

On Extending a Full-Sharing 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. Keywords: Multithreading, Tabling, Concurrency, Batched 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 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 [8], 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 [7] and Yap [2] were the only Prolog systems that were able to combine both multithreading with tabling. 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: (i) The **No-Sharing (NS)** design, where each thread allocates fully private tables for each new tabled subgoal called during its computation. (ii) **Subgoal-Sharing (SS)**, where threads share part of the table space and the (iii) **Full-Sharing (FS)**, where threads share the complete table space.

In this paper we discuss a novel approach for solving the problem of supporting multithreaded batched scheduling in the FS design and we present a performance analysis comparison between local scheduling with batched scheduling.

Experimental results show that despite the extra data structures required to extend the FS design with batched scheduling, the execution time of the FS design with batched scheduling is still quite competitive when comparing against 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. Next, we show the most important implementation details and finally, we discuss experimental results and we end by outlining some conclusions.

2 Background

This section introduces some background needed for the following sections.

2.1 No-Sharing, Subgoal-Sharing and Full-Sharing Designs

2.2 Batched Scheduling

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 strategy schedules the evaluation of a program in a breath-first manner. It favors the backtracking first with completion instead of the forward execution, leaving the consumption of answers for last. Thus, it only allows a Cluster of Dependent Subgoals (CDS) to return answers only after the completion point has been reached [5]. In other words, the local scheduling tries to keep a 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 [8] and delaying the completion point to an older generator node.

3 Full-Sharing with Batched Scheduling

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 1 shows the key data structures for supporting the implementation of the PAC procedure during the evaluation of a tabled subgoal call $P_{i,j}$ using the *FS* design. The *FS* design, uses a subgoal entry data structure to store common information for a subgoal call and a subgoal frame (*SF*) data structure to store private information about the execution of each thread. The PAC procedure works at the subgoal frame level, which is private to each thread.

Figure 1(a) shows then a situation where two threads, T_1 and T_{k-1} , are sharing the same subgoal entry call $P_{i,j}$ 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 $P_{i,j}$. For the sake of simplicity, we are omitting the internal answer trie nodes and we are only showing the leaf nodes LN_1 , LN_2 and LN_3 , in the figure. With the PAC procedure, the leaf nodes are not chained in the *AT* data structure. Now, the chaining process is done privately, and for that, we use the subgoal frame structure of each thread. 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. In particular, on each hash trie level, we have removed the tools necessary to support concurrency, such as useless pointers and compare-and-swap operations. We have chosen this hashing mechanism, because it showed a good balance between lookup and insert operations [3], but the major reason was mostly because of the integration in the TabMalloc memory allocator [1].

Going back to Figure 1(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

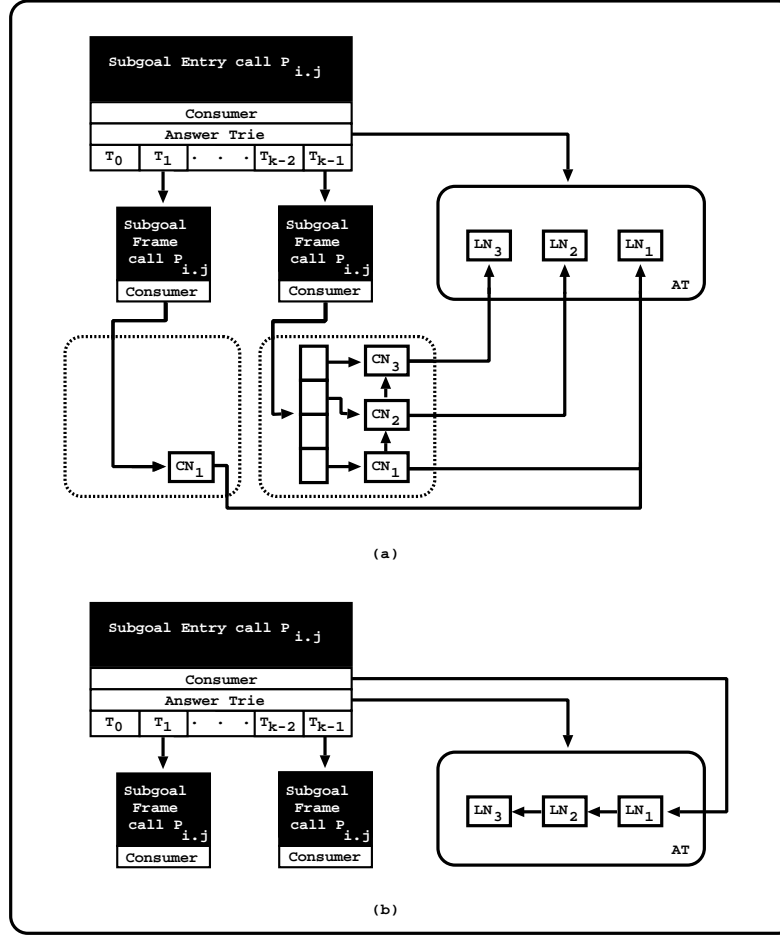


Fig. 1. The *FS* design with the PAC procedure - (a) private chaining and (b) public chaining

that consumer nodes belonging to thread T_{k-1} can consume the answers as in the original mechanism.

Figure 1(b) shows the state of the subgoal call after completion (recall that after completion of a subgoal call, the threads use loader nodes to consume the answers). When a thread T completes a subgoal call, it frees its private consumer structures, but before doing that, it checks whether another thread has 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 reproduce the 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 1(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 implementation level, 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. For the NS and SS designs the support for batched scheduling is immediate, since the answer trie data structure is not shared among threads. The usage of batched scheduling with the FS design requires further support since with batched scheduling, answers are immediately propagated and we have to ensure that the propagation of an answer occurs on all subgoal calls one and only once. To do so, we take advantage of the private chaining procedure, presented in the previous subsection, as a way to keep, for every subgoal call of every thread, track of all the answers that were already propagated. This requires minor changes to the *tabled new answer* tabling operation. Algorithm 1 shows how we have extended the tabled new answer operation to support the 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: if NS_design or SS_design then
3:   ... {without changes}
4: else {FS design}
5:   chain  $\leftarrow$  check_insert_consumer_chain(leaf, SF)
6:   if is_answer_marked_as_found(chain) = True then
7:     return failure
8:   else {the answer is new}
9:     mark_answer_as_found(chain)
10:    if local_scheduling_mode(SF) then
11:      return failure
12:    else {batched scheduling mode}
13:      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 *NS_design*, *SS_design* and *FS_design* macros define which table design is enabled.

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). In line 2, it then tests whether one of the NS or SS designs are active, and in such a case, the algorithm is remains unchanged.

Otherwise, for the FS design (lines 4 to 13), it checks/inserts the given *leaf* node into the private consumer chain for the current thread, which will return the corresponding chain node. In line 6, it then 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 7), or it proceed with marking the answer *ANS* has found (line 9). At the end (lines 10 to 13), it returns failure if local scheduling is active (line 11), otherwise, the batched scheduling is active, thus it propagates the answer *ANS* (line 13).

4 Performance Analysis

We now present experimental results about the usage of the batched scheduling on the *NS*, *SS* and *FS* designs. 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, each processor with caches *L1*, *L2* and *L3* respectively with the sizes of 64K, 2048K and 6144K, running the Linux kernel is the 3.16.7-200.fc20.x86_64, with Yap 6.3 compiled with gcc 4.8.

We used five sets of benchmarks. The *Large Joins* and *WordNet* sets were obtained from the OpenRuleBench project [6]; 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. We experimented graphs with the *BTree* configuration with depth 17, the *Pyramid* and *Cycle* configurations with depth 2000 and the *Grid* configuration with depth 35.

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 1 have the following meaning: (i) **calls:** is the number of different calls to tabled subgoals. It corresponds to the number of paths in the subgoal tries. (ii) **trie nodes:** is the total number of trie nodes allocated in the corresponding subgoal/answer trie structures. (iii) **trie depth:** is the minimum/average/maximum number of trie node levels required to represent a path in the corresponding subgoal/answer trie structures. Trie structures with smaller average values are more amenable to higher lock contention. (iv) **unique:** is the number of different tabled answers found. It corresponds to the number of paths in the answer tries. (v) **repeated:** is the number of redundant tabled answers found. (vi) **NS:** is the average execution time, in seconds, of ten runs for 1 thread with the NS design without the TabMalloc memory allocator. In what follows, we will use these times as the base times when computing the overhead ratios for the other designs.

By observing Table 1, 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.

Table 1. Characteristics of the benchmark programs

Bench	Tabled Subgoals			Tabled Answers				Time (sec)
	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	2.85
Mondial	35	42	3/4/4	2,664	2,452,890	14,334	6/7/7	0.84
WordNet								
Clusters	117,659	235,319	2/2/2	166,877	161,853	284,536	1/1/1	0.83
Holo	117,657	235,315	2/2/2	74,838	54	192,495	1/1/1	0.75
Hyper	117,657	235,315	2/2/2	698,472	8,658	816,129	1/1/1	1.42
Hypo	117,657	117,659	2/2/2	698,472	20,341	816,129	1/1/1	1.53
Mero	117,657	117,659	2/2/2	74,838	13	192,495	1/1/1	0.74
Tropo	117,657	235,315	2/2/2	472	0	118,129	1/1/1	0.66
Model Checking								
IPProto	1	6	5/5/5	134,361	385,423	1,554,896	4/51/67	2.70
Leader	1	5	4/4/4	1,728	574,786	41,788	15/80/97	3.51
Sieve	1	7	6/6/6	380	1,386,181	8,624	21/53/58	18.50
Path Left								
BTree	1	3	2/2/2	1,966,082	0	2,031,618	2/2/2	1.53
Cycle	1	3	2/2/2	4,000,000	2,000	4,002,001	2/2/2	3.52
Grid	1	3	2/2/2	1,500,625	4,335,135	1,501,851	2/2/2	1.93
Pyramid	1	3	2/2/2	3,374,250	1,124,250	3,377,250	2/2/2	3.08
Path Right								
BTree	131,071	262,143	2/2/2	3,801,094	0	3,997,700	1/2/2	2.33
Cycle	2,001	4,003	2/2/2	8,000,000	4,000	8,004,001	1/2/2	3.55
Grid	1,226	2,453	2/2/2	3,001,250	8,670,270	3,003,701	1/2/2	2.32
Pyramid	3,000	6,001	2/2/2	6,745,501	2,247,001	6,751,500	1/2/2	3.17

Note that for the moment our goal is to evaluate the robustness of our implementation when exposed to *worst case* scenarios. By focusing on *worst case* scenarios, we show the lowest bounds in terms of performance that each design might achieve when applied/used with other real world applications/programs. Moreover, by testing the framework with worst case scenarios, we avoid the peculiarities of the program at hand and we try to focus on measuring the real value of our designs. Thus, we will follow a common approach to create worst case scenarios and we will run all threads starting with the same query goal. By doing this, it is expected that all threads will access the table space, to check/insert for subgoals and answers, at similar times, thus causing a huge stress on the same critical regions. To put the results in perspective, we experimented with 1, 8, 16, 24 and 32 threads (the maximum number of cores available in our machine)

with local and batched scheduling on the five sets of benchmarks presented in the early subsection.

Table 2 shows the overhead ratios, when compared with the NS design with 1 thread (running with local scheduling), for the NS, SS and FS designs (all running TabMalloc), when running 1, 8, 16, 24 and 32 threads with local and batched scheduling on the five sets of benchmarks. For each design, the table has then two columns, a column with **Local** that shows results for the local scheduling and a column with **Batched** with results for batched scheduling. The overhead results presented in both **Local** and **Batched** columns use as base time the execution times presented in the **NS** column of the Table 1.

Table 2. Overhead ratios, when compared with the NS design with 1 thread (running with local scheduling without TabMalloc) for the NS, SS, FS designs (with TabMalloc), when running 1, 8, 16, 24 and 32 threads with local and batched scheduling on the five sets of benchmarks (best ratios by row and by design for the Minimum, Average and Maximum are in bold)

Threads	NS		SS		FS	
	Local	Batched	Local	Batched	Local	Batched
1	Min	0.53 0.55	0.54 0.55		1.01	0.95
	Avg	0.78 0.82	0.84 0.90		1.30	1.46
	Max	1.06 1.05	1.04 1.04		1.76	2.33
	StD	0.15 0.14	0.17 0.16		0.22	0.44
8	Min	0.66 0.63	0.66 0.63		1.16	0.99
	Avg	0.85 0.88	0.92 0.93		1.88	1.95
	Max	1.12 1.14	1.20 1.15		2.82	3.49
	StD	0.13 0.14	0.15 0.14		0.60	0.79
16	Min	0.85 0.75	0.82 0.77		1.17	1.06
	Avg	0.98 1.00	1.04 1.05		1.97	2.08
	Max	1.16 1.31	1.31 1.28		3.14	3.69
	StD	0.09 0.17	0.12 0.13		0.65	0.83
24	Min	0.91 0.93	1.02 0.98		1.16	1.09
	Avg	1.15 1.16	1.22 1.19		2.06	2.19
	Max	1.72 1.60	1.81 1.61		3.49	4.08
	StD	0.20 0.21	0.18 0.16		0.70	0.91
32	Min	1.05 1.04	1.07 1.12		1.33	1.26
	Avg	1.51 1.49	1.54 1.51		2.24	2.41
	Max	2.52 2.63	2.52 2.62		3.71	4.51
	StD	0.45 0.45	0.42 0.43		0.74	1.02

By observing Table 2, we can see that, for one thread, on average, local scheduling is slightly better than batched on the three designs. For the NS design we have 0.78 and 0.82, for the SS design we have 0.84 and 0.90 and for the FS design we have 1.30 and 1.46 average overhead ratios, for the local and batched scheduling strategies, respectively. As we scale the number of threads, one can observe that, for the NS and SS designs both scheduling strategies have

similar minimum, average and maximum overhead ratios. For the FS design, the best minimum overhead ratio is always for batched scheduling. For the average and maximum overhead ratio, local scheduling is always better than batched scheduling. In summary, we can say that both the local and batched scheduling strategies have similar overhead results on worst case scenarios for the NS, SS and FS designs.

5 Conclusions and Further Work

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. Liang, S., P.Fodor, Wan, H., M.Kifer: OpenRuleBench: An Analysis of the Performance of Rule Engines. In: International World Wide Web Conference. pp. 601–610. ACM (2009)
7. 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)
8. 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)