Towards Multi-Threaded Local Tabling Using a Common Table Space

MIGUEL AREIAS and RICARDO ROCHA

CRACS & INESC TEC, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
(*e-mail:* {miguel-areias,ricroc}@dcc.fc.up.pt)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Multi-threading is currently supported by several well-known Prolog systems providing a highly portable solution for applications that can benefit from concurrency. When multi-threading is combined with tabling, we can exploit the power of higher procedural control and declarative semantics. However, despite the availability of both threads and tabling in some Prolog systems, the implementation of these two features implies complex ties to each other and to the underlying engine. Until now, XSB was the only Prolog system combining multi-threading with tabling. In XSB, tables may be either private or shared between threads. While thread-private tables are easier to implement, shared tables have all the associated issues of locking, synchronization and potential deadlocks. In this paper, we propose an alternative view to XSB's approach. In our proposal, each thread views its tables as private but, at the engine level, we use a *common table space* where tables are shared among all threads. We present three designs for our common table space approach: *No-Sharing (NS)* (similar to XSB's private tables), *Subgoal-Sharing (SS)* and *Full-Sharing (FS)*. The primary goal of this work was to reduce the memory usage for the table space but, our experimental results, using the YapTab tabling system with a local evaluation strategy, show that we can also achieve significant reductions on running time.

KEYWORDS: Tabling, Multi-Threading, Implementation.

1 Introduction

Tabling (Chen and Warren 1996) is a recognized and powerful implementation technique that overcomes some limitations of traditional Prolog systems in dealing with recursion and redundant sub-computations. In a nutshell, tabling is a refinement of SLD resolution that stems from one simple idea: save intermediate answers from past computations so that they can be reused when a *similar call* appears during the resolution process[1]. Tabling based models are able to reduce the search space,

[1] We can distinguish two main approaches to determine similarity between tabled subgoal calls: *variant-based tabling* and *subsumption-based tabling*.

avoid looping, and always terminate for programs with the *bounded term-size property* (Chen and Warren 1996). Work on tabling, as initially implemented in the XSB system (Sagonas and Swift 1998), proved its viability for application areas such as natural language processing, knowledge based systems, model checking, program analysis, among others. Currently, tabling is widely available in systems like XSB, Yap, B-Prolog, ALS-Prolog, Mercury and Ciao.

Nowadays, the increasing availability of computing systems with multiple cores sharing the main memory is already a standardized, high-performance and viable alternative to the traditional (and often expensive) shared memory architectures. The number of cores per processor is expected to continue to increase, further expanding the potential for taking advantage of multi-threading support. The ISO Prolog multi-threading standardization proposal (Moura 2008) is currently implemented in several Prolog systems including XSB, Yap, Ciao and SWI-Prolog, providing a highly portable solution given the number of operating systems supported by these systems. Multi-threading in Prolog is the ability to concurrently perform multiple computations, in which each computation runs independently but shares the database (clauses).

When multi-threading is combined with tabling, we have the best of both worlds, since we can exploit the combination of higher procedural control with higher declarative semantics. In a multi-threaded tabling system, tables may be either private or shared between threads. While thread-private tables are easier to implement, shared tables have all the associated issues of locking, synchronization and potential deadlocks. Here, the problem is even more complex because we need to ensure the correctness and completeness of the answers found and stored in the shared tables. Thus, despite the availability of both threads and tabling in Prolog compilers such as XSB, Yap, and Ciao, the implementation of these two features such that they work together seamlessly implies complex ties to one another and to the underlying engine. Until now, XSB was the only system combining tabling with multi-threading, for both private and shared tables (Marques and Swift 2008; Swift and Warren 2012). For shared tables, XSB uses a semi-naive approach that, when a set of subgoals computed by different threads is mutually dependent, then a *usurpation operation* (Marques 2007; Marques et al. 2010) synchronizes threads and a single thread assumes the computation of all subgoals, turning the remaining threads into consumer threads.

The basis for our work is also on multi-threaded tabling using private tables, but we propose an alternative view to XSB's approach. In our proposal, each thread has its own tables, i.e., from the thread point of view the tables are private, but at the engine level we use a *common table space*, i.e., from the implementation point of view the tables are shared among all threads. We present three designs for our common table space approach: *No-Sharing (NS)* (similar to XSB with private tables), *Subgoal-Sharing (SS)* and *Full-Sharing (FS)*. Experimental results, using the YapTab tabling system (Rocha et al. 2005) with a local evaluation strategy (Freire et al. 1995), show that the FS design can achieve significant reductions on memory usage and on execution time, compared to the NS and SS designs, for a set of worst case scenarios where all threads start with the same query goal.

The remainder of the paper is organized as follows. First, we describe YapTab's table space organization and XSB's approach for multi-threaded tabling. Next, we introduce our three designs and discuss important implementation details. We then present some experimental results and outline some conclusions.

2 Basic Concepts

In this section, we introduce some background needed for the following sections. We begin by describing the actual YapTab's table space organization, and then we briefly present XSB's approach for supporting multi-threaded tabling.

2.1 YapTab's Table Space Organization

The basic idea behind tabling is straightforward: programs are evaluated by storing answers for tabled subgoals in an appropriate data space, called the *table space*. Similar calls to tabled subgoals are not re-evaluated against the program clauses, instead they are resolved by consuming the answers already stored in their table entries. During this process, as further new answers are found, they are stored in their tables and later returned to all similar calls.

A critical component in the implementation of an efficient tabling system is thus the design of the data structures and algorithms to access and manipulate tabled data. The most successful data structure for tabling is *tries* (Ramakrishnan et al. 1999). Tries are trees in which common prefixes are represented only once. The trie data structure provides complete discrimination for terms and permits look up and possibly insertion to be performed in a single pass through a term, hence resulting in a very efficient and compact data structure for term representation. Figure 1 shows the general table space organization for a tabled predicate in YapTab.

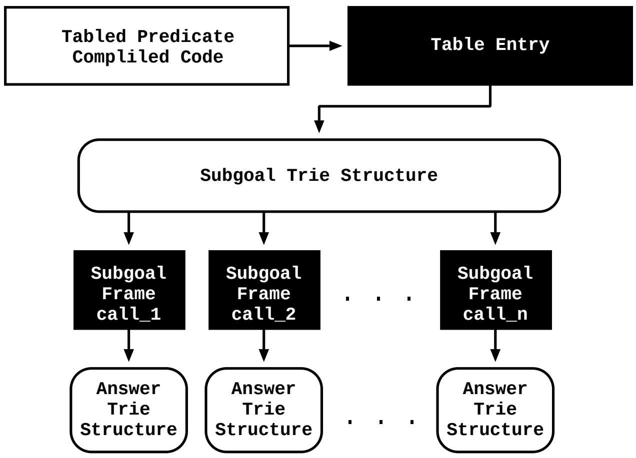

Fig. 1. YapTab's table space organization

At the entry point we have the *table entry* data structure. This structure is allocated when a tabled predicate is being compiled, so that a pointer to the table entry can be included in its compiled code. This guarantees that further calls to the predicate will access the table space starting from the same point. Below the table entry, we have the *subgoal trie structure*. Each different tabled subgoal call to the predicate at hand corresponds to a unique path through the subgoal trie structure, always starting from the table entry, passing by several subgoal trie data units, the *subgoal trie nodes*, and reaching a leaf data structure, the *subgoal frame*. The subgoal frame stores additional information about the subgoal and acts like an entry point to the *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.

2.2 XSB's Approach to Multi-Threaded Tabling

XSB offers two types of models for supporting multi-threaded tabling: *private tables* and *shared tables* (Swift and Warren 2012).

For private tables, each thread keeps its own copy of the table space. On one hand, this avoids concurrency over the tables but, on the other hand, the same table can be computed by several threads, thus increasing the memory usage necessary to represent the table space.

For shared tables, the running threads store only once the same table, even if multiple threads use it. This model can be viewed as a variation of the *table-parallelism* proposal (Freire et al. 1995), where a tabled computation can be decomposed into a set of smaller sub-computations, each being performed by a different thread. Each tabled subgoal is computed independently by the first thread calling it, the *generator thread*, and each generator is the sole responsible for fully exploiting and obtaining the complete set of answers for the subgoal. Similar calls by other threads are resolved by consuming the answers stored by the generator thread.

In a tabled evaluation, 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 determined by the evaluation strategy. The two most successful strategies are *batched evaluation* and *local evaluation* (Freire et al. 1996). Batched evaluation 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 evaluation 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 at hand were resolved.

Based on these two strategies, XSB supports two types of concurrent evaluations: *concurrent local evaluation* and *concurrent batched evaluation*. In the concurrent local evaluation, similar calls by other threads are resolved by consuming the answers stored by the generator thread, but a consumer thread suspends execution until the table is completed. In the concurrent batched evaluation, new answers are consumed as they are found, leading to more complex dependencies between threads. In both evaluation strategies, when a set of subgoals computed by different threads is mutually dependent, then a *usurpation operation* (Marques et al. 2010) synchronizes threads and a single thread assumes the computation of all subgoals, turning the remaining threads into consumer threads.

3 Our Approach

Yap implements a SWI-Prolog compatible multi-threading library (Wielemaker 2003). Like in SWI-Prolog, Yap's threads have their own execution stacks and only share the code area where predicates, records, flags and other global non-backtrackable data are stored. Our approach for multi-threaded tabling is still based on this idea in which each computational thread runs independently. This means that each tabled evaluation depends only on the computations being performed by the thread itself, i.e., there isn't the notion of being a consumer thread since, from each thread point of view, a thread is always the generator for all of its subgoal calls. We next introduce the three alternative designs for our approach: *No-Sharing (NS)*, *Subgoal-Sharing (SS)* and *Full-Sharing (FS)*. In what follows, we assume a local evaluation strategy.

3.1 No-Sharing

The starting point of our work is the situation where each thread allocates fully private tables for each new subgoal called during its computation. Figure 2 shows the configuration of the table space if several different threads call the same tabled subgoal *call_i*. One can observe that the table entry data structure still stores the common information for the predicate (such as the arity or the evaluation strategy), and then each thread t has its own cell T_t inside a *bucket array* which points to the private data structures. The subgoal trie structure, the subgoal frames and the answer trie structures are private to each thread and they are removed when the thread finishes execution.

The memory usage for this design for a particular tabled predicate P, assuming that all running threads NT have completely evaluated the same number NS of subgoals, is $sizeof(TE_P) + sizeof(BA_P) + [sizeof(STS_P) + [sizeof(SF_P) + sizeof(ATS_P)] * NS] * NT$, where TE_P and BA_P represent the common table entry and bucket array data structures, STS_P and ATS_P represent the nodes inside the subgoal and answer trie structures, and SF_P represents the subgoal frames.

3.2 Subgoal-Sharing

In our second design, the threads share part of the table space. Figure 3 shows again the configuration of the table space if several different threads call the same tabled subgoal *call_i*. In this design, the subgoal trie structure is now shared among the threads and the leaf data structures in each subgoal trie path, instead of pointing to a subgoal frame, they now point to a bucket array. Each thread t has its own

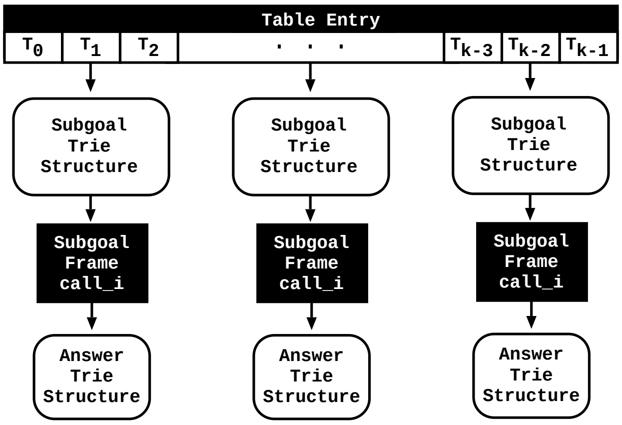

Fig. 2. Table space organization for the NS design

cell T_t inside the bucket array which then points to a private subgoal frame and answer trie structure.

In this design, concurrency among threads is restricted to the allocation of new entries on the subgoal trie structure. Whenever a thread finishes execution, its private structures are removed, but the shared part remains present as it can be in use or be further used by other threads. Assuming again that all running threads NT have completely evaluated the same number NS of subgoals, the memory usage for this design for a particular tabled predicate P is $sizeof(TE_P) + sizeof(STS_P) + [sizeof(BA_P) + [sizeof(SF_P) + sizeof(ATS_P)] * NT] * NS$, where BA_P represents the bucket array pointing to the private data structures.

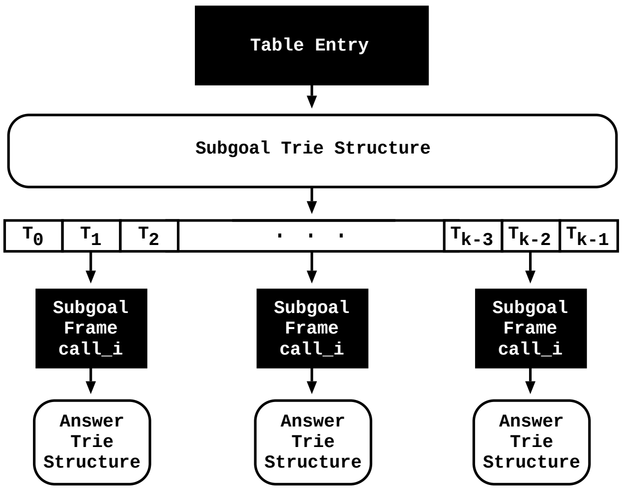

Fig. 3. Table space organization for the SS design

3.3 Full-Sharing

Our third design is the most sophisticated among three. Figure 4 shows its table space organization if considering several different threads calling the same tabled subgoal *call_i*. In this design, part of the subgoal frame information (the *subgoal entry* data structure in Fig. 4) and the answer trie structure are now also shared among all threads. The previous subgoal frame data structure was split into two: the *subgoal entry* stores common information for the subgoal call (such as the pointer to the shared answer trie structure); the remaining information (the *subgoal frame* data structure in Fig. 4) remains private to each thread.

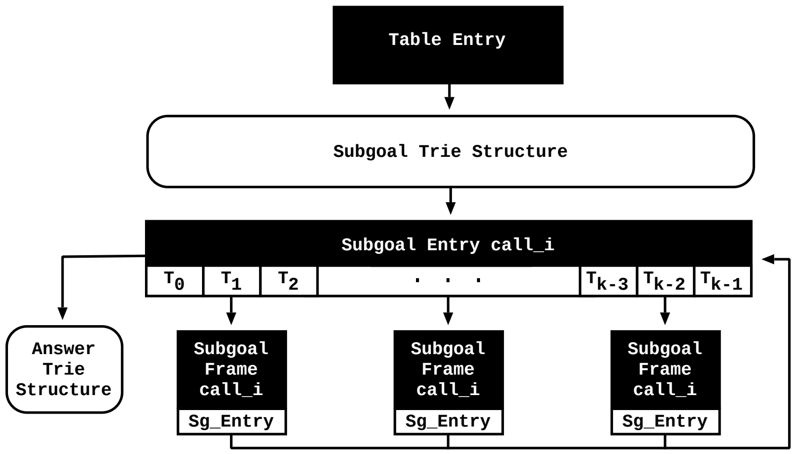

Fig. 4. Table space organization for the FS design

The subgoal entry also includes a bucket array, in which each cell T_t points to the private subgoal frame of each thread t. The private subgoal frames include an extra field which is a back pointer to the common subgoal entry. This is important because, with that, we can keep unaltered all the tabling data structures that point to subgoal frames. To access the private information on the subgoal frames there is no extra cost (we still use a direct pointer), and only for the common information on the subgoal entry we pay the extra cost of following an indirect pointer.

Again, assuming that all running threads NT have completely evaluated the same number NS of subgoals, the memory usage for this design for a particular tabled predicate P is $sizeof(TE_P) + sizeof(STS_P) + [sizeof(SE_P) + sizeof(BA_P) + sizeof(ATS_P) + sizeof(SF_P) * NT] * NS$, where SE_P and SF_P represent, respectively, the shared subgoal entry and the private subgoal frame data structures.

In this design, concurrency among threads now also includes the access to the subgoal entry data structure and the allocation of new entries on the answer trie structures. However, this latest design has two major advantages. First, memory usage is reduced to a minimum. The only memory overhead, when compared with a single threaded evaluation, is the bucket array associated with each subgoal entry, and apart from the split on the subgoal frame data structure, all the remaining structures remain unchanged. Second, since threads are sharing the same answer trie structures, answers inserted by a thread for a particular subgoal call are automatically made available to all other threads when they call the same subgoal. As we will see in section 5, this can lead to reductions on the execution time.

4 Implementation

In this section, we discuss some low level details regarding the implementation of the three designs. We begin by describing the expansion of the table space to efficiently support multiple threads, next we discuss the locking schemes used to ensure mutual exclusion over the table space, and then we discuss how the most important tabling operations were extended for multi-threaded tabling support.

4.1 Efficient Support for Multiple Threads

Our proposals already include support for any number of threads working on the same table. For that, we extended the original table data structures with bucket arrays. For example, for the NS design, we introduced a bucket array in the table entry (see Fig. 2), for the SS design, the bucket array follows a subgoal trie path (see Fig. 3), and for the FS design, the bucket array is part of the new subgoal entry data structure (see Fig. 4).

These bucket arrays contain as much entry cells as the maximum number of threads that can be created in Yap (currently 1024). However, in practice, this solution is highly inefficient and memory consuming, as we must always allocate this huge bucket array even when only one thread will use it.

To solve this problem, we introduce a kind of inode pointer structure, where the bucket array is split into direct bucket cells and indirect bucket cells. The direct bucket cells are used as before and the indirect bucket cells are only allocated as needed. This new structure applies to all bucket arrays in the three designs. Figure 5 shows an example on how this new structure is used in the FS design.

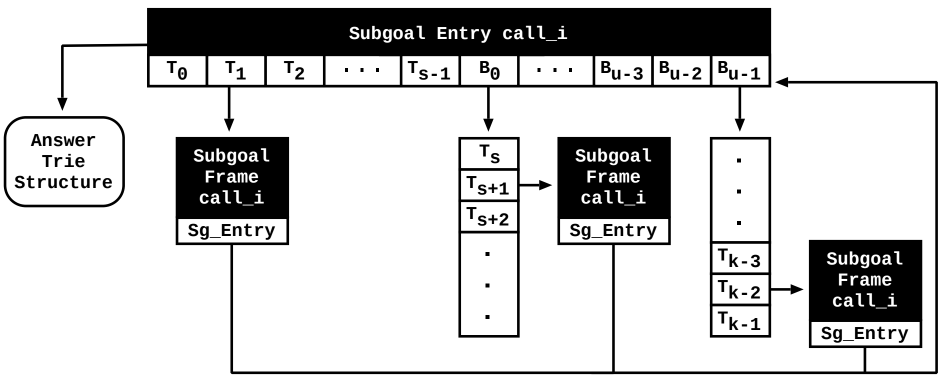

Fig. 5. Using direct and indirect bucket cells in the FS design

A bucket array has now two operating modes. If it is being used by a thread with an identification number t lower than a default starting size s (32 in our implementation), then the buckets are used as before, meaning that the entry cell T_t still points to the private information of the corresponding thread. But now, if a thread with an identification number equal or higher than s appears, the thread is mapped into one of the u undirected buckets (entry cells B_0 until B_{u-1} in Fig. 5), which becomes a pointer to a second level bucket array that will now contain the entry cells pointing to the private thread information. Given a thread t ($t \geq s$), its index in the first and in the second level bucket arrays is given by the division and the remainder of $(t - s)$ by u, respectively.

4.2 Table Locking Schemes

Remember that the SS and FS designs introduce concurrency among threads when accessing shared resources of the table space. Here, we discuss how we use locking schemes to ensure mutual exclusion when manipulating such shared resources.

We can say that there are two critical issues that determine the efficiency of a locking scheme. One is the *lock duration*, that is, the amount of time a data structure is locked. The other is the *lock grain*, that is, the amount of data structures that are protected through a single lock request. It is the balance between lock duration and lock grain that compromises the efficiency of different locking schemes.

The or-parallel tabling engine of Yap (Rocha et al. 2005) already implements four alternative locking schemes to deal with concurrent table accesses: the *Table Lock at Entry Level* (TLEL) scheme, the *Table Lock at Node Level* (TLNL) scheme, the *Table Lock at Write Level* (TLWL) scheme, and the *Table Lock at Write Level - Allocate Before Check* (TLWL-ABC) scheme. Currently, the first three are also available on our multi-threaded engine. However, in what follows, we will focus our attention only on the TLWL locking scheme, since its performance showed to be clearly better than the other two (Rocha et al. 2004).

The TLWL scheme allows a *single writer* per chain of sibling nodes that represent alternative paths from a common parent node (see Fig. 6). This means that each node in the subgoal/answer trie structures is expanded with a *locking field* that, once activated, synchronizes updates to the chain of sibling nodes, meaning that only one thread at a time can be inserting a new child node starting from the same parent node.

With the TLWL scheme, the process of check/insert a term t in a chain of sibling nodes works as follows. Initially, the working thread starts by searching for t in the available child nodes (the non-critical region) and only if the term is not found, it will enter the critical region in order to insert it on the chain. At that point, it waits until the lock be available, which can cause a delay proportional to the number of threads that are accessing the same critical region at the same time.

In order to reduce the lock duration to a minimum, we have improved the original TLWL scheme to use *trylocks* instead of traditional locks. With trylocks, when a thread fails to get access to the lock, instead of waiting, it returns to the non-critical region, i.e., it traverses the newly inserted nodes, if any, checking if t was, in the meantime, inserted in the chain by another thread. If t is not found, the process

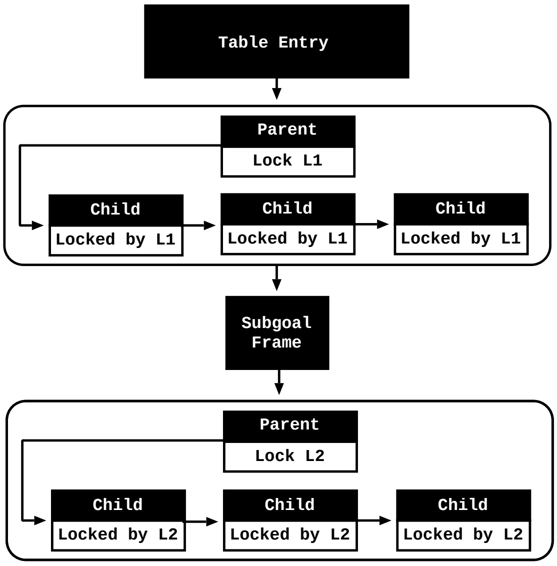

Fig. 6. The TLWL locking scheme

repeats until the thread get access to the lock, in order to insert t, or until t be found. Figure 7 shows the pseudo-code for the implementation of this procedure using the TLWL scheme with trylocks.

```
trie_node_check_insert(term T, parent trie node P)

last_child = NULL  // used to mark the last child to be checked
do {  // non-critical region
  first_child = TrNode_first_child(P)
  child = first_child
  while (child != last_child)  // traverse the chain of sibling nodes ...
    if (TrNode_term(child) == T)  // ... searching for T
      return child
    child = TrNode_sibling(child)
  last_child = first_child
} while (! trylock(TrNode_lock(P)))
// critical region, lock is set
child = TrNode_first_child(P)
while (child != last_child)  // traverse the chain of sibling nodes ...
  if (TrNode_entry(child) == T)  // ... searching for T
    unlock(TrNode_lock(P))  // unlocking before return
    return child
  child = TrNode_sibling(child)
// create a new node to represent T
child = new_trie_node(T)
TrNode_sibling(child) = TrNode_first_child(P)
TrNode_first_child(P) = child
unlock(TrNode_lock(P))  // unlocking before return
return child
```

Fig. 7. Pseudo-code for the trie node check/insert operation

Initially, the procedure traverses the chain of sibling nodes, that represent alternative paths from the given parent node `P`, and checks for one representing the given term `T`. If such a node is found (line 6) then execution is stopped and the node returned (line 7). Otherwise, this process repeats (lines 3 to 10) until the working thread gets access to the lock field of the parent node `P`. In each round, the `last_child` auxiliary variable marks the last node to be checked. It is initially set to `NULL` (line 1) and then updated, at the end of each round, to the new first child of the current round (line 9).

Otherwise, the thread gets access to the lock and enters the critical region (lines 12 to 23). Here, it first checks if T was, in the meantime, inserted in the chain by another thread (lines 13 to 17). If this is not the case, then a new trie node representing T is allocated (line 19) and inserted in the beginning of the chain (lines 20 and 21). The procedure then unlocks the parent node (line 22) and ends returning the newly allocated child node (line 23).

4.3 Tabling Operations

In YapTab, programs using tabling are compiled to include *tabling operations* that enable the tabling engine to properly schedule the evaluation process. One of the most important operations is the *tabled subgoal call*. This operation inspects the table space looking for a subgoal similar to the current subgoal being called. If a similar subgoal is found, then the corresponding subgoal frame is returned. Otherwise, if no such subgoal exists, it inserts a new path into the subgoal trie structure, representing the current subgoal, and allocates a new subgoal frame as a leaf of the new inserted path. Figure 8 shows how we have extended the tabled subgoal call operation for multi-threaded tabling support.

```
tabled_subgoal_call(table entry TE, subgoal call SC, thread id TI)

root = get_subgoal_trie_root_node(TE, TI)
leaf = check_insert_subgoal_trie(root, SC)
if (NS_design)
  sg_fr = get_subgoal_frame(leaf)
  if (not_exists(sg_fr))
    sg_fr = new_subgoal_frame(leaf)
  return sg_fr
else if (SS_design)
  bucket = get_bucket_array(leaf)
  if (not_exists(bucket))
    bucket = new_bucket_array(leaf)
else if (FS_design)
  sg_entry = get_subgoal_entry(leaf)
  if (not_exists(sg_entry))
    sg_entry = new_subgoal_entry(leaf)
  bucket = get_bucket_array(sg_entry)
sg_fr = get_subgoal_frame(bucket)
if (not_exists(sg_fr))
  sg_fr = new_subgoal_frame(bucket)
return sg_fr
```

Fig. 8. Pseudo-code for the tabled subgoal call operation

The procedure receives three arguments: the table entry for the predicate at hand (`TE`), the current subgoal being called (`SC`), and the *id* of the working thread (`TI`). The `NS_design`, `SS_design` and `FS_design` macros define which table design is enabled.

The procedure starts by getting the root trie node for the subgoal trie structure that matches with the given thread *id* (line 1). Next, it checks/inserts the given `SC` into the subgoal trie structure, which will return the leaf node for the path representing `SC` (line 2). Then, if the NS design is enable, it uses the leaf node to obtain the corresponding subgoal frame (line 4). If the subgoal call is new, no subgoal frame still exists and a new one is created (line 6). Then, the procedure ends by returning the subgoal frame (line 7). This code sequence corresponds to the usual tabled subgoal call operation.

Otherwise, for the SS design, it follows the leaf node to obtain the bucket array (line 9). If the subgoal call is new, no bucket exists and a new one is created (line 11). On the other hand, for the FS design, it follows the leaf node to obtain the subgoal entry (line 13) and, again, if the subgoal call is new, no subgoal entry exists and a new one is created (line 15). From the subgoal entry, it then obtains the bucket array (line 16).

Finally, for both SS and FS designs, the bucket array is then used to obtain the subgoal frame (line 17) and one more time, if the given subgoal call is new, a new subgoal frame needs to be created (line 19). The procedure ends by returning the subgoal frame (line 20). Note that, for the sake of simplicity, we omitted some of the low level details in manipulating the bucket arrays, such as in computing the bucket cells or in expanding the indirect bucket cels.

Another important tabling operation is the *new answer*. This operation checks whether a newly found answer is already in the corresponding answer trie structure and, if not, inserts it. Remember from section 2.2 that, with local evaluation, the new answer operation always fails, regardless of the answer being new or repeated, and that, with batched evaluation, when new answers are inserted the evaluation should continue, failing otherwise. With the FS design, the answer trie structures are shared. Thus, when several threads are inserting answers in the same trie structure, it may be not possible to determine when an answer is new or repeated for a certain thread. This is the reason why the FS design can be only safely used with local evaluation. We are currently studying how to bypass this constraint in order to also support the FS design with batched evaluation.

5 Experimental Results

In this section, we present some experimental results obtained for the three proposed table designs using the TLWL scheme with traditional locks and with trylocks. The environment for our experiments was a machine with 4 Six-Core AMD Opteron (tm) Processor 8425 HE (24 cores in total) with 64 GBytes of main memory and running the Linux kernel 2.6.34.9-69.fc13.x86_64 with Yap 6.3. To put our results in perspective, we make a comparison with the multi-threaded implementation of XSB, version 3.3.6, using thread-private tables.

We used five sets of benchmarks. The **Large Joins** and **WordNet** sets were obtained from the OpenRuleBench project[2]; the **Model Checking** set includes three different specifications and transition relation graphs usually used in model checking applications; the **Path Left** and **Path Right** sets implement two recursive definitions of the well-known *path*/2 predicate, that computes the transitive closure in a graph, using several different configurations of *edge*/2 facts (Fig. 9 shows an example for each configuration). We experimented the **BTree** configuration with depth 18, the **Pyramid** and **Cycle** configurations with depth 2000 and the **Grid** configuration with depth 35. All benchmarks find all the solutions for the problem.

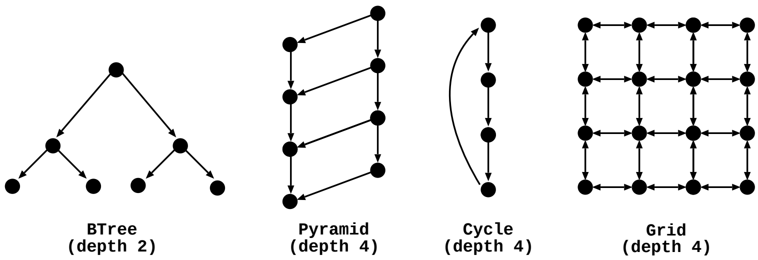

Fig. 9. Edge configurations

Table 1 shows the execution time, in milliseconds, when running 1 working thread with local scheduling, for our three table designs, using the TLWL scheme with traditional locks (columns **NS** and **FS**) and with trylocks (columns $\mathbf{SS}_T$ and $\mathbf{FS}_T$)[3], and for XSB. In parentheses, it also shows the respective overhead ratios when compared with the NS design. The running times are the average of five runs. The ratios marked with *n.c.* for XSB mean that we are *not considering* them in the average results (we opted to do that since they correspond to running times much higher than the other designs, which may suggest that something was wrong).

One can observe that, on average, the SS design and the XSB implementation have a lower overhead ratio (around 10%) than the FS and FS_T designs (around 20%). For the SS and, mainly, for the FS approaches, this can be explained by the higher complexity of the implementation and, in particular, by the cost incurred with the extra code necessary to implement the TLWL locking scheme. Note that, even with a single working thread, this code has to be executed.

Starting from these base results, Table 2 shows the overhead ratios, when compared with the NS design with 1 thread, for our table designs and XSB, when running 16 and 24 working threads (the results are the average of five runs).

In order to create a worst case scenario that stresses the trie data structures, we ran all threads starting with the same query goal. By doing this, it is expected that they will access the table space, to check/insert for subgoals and answers, at

[2] Available from `http://rulebench.projects.semwebcentral.org`. We also have results for the other benchmarks proposed by the OpenRuleBench project (Liang et al. 2009) but, due to lack of space, here we only include these two sets.

[3] In general, for this set of benchmarks, the SS design presented similar results with traditional locks and with trylocks and, thus, here we only show the results with trylocks.

Table 1. *Execution time, in milliseconds, when running 1 working thread with local scheduling, for the NS, SS_T, FS and FS_T designs and for XSB, and the respective overhead ratios when compared with the NS design*

Bench	NS	$\mathbf{SS}_T$	FS	$\mathbf{FS}_T$	XSB
Large Joins					
Join2	3,419	3,418 (1.00)	3,868 (1.13)	3,842 (1.12)	3,444 (1.01)
Mondial	730	725 (0.99)	856 (1.17)	887 (1.21)	1,637 (2.24)
	Average	(1.00)	(1.15)	(1.17)	(1.62)
WordNet					
Clusters	789	990 (1.26)	981 (1.24)	982 (1.24)	549 (0.70)
Hypo	1,488	1,671 (1.12)	1,728 (1.16)	1,720 (1.16)	992,388 (*n.c.*)
Holo	694	902 (1.30)	881 (1.27)	884 (1.27)	425 (0.61)
Hyper	1,386	1,587 (1.15)	1,565 (1.13)	1,576 (1.14)	1,320 (0.95)
Tropo	598	784 (1.31)	763 (1.28)	762 (1.27)	271 (0.45)
Mero	678	892 (1.32)	869 (1.28)	864 (1.27)	131,830 (*n.c.*)
	Average	(1.24)	(1.23)	(1.23)	(0.68)
Model Checking					
IProto	2,517	2,449 (0.97)	2,816 (1.12)	2,828 (1.12)	3,675 (1.46)
Leader	3,726	3,800 (1.02)	3,830 (1.03)	3,897 (1.05)	10,354 (2.78)
Sieve	23,645	24,402 (1.03)	24,479 (1.04)	25,201 (1.07)	27,136 (1.15)
	Average	(1.01)	(1.06)	(1.08)	(1.80)
Path Left					
BTree	2,966	2,998 (1.01)	3,826 (1.29)	3,864 (1.30)	2,798 (0.94)
Pyramid	3,085	3,159 (1.02)	3,256 (1.06)	3,256 (1.06)	2,928 (0.95)
Cycle	3,828	3,921 (1.02)	3,775 (0.99)	3,798 (0.99)	3,357 (0.88)
Grid	1,743	1,791 (1.03)	2,280 (1.31)	2,293 (1.32)	2,034 (1.17)
	Average	(1.02)	(1.16)	(1.17)	(0.98)
Path Right					
BTree	4,568	5,048 (1.11)	5,673 (1.24)	5,701 (1.25)	3,551 (0.78)
Pyramid	2,520	2,531 (1.00)	3,664 (1.45)	3,673 (1.46)	2,350 (0.93)
Cycle	2,761	2,773 (1.00)	3,994 (1.45)	3,992 (1.45)	2,817 (1.02)
Grid	2,109	2,110 (1.00)	3,097 (1.47)	3,117 (1.48)	2,462 (1.17)
	Average	(1.03)	(1.40)	(1.41)	(0.97)
	Total Average	(1.08)	(1.21)	(1.22)	(1.12)

similar times, thus causing a huge stress on the same critical regions. In particular, for this set of benchmarks, this will be specially the case for the answer tries (and thus, for the FS and FS_T designs), since the number of answers clearly exceeds the number of subgoals. Analyzing the general picture of Table 2, one can observe that, on average, the NS and SS_T designs show very poor results for 16 and 24 threads. In particular, these bad results are more clear in the benchmarks that allocate a higher

Table 2. *Overhead ratios, when compared with the NS design with 1 thread, for the NS, SS_T, FS and FS_T designs and for XSB, when running 16 and 24 working threads with local scheduling (best ratios are in bold)*

Bench	16 Threads					24 Threads				
	NS	**SS$_T$**	**FS**	**FS$_T$**	**XSB**	**NS**	**SS$_T$**	**FS**	**FS$_T$**	**XSB**
Large Joins										
Join2	7.96	8.05	**3.14**	**3.14**	5.74	24.78	24.84	3.77	**3.76**	8.64
Mondial	**1.05**	1.07	1.46	1.53	2.43	**1.13**	**1.13**	1.60	1.64	2.53
Average	4.51	4.56	**2.30**	2.34	4.08	12.96	12.98	**2.68**	2.70	5.58
WordNet										
Clusters	6.29	5.61	3.92	3.94	**2.82**	12.23	8.67	**4.52**	4.55	4.87
Hypo	5.33	5.09	4.56	**2.99**	*n.c.*	9.20	8.33	5.21	**4.15**	*n.c.*
Holo	6.15	5.41	3.73	3.72	**2.77**	10.92	9.87	4.67	4.55	**4.37**
Hyper	8.03	7.65	3.57	**2.94**	4.26	21.34	16.82	4.59	**3.34**	7.14
Tropo	6.03	4.96	3.93	3.95	**2.93**	13.46	8.44	5.64	5.68	**4.69**
Mero	4.90	4.92	3.90	**3.71**	*n.c.*	8.93	7.96	4.59	**4.44**	*n.c.*
Average	6.12	5.61	3.93	3.54	**3.19**	12.68	10.02	4.87	**4.45**	5.27
Model Checking										
IProto	4.15	4.20	1.60	**1.55**	1.92	7.16	7.31	1.71	**1.63**	2.14
Leader	**1.02**	1.04	1.05	1.07	2.80	**1.02**	1.04	1.05	1.07	2.79
Sieve	**1.01**	1.04	1.05	1.08	1.15	**1.02**	1.04	1.06	1.08	1.15
Average	2.06	2.09	1.24	**1.23**	1.95	3.07	3.13	1.27	**1.26**	2.03
Path Left										
BTree	9.85	9.78	6.88	**4.81**	5.11	25.65	25.42	8.03	**5.97**	8.09
Pyramid	7.67	7.79	3.74	**3.40**	4.40	24.92	24.88	5.86	**4.48**	7.02
Cycle	7.32	7.38	3.73	**3.25**	4.36	22.39	23.05	5.95	**4.08**	6.99
Grid	5.99	6.00	3.77	3.15	**2.41**	19.82	19.80	4.65	**4.46**	5.30
Average	7.71	7.74	4.53	**3.65**	4.07	23.20	23.29	6.12	**4.75**	6.85
Path Right										
BTree	13.82	13.13	10.57	**5.54**	6.33	29.53	27.36	10.16	**6.76**	10.38
Pyramid	17.09	17.00	14.85	8.15	**5.94**	46.25	45.31	10.86	10.42	**10.31**
Cycle	17.96	18.17	17.05	8.36	**6.63**	47.89	47.60	11.49	**10.76**	10.99
Grid	9.52	9.48	7.13	5.53	**3.75**	26.58	27.80	7.50	6.96	**6.41**
Average	14.60	14.44	12.40	6.90	**5.66**	37.56	37.02	10.00	**8.73**	9.52
Total Average	7.43	7.25	5.24	**3.78**	3.87	18.64	17.72	5.42	**4.73**	6.11

number of trie nodes. The explanation for this is the fact that we are using Yap's memory allocator, that is based on Linux system's *malloc*, which can be a problem, when making a lot of memory requests, since these requests require synchronization at the low level implementation.

For the FS and FS_T designs, the results are significantly better and, in particular for FS_T, the results show that its trylock implementation is quite effective in reducing contention and, consequently, the running times for most of the experiments. Regarding XSB, for 16 threads, the results are similar to the FS_T design (3.87 for XSB and 3.78 for FS_T, on average) but, for 24 threads, the FS_T is noticeable better (6.11 for XSB and 4.73 for FS_T, on average). These results are more important since XSB shows base execution times (with 1 thread) lower than FS_T (please revisit Table 1) and since FS_T also pays the cost of using Yap's memory allocator based on Linux system's *malloc*.

We can say that there are two main reasons for the good results of the FS design. The first, and most important, is that the FS design can effectively reduce the memory usage of the table space, almost linearly in the number of threads[4], which has the collateral effect of also reducing the impact of Yap's memory allocator. The second reason is that, since threads are sharing the same answer trie structures, answers inserted by a thread are automatically made available to all other threads when they call the same subgoal. We observed that this collateral effect can also lead to unexpected reductions on the execution time.

6 Conclusions

We have presented a new approach to multi-threaded tabled evaluation of logic programs using a local evaluation strategy. In our proposal, each thread views its tables as private but, at the engine level, the tables are shared among all threads. The primary goal of our work was, in fact, to reduce the memory table space but, our experimental results, showed that we can also significantly reduce the running times. Since our implementation achieved very encouraging results on worst case scenario tests, it should keep at least the same level of efficiency on any other tests. Moreover, we believe that there is still considerable space for improvements, mainly related to the low-level issues of Yap's memory allocator for multi-threaded support. The goal would be to implement strategies that pre-allocate bunches of memory in order to minimize the performance degradation that the system suffers, when it is exposed to simultaneous memory requests made by multiple threads. Further work will also include extending the FS design to support batched evaluation.

Acknowledgments

This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Technology) within projects HORUS (PTDC/EIA-EIA/100897/2008) and LEAP (PTDC/EIA-CCO/112158/2009). Miguel Areias is funded by the FCT grant SFRH/BD/69673/2010.

[4] We have experimental results confirming the memory usage formulas introduced on section 3 but, due to lack of space, we are not including them here.

References

Chen, W. and Warren, D. S. 1996. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM 43,* 1, 20–74.

Freire, J., Hu, R., Swift, T., and Warren, D. S. 1995. Exploiting Parallelism in Tabled Evaluations. In *International Symposium on Programming Languages: Implementations, Logics and Programs*. Number 982 in LNCS. Springer-Verlag, 115–132.

Freire, J., Swift, T., and Warren, D. S. 1996. 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, 243–258.

Liang, S., P.Fodor, Wan, H., and M.Kifer. 2009. OpenRuleBench: An Analysis of the Performance of Rule Engines. In *Internacional World Wide Web Conference*. ACM Press, 601–610.

Marques, R. 2007. Concurrent Tabling: Algorithms and Implementation. Ph.D. thesis, Department of Computer Science, New University of Lisbon.

Marques, R. and Swift, T. 2008. Concurrent and Local Evaluation of Normal Programs. In *International Conference on Logic Programming*. Number 5366 in LNCS. Springer-Verlag, 206–222.

Marques, R., Swift, T., and Cunha, J. C. 2010. A Simple and Efficient Implementation of Concurrent Local Tabling. In *International Symposium on Practical Aspects of Declarative Languages*. Number 5937 in LNCS. Springer-Verlag, 264–278.

Moura, P. 2008. ISO/IEC DTR 13211–5:2007 Prolog Multi-threading Predicates. Available from `http://logtalk.org/plstd/threads.pdf`.

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

Rocha, R., Silva, F., and Santos Costa, V. 2004. Concurrent Table Accesses in Parallel Tabled Logic Programs. In *International Euro-Par Conference*. Number 3149 in LNCS. Springer-Verlag, 662–670.

Rocha, R., Silva, F., and Santos Costa, V. 2005. On applying or-parallelism and tabling to logic programs. *Theory and Practice of Logic Programming 5,* 1 & 2, 161–205.

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

Swift, T. and Warren, D. S. 2012. XSB: Extending Prolog with Tabled Logic Programming. *Theory and Practice of Logic Programming 12,* 1 & 2, 157–187.

Wielemaker, J. 2003. Native Preemptive Threads in SWI-Prolog. In *International Conference on Logic Programming*. Number 2916 in LNCS. Springer-Verlag, 331–345.