stems from having several engines that implement SLG-resolution, instead of implementing Prolog's

SLD-resolution.

Tabling is the base component of the OPT computational model. Each computational agent, or *worker*, can be considered a full sequential tabling engine and should spend most of its computation time exploiting the search tree involved in such an evaluation. It allocates all three types of nodes, fully implements suspension of tabled subgoals, and resumes subcomputations to consume newly found answers. Or-parallelism is only triggered when a worker runs out of alternatives to exploit. Unexploited alternatives should be made available for parallel execution, regardless of whether they originate from a generator, consumer or interior node. Therefore, parallelism stems from both tabled and non-tabled subgoals. This contrasts with the table-parallelism model [7] that only considers tabled subgoals as candidates for parallel execution.

To implement the OPT approach, we have decided to use the YapOr system [14] as the parallel component of the model. The YapOr system is an or-parallel Prolog system based on environment copying. Thus, we have designed the YapTab tabling engine in order to meet the requirements of the OPT approach based on environment copying. The YapTab tabling engine is SLG-WAM based, not copying or recomputation-based. We decided to use the SLG-WAM in order to better isolate the interactions between tabling and parallelism. Using recomputation would simplify the implementation [9] but may result in a larger search space, so it would be hard to evaluate our parallel implementation versus current sequential technology. As we shall see next, most of our work supports both copying and the SLG-WAM's cactus stack.

In an environment copying model, sharing is implemented through copying of the execution stacks between workers and, thus, a shared branch may exist several times for the workers that are sharing the branch. The duplication of items is a major source of overhead. It implies larger stack areas to be copied when sharing, and it requires synchronization mechanisms when updating common items and when replicating the new values. Hence, in order to efficiently integrate the tabling and the or-parallel components of the OPT model, we should minimize this duplication. To address this need, YapTab introduces a new data structure, the *dependency frame*, that resides in a single shared space that we called the *dependency space*.

The dependency frame data structure keeps track of all data related with tabling suspensions. This allows us to reduce the number of extra fields in tabled choice points and to eliminate the need for a completion stack area. Moreover, a smaller number of extra fields in tabled choice points minimizes the time needed to perform copying. Eliminating the completion stack area from the YapTab design reduces the number of stack areas to be copied when sharing, and simplifies the complexity in managing shared tabling suspensions.

In practice, we found that this solution simplifies the parallel implementation of fundamental aspects to the system's efficiency. Sharing tabling suspensions is straightforward, the worker requesting work only needs to update its private top dependency frame pointer to the one's of the sharing worker. Concurrent accesses or updates to the shared suspension data can be synchronized through the use of a locking mechanism at the dependency frame level. The OPT completion algorithm [13] for shared branches is mainly based on the dependency frame data structure which

avoids explicit communication and synchronization between workers.

4 Extending Yap to Support Tabling

The YapTab design is WAM based, as is the SLG-WAM. It implements two tabling scheduling strategies, batched and local [8], and in the initial design it only considers table predicates without any kind of negative calls. As in the original SLG-WAM, 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 the four main tabling operations: tabled subgoal call, new answer, answer resolution and completion.

The substantial differences between the two designs, and corresponding implementations, reside in the aspects that can be a potential source of overheads when the tabling engine is extended to a parallel model. In order to efficiently integrate the tabling and the or-parallel components of the OPT computational model, YapTab introduces the dependency frame data structure. To take advantage of the philosophy behind the dependency frame data structure, all the algorithms related with suspension, resumption and completion were redesigned. We then present the main data areas, data structures and algorithms implemented to extend the Yap system to support tabling. All the algorithms described assume a batched scheduling strategy implementation [8].

4.1 Table Space

The table space can be accessed in different ways during the course of an evaluation: to look up if a subgoal is in the table, and if not insert it; to verify whether a newly found answer is already in the table, and if not insert it; to pick up answers to consumer nodes; and to mark subgoals as completed.

Hence, the correct design of the table data structures is the base component to implement an efficient tabling system. Our implementation uses tries as the basis for tables, as proposed in [11]. Tries provide complete discrimination for terms and permit a lookup and possible insertions to be performed in a single pass through a term. Tries are also easily parallelizable.

Figure 1 presents the table data structures for a particular predicate `t/2` after the execution of some `tabled_subgoal_call` and `new_answer` instructions. As in SLG-WAM, each invocation of the `tabled_subgoal_call` instruction leads to either finding a path through the subgoal trie nodes, always starting from the table entry, until a matching subgoal frame is reached, or creating the correspondent path of subgoal trie nodes, otherwise. Each invocation of the `new_answer` instruction corresponds to the definition of a path through the answer trie nodes, starting from the corresponding subgoal frame.

Searching through a chain of sibling nodes that represent alternative paths for a table entry is done sequentially. However, if the chain becomes larger then a threshold value, we dynamically index the nodes through a hash table to provide direct node access and therefore optimizing the search.

The table subgoal frames delimit the subgoal and answer trie nodes space. They

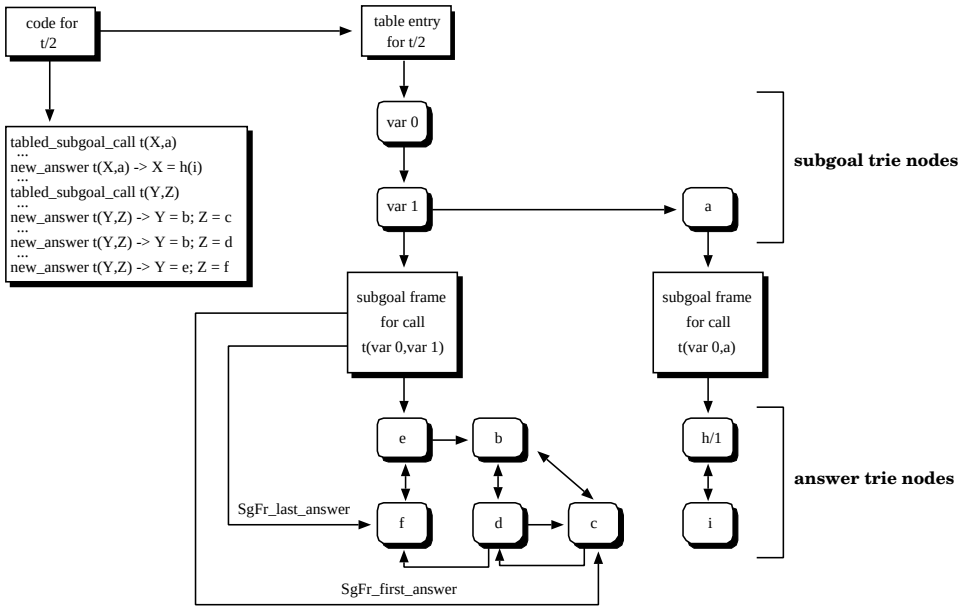

Figure 1: Using tries to organize the table space.

include a `SgFr_first_answer` pointer to provide access to the answers already found for a particular subgoal. Another pointer, `SgFr_last_answer`, marks the last answer that has been found for the subgoal. Whenever a consumer node needs to check whether new answers for its subgoal have been added to table, it compares the subgoal frame `SgFr_last_answer` pointer with its pointer to the last consumed answer. The subgoal frames are also used to mark subgoals as completed.

4.2 Generator and Consumer Nodes

Remember that interior nodes correspond to normal (not tabled) subgoals and they are evaluated by the standard SLD-resolution. Generator and consumer nodes correspond, respectively, to first and variant calls to tabled subgoals. The generator nodes use program clause resolution to produce and store answers in the table space for the corresponding tabled subgoal. The consumer nodes load from the table space the answers previously stored by the associated generator node.

Interior nodes are implemented as normal WAM choice points (see Fig. 2). The `CP_TR`, `CP_H`, `CP_B`, `CP_CP`, `CP_AP` and `CP_ENV` choice point fields [19] 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.

In the SLG-WAM, generator nodes are WAM choice points extended with a few extra fields to control tabling execution. In our implementation the generator choice point only requires one of the extra fields used in the SLG-WAM, the `CP_SG_FR` field, that is a pointer to the associated subgoal frame in the table space.

In SLG-WAM, the consumer choice points store supplementary information about the suspension point. In our case, we move that information to a dependency frame and leave a pointer to this frame in the `CP_DEP_FR` consumer choice point field.

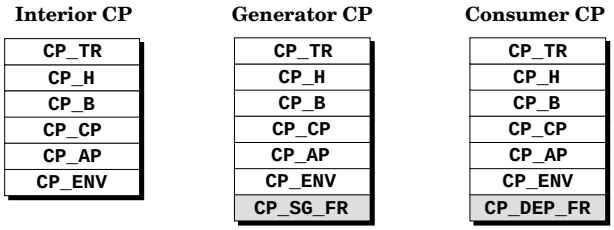

Figure 2: Structure of interior, generator and consumer choice points.

Hence, the consumer choice point only requires an extra field. The dependency frames are linked forming a dependency graph between consumer nodes. Additionally, the dependency frame stores information to efficiently check for completion points, and to efficiently move across the consumer nodes with unconsumed answers. As we shall see, this additional information replaces the need for a completion stack.

4.3 Subgoal and Dependency Frames

The subgoal and dependency frames are the key data structures required to control the flow of a tabling computation. As mention before, the subgoal frames are used to access, insert and load the particular answers found for a subgoal and to check for completed subgoals. The dependency frames are used to synchronize suspension, resumption and completion of subcomputations. Figure 3 shows the subgoal and dependency frame structures in detail.

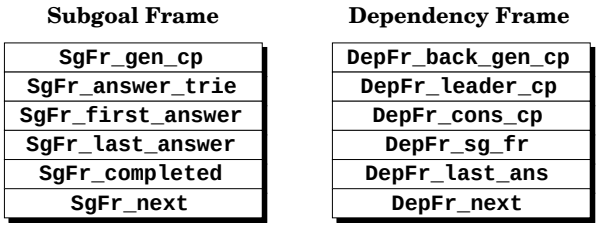

Figure 3: Structure of subgoal and dependency frames.

A subgoal frame is a six field data structure. The `SgFr_gen_cp` is a back pointer to the correspondent generator choice point; the `SgFr_answer_trie` is a pointer to the top answer trie node and it is used to check for/insert new answers; the `SgFr_first_answer` is a pointer to the bottom answer trie node of the first available answer; the `SgFr_last_answer` is a pointer to the bottom answer trie node of the last available answer; the `SgFr_completed` is a flag that indicates if a subgoal is completed or not; and the `SgFr_next` is a pointer to the next subgoal frame and it is used to efficiently go through the frames when performing completion. To access the subgoal frames chain, we use a global `TOP_SG_FR` variable that points to the younger subgoal frame.

Each dependency frame is also a six field data structure. The `DepFr_back_gen_cp` is a pointer to the older generator choice point when we perform an unsuccessful completion operation and it is used to efficiently schedule a backtracking node (see 4.7

for details); the `DepFr_leader_cp` is a pointer to the leader choice point and it is used to check for completion points; the `DepFr_cons_cp` is a back pointer to the consumer choice point; the `DepFr_sg_fr` and the `DepFr_last_ans` are pointers respectively to the correspondent subgoal frame and to the last consumed answer and they are used to connect consumer nodes with the table space in order to search for and to pick up new answers; and the `DepFr_next` is a pointer to the next dependency frame and it is used to form a dependency graph between consumer nodes, used to efficiently check for leader nodes and perform completion. To access the dependency graph, we use a global `TOP_DEP_FR` variable that points to the younger dependency frame.

Figure 4 shows an example of how the data structures presented are used in a particular evaluation. The left sub-figure presents the dependencies between the predicates involved in the example.

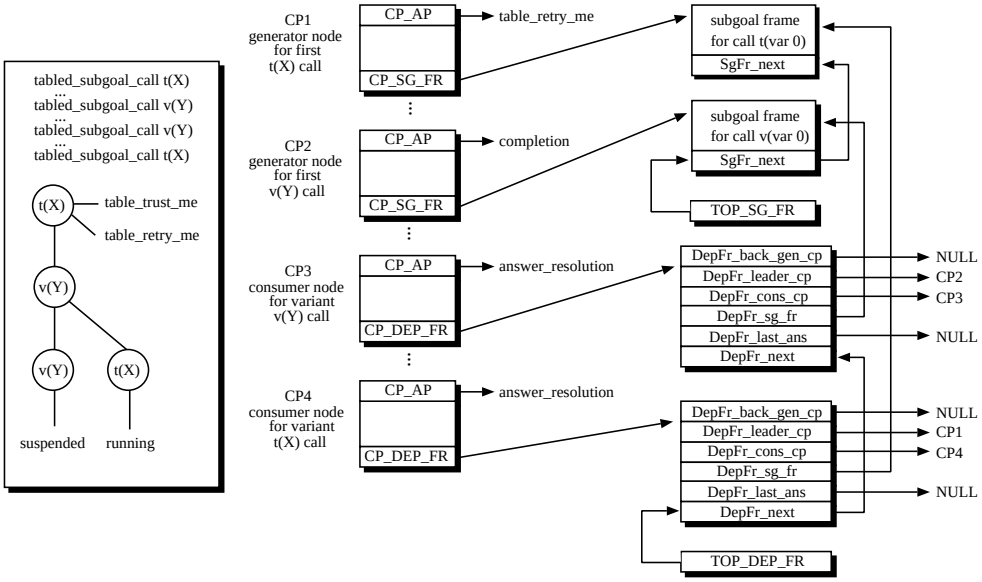

Figure 4: Dependencies between choice points, subgoal and dependency frames.

The first instance of `tabled_subgoal_call` searches the table space for the corresponding subgoal `t(X)`. As this is the first call to the subgoal, it must allocate a subgoal frame and store a generator choice point. Supposing that `t/1` is a three clause predicate, the `CP_AP` generator field is initialized with the `table_retry_me` instruction corresponding to the WAM code of the second clause. Assuming that `v/1` is a two clause predicate, an analogous situation occurs with the first call to subgoal `v(Y)`. The only difference resides in the `table_trust_me` instruction used to initialize the `CP_AP` generator choice point field.

Following the example, the second call to `v(Y)` searches the table space and finds that it is a variant call to subgoal `v(var0)`. Thus, it allocates a dependency frame and stores a consumer choice point. A consumer choice point is initialized with its `CP_AP` field pointing to the `answer_resolution` pseudo instruction. Assuming that no answers were found for subgoal `v(var0)`, the computation will backtrack to the previous choice point `CP2`. The `table_trust_me` instruction gets executed, and the `CP_AP` generator choice point field is update to the `completion` pseudo instruction.

The second call to `t(X)` implies a similar procedure to the previous one.

The dependency frame fields `DepFr_back_gen_cp` and `DepFr_leader_cp` and the pseudo instructions `answer_resolution` and `completion` are detailed next.

4.4 Freeze Registers

A tabling evaluation can be seen as a sequence of suspension and resumptions of subcomputations. To preserve the environment of a suspended computation the stacks are frozen using a set of freeze registers, one per stack. This way, and until the completion of the appropriate subgoal call does not take place, the space belonging to the suspended branch is safe from being erased. It is only upon completion that we can release the space previously frozen and adjust the freeze registers.

In the SLG-WAM, the generator choice points are extended to store the freeze registers at choice point creation, so that they can be adjusted if completion takes place. In our approach, we adjust the freeze registers using the top stack pointers saved in the previous suspension point, that is, the younger consumer node kept in the preserved stacks. To access that node we check for the top dependency frame, using `TOP_DEP_FR`.

4.5 Forward Trail

The forward trail is an extension of the standard WAM trail that records the variable bindings. In SLG-WAM, each forward trail frame records the address of the trailed variable, the value to which the variable was bound and a pointer to the parent trail frame. The parent trail frame pointer is used to correctly move across the variables in a branch, hence avoiding variables in frozen segments [16].

In YapTab, the forward trail is implemented without parent trail frame pointers. As Yap already uses the trail to store information beyond the normal variable trailing (to control dynamic predicates and multi-assignment variables), we extend this information to also control the chain between frozen segments. In terms of computation complexity the two approaches are equivalent. The main advantage of our scheme is that Yap already tests the trail frames to check if they are of a special type, and so, we do not introduce further overheads which would be the case if we had chosen the SLG-WAM approach.

Figure 5 illustrates our implementation scheme. Suppose that the execution has reached the consumer node marked as (a). At this point the trail register `TR` and the trail freeze register `TR_FZ` are the same. Now if backtracking takes place up to the node marked by (b), the variables corresponding to the backtracked segment are untrailed and the trail register is made to point to the last untrailed frame. At this point, the trail register points to a position preceding the one pointed by the trail freeze register. The trail segment in between these registers must become frozen as it may be resumed later. To ensure that the bindings in the frozen segment are not erased and are not seen by a new untrailing operation, we use a special trail frame to mark the existence of a frozen segment just above it. This frame records the continuation trail frame that allows for the frozen segment to be ignored in a backtracking operation. The trail register is also updated to point to this new trail

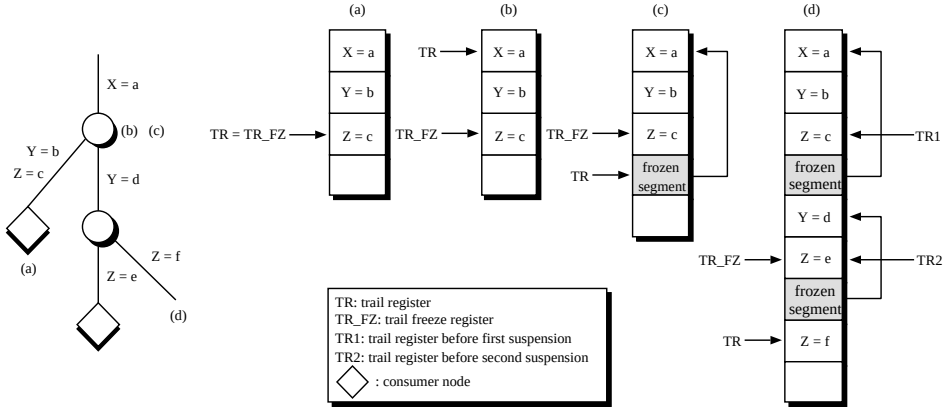

Figure 5: The forward trail implementation.

frame (as illustrated in (c)). Suppose the execution has evolved to situation (d) in which the trail shows a more complex chaining of segments. At this point, if resumption takes place at one of the consumer nodes, then the variable bindings belonging to the respective branch can be correctly untrailed to the previous values. This is accomplished by following the trail register saved in the consumer node before the suspension (`TR1` or `TR2` in situation (d)).

4.6 Completion and Leader Nodes

The completion operation takes place when a generator node exhausts all alternatives and it finds itself as a leader node. We designed our algorithms to quickly determine whether a generator node is a leader node. A special field in the dependency frame structure is used to hold a pointer to the leader node of the SCC that includes the current suspension point. To compute the leader node information when allocating a dependency frame, we first hypothesize that the leader node is the generator node for the current variant subgoal call, say $\mathcal{N}$. Next, for all consumer nodes between $\mathcal{N}$ and the current consumer node, we check whether they depend on an older generator node. Consider that the oldest dependency is for the generator node $\mathcal{N}'$. If this is the case, then $\mathcal{N}'$ is the leader node, otherwise our hypothesis was correct and the leader is indeed the initially found generator node $\mathcal{N}$.

By using the leader node information from the dependency frames, the generator nodes can quickly determine whether they are leader nodes. A generator node finds itself as a leader node if there are no younger dependencies (i.e., no younger consumer nodes) or if it is the leader node referred in the top dependency frame. Figure 6 illustrates a small example of node dependencies.

In Fig. 6(a), the generator node `N3` is the top leader node because there are no younger consumer nodes. By top leader node we mean the generator node where the next completion operation may take place, i.e., the younger generator node that is a leader node. Suppose that a new consumer node is created, that is node `N4` in Fig. 6(b). Then a dependency frame associated with `N4` has to be allocated and the leader node information must be computed. In this case, the leader node is `N1` because `N4` is a variant subgoal `a` of the generator node `N1` and there are no other

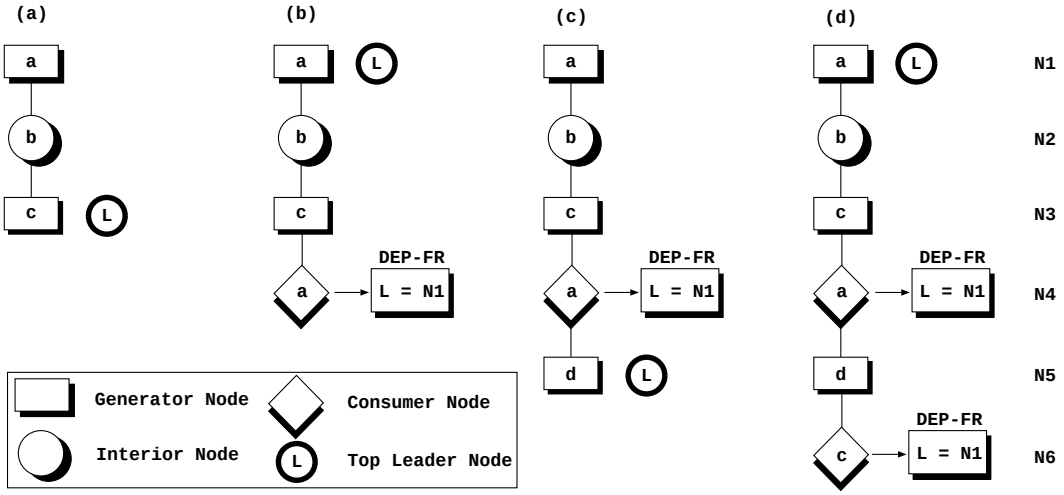

Figure 6: Spotting the top leader node.

consumer nodes in between. As a result, the top leader node for the set of nodes including `N4` becomes `N1`. Figure 6(c) illustrates a similar case to the first one, and `N5` becomes the new top leader node. The consumer node `N6` (Fig. 6(d)) for the variant subgoal `c` has its generator node at node `N3`. Since in between nodes `N6` and `N3` there is a dependency for an older generator node, `N1`, given by the frame associated with consumer node `N4`, the leader node information for the dependency frame associated with `N6` is also `N1`. This turns `N1` again as the top leader node.

We then present in Fig. 7 the pseudo-code for the `completion()` instruction. It gets executed when the computation fails to a generator choice point with no alternatives left.

```
completion (generator node GN) {
  if (GN is the top leader node) {
    df = TOP_DEP_FR
    while (DepFr_cons_cp(df) is younger than GN)) {
      if (DepFr_last_ans(df) != SgFr_last_answer(DepFr_sg_fr(df))) {
        CN = DepFr_cons_cp(df)  % younger consumer node with unconsumed answers
        DepFr_back_gen_cp(df) = GN
        restore_variables(from CP_TR(GN) to CP_TR(CN))
        goto answer_resolution(CN)  % resume computation to CN
      }
      df = DepFr_next(df)
    }
    complete(GN)  % perform completion in GN
  }
  backtrack_to(CP_B(GN))
}
```

Figure 7: Pseudo-code for `completion()`.

Whenever a generator node finds that it is the top leader node, it starts to check if there are younger consumer nodes with unconsumed answers. This can be done by going through the chain of dependency frames looking for a frame with unconsumed answers. If there is such a frame, it resumes the computation to the corresponding consumer node. Before resuming, it musts update the `DepFr_back_gen_cp` field (more details in 4.7) and use the forward trail to restore conditional bindings.

Otherwise, we can perform completion. This includes marking as completed all the subgoals in the SCC, using the TOP_SG_FR to go through the subgoals frames; deallocating all the younger dependency frames, using the TOP_DEP_FR to go through the dependency frames; and readjusting the TOP_SG_FR and TOP_DEP_FR top pointers.

4.7 Answer Resolution

When a consumer choice point is allocated, its CP_AP field is made to point to an answer_resolution instruction. This instruction is responsible for resuming the computation guaranteeing that every answer is consumed once and just once. Figure 8 shows the pseudo-code for the answer_resolution() instruction.

```
answer_resolution (consumer node CN) {
  DEP_FR = CP_DEP_FR(CN)
  if (DepFr_last_ans(DEP_FR) != SgFr_last_answer(DepFr_sg_fr(DEP_FR))) {
    load_next_unconsumed_answer_for_subgoal(DepFr_sg_fr(DEP_FR))
    proceed
  }
  back_cp = DepFr_back_gen_cp(DEP_FR)
  if (back_cp == NULL)
    backtrack_to(CP_B(CN))
  df = DepFr_next(DEP_FR)
  while (DepFr_cons_cp(df) is younger than back_cp)) {
    if (DepFr_last_ans(df) != SgFr_last_answer(DepFr_sg_fr(df))) {
      DepFr_back_gen_cp(df) = back_cp
      back_cp = DepFr_cons_cp(df)  % next consumer node with unconsumed answers
      restore_variables(from CP_TR(CN) to CP_TR(back_cp))
      goto answer_resolution(back_cp)  % resume computation to back_cp
    }
    df = DepFr_next(df)
  }
  unbind_variables(from CP_TR(CN) to CP_TR(back_cp))
  goto completion(back_cp)  % resume computation to older generator node
}
```

Figure 8: Pseudo-code for answer_resolution().

The answer_resolution algorithm first checks the table space for unconsumed answers for the subgoal in hand. If there are new answers, it loads the next available answer and proceeds the execution. Otherwise, it schedules for a backtracking node.

If this is the first time that backtracking from that consumer node takes place, then backtracking is performed as usual to the previous node. This is the case when the DepFr_back_gen_cp dependency frame field is NULL. Otherwise, we know that the DepFr_back_gen_cp field stores the pointer to the older generator node $\mathcal{N}$ from where the computation has been resumed during an unsuccessful completion operation. Therefore, backtracking must be done to the next consumer node that has unconsumed answers and is younger than $\mathcal{N}$. If there are no such consumer nodes then backtracking must be done to the $\mathcal{N}$ generator node.

5 Local Scheduling

All the algorithms described in the previous section assume a batched scheduling strategy. Local scheduling is an alternative tabling scheduling strategy that tries to

evaluate subgoals as soon as possible [8]. Evaluation is done one SCC at a time, and answers are returned outside of an SCC only after that SCC is completely evaluated. In other words, local scheduling prevents answers from being returned to the calling environment of the leader while its SCC is not completely evaluated. We are interested in alternative tabling scheduling strategies to study its impact when combining tabling with parallelism. As local scheduling completes subgoals sooner, this can decrease some complex dependencies when running in parallel. We then present how local scheduling is implemented on top of batched scheduling.

As the reader will see, it is straightforward to extend the engine to local scheduling. To prevent answers from being returned to the calling environment of a generator node, after a new answer is found for a particular subgoal, local scheduling fails and backtracks to search for the complete set of answers. If a generator node finds that it is not a leader node, then it must act like a consumer node to consume the answers that were prevented from being returned to its environment. In our approach, a generator choice point is implemented as a consumer choice point. Hence, when we store a generator node we must allocate a dependency frame and initialize it as described before.

In batched scheduling we use the `CP_SG_FR` generator choice point field to access the correspondent subgoal frame. In local scheduling we must use the `DepFr_sg_fr` field of the dependency frame pointed by the `CP_DEF_FR` generator choice point field. To fully implement local scheduling, we need to slightly change the `completion` instruction. Figure 9 shows the modified pseudo-code for `completion`.

```
completion (generator node GN) {
  if (GN is the top leader node) {
    ...
    complete(GN)
    load_first_unconsumed_answer_for_subgoal(DepFr_sg_fr(CP_DEP_FR(GN)))  % new
    proceed                                                               % new
  }
  CP_AP(GN) = answer_resolution                                           % new
  backtrack_to(CP_B(GN))
}
```

Figure 9: Pseudo-code for `completion()` using a local scheduling strategy.

As the newly founded answers are prevented from being immediately returned, we need to consume them at a later point. If we perform completion with success, instead of backtracking to the previous node, we start consuming the set of answers that have been founded to the completed subgoal. Otherwise, if a generator node finds that it is not a leader node then it must act like a consumer node. To implement this idea we update the `CP_AP` choice point field to the `answer_resolution` pseudo-instruction before backtracking.

6 Initial Performance Evaluation

We have implemented two scheduling strategies, batched and local scheduling, that constraint differently the tabling execution. The performance study will be made using both strategies. We start by analyzing the overheads of supporting tabling in Yap on a set of non tabled Prolog programs, and by measuring the XSB behavior

on the same set of programs. We then use four versions of a tabled program with varying complexity of the search space to compare YapTab with XSB.

YapTab is based on the Yap4.2.1 engine and the results were obtained on a 200 MHz PentiumPro with 128MB of main memory, a 256KB cache, and running Linux2.2.5. We used the same compilation flags for Yap and for YapTab. Regarding XSB, we used version 2.2 with the default configuration and the default execution parameters (chat engine and batched scheduling).

To put the performance results in perspective we first use a standard set of non tabled logic programming benchmarks to compare the performance of YapTab with Yap and XSB. The benchmarks include the n-queens problem, the puzzle and cubes problems from Evan Tick's book, an hamiltonian graph problem and a naïve sorting resolution.

Benchmark	**YapTab**	**Yap Prolog**	**XSB Prolog**
9-queens	740	740(1.00)	1819(2.46)
cubes	210	210(1.00)	589(2.80)
ham	460	430(0.93)	1139(2.48)
nsort	390	370(0.95)	1101(2.82)
puzzle	2430	2120(0.87)	5819(2.39)
Average		(0.95)	(2.59)

Table 1: YapTab, Yap and XSB running times on a set of non tabled benchmarks.

Table 1 shows the base running times, in milliseconds, for YapTab, Yap Prolog and XSB Prolog for the set of non tabled benchmarks. In parentheses, it shows the performance of Yap and XSB over the YapTab running times. The results indicate that YapTab introduce, on average, an overhead of about 5% over standard Yap. These overheads are due to the handling of the freeze registers and the forward trail. Regarding XSB, the results show that, on average, YapTab is 2.59 times faster than XSB. This is almost due to the faster Yap engine.

To assess the performance of YapTab when running tabled programs and compare it with XSB, we wrote four versions of the following tabled program:

```
:- table path/3.

path(X,Y,[X,Y]) :- arc(X,Y).
path(X,Y,P) :- path(X,Z,P1), arc(Z,Y), insert_if_new(Y,P1,P).

insert_if_new(X,[],[X]).
insert_if_new(X,[H|T],[H|NT]) :- X \= H, insert_if_new(X,T,NT).
```

The four versions differ mainly in the number of nodes and graph topology, hence defining search spaces of varying complexity. In all versions, the query goal is to find the transitive closure of the given graph.

Table 2 shows the running times, in milliseconds, for YapTab, using batched (YapTab Batched) and local (YapTab Local) scheduling strategies, and XSB Prolog for the four benchmarks. The results show that YapTab Batched on average performs better than YapTab Local and XSB. In parentheses it shows the overheads of YapTab Local and XSB over YapTab Batched. The results obtained for batched and

Benchmark	YapTab Batched	YapTab Local	XSB Prolog
binary tree (depth 10)	180	230(1.27)	451(2.50)
chain (64 nodes)	130	160(1.23)	399(3.06)
cycle (64 nodes)	390	490(1.25)	1121(2.87)
grid (4x4 nodes)	1330	1450(1.09)	5740(4.31)
Average		(1.21)	(3.18)

Table 2: YapTab and XSB running times on a four version tabled benchmark.

local scheduling confirm previous results obtained for XSB using the same scheduling strategies [8]. The results also show that, for these programs, YapTab is about 3.18 times faster than current XSB. This is partly due to the faster Yap engine, as seen in table 1, and to the fact that XSB implements extra functionalities that are still lacking in YapTab and those may cause delays during execution.

In comparison with the results presented in [15], the running time for the binary tree benchmark was reduced by a factor of 2.5 just by employing hashing in the implementation of tries. This benchmark is a good example in which large chains of sibling nodes, representing alternative paths for a table entry, are formed and therefore using hashing to directly access each node in the chain gives much improvement in the execution time.

7 Conclusions

We have presented the design and implementation of YapTab, an extension of the Yap Prolog system that implements sequential tabling. Our system includes all the machinery required to execute programs with tabling in or-parallel. YapTab reuses the principles of XSB-Prolog's SLG-WAM engine, whilst innovating by separating the tabling suspension data in a single space, the dependency space, and by proposing a new completion detection algorithm not based on the intrinsically sequential completion stack.

Our first results are very encouraging. Overheads over standard Yap are low and performance in tabling benchmarks is quite satisfactory even when compared with the arguably more mature and complete XSB system.

We have obtained very initial timings for parallel execution on a shared memory PentiumPro machine. The results show significant speedups for a tabled application increasing up to the four processors and encourage us in our believe that tabling and parallelism may together contribute to increasing the range of applications for Logic Programming.

References

[1] Khayri A. M. Ali and Roland Karlsson. Scheduling speculative work in MUSE and performance results. *International Journal of Parallel Programming*, 21(6):449–476, December 1992.

[2] W. Chen and D. S. Warren. Query Evaluation under the Well Founded Semantics. In *Proceedings of PODS'93*, pages 168–179, 1993.

[3] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[4] L. Damas, V. S. Costa, R. Reis, and R. Azevedo. *YAP User's Guide and Reference Manual*. Centro de Informática da Universidade do Porto, 1989.

[5] B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. In *Proceedings of PLILP/ALP98*. Springer Verlag, September 1998.

[6] B. Demoen and K. Sagonas. CHAT: The Copy-Hybrid Approach to Tabling. *Lecture Notes in Computer Science*, 1551:106–121, 1999.

[7] J. Freire, R. Hu, T. Swift, and D. S. Warren. Exploiting Parallelism in Tabled Evaluations. In *Proceedings of PLILP'95*, pages 115–132, 1995.

[8] J. Freire, T. Swift, and D. S. Warren. Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In *Proceedings of PLILP'96*, pages 243–258. Springer-Verlag, Sep. 1996.

[9] Hai-Feng Guo and G. Gupta. A New Tabling Scheme with Dynamic Reordering of Alternatives. In *Workshop on Parallelism and Implementation Technology for (Constraint) Logic Languages*, July 2000.

[10] Robert A. Kowalski. *Logic for Problem Solving*. Elsevier North-Holland Inc., 1979.

[11] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Tabling Mechanisms for Logic Programs. In *Proceedings of ICLP'95*, pages 687–711. The MIT Press, June 1995.

[12] R. Rocha, F. Silva, and V. S. Costa. On Applying Or-Parallelism to Tabled Evaluations. In *Proceedings of the First International Workshop on Tabling in Logic Programming*, pages 33–45, Leuven, Belgium, June 1997.

[13] R. Rocha, F. Silva, and V. S. Costa. Or-Parallelism within Tabling. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages, PADL'99*, number 1551 in LNCS, pages 137–151, San Antonio, Texas, USA, January 1999. Springer-Verlag.

[14] R. Rocha, F. Silva, and V. S. Costa. YapOr: an Or-Parallel Prolog System Based on Environment Copying. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence, EPIA'99*, number 1695 in LNAI, pages 178–192, Évora, Portugal, September 1999. Springer-Verlag.

[15] R. Rocha, F. Silva, and V. S. Costa. YapTab: A Tabling Engine Designed to Support Parallelism. In *Proceedings of the 2nd Conference on Tabulation in Parsing and Deduction, TAPD'2000*, pages 77–87, Vigo, Spain, September 2000.

[16] 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, 1998.

[17] Terrance Swift and David S. Warren. An abstract machine for SLG resolution: definite programs. In Maurice Bruynooghe, editor, *Logic Programming - Proceedings of the 1994 International Symposium*, pages 633–652, Massachusetts Institute of Technology, 1994. The MIT Press.

[18] Paul Tarau, Koenraad De Bosschere, and Bart Demoen. On Delphi lemmas and other memoing techniques for deterministic logic programs. *Journal of Logic Programming*, 30(2):145–163, February 1997.

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

[20] Neng-Fa Zhou. Implementation of a Linear Tabling Mechanism. In *Proceedings of PADL'2000*, number 1753 in LNCS, pages 109–123. Springer-Verlag, January 2000.