An Efficient and Scalable Memory Allocator for Multithreaded Tabled Evaluation of Logic Programs

Miguel Areias and Ricardo Rocha
CRACS & INESC TEC, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
Email: {miguel-areias,ricroc}@dcc.fc.up.pt

***Abstract*—Despite the availability of both multithreading and tabling in some Prolog systems, the implementation of these two features, such that they work together, implies complex ties to one another and to the underlying engine. In recent work, we have proposed an approach to combine multithreading with tabling, implemented on top of the Yap Prolog system, whose primary goal was to reduce memory usage for the table space. Regarding the execution times, we observed some problems related to Yap's memory allocator, which is based on the operating system's default memory allocator, when running programs that allocate a higher number of data structures in the table space.**

In this paper, we propose a more efficient and scalable memory allocator for multithreaded tabled evaluation of logic programs. Our goal is to minimize the performance degradation that the system suffers when it is exposed to simultaneous memory requests made by multiple threads. For that, we propose a memory allocator based on local and global pages, to split memory among specific data structures and different threads, together with a strategy where data structures of the same type are pre-allocated within a page. Experimental results show that our new memory allocator can effectively reduce the execution time and scale better, when increasing the number of threads, than the original allocator.

***Keywords*-Multithreading, Logic Programming, Tabling, Memory Allocation, Performance, Scalability.**

I. Introduction

Tabling [1] is a recognized and powerful implementation technique that overcomes some limitations of traditional Prolog systems in dealing with recursion and redundant subcomputations. Tabling is a refinement of Prolog's SLD resolution that stems from one simple idea: save intermediate answers for current computations, in an appropriate data area called the *table space*, so that they can be reused when a *similar computation* appears during the resolution process. Tabled evaluation can reduce the search space, avoid looping and have better termination properties than SLD resolution. Work on tabling 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 ALS-Prolog, B-Prolog, Ciao, Mercury, XSB and Yap.

Multithreading in Prolog is the ability to concurrently perform computations, in which each computation runs independently but shares the program clauses. The ISO Prolog multithreading standardization proposal [2] is currently implemented in several Prolog systems including Ciao, SWI-Prolog, XSB and Yap, providing a highly portable solution given the number of operating systems supported by these systems.

When multithreading 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 multithreaded tabling system, tables may be either private or shared between threads. While 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 a Prolog system, the implementation of these two features such that they work together implies complex ties to one another and to the underlying engine.

XSB was the first Prolog system to combine tabling with multithreading [3]. In recent work [4], we have presented an alternative view to XSB's approach, implemented on top of the Yap Prolog system [5], where each thread views its tables as private but, at the engine level, we use a *common table space*, i.e., from the thread point of view, the tables are private but, from the implementation point of view, the tables are shared among all threads. The primary goal of this work was to reduce memory usage for the table space and, for that, we have proposed three designs for the common table space: *No-Sharing (NS)*, *Subgoal-Sharing (SS)* and *Full-Sharing (FS)*. Our results showed that the SS design and, mainly, the FS design can, indeed, significantly reduce the memory usage for multithreaded tabled logic programs [4].

Regarding the execution times, we observed some problems related to Yap's memory allocator for multithreaded support, in particular, when running programs that allocate a higher number of data structures in the table space. Yap's memory allocator is based on the operating system's default memory allocator, which can be a problem when making a lot of memory requests, since such requests may require synchronization at the low-level implementation.

In this paper, we focus our discussion on the efficient and scalable memory allocation for multithreaded tabled evaluation of logic programs, within the common table space approach. Our goal is to minimize the performance degradation that the system suffers, when it is exposed to simultaneous memory requests made by multiple threads. In order to avoid this memory contention, we follow the general approach of the

current state-of-the-art user-level memory allocators, such as Hoard [6], Ptmalloc [7], Tcmalloc [8], and Jemalloc [9], but instead of using thread caches, local and global heaps with different block sizes, we use proper *local* and *global* pages, to split memory among specific data structures and different threads, together with a kind of *slab allocation* [10] mechanism where tabled data structures of the same type are pre-allocated within a page. When a page P is made local to a thread T, this means that T has exclusive permission to allocate and deallocate data structures from P. On the other hand, global pages have no owners and, thus, they are free from allocate/deallocate operations. In both cases, all threads can access (for read or write operations) the data structures on local or global pages. This is very important since it allows to significantly reduce memory contention without introducing any overhead for multithreaded tabled evaluation.

Experimental results show that our new memory allocator can effectively reduce the execution time and scale better, when increasing the number of threads, than the original allocator, for multithreaded tabled programs using the NS, SS and FS designs. The present work is already fully integrated and available with the latest development version of Yap[1].

The remainder of the paper is organized as follows. First, we briefly introduce some background about Yap's table space organization and the NS, SS and FS designs. Next, we describe our new memory allocator and we present some important implementation details. Finally, we discuss experimental results and we end by outlining some conclusions.

II. Background

Yap implements a multithreading library that follows the original SWI-Prolog proposal [11]. Like in SWI-Prolog, Yap's threads run independently, i.e., all threads have their own stacks and only share the Prolog area where predicates, records, flags and other global non-backtrackable data are stored. For multithreaded tabling, our approach is still based on this idea in which each thread runs independently.

A. Yap's Table Space Organization

A critical component in the implementation of an efficient tabling system is the design of the data structures and algorithms to access and manipulate tabled data. The most successful data structure for tabling is *tries*. 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 [12].

Yap's table space is organized as follows. 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 tabled answer to the entry subgoal.

[1]http://www.dcc.fc.up.pt/~vsc/Yap

B. The No-Sharing Design

The starting point for our multithreading approach was the situation where each thread allocates fully private tables for each new tabled subgoal called during its computation. Figure 1 shows the configuration of the table space if several different threads call the same tabled subgoal $call\_i$.

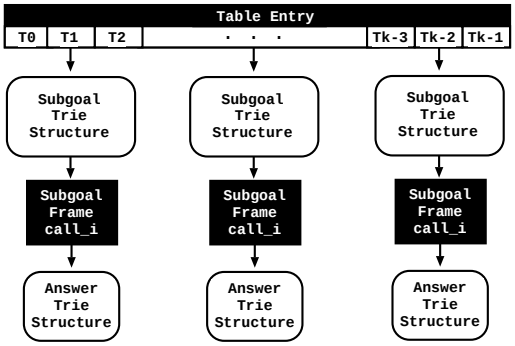

Fig. 1. Table space organization for the NS design

One can observe that the table entry data structure is still the entry point for tabled predicates, but now each thread has its own cell 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 itself abolishes the tables or finishes execution.

The memory usage for this design for a particular table entry T, assuming that all running threads NT have completely evaluated the same number NS of subgoals, is $sizeof(TE_T) + sizeof(BA_T) + [sizeof(STS_T) + [sizeof(SF_T) + sizeof(ATS_T)] * NS] * NT$, where TE_T and BA_T represent the common table entry and bucket array data structures, STS_T and ATS_T represent the nodes inside the subgoal and answer trie structures, and SF_T represents the subgoal frames.

C. The Subgoal-Sharing Design

In our second design, the threads share part of the tables. Figure 2 shows the configuration of the table space if several different threads call the same tabled subgoal $call\_i$.

In the SS 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 has its own cell inside

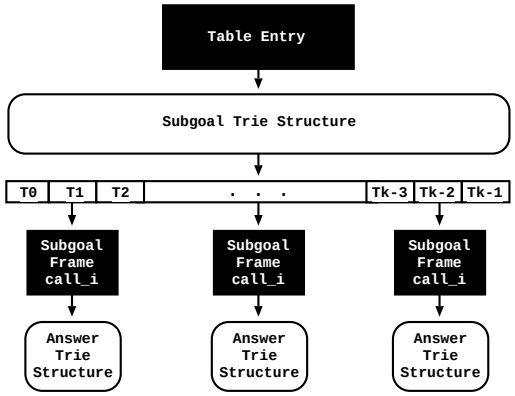

Fig. 2. Table space organization for the SS design

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 nodes on the subgoal trie structure. Whenever a thread finishes execution, its private structures are removed, but the shared structures remain present as they can be in use or be further used by other threads. Shared structures are only removed when the last running thread (usually thread 0) abolish the tables.

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 table entry T is $sizeof(TE_T) + sizeof(STS_T) + [sizeof(BA_T) + [sizeof(SF_T) + sizeof(ATS_T)] * NT] * NS$.

D. The Full-Sharing Design

Our third design is the more sophisticated one. Figure 3 shows its table space organization if considering again several different threads calling the same tabled subgoal $call\_i$.

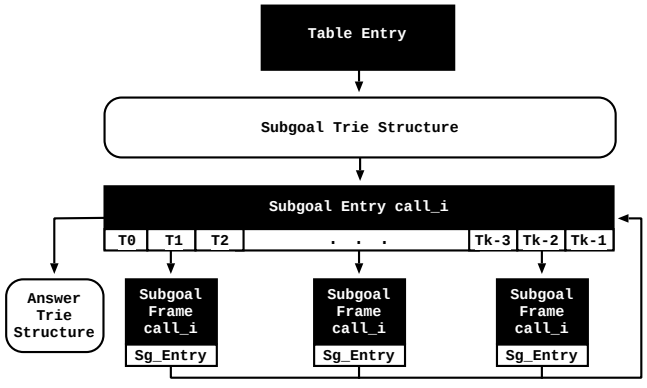

Fig. 3. Table space organization for the FS design

In the FS design, part of the subgoal frame information (the *subgoal entry* data structure in Fig. 3) 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. 3) remains private to each thread. The subgoal entry also includes a bucket array, in which each cell points to the private subgoal frame of each thread. 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. In this design, concurrency among threads now also includes the access to the subgoal entry data structure and the allocation of nodes on the answer trie structures.

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 table entry T is $sizeof(TE_T) + sizeof(STS_T) + [sizeof(SE_T) + sizeof(BA_T) + sizeof(ATS_T) + sizeof(SF_T) * NT] * NS$, where SE_T and SF_T represent, respectively, the shared subgoal entry and the private subgoal frame data structures.

The FS 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 are kept 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.

E. Table Locking Schemes

To deal with concurrent table accesses, we use a *Table Lock at Write Level (TLWL)* scheme [4]. The TLWL scheme allows a *single writer* per chain of sibling nodes that represent alternative paths from a common parent node, meaning that only one thread at a time can be inserting a new child node starting from the same parent node. In order to reduce the lock duration to a minimum, we also 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 the token t at hand was, in the meantime, inserted by another thread. If t is not found, the process repeats until the thread get access to the lock or until t be found. For locking, we use either a *locking field* per trie node or a *global array* of lock entries.

III. New Memory Allocator

Modern computer architectures use *pages* to handle memory. Pages are fixed size blocks of contiguous memory cells. Based on this characteristic, we adopted an allocation scheme based also on pages, where each memory page only contains data structures of the same type. In order to split memory among different threads, in our approach, a page can be considered a *local page*, if owned by a particular thread, or a *global page*, otherwise. Figure 4 gives an overview of the new memory allocator based on pages.

A thread can own any number of pages of the same type, of different types and/or free pages. Any type of page (including free pages) can be local to a thread or global, and each

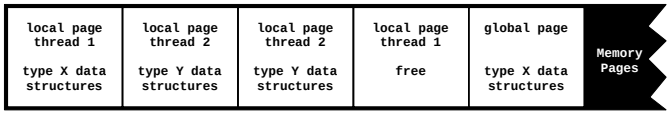

Fig. 4. Using pages as the basis for the new memory allocator

particular page only contains data structures of the same type. When a page P is made local to a thread T, this means that T has exclusive permission to allocate and deallocate data structures from P. On the other hand, global pages have no owners and, thus, they are free from allocate/deallocate operations. To allocate/deallocate data structures on global pages, first the corresponding pages should be moved to a particular thread. All running threads can access (for read or write operations) the data structures allocated on a page, independently of being a local or global page.

Access to the chain of available pages for a given data type is synchronized by a *page entry* data structure. For each different data type, there is a *global page entry* and a *local page entry* per thread. For example, for the subgoal frames, there is a **GB_PG_sg_fr** global page entry and a **LC_PG_sg_fr** local page entry per thread. Access to free pages (i.e., pages with all data structures unused) is also synchronized by proper global/local page entries, named **GB_PG_void** and **LC_PG_void**, respectively. Full pages (i.e., pages with all data structures in use) are not accessed from any local or global page entry. A page entry data structure includes a **PgEnt_first** and a **PgEnt_last** field that point, respectively, to the first and last page in the chain of pages. For the global pages, an extra **PgEnt_lock** field implements a lock mechanism that synchronizes access to the respective chain of pages.

The management of pages and data structures within pages is achieved by allocating a special *page header* structure at the beginning of each page and by uniformly dividing the remaining space in equal-size data structures of the data type being handled. Figure 5 shows an example that better illustrates how page entries and page headers work together. A page header consists of four fields. The **PgHd_next** and **PgHd_prev** fields point, respectively, to the next and previous pages in the chain of pages. The **PgHd_strs_in_use** field stores the number of data structures in use within a page. When it reaches zero the page is freed and moved to the **LC_PG_void** page entry of the thread at hand. The **PgHd_first** field points to the first unused data structure within a page and the remaining unused data structures are linked through their **next** fields. When all data structures are in use (i.e., when a page is full and **PgHd_first** is **Null**), the page is simply released from the respective chain.

Allocating and freeing data structures are constant-time operations, all we have to do is to move a structure to or from a list of free structures. Whenever a thread T requests to allocate memory for a data structure of type S, it can instantly satisfy the request by returning the first unused slot on the first available local page with type S. If there are no available local pages with type S, then a new page must be requested. If there are free local pages in **LC_PG_void**, then the first one is made

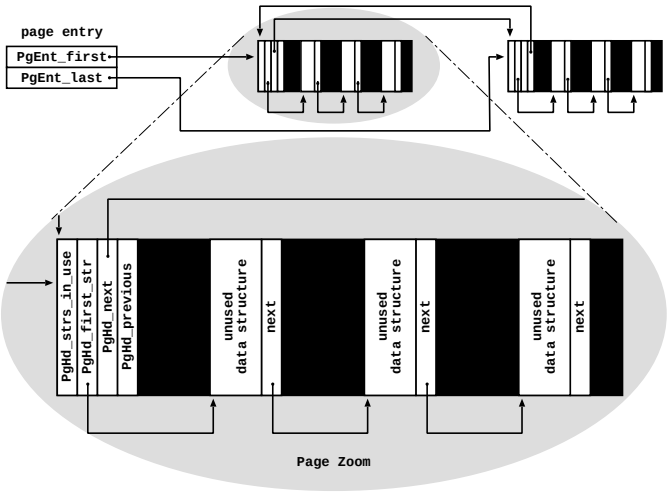

Fig. 5. Page entries and page headers in the new memory allocator

to be of type S. Otherwise, thread T must synchronize with the other threads in order to access the shared resources. Then, it first tries the **GB_PG_void** chain of free global pages and, if no free page exists there, it asks for new memory pages from the operating system's memory allocator (such pages are then chained in the **GB_PG_void** page entry).

Deallocation of a data structure of type S does not free up the memory, but only opens an unused slot on the chain of available local pages for type S. Further requests to allocate memory of type S will later return the now unused memory slot. When all data structures in a page are unused, the page is moved to the chain of free local pages. A free local page can be reassigned later to a different data type. This process eliminates the need to search for suitable memory space and greatly alleviates memory fragmentation. The only wasted space is the unused portion at the end of a page when it cannot fit exactly with the size of the corresponding data structures.

When a thread finishes execution, it deallocates all its private data structures and then moves its local pages to the corresponding global page entries. Remember from Section II that shared structures are only deallocated when the last running thread (usually thread 0) abolish the tables. Thus, if a thread T allocates a data structure D, then it will be also responsible for deallocating D, if D is private to T, or D will remain live in the tables, if D is shared, even when T finish execution. In the latter case, D can be only deallocated by the last running thread L. In such case, D is made to be local to L and the deallocation process follows as usual.

IV. Implementation Details

In this section, we present in more detail the algorithms that implement the key aspects of the new memory allocator.

Algorithm 1 shows the pseudo-code for allocating a new data structure given the corresponding local page entry pg_ent. Initially, it checks for available pages and, if no page exists, a new one is requested through a call to the *alloc_page()* procedure (lines 1–3). Next, it gets the first unused structure

from the page obtained and updates the page header to point to the next unused structure (lines 4–5). If no more unused structures exist then the page is full and the page entry at hand is updated to point to the next available page (lines 7–11).

Algorithm 1 $alloc\_struct(pg\_ent)$

```
1:  pg ← PgEnt_first(pg_ent)
2:  if pg = Null then {no available pages}
3:    pg ← alloc_page()
4:  str ← PgHd_first(pg)
5:  PgHd_first(pg) ← struct_next(str)
6:  if PgHd_first(pg) = Null then {page is full}
7:    if PgHd_next(pg) = Null then
8:      PgEnt_last(pg_ent) ← Null
9:    else
10:     PgHd_prev(PgHd_next(pg)) ← Null
11:   PgEnt_first(pg_ent) ← PgHd_next(pg)
12: PgHd_strs_in_use(header) + +
13: return str
```

Algorithm 2 shows the pseudo-code for the *alloc_page()* procedure. Initially, the procedure checks for free local pages (lines 1–2). If there is at least one such page, it updates the chain of free local pages (lines 3–5) and returns it. Otherwise, it locks the free global pages and tries to get a page from there (lines 7–15) and, if no free page exists, it asks the operating system for new memory pages (procedure *alloc_init_new_pages_from_OS()*).

Algorithm 2 $alloc\_page()$

```
1:  pg ← PgEnt_first(LC_PG_void)
2:  if pg ≠ Null then
3:    if PgHd_next(pg) = Null then
4:      PgEnt_last(LC_PG_void) ← Null
5:    PgEnt_first(LC_PG_void) ← PgHd_next(pg)
6:  else {no free local pages}
7:    lock(PgEnt_lock(GB_PG_void))
8:    pg = PgEnt_first(GB_PG_void)
9:    if pg = Null then {no free global pages}
10:     alloc_init_new_pages_from_OS()
11:     pg = PgEnt_first(GB_PG_void)
12:   if PgHd_next(pg) = Null then
13:     PgEnt_last(GB_PG_void) ← Null
14:   PgEnt_first(GB_PG_void) ← PgHd_next(pg)
15:   unlock(PgEnt_lock(GB_PG_void))
16: return pg
```

Algorithm 3 shows the pseudo-code for the *free_struct()* procedure given a data structure str and the corresponding local page entry pg_ent. Initially, it determines the corresponding page pg for str (line 1) and checks if pg contains other structures in use (lines 2–3). If so, str is chained in the list of unused structures within the page (lines 10–11), and if str is the first structure being made available, then pg is also inserted in the chain of available pages of that type (lines 5–9). Otherwise, if pg does not contain other structures in use, the page stops being of the current type and is moved to the chain of free local pages (lines 13–25). The *free_page()* procedure inserts a page into the chain of available free pages.

Algorithm 3 $free\_struct(str, pg\_ent)$

```
1:  pg ← get_page(str)
2:  PgHd_strs_in_use(pg) − −
3:  if PgHd_strs_in_use(pg) ≠ 0 then
4:    if PgHd_first(pg) = Null then {first unused struct}
5:      PgHd_next(pg) ← Null
6:      PgHd_prev(pg) = PgEnt_last(pg_ent)
7:      if PgHd_prev(pg) ≠ Null then
8:        PgHd_next(PgHd_prev(pg)) ← pg
9:      PgEnt_last(pg_ent) ← pg
10:   struct_next(str) ← PgHd_first(pg)
11:   PgHd_first(pg) ← str
12: else {no other structures in use}
13:   if PgHd_prev(pg) ≠ Null then
14:     if PgHd_next(pg) = Null then
15:       PgEnt_last(pg_ent) ← PgHd_prev(pg)
16:     else
17:       PgHd_prev(PgHd_next(pg)) ← PgHd_prev(pg)
18:     PgHd_next(PgHd_prev(pg)) ← PgHd_next(pg)
19:   else
20:     if PgHd_next(pg) = Null then
21:       PgEnt_last(pg_ent) ← Null
22:     else
23:       PgHd_prev(PgHd_next(pg)) ← Null
24:     PgEnt_first(pg_ent) ← PgHd_next(pg)
25:   free_page(pg, LC_PG_void)
```

V. Experimental Results

We now present experimental results comparing the old and the new memory allocators for the NS, SS and FS table designs. 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.

A. Benchmark Programs

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. 6 shows an example for each configuration). We experimented the **BTree** configuration with depth 17, the **Pyramid** and **Cycle** configurations with depth 2000 and the **Grid** configuration with depth 35. All benchmarks find all the solutions for the problem.

[2]http://rulebench.projects.semwebcentral.org

TABLE I
CHARACTERISTICS OF THE BENCHMARK PROGRAMS

Bench	Tabled Subgoals			Tabled Answers				Time (ms)
	calls	trie nodes	trie depth	unique	repeated	trie nodes	trie depth	NS
Large Joins								
Join2	1	6	5/5/5	2,476,099	0	2,613,660	5/5/5	3,747
Mondial	35	42	3/4/4	2,664	2,452,890	14,334	6/7/7	737
WordNet								
Clusters	117,659	235,319	2/2/2	166,877	161,853	284,536	1/1/1	822
Hypo	117,657	117,659	2/2/2	698,472	20,341	816,129	1/1/1	1,551
Holo	117,657	235,315	2/2/2	74,838	54	192,495	1/1/1	711
Hyper	117,657	235,315	2/2/2	698,472	8,658	816,129	1/1/1	1,413
Tropo	117,657	235,315	2/2/2	472	0	118,129	1/1/1	611
Mero	117,657	117,659	2/2/2	74,838	13	192,495	1/1/1	695
Model Checking								
IProto	1	6	5/5/5	134,361	385,423	1,554,896	4/51/67	2,466
Leader	1	5	4/4/4	1,728	574,786	41,788	15/80/97	3,761
Sieve	1	7	6/6/6	380	1,386,181	8,624	21/53/58	24,688
Path Left								
BTree	1	3	2/2/2	1,966,082	0	2,031,618	2/2/2	1,517
Pyramid	1	3	2/2/2	3,374,250	1,124,250	3,377,250	2/2/2	3,393
Cycle	1	3	2/2/2	4,000,000	2,000	4,002,001	2/2/2	4,035
Grid	1	3	2/2/2	1,500,625	4,335,135	1,501,851	2/2/2	1,929
Path Right								
BTree	131,071	262,143	2/2/2	3,801,094	0	3,997,700	1/2/2	2,334
Pyramid	3,000	6,001	2/2/2	6,745,501	2,247,001	6,751,500	1/2/2	2,770
Cycle	2,001	4,003	2/2/2	8,000,000	4,000	8,004,001	1/2/2	3,025
Grid	1,226	2,453	2/2/2	3,001,250	8,670,270	3,003,701	1/2/2	2,346

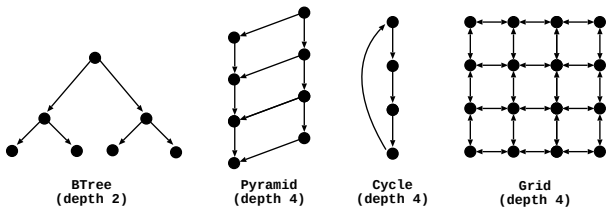

Fig. 6. Edge configurations for the path benchmarks

In order to have a deeper insight on the behavior of each benchmark, and therefore clarify some of the results that are presented next, we first characterize the benchmarks. The columns in Table I have the following meaning:

- **calls:** is the number of different calls to tabled subgoals. It corresponds to the number of paths in the subgoal tries.
- **trie nodes:** is the total number of trie nodes allocated in the corresponding subgoal/answer trie structures.
- **trie depth:** is the minimum/average/maximum number of trie nodes required to represent a path in the corresponding subgoal/answer trie structures. Trie structures with smaller average depth values are more amenable to higher lock contention.
- **unique:** is the number of different tabled answers found. It corresponds to the number of paths in the answer tries.
- **repeated:** is the number of redundant tabled answers found. With the TLWL locking scheme, redundant answers do not lock the table space.
- **NS:** is the average execution time, in milliseconds, of five first runs for 1 thread with the NS design and the old memory allocator. In what follows, we will use these times as the base times when computing the overhead ratios for the execution with 16 and 24 threads.

By observing Table I, the **Mondial** benchmark, from the **Large Joins** set, and the three **Model Checking** benchmarks seem to be the benchmarks least amenable to lock contention since they are the ones that find less unique answers and that have the deepest trie structures. In this regard, the **Path Left** and **Path Right** sets correspond to the opposite case. They find a huge number of answers and have very shallow trie structures. On the other hand, the **WordNet** and **Path Right** sets have the benchmarks with the largest number of different subgoal calls, which can reduce the probability of lock contention because answers can be found for different subgoal calls and therefore be inserted with minimum overlap. On the opposite side are the **Join2** benchmark, from the **Large Joins** set, and the **Path Left** benchmarks, which have only a single tabled subgoal call.

B. Performance Analysis

During experimentation, we observed some huge differences, in the execution time, between the first and the following runs of a program, being such differences more clear for the operating system's default memory allocator. This happens because the memory allocated for the first runs is not really freed, which allows to be reused in the following runs. Hence, and in order to differentiate such runs, in what follows, we show two types of evaluations: *first runs*, where we just consider the first execution times; and *second runs*, where we discard the first runs and consider only the second execution times.

Tables II and III show the overhead ratios for the average of five first and five second runs, respectively, when running

TABLE II

OVERHEAD RATIOS (AVERAGE OF FIVE FIRST RUNS) COMPARING THE NS, NS_P, SS, SS_P, FS, FS_P, FS_G AND FS_{GP} DESIGNS WITH 16 AND 24 THREADS AGAINST THE NS DESIGN WITH 1 THREAD (BEST RATIOS ARE IN BOLD)

Bench	16 Threads								24 Threads							
	NS	**NS_P**	**SS**	**SS_P**	**FS**	**FS_P**	**FS_G**	**FS_{GP}**	**NS**	**NS_P**	**SS**	**SS_P**	**FS**	**FS_P**	**FS_G**	**FS_{GP}**
Large Joins																
Join2	7.39	3.83	7.41	3.87	2.95	2.01	2.40	**1.85**	23.05	5.86	23.39	6.02	3.80	2.58	2.77	**2.29**
Mondial	1.10	1.13	**1.08**	1.15	3.10	1.43	1.48	1.45	1.20	**1.14**	1.18	1.15	4.10	1.50	1.91	1.60
WordNet																
Clusters	6.35	**1.61**	5.45	2.14	3.82	2.13	3.65	2.07	19.84	2.58	11.94	3.08	4.53	2.36	4.26	**2.23**
Hypo	5.71	**2.05**	5.43	2.42	2.91	2.31	2.97	2.14	15.84	2.92	9.51	3.29	4.25	2.69	4.35	**2.47**
Holo	6.30	**1.52**	5.40	2.06	3.67	1.95	3.56	1.90	17.57	2.58	9.41	2.59	4.88	2.04	4.59	**1.99**
Hyper	8.10	3.64	7.48	4.20	3.01	2.20	2.91	**2.10**	23.79	5.94	15.29	6.31	3.38	2.30	3.26	**2.24**
Tropo	6.24	**1.19**	4.94	2.07	3.93	2.20	3.83	2.14	17.95	**1.88**	8.02	2.29	5.80	2.37	5.54	2.25
Mero	5.10	**1.37**	4.91	1.93	3.66	1.99	3.36	1.91	13.99	**1.95**	8.37	2.30	4.81	2.19	4.27	2.15
Model Checking																
IProto	4.36	1.09	4.39	**1.07**	1.59	1.26	1.58	1.39	15.05	1.18	14.94	**1.17**	1.70	1.37	1.63	1.36
Leader	**1.02**	1.06	1.03	1.06	1.03	**1.02**	1.03	1.03	**1.02**	1.07	1.04	1.06	1.03	1.03	1.03	1.03
Sieve	1.01	1.04	1.01	1.04	**0.99**	1.00	1.00	1.01	1.01	1.04	1.01	1.04	**0.99**	1.00	1.00	1.01
Path Left																
BTree	9.89	**2.66**	9.79	**2.66**	4.09	3.31	3.59	2.74	32.13	4.48	33.45	4.44	5.62	4.41	4.52	**3.66**
Pyramid	7.48	1.87	7.49	**1.83**	3.15	2.26	2.78	2.19	23.73	3.31	24.24	3.33	4.34	3.03	3.43	**2.90**
Cycle	7.39	1.73	7.32	**1.66**	2.82	2.21	2.95	2.10	22.70	3.18	22.79	3.25	4.10	**2.89**	3.84	3.08
Grid	5.75	**1.43**	5.71	1.50	2.92	2.18	2.56	2.16	16.55	2.30	16.30	**2.29**	4.09	2.73	3.32	2.66
Path Right																
BTree	13.74	3.77	12.97	4.01	4.99	3.86	4.47	**3.42**	43.98	6.09	34.15	6.32	6.22	4.86	6.46	**4.24**
Pyramid	16.76	4.78	16.80	**4.70**	7.47	5.44	6.23	4.85	55.54	8.15	52.79	8.15	9.32	7.36	7.58	**6.39**
Cycle	17.71	4.71	17.68	**4.67**	7.56	6.20	6.56	4.95	52.25	8.11	53.73	8.12	9.81	7.94	8.10	**6.72**
Grid	9.47	2.60	9.46	2.64	4.64	**2.46**	4.59	2.81	30.84	4.25	31.02	4.30	6.16	**2.93**	5.59	4.23
Total Average	7.41	**2.27**	7.14	2.46	3.59	2.50	3.24	2.33	22.53	3.58	19.61	3.71	4.68	3.03	4.08	**2.87**

TABLE III

OVERHEAD RATIOS (AVERAGE OF FIVE SECOND RUNS) COMPARING THE NS, NS_P, SS, SS_P, FS, FS_P, FS_G AND FS_{GP} DESIGNS WITH 16 AND 24 THREADS AGAINST THE NS DESIGN WITH 1 THREAD (BEST RATIOS ARE IN BOLD)

Bench	16 Threads								24 Threads							
	NS	**NS_P**	**SS**	**SS_P**	**FS**	**FS_P**	**FS_G**	**FS_{GP}**	**NS**	**NS_P**	**SS**	**SS_P**	**FS**	**FS_P**	**FS_G**	**FS_{GP}**
Large Joins																
Join2	1.15	**0.94**	1.17	0.96	2.30	1.86	2.29	1.80	1.17	**0.99**	1.15	**0.99**	2.96	2.60	2.72	2.29
Mondial	0.90	1.02	**0.88**	1.03	3.15	1.29	1.62	1.30	**0.91**	1.02	0.95	1.05	4.20	1.37	2.01	1.45
WordNet																
Clusters	1.01	**0.65**	1.16	1.05	1.54	1.46	1.69	1.47	1.26	**0.89**	1.41	1.16	1.75	1.59	1.82	1.58
Hypo	1.56	1.17	1.48	**1.13**	2.06	1.96	2.06	1.80	2.44	**1.48**	2.13	1.59	2.60	2.29	2.43	2.23
Holo	1.01	**0.58**	1.27	1.14	1.45	1.36	1.62	1.31	1.36	**0.76**	1.47	1.19	1.57	1.41	1.79	1.38
Hyper	1.03	**0.80**	1.10	1.01	1.97	1.78	2.07	1.73	1.19	**0.93**	1.18	1.07	2.11	1.87	2.16	1.82
Tropo	0.96	**0.50**	1.24	1.07	1.29	1.20	1.42	1.20	1.29	**0.65**	1.52	1.20	1.49	1.31	1.54	1.29
Mero	1.08	**0.57**	1.23	1.10	1.42	1.32	1.51	1.29	1.56	**0.70**	1.53	1.24	1.64	1.52	1.74	1.38
Model Checking																
IProto	1.26	**1.14**	1.24	**1.14**	1.44	1.31	1.41	1.37	1.27	1.24	**1.20**	1.26	1.49	1.35	1.45	1.48
Leader	**0.99**	1.06	1.01	1.06	1.03	1.02	1.03	1.04	**1.00**	1.06	**1.00**	1.06	1.03	1.03	1.03	1.03
Sieve	**0.99**	1.05	1.00	1.05	1.00	1.00	1.00	1.01	**1.00**	1.04	**1.00**	1.03	**1.00**	**1.00**	1.01	1.01
Path Left																
BTree	2.05	1.30	1.87	**1.28**	3.62	3.39	3.35	2.60	2.98	1.67	2.58	**1.60**	4.89	4.23	3.50	3.60
Pyramid	1.87	**1.27**	1.86	1.30	2.49	2.16	2.39	2.20	3.08	**1.58**	2.84	1.61	3.61	2.93	3.58	3.03
Cycle	1.71	1.23	1.70	**1.22**	2.46	2.11	2.60	2.04	2.44	1.50	2.43	**1.49**	3.41	2.84	3.28	2.85
Grid	1.47	1.16	1.44	**1.15**	2.57	2.33	2.22	2.12	1.73	**1.28**	1.70	1.37	3.26	2.77	3.09	2.59
Path Right																
BTree	2.08	**1.42**	2.08	1.54	4.36	3.70	3.95	3.30	3.26	**2.32**	3.88	2.96	5.15	4.75	4.65	4.19
Pyramid	2.35	1.95	2.39	**1.93**	6.19	5.31	5.41	4.88	5.88	4.96	5.27	**4.54**	8.14	6.26	7.66	6.52
Cycle	2.30	1.91	2.42	**1.87**	6.63	6.18	6.47	4.99	4.83	4.69	4.93	**3.85**	8.83	7.68	7.01	6.84
Grid	1.58	**1.17**	1.57	**1.17**	3.26	2.45	3.20	2.66	2.63	2.49	2.29	**1.81**	3.86	3.21	4.47	3.61
Total Average	1.44	**1.10**	1.48	1.22	2.64	2.27	2.49	2.11	2.17	**1.64**	2.13	1.69	3.31	2.74	3.00	2.64

the benchmark set with 16 and 24 threads for the three table designs using the old and the new memory allocator. To compute the ratios we used the NS base times from Table I. Columns NS, SS and FS correspond to the execution using the operating system's default memory allocator, and columns NS_P, SS_P, FS_P show the results for the new page based memory allocator (in all cases, both SS and FS designs implement the TLWL locking scheme using a locking field in the trie nodes). Additionally, columns FS_G and FS_{GP} show the results for the FS design implementing the TLWL locking

scheme using a global array of lock entries.

In order to create a worst case scenario that stresses either the memory allocator and both trie data structures, we ran all threads starting with the same query goal. By doing this, it is expected that all threads will request memory (specially the NS design) and/or access the table space, to check/insert for subgoals (the NS and FS designs) and answers (the FS design), at similar times, thus causing a huge stress on the same critical regions. In particular, this will be specially the case for the answer tries, since the number of answers clearly exceeds the number of subgoals on most benchmarks.

In general, the experiments on Tables II and III show that the results with the new memory allocator are, on average, always better (no exceptions) than the operating system's default memory allocator. In particular, for first runs executions with the NS and SS designs, the gain is overwhelming. For example, with 24 threads, the reduction goes from 22.53 to 3.58, for the NS design, and from 19.61 to 3.71, for the SS design. Moreover, our experiments also show that the new memory allocator scales better than the previous implementation, when we increase the number of threads. On average, the cost of moving from 16 to 24 threads is always less (no exceptions) with the new memory allocator.

By comparing the NS with the NS_P ratios, we can observe the impact of the new memory allocator in reducing synchronization when requesting memory. This reduction is more clear for the benchmarks that allocate an higher number of trie nodes, such as the **Join2** and **IProto** benchmarks and the **WordNet**, **Path Left** and **Path Right** sets.

When we compare the NS_P with the SS_P ratios, and since both designs benefit from the gain introduced by the new memory allocator, we can observe the impact of having the subgoal tries shared across threads. On one hand, with the SS design, we request less trie nodes for the subgoal tries (thus reducing synchronization when requesting memory for the memory allocator) but, on the other hand, we are introducing a new cost when synchronizing the insertion of nodes into the shared subgoal trie structures. This cost is more clear for the benchmarks that allocate an higher number of subgoal trie nodes, such as the **WordNet** benchmark set.

Finally, the FS_P ratios show the impact of having also the answer tries shared across threads. On one hand, we request less trie nodes for the answer tries (thus reducing synchronization when requesting memory for the memory allocator) but, on the other hand, we introduce an extra cost when synchronizing the insertion of nodes into the shared answer trie structures. Our results confirm that the **Mondial**, **Leader** and **Sieve** benchmarks are least susceptible to this extra cost.

Our results also show that, with a global array of lock entries, we can still improve performance by using the new memory allocator (the FS_{GP} design). Note that both mechanisms are orthogonal to each other. From Table II, we can observe that, the total average for 24 threads, moves from a ratio of 4.68 and 4.08, for the FS and FS_G designs, to a ratio of 3.03 and 2.87, for the FS_P and FS_{GP} designs, on average.

VI. Conclusions

We have presented a novel, efficient and scalable memory allocator for multithreaded tabled evaluation of logic programs based on local and global pages, to split memory among specific data structures and different threads, together with a page based mechanism, where data structures of the same type are pre-allocated within a page. Our main goal is to minimize the performance degradation that the system suffers, when it is exposed to simultaneous memory requests made by multiple threads. Experimental results show that our new memory allocator is always better than the previous implementation and that, for first runs executions with the NS and SS designs, it can achieve significant reductions on the execution time. Our experiments also show that the new memory allocator scales better when we increase the number of threads. Further work will include studying alternative memory allocators and new approaches to reduce lock contention when inserting nodes into the shared trie structures.

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 PEst (FCOMP-01-0124-FEDER-022701), 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.

References

[1] W. Chen and D. S. Warren, "Tabled Evaluation with Delaying for General Logic Programs," *Journal of the ACM*, vol. 43, no. 1, pp. 20–74, 1996.

[2] P. Moura, "ISO/IEC DTR 13211–5:2007 Prolog Multi-threading Predicates," 2008. [Online]. Available: http://logtalk.org/plstd/threads.pdf

[3] R. Marques and T. Swift, "Concurrent and Local Evaluation of Normal Programs," in *International Conference on Logic Programming*, ser. LNCS, no. 5366. Springer-Verlag, 2008, pp. 206–222.

[4] M. Areias and R. Rocha, "Towards Multi-Threaded Local Tabling Using a Common Table Space," *Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue*, vol. 12, no. 4 & 5, pp. 427–443, 2012.

[5] V. Santos Costa, R. Rocha, and L. Damas, "The YAP Prolog System," *Journal of Theory and Practice of Logic Programming*, vol. 12, no. 1 & 2, pp. 5–34, 2012.

[6] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A Scalable Memory Allocator for Multithreaded Applications," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 117–128, 2000.

[7] W. Gloger, "Ptmalloc." [Online]. Available: http://www.malloc.de/en/

[8] S. Ghemawat and P. Menage, "TCMalloc: Thread-Caching Malloc." [Online]. Available: http://goog-perftools.sourceforge.net/doc/tcmalloc.html

[9] J. Evans, "A Scalable Concurrent malloc(3) Implementation for FreeBSD," in *The Technical BSD Conference*, 2006.

[10] J. Bonwick, "The Slab Allocator: An Object-Caching Kernel Memory Allocator," in *Usenix Summer 1994 Technical Conference*. Usenix Association, 1994, pp. 87–98.

[11] J. Wielemaker, "Native Preemptive Threads in SWI-Prolog," in *International Conference on Logic Programming*, ser. LNCS, no. 2916. Springer-Verlag, 2003, pp. 331–345.

[12] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren, "Efficient Access Mechanisms for Tabled Logic Programs," *Journal of Logic Programming*, vol. 38, no. 1, pp. 31–54, 1999.