On Extending a Full-Sharing Multithreaded Tabling Design with Batched Scheduling

Miguel Areias and Ricardo Rocha

CRACS & INESC TEC and Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021, 4169-007 Porto, Portugal
{miguel-areias,ricroc}@dcc.fc.up.pt

Abstract. Tabling is an implementation technique that overcomes some limitations of traditional Prolog systems in dealing with redundant sub-computations and recursion. Multithreading in Prolog is the ability to concurrently perform computations, in which each computation runs independently but shares the program clauses. 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. To support this combination, the Yap system has at engine level, multiple designs that vary from a No-Sharing design, where each thread allocates fully private tables for each new tabled subgoal called during its computation, to a Full-Sharing design, where threads share the complete table space. Arguably, batched scheduling is one of the most successful tabling scheduling strategies. In this work, we propose an extension of the Full-Sharing design to support batched scheduling. Experimental results show that, despite the extra data structures required to support the extension, the FS design remains quite competitive.
Keywords: Logic Programming, Multithreading, Tabling, Scheduling.

1 Introduction

Tabling [4] is an implementation technique that overcomes some limitations of traditional Prolog systems in dealing with redundant sub-computations and recursion. Tabling consists of storing intermediate answers for subgoals in a proper data structure, called the *table space*, so that they can be reused when a repeated subgoal appears during the resolution process. Tabling has become a popular and successful technique thanks to the ground-breaking work in the XSB Prolog system and in particular in the SLG-WAM engine [7], the most successful engine of XSB. The success of SLG-WAM led to several alternative implementations that differ in the execution rule, in the data-structures used to implement tabling, and in the changes to the underlying Prolog engine. Implementations of tabling are now widely available in systems like Yap Prolog, B-Prolog, ALS-Prolog, Mercury, Ciao Prolog and more recently Picat.

Multithreading in Prolog is the ability to concurrently perform computations, in which each computation runs independently but shares the program clauses. 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. To the best of our knowledge, XSB [6] and Yap [2] are the only Prolog systems that support the combination of multithreading with tabling. Yap implements a SWI-Prolog compatible multithreading library [8]. 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. For tabled evaluation, a thread views its tables as private but, at the engine level, Yap has three designs [2] that vary from a *No-Sharing* (NS) design, where each thread allocates fully private tables for each new tabled subgoal called during its computation, to a *Full-Sharing* (FS) design, where threads share the complete table space.

The decision about the evaluation flow is determined by the *scheduling strategy*. Different strategies may have a significant impact on performance, and may lead to a different ordering of solutions to the query goal. Arguably, the two most successful tabling scheduling strategies are *local scheduling* and *batched scheduling* [5]. Local scheduling tries to complete subgoals as soon as possible. When new answers are found, they are added to the table space and the evaluation fails. Answers are only returned when all program clauses for the subgoal at hand were resolved. Batched scheduling favors forward execution first, backtracking next, and consuming answers or completion last. It thus tries to delay the need to move around the search tree by batching the return of answers to repeated subgoals. When new answers are found for a particular tabled subgoal, they are added to the table space and the evaluation continues.

With the FS design, all tables are shared. Thus, since several threads can be inserting answers in the same table, when an answer already exists, it is not possible to determine if the answer is new or repeated for a certain thread without further support. However, for local scheduling this is not a problem because, for repeated and new answers, local scheduling always fails. The problem is with batched scheduling that requires further support to work with the FS design since, with batched scheduling, only repeated answers should fail. We have thus to detect when a new found answer is also a new answer for the thread at hand in order to ensure that new answers for a given subgoal call are correctly propagated one and only once to all thread's calls.

In this work, we propose an extension to the table space data structures, which we named *Private Answer Chaining (PAC)*, as a way to keep track, per thread and subgoal call, of the answers that were already found and propagated to all repeated calls. We discuss in detail our proposal for extending the FS design with batched scheduling and we present a performance analysis comparison between local and batched scheduling. Experimental results show that, despite the extra PAC data structures required to support batched scheduling with the FS design, the execution time of the combination is still quite competitive when comparing against the FS design with local scheduling.

The remainder of the paper is organized as follows. First, we briefly introduce some background and related work. Then, we describe our approach for combining batched scheduling with the FS design and we discuss the most important implementation details. Finally, we present experimental results and we end by outlining some conclusions.

2 Background

This section introduces some background needed for the following sections.

2.1 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.

Figure 1 shows the table space organization of the Yap system. 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.

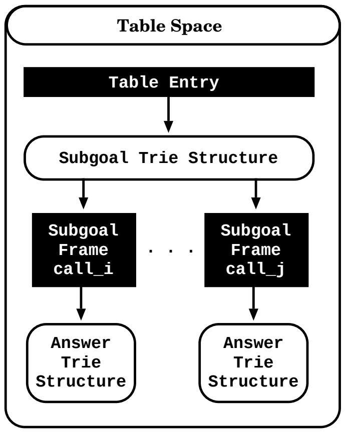

Fig. 1. Table space organization

2.2 Yap's Multithreaded Tabling Support

In Yap, a thread views its tables as private but, at the engine level, it uses a common table space where tables are shared among all threads. Yap implements three designs for concurrent tabling support that vary from a *No-Sharing* (NS) design, where each thread allocates fully private tables for each new tabled subgoal called during its computation, to a *Full-Sharing* (FS) design, where threads share the complete table space. Figure 1 shows Yap's multithreaded table space organization for the NS and FS designs, where an interface layer abstracts the design being used at the engine level. The figure illustrates the main differences between the two designs for a situation where several threads are evaluating the same tabled subgoal call *call_i*.

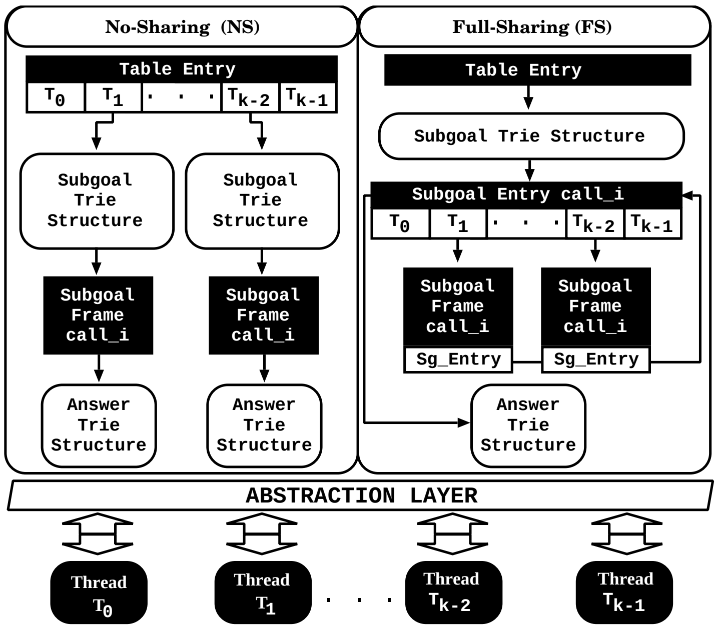

Fig. 2. Yap's multithreaded table space organization for the NS and FS designs

When using the NS design, each thread allocates fully private tables for each new subgoal called during its computation. 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.

When using the FS design, the subgoal and answer trie structures and part of the subgoal frame information (the *subgoal entry* data structure in Fig. 2) are shared among all threads. The previous subgoal frame data structure was split in two: the *subgoal entry* stores common information for the subgoal call (such as the pointer to the shared answer trie structure); the remaining information is kept private to each thread in the *subgoal frame* data structure.

2.3 Scheduling Strategies

Local scheduling schedules the evaluation of a program in a breath-first manner. It favors backtracking first with completion instead of forward execution, leaving the consumption of answers for last. Local scheduling only allows a *Cluster of Dependent Subgoals* (CDS) to return answers after a fixpoint point has been reached [5].

In other words, local scheduling tries to keep CDS as minimal as possible. When new answers are found, they are added to the table space and the computation fails as consequence, tabled subgoals inside a CDS propagate their answers to outside the CDS only after its completion point is found.

Local scheduling causes a sooner completion of subgoals, which creates less complex dependencies between them.

On the other hand, batched scheduling schedules the evaluation of a program in a depth-first manner. It favors the forward execution first instead of backtracking, leaving the consumption of answers and completion for 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 execution continues. For some situations, this results in creating dependencies to older subgoals, therefore enlarging the current CDS [7] and delaying the completion point to an older generator node.

3 Extending Full-Sharing with Batched Scheduling

In this section, we show how we have extended the FS design to support batched scheduling. The usage of FS design with batched scheduling, requires further support than with local scheduling, since when a new answer is found it must be immediately propagated, and we have to ensure that the propagation occurs on all subgoal calls once and only once. To do so, next we show a procedure that, for every subgoal call of every thread, keeps track of all the answers that were already propagated.

3.1 Our Approach

The key idea of our approach, which we named Privately-consumed Answer Chaining (PAC), is then to extend the *FS* design with batched scheduling, by chaining privately for each subgoal call, the answers that were already consumed by a thread. Since the procedure is private, it will only affect the thread that is doing it. At the end, when the evaluation is complete, i.e, when a subgoal call is marked as complete, we put one of the private chain as public, so that from that point on all threads can use that chain in complete (only reading) mode.

Figure 3 shows the key data structures for supporting the implementation of the PAC procedure during the evaluation of a tabled subgoal call i using the FS design. The PAC procedure works at the subgoal frame level, which is private to each thread.

Figure 3(a) shows then a situation where two threads, T_1 and T_{k-1}, are sharing the same subgoal entry call i when the subgoal is still under evaluation, i.e., the subgoal is not yet complete. The current state of the evaluation shows an answer trie with 3 answers found for the subgoal call i. For the sake of

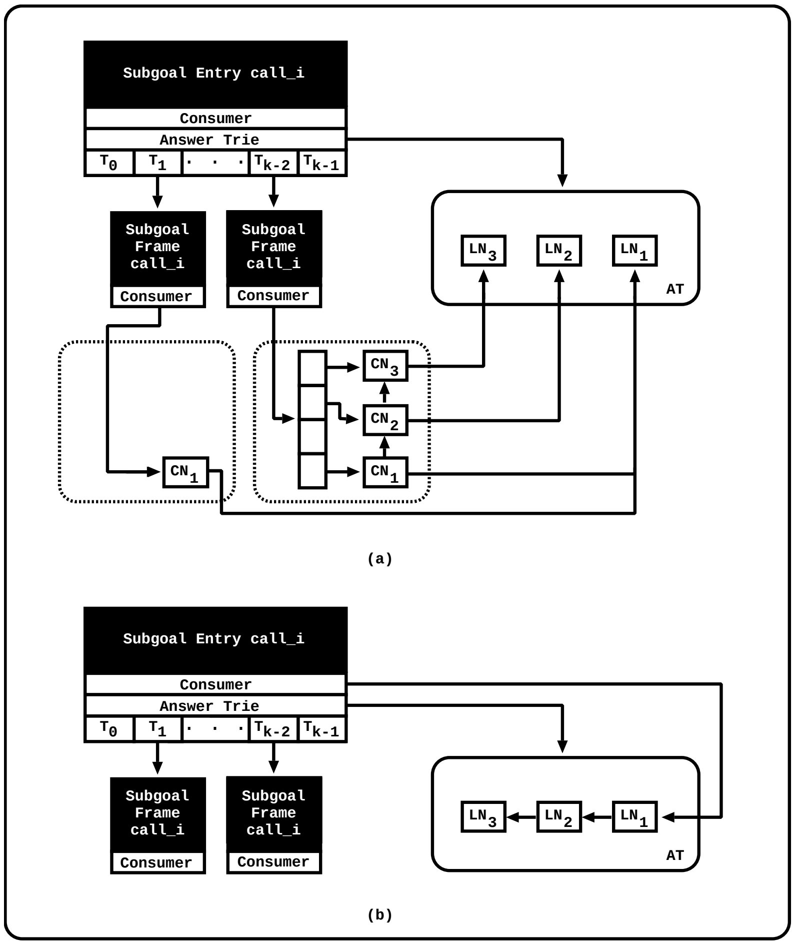

Fig. 3. The PAC procedure using (a) private chaining and (b) public chaining

simplicity, we are omitting the internal answer trie nodes and we are only showing the leaf nodes nodes LN_1, LN_2 and LN_3, in the figure. On the subgoal frame structure we added a new field, called *consumer*, to store the answers found within the execution of the thread. In order to minimize the impact of the PAC optimization, each node within the new consumer answer structure has two fields: (i) an entry pointer, which points to the corresponding leaf node in the answer trie data structure; (ii) a next pointer to chain the answers within the consumer structure. To maintain a good performance, when the number of nodes exceeds a certain threshold, we use a hash trie mechanism design similar to the one presented in the work [3]. However, since this mechanism is private to each thread, it does not require any of the tools that were necessary to support concurrency, thus we have removed them from the mechanism.

Going back to Figure 3(a), the consumer answer structures represent then two different situations where threads can be evaluating a subgoal call. Thread T_1 has only found one answer and it is using a direct consumer answer chaining to access the node LN_1. Thread T_{k-1} was already found three answers for the subgoal call and it is already using the hash trie mechanism within its consumer answer structure. The consumer nodes are chained between themselves, thus that consumer nodes belonging to thread T_{k-1} can consume the answers as in the original mechanism.

Figure 3(b) shows the state of the subgoal call after completion. When a thread T completes a subgoal call, it frees its private consumer structures, but before doing that, it checks whether another thread as already marked the subgoal as completed. If no other thread has done that, then thread T not only follows its private chaining mechanism as it would for freeing its private nodes, but also, follows the pointers to the answer trie leaf nodes in order to create a chain inside the answer trie. Since this procedure is done inside a critical region, no more than one thread can be doing this chaining process. Thus, in Figure 3(b), we are showing a situation where the subgoal call is completed and both threads T_1 and T_{k-1} have already removed their consumer answer structures and chained the leaf nodes inside the answer trie.

3.2 Implementations Details

At the tabling engine, the major difference between local and batched scheduling is in the tabling operation *tabled new answer*, where we decide what to do when an answer is found during the evaluation. This operation checks whether a newly found answer is already in the corresponding answer trie structure and, if not, inserts it. Algorithm 1 shows how we have extended this operation to support FS design with batched scheduling.

Algorithm 1 tabled_new_answer(answer ANS, subgoal frame SF)

```
1: $leaf \leftarrow check\_insert\_answer\_trie(ANS, SF)$
2: $chain \leftarrow check\_insert\_consumer\_chain(leaf, SF)$
3: if $is\_answer\_marked\_as\_found(chain) = True$ then
4:     return $failure$
5: else {the answer is new}
6:     $mark\_answer\_as\_found(chain)$
7:     if $local\_scheduling\_mode(SF)$ then
8:         return $failure$
9:     else {batched scheduling mode}
10:        return $proceed$
```

The algorithm receives two arguments: the new answer found during the evaluation (ANS) and the subgoal frame which corresponds to the call at hand (SF). The algorithm begins by checking/inserting the given ANS into the answer trie structure, which will return the leaf node for the path representing ANS (line 1).

Then, it checks/inserts the given *leaf* node into the private consumer chain for the current thread, which will return the corresponding chain node (line 2). Next in line 3, it tests whether the chain node already existed in the consumer chain, i.e., if it was inserted or not by the current check/insert operation in order to return failure (line 4), or it proceeds with marking the answer ANS has found (line 6). At the end (lines 7 to 10), it returns failure if local scheduling is active (line 8), otherwise, the batched scheduling is active, thus it propagates the answer ANS (line 10).

4 Performance Analysis

We now present experimental results about the usage of the batched scheduling on FS design. To put results in perspective, we will be showing also the results for the NS design. The environment for our experiments was a machine with 32-Core AMD Opteron (TM) Processor 6274 (2 sockets with 16 cores each) with 32G of main memory, running the Linux kernel is the 3.16.7-200.fc20.x86_64 with Yap 6.3.

For the experimentation, we used our memory allocator described in the work [1], later named *TabMalloc*, and we the same five sets of benchmarks. These benchmarks create *worst case* scenarios, where we are able to show the lowest bounds in terms of performance that each design might achieve when applied/used with other real world applications/programs.

Table 1 shows the overhead ratios, when compared with the NS design with 1 thread (running with local scheduling without TabMalloc), for the NS and FS designs, when running 1, 8, 16, 24 and 32 threads, with TabMalloc and, local scheduling (column *Local*) and batched scheduling (column *Batched*) strategies. In order to give a fair weight to each benchmark, the overhead ratio is calculated as follows. We begin by running ten times each benchmark B for each design D with T threads. Then, we calculate the average of those ten runs and use that value (D_{BT}) to put it in perspective against the base time, which is the average of the ten runs of the NS design with 1 thread (NS_{B1}). For that, we use the following formula for the overhead $O_{DBT} = D_{BT}/NS_{B1}$. After calculating all the overheads O_{DBT} for a certain design D and number of threads T corresponding to the several benchmarks B, we calculate the respective minimum, average, maximum and standard deviation overhead ratios.

By observing Table 1, we can see that, for 1 thread, on the minimum overhead ratio, local scheduling is better on the NS design, while batched scheduling is better in the FS design. For the average and maximum overhead ratios, in the FS design, the best strategy is always the local scheduling.

As we scale the number of threads, for the NS design both scheduling strategies have similar minimum, average and maximum overhead ratios. For the FS design, the best minimum overhead ratio is always for the batched scheduling. For the average and maximum overhead ratio, local scheduling is always better

Table 1. Overhead ratios, when compared with the NS design with 1 thread for the NS and FS designs, when running 1, 8, 16, 24 and 32 threads with local and batched scheduling (lowest is the best and it is marked in bold by row and by design for the Minimum, Average and Maximum)

Threads		**NS**		**FS**	
		Local	**Batched**	**Local**	**Batched**
1	**Min**	**0.53**	0.55	1.01	**0.95**
	Avg	**0.78**	0.82	**1.30**	1.46
	Max	1.06	**1.05**	**1.76**	2.33
	StD	0.15	0.14	0.22	0.44
8	**Min**	0.66	**0.63**	1.16	**0.99**
	Avg	**0.85**	0.88	**1.88**	1.95
	Max	**1.12**	1.14	**2.82**	3.49
	StD	0.13	0.14	0.60	0.79
16	**Min**	0.85	**0.75**	1.17	**1.06**
	Avg	**0.98**	1.00	**1.97**	2.08
	Max	**1.16**	1.31	**3.14**	3.69
	StD	0.09	0.17	0.65	0.83
24	**Min**	**0.91**	0.93	1.16	**1.09**
	Avg	**1.15**	1.16	**2.06**	2.19
	Max	1.72	**1.60**	**3.49**	4.08
	StD	0.20	0.21	0.70	0.91
32	**Min**	1.05	**1.04**	1.33	**1.26**
	Avg	1.51	**1.49**	**2.24**	2.41
	Max	**2.52**	2.63	**3.71**	4.51
	StD	0.45	0.45	0.74	1.02

than batched scheduling. However, for the average ratio, the results between both strategies is quite close. For the maximum overhead ratio, the difference between both strategies is significantly higher.

5 Conclusions and Further Work

We have presented a simple and novel approach for combining the FS design with batched scheduling. Experimental results showed that on average, the extra structures required for the combination do not seem to have a big impact in the performance. Further work will include experimentation in other real world applications.

References

1. Areias, M., Rocha, R.: An Efficient and Scalable Memory Allocator for Multi-threaded Tabled Evaluation of Logic Programs. In: International Conference on Parallel and Distributed Systems. pp. 636–643. IEEE Computer Society (2012)

2. Areias, M., Rocha, R.: 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 12(4 & 5), 427–443 (2012)
3. Areias, M., Rocha, R.: A lock-free hash trie design for concurrent tabled logic programs. International Journal of Parallel Programming pp. 1–21 (2015)
4. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM 43(1), 20–74 (1996)
5. Freire, J., Swift, T., Warren, D.S.: Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In: International Symposium on Programming Language Implementation and Logic Programming. pp. 243–258. No. 1140 in LNCS, Springer (1996)
6. Marques, R., Swift, T.: Concurrent and Local Evaluation of Normal Programs. In: International Conference on Logic Programming. pp. 206–222. No. 5366 in LNCS, Springer (2008)
7. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems 20(3), 586–634 (1998)
8. Wielemaker, J.: Native Preemptive Threads in SWI-Prolog. In: International Conference on Logic Programming. pp. 331–345. No. 2916 in LNCS, Springer (2003)