Concurrent Table Accesses in Parallel Tabled Logic Programs

Ricardo Rocha[1], Fernando Silva[1], and Vítor Santos Costa[2]

[1] DCC-FC & LIACC
University of Porto, Portugal
{ricroc,fds}@ncc.up.pt
[2] COPPE Systems & LIACC
Federal University of Rio de Janeiro, Brazil
vitor@cos.ufrj.br

Abstract. Tabling is an implementation technique that improves the declarativeness and expressiveness of Prolog by reusing answers to subgoals. The declarative nature of tabled logic programming suggests that it might be amenable to parallel execution. On the other hand, the complexity of the tabling mechanism, and the existence of a shared resource, the table, may suggest that parallelism might be limited and never scale for real applications. In this work, we propose three alternative locking schemes to deal with concurrent table accesses, and we study their impact on the OPTYap parallel tabling system using a set of tabled programs.

1 Introduction

Tabling (or *tabulation* or *memoing*) is an implementation technique where results for subcomputations are stored and reused. Tabling has proven to be particularly effective in logic programs: tabling can reduce the search space, avoid looping, and in general have better termination properties than traditional Prolog. The XSB Prolog system [1] is the most well known tabling Prolog system, with excellent results in application areas such as Natural Language Processing, Knowledge Based Systems, Model Checking, and Program Analysis.

One extra advantage of tabling is that tabled programs are most often pure logic programs, and are thus amenable to the implicit exploitation of parallelism. Because tabling has often been used to reduce search space, or-parallelism is most interesting. We thus proposed OPTYap [2], a design for combining implementation techniques for or-parallelism in shared-memory machines, namely environment copying [3], with the WAM extensions originally proposed in XSB [4]. Results have shown that OPTYap can achieve excellent speedups, while introducing low overheads for sequential execution [5].

The performance of tabling largely depends on the implementation of the table itself. The table will be called very often, therefore fast lookup and insertion is mandatory. Applications can make millions of different calls, hence compactness is also required. The XSB design used *tries* to implement this goal [6]. Tries are trees in which there is one node for every common prefix [7]. Tries have

proven to be one of the main assets of XSB, because they are quite compact for most applications, while having fast lookup and insertion.

One critical issue in our parallel design was whether tries would be effective in the presence of concurrent accesses. One of the first implementations of tries in a parallel environment was the work by Chan and Lim [8], where tries were used to index words for alphabets with a finite number of symbols. In our work, we use tries to index Prolog terms, which can have an infinite number of symbols. We address concurrency by extending the trie structure originally proposed in XSB to support locking mechanisms. To achieve best performance, different implementations may be pursued. We can have one lock per table entry, one lock per path, or one lock per node. We can also have hybrid locking schemes combining the above. Our initial results did show that naive approaches could generate significant overheads or result in minimal concurrency. We thus studied alternative locking schemes that try to reduce overheads by only locking part of the tree when strictly necessary, and evaluated their performance. Our results show almost-linear speedups up to 32 CPUs on an Origin2000. Although our context is parallel tabling, we believe that these experiments will be of interest to application areas that rely on access to frequently updated trees.

The remainder of the paper is organized as follows. First, we introduce the trie data structure and the table space organization. Next, we describe the three alternative locking schemes implemented in OPTYap. We then end by presenting some initial results and outlining some conclusions.

2 Table Space

Tabling is about storing intermediate answers for subgoals so that they can be reused when a repeated subgoal appears. Execution proceeds as follows. Whenever a tabled subgoal S is first called, an entry for S is allocated in the *table space*. This entry will collect all the answers found for S. Repeated calls to *variants* of S are resolved by consuming the answers already stored in the table. Meanwhile, as new answers are generated, they are inserted into the table and returned to all variant subgoals.

The table space can be accessed in a number of ways: **(i)** to look up if a subgoal is in the table, and if not insert it; **(ii)** to verify whether a newly found answer is already in the table, and if not insert it; and, **(iii)** to load answers to variant subgoals. Hence, a correct design of the algorithms to access and manipulate the table data is critical to achieve an efficient implementation. Our implementation uses tries as proposed by Ramakrishnan *et al.* [6].

Tries were first proposed to index dictionaries [7] and have since been generalized to index recursive data structures such as terms (see [6, 9–11] for use of tries in tabled logic programs, automated theorem proving and term rewriting). An essential property of the trie structure is that common prefixes are represented only once. The efficiency and memory consumption of a particular trie largely depends on the percentage of terms that have common prefixes. For tabled logic programs, we often can take advantage of this property.

Each different path through the nodes in the trie, the *trie nodes*, corresponds to a term. The entry point is called the root node, internal nodes represent symbols in terms and leaf nodes specify completed terms. Terms with common prefixes branch off from each other at the first distinguishing symbol. Inserting a term requires in the worst case allocating as many nodes as necessary to represent its complete path. On the other hand, inserting repeated terms requires traversing the trie structure until reaching the corresponding leaf node, without allocating any new node. During traversal, each child node specifies the next symbol to be inspected in the input term. A transition is taken if the symbol in the input term at a given position matches a symbol on a child node. Otherwise, a new child node representing the current symbol is added and an outgoing transition from the current node is made to point to the new child. On reaching the last symbol in the input term, we reach a leaf node in the trie.

Figure 1 shows an example for a trie with three terms. Initially, the trie contains the root node only. Next, we insert $f(X, a)$. As a result, we create three nodes: one for the functor $f/2$, next for the variable X, and last for the constant a. The second step is to insert $g(X, b, Y)$. The two terms differ on the main functor, so tries bring no benefit here. In the last step, we insert $f(Y, 1)$ and we save two common nodes with $f(X, a)$. Notice the way variables are represented. We follow the formalism proposed by Bachmair *et al.* [10], where each variable is represented as a distinct constant.

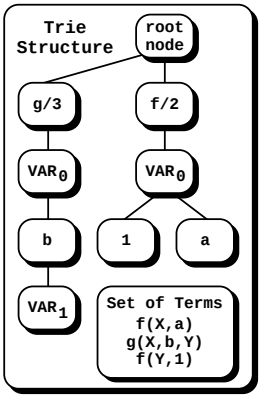

Fig. 1. Tries for terms

The implementation of tries requires four fields per trie node. The first field (`TrNode_symbol`) stores the symbol for the node. The second (`TrNode_child`) and third (`TrNode_parent`) fields store pointers respectively to the first child node and to the parent node. The fourth field (`TrNode_next`) stores a pointer to the sibling node, in such a way that the outgoing transitions from a node can be collected by following its first child pointer and then the list of sibling pointers. We next present how tries are used to implement tabled predicates. Figure 2 shows an example for a predicate $f/2$ after the execution of the following operations:

```
tabled_subgoal_call: f(X,a)
tabled_subgoal_call: f(Y,1)
tabled_new_answer:   f(0,1)
tabled_new_answer:   f(n(0),1)
```

We use two levels of tries: one stores the subgoal calls, the other the answers. Each different call to a tabled predicate corresponds to a unique path through the *subgoal trie structure*. Such a path always starts from a *table entry* data structure, follows a sequence of the *subgoal trie nodes*, and terminates at a leaf data structure, the *subgoal frame*. Each subgoal frame stores information about the subgoal, namely an entry point to its *answer trie structure*. Each unique path through the *answer trie nodes* corresponds to a different answer to the entry subgoal. When inserting new answers, we only store the substitutions for the unbound variables in the subgoal call. This optimization is called *substitution*

factoring [6]. Leaf answer nodes are chained in a linked list in insertion time order, so that we can recover answers in the same order they were inserted. The subgoal frame points to the first and last answer in this list. Thus, a variant subgoal only needs to point at the leaf node for its last loaded answer, and consumes more answers just by following the chain. To load an answer, the trie nodes are traversed in bottom-up order and the answer is reconstructed.

Traversing a trie to check/insert for new calls or for new answers is implemented by repeatedly invoking a `trie_check_insert()` procedure for each symbol that represents the term being checked. Given a symbol S and a parent node P, the procedure returns the child node of P that represents the given symbol S. Figure 3 shows the pseudo-code. Initially it traverses the chain of sibling nodes that represent alternative paths from the given parent node and checks for one representing the given symbol. If such a node is found then execution is stopped and the node returned. Otherwise, a new trie node is allocated and inserted in the beginning of the chain. To allocate new trie nodes, we use a `new_trie_node()` procedure with four arguments, each one corresponding to the initial values to be stored respectively in the `TrNode_symbol`, `TrNode_child`, `TrNode_parent` and `TrNode_next` fields of the new allocated node.

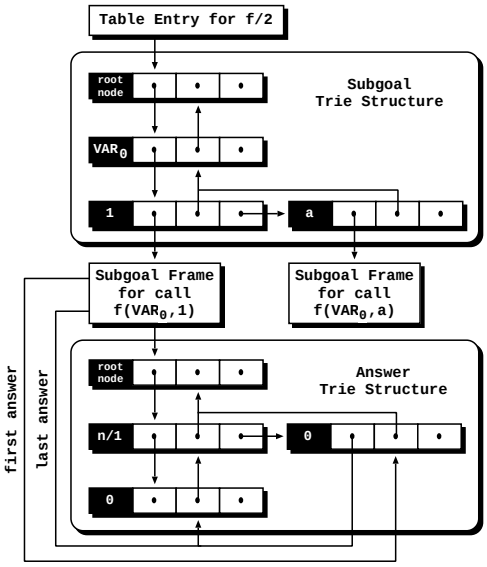

Fig. 2. Tries to organize the table space

```
trie_check_insert(symbol s, trie node parent) {
   child = TrNode_child(parent)
   while (child) {          // check if a node for s was already inserted
      if (TrNode_symbol(child) == s)
         return child                                    // node found
      child = TrNode_next(child)
   }
   child = new_trie_node(s, NULL, parent, TrNode_child(parent))
   TrNode_child(parent) = child              // insert the new node for s
   return child
}
```

Fig. 3. Pseudo-code for `trie_check_insert()`

3 Table Locking Schemes

There are two critical issues that determine the efficiency of a locking scheme for the table. One is *lock duration*, that is, the amount of time a data structure is held. The other is *lock grain*, that is, the number 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 table locking approaches. For instance, if the lock scheme is short duration or fine grained, then inserting many trie nodes in sequence, corresponding to a long trie path, may result in a large number of lock requests. On the other hand, if the lock scheme is long duration or coarse grain, then going through a trie path without extending or updating its trie structure, may unnecessarily lock data and prevent possible concurrent access by others. Unfortunately, it is impossible beforehand to know which locking scheme would be optimal.

The *Table Lock at Node Level (TLNL)* was our first implemented scheme. It only enables a single writer per chain of sibling nodes that represent alternative paths from a common parent node. Figure 4 shows the pseudo-code that implements it. The main difference from the original procedure is that in TLNL we lock the parent node while accessing its children nodes. Locking is done by applying a mask to the node address in order to index a global array of lock entries. Within this scheme, the period of time a node is locked is proportional to the average time needed to traverse its children nodes, and the number of lock requests is proportional to the length of the path.

```
trie_check_insert(symbol s, trie node parent) {
   lock(parent)                                  // locking the parent node
   child = TrNode_child(parent)
   while (child) {
      if (TrNode_symbol(child) == s) {
         unlock(parent)                          // unlocking before return
         return child
      }
      child = TrNode_next(child)
   }
   child = new_trie_node(s, NULL, parent, TrNode_child(parent))
   TrNode_child(parent) = child
   unlock(parent)                                // unlocking before return
   return child
}
```

Fig. 4. Pseudo-code for the TLNL scheme

The *Table Lock at Write Level (TLWL)* scheme improves TLNL by reducing lock duration. Like TLNL, TLWL only enables a single writer per chain of sibling nodes, but the common parent node is only locked when writing to the table is likely. Figure 5 shows the pseudo-code for TLWL. Initially, the chain of sibling nodes that follow the given parent node is traversed without locking. The parent node must be locked only when the given symbol is not found. This avoids locking when the symbol already exists in the chain. Moreover, it delays locking while insertion of a new node to represent the symbol is not likely. Note that we need to check if, during our attempt to lock, other worker expanded the chain to include the given symbol. Within this scheme, the number of lock requests is, on average, lower than TLNL. It ranges from zero to the number of nodes in path. Similarly, the amount of time a node is locked is also, on average, smaller. It includes the time needed to check the nodes that in the meantime were inserted by other workers, if any, plus the time to allocate and initialize the new node.

Last, the *Table Lock at Write Level - Allocate Before Check (TLWL-ABC)* scheme is a variant of TLWL. It also follows the *probable node insertion notion*,

```
trie_check_insert(symbol s, trie node parent) {
   child = TrNode_child(parent)
   initial_child = child                  // keep the initial child node
   while (child) {         // traverse the initial chain of sibling nodes
      if (TrNode_symbol(child) == s)
         return child
      child = TrNode_next(child)
   }
   lock(parent)
   child = TrNode_child(parent) // traverse the nodes inserted in the ...
   while (child != initial_child) {     // ... meantime by others, if any
      ...                                                 // the same as TLNL
   }
   ...                                                    // the same as TLNL
}
```

Fig. 5. Pseudo-code for the TLWL scheme

but uses a different strategy to decide on when to allocate a node. In order to reduce to a minimum the lock duration, it anticipates the allocation and initialization of nodes that are likely to be inserted in the table before locking. However, if in the meantime a different worker introduces first an identical node, we pay the cost of having pre-allocated an unnecessary node that has to be additionally freed. Figure 6 presents the pseudo-code that implements this scheme.

```
trie_check_insert(symbol s, trie node parent) {
   ...                                                    // the same as TLWL
   pre_alloc = new_trie_node(s, NULL, parent, NULL)   // pre-allocate ...
   lock(parent)                         // ... a node for s before locking
   child = TrNode_child(parent)
   while (child != initial_child) {
      if (TrNode_symbol(child) == s) {
         unlock(parent)
         free(pre_alloc)                  // free the pre-allocated node
         return child
      }
      child = TrNode_next(child)
   }
   TrNode_next(pre_alloc) = TrNode_child(parent)
   TrNode_child(parent) = pre_alloc      // insert the pre-allocated node
   unlock(parent)
   return pre_alloc
}
```

Fig. 6. Pseudo-code for the TLWL-ABC scheme

4 Preliminary Results

In order to evaluate the scalability of our locking schemes, we ran OPTYap for a set of selected programs in a Cray Origin2000 parallel computer with 96 MIPS 195 MHz R10000 processors. We selected the programs that showed significant speedups for parallel execution in previous work [2, 5]. The programs include the transition relation graphs for two model-checking specifications, a same generation problem for a 24x24x2 data cylinder, and a transitive closure of a 25x25 grid using left recursion. All programs find all the solutions for the problem.

In order to get a deeper insight on the behavior of each program, we first characterize the programs in Table 1. The columns have the following meaning: *time* is the execution time in seconds with a single worker; *sg* is the number

of different tabled subgoal calls; *unique* is the number of answers stored in the table; *repeated* is the number of redundant answers found; *nodes* is the number of trie nodes allocated to represent the set of answers; and *depth* is the average number of trie nodes required to represent an answer. In parentheses, it shows the percentage of saving that the trie's design achieves on these data structures. Consider Figure 1 as an example, it requires 10 nodes to represent individually all answers but it uses only 8, thus achieving a saving of 20%. Smaller depth values or higher percentages of saving reflect higher probabilities of lock contention when concurrently accessing the table space.

Program	time	sg	unique	repeated	nodes	depth
mc-sieve	268	1	380	1386181	8624	53(57%)
mc-iproto	24	1	134361	385423	1554896	51(77%)
samegen	26	485	23152	65597	24190	1.5(33%)
lgrid	69	1	160000	449520	160401	2(49%)

Table 1. Program characteristics

Table 1 indicates that *mc-sieve* is the program least amenable to lock contention because it finds the least number of answers and has the deepest trie structures. In this regard, *lgrid* is the opposite case. It finds the largest number of answers and it has very shallow trie structures. Likewise, *samegen* also shows a very shallow trie structure, despite that it can benefit from its large number of different tabled subgoal calls. It is the case, because the answers found for different subgoals can be inserted without overlap. Finally, *mc-iproto* can also lead to higher ratios of lock contention. It shows the highest percentage of saving and it inserts a huge number of trie nodes in the table.

Table 2 shows the speedups for the three locking schemes with varying number of workers. The speedups are relative to the single worker case. A main conclusion can be easily drawn from the results presented: TLWL and TLWL-ABC show identical speedup ratios and they are the only schemes showing scalability. In particular, for the *mc-sieve* program they show superb speedups up to 32 workers. Closer analysis allows us to observe other interesting aspects: the more refined strategy of TLWL-ABC does not show to perform better than TLWL; all schemes show identical speedups for *samegen*; and TLNL clearly slows down for more than 16 workers. The good behavior with *samegen* arises from the fact that this program calls 485 different tabled subgoals. This increases the number of entries where answers can be stored thus reducing contention points. The TLNL slowdown is related to the fact that this scheme locks the table even when writing is not likely. In particular, for repeated answers it pays the cost of performing locking operations without inserting any new node.

	Workers			
Schemes	**8**	**16**	**24**	**32**
mc-sieve				
TLNL	7.2	11.8	3.9	4.7
TLWL	7.9	15.8	23.7	31.5
TLWL-ABC	7.9	15.8	23.7	31.4
mc-iproto				
TLNL	2.6	1.8	1.0	1.0
TLWL	5.0	9.0	8.8	7.2
TLWL-ABC	5.1	7.7	8.4	7.1
samegen				
TLNL	7.2	13.8	19.6	24.0
TLWL	7.2	13.9	19.7	24.1
TLWL-ABC	7.2	13.9	19.7	24.2
lgrid				
TLNL	6.7	12.1	6.2	5.3
TLWL	7.1	13.5	19.9	24.3
TLWL-ABC	6.9	13.4	18.9	24.2

Table 2. Speedups

During parallel execution, not only concurrency can be a major source of overhead. For some programs, the complexity of the tabling mechanism can induce some intricate dependencies that may always constraint parallel execution. Besides, as tabling, by nature, reduces the potential non-determinism available in logic programs, the source of parallelism may also be intrinsically limited.

To better understand the parallel execution behavior of our set of programs, we gathered in Table 3 a set of statistics regarding the number of contention points when using our best table locking scheme – TLWL. By contention points we mean the number of unsuccessful first attempts to lock a data structure. We distinguish three kind of locking attempts: **(i)** locking related with *trie nodes*, when inserting new subgoal calls or new answers; **(ii)** locking related with *subgoal frames*, when updating the subgoal frame pointers to point to the last found answer; and **(iii)** locking related with *variant subgoals*, when synchronizing access to check for available answers to variant subgoals. Note that TLWL only affects contention related with trie nodes.

Contention Points	**Workers**			
	8	**16**	**24**	**32**
mc-sieve				
trie nodes	188	415	677	1979
subgoal frames	0	0	0	2
variant subgoals	0	1	0	4
mc-iproto				
trie nodes	6579	10537	11816	11736
subgoal frames	9894	21271	33162	33307
variant subgoals	4685	25006	66334	81515
samegen				
trie nodes	119	201	364	417
subgoal frames	52	112	283	493
variant subgoals	0	1	0	0
lgrid				
trie nodes	5292	10341	12870	12925
subgoal frames	1124	7319	17440	27834
variant subgoals	1209	5987	23357	35991

Table 3. Contention points with TLWL

The insignificant number of contention points obtained for *mc-sieve* supports the excellent speedups observed for parallel execution. In this regard, the contention also obtained for *samegen* indicates that locking is not a problem, and that the small overhead observed for *samegen* in Table 2 is thus mainly related with the complexity of the tabling mechanism. On the other hand, regarding *mc-iproto* and *lgrid*, lock contention is a major problem. For trie nodes they show identical numbers, but *mc-iproto* inserts about 10 times more answer trie nodes than *lgrid*. For subgoal frames and variant subgoals they show a similar pattern, but *mc-iproto* has higher contention ratios per time unit (remember from Table 1 that *mc-iproto* is about 3 times faster than *lgrid*), hence justifying its worst behavior with the increase in the number of workers. For these programs, the sequential order by which leaf answer nodes are chained in the trie seems to be the key issue that reflects the high number of contention points. After inserting a new answer we need to update the subgoal frame to point to the last found answer. When checking for answers to variant subgoals we need to lock and possibly update the variant subgoal to point to the last loaded answer. For programs that find a large number of answers per time unit, this obviously increases contention when accessing such pointers, and thus obtaining good speedups in the presence of these conditions will always be a difficult task.

5 Concluding Remarks

We studied the impact of using alternative locking schemes to deal with concurrent table accesses in parallel tabling. We used OPTYap, that to the best of our knowledge, is the only available parallel tabling system for logic programming. Through experimentation, we observed that there are locking schemes that can obtain good speedup ratios and achieve scalability. Our results show that a main problem is not only how we lock trie nodes to insert items, but also how we use auxiliary data structures to synchronize access to the table space. In this regard, a key issue is the sequential order by which leaf answer nodes are chained in the trie structure. In the future, we plan to investigate whether alternative designs can obtain scalable speedups even when frequently updating/accessing tables.

Acknowledgments

This work has been partially supported by APRIL (POSI/SRI/40749/2001) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSI.

References

1. Sagonas, K., Swift, T., Warren, D.S.: XSB as an Efficient Deductive Database Engine. In: ACM SIGMOD International Conference on the Management of Data, ACM Press (1994) 442–453
2. Rocha, R., Silva, F., Santos Costa, V.: On a Tabling Engine that Can Exploit Or-Parallelism. In: International Conference on Logic Programming. Number 2237 in LNCS, Springer-Verlag (2001) 43–58
3. Ali, K., Karlsson, R.: The Muse Approach to OR-Parallel Prolog. International Journal of Parallel Programming **19** (1990) 129–162
4. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems **20** (1998) 586–634
5. Rocha, R., Silva, F., Costa, V.S.: Achieving Scalability in Parallel Tabled Logic Programs. In: International Parallel and Distributed Processing Symposium, IEEE Computer Society (2002)
6. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38** (1999) 31–54
7. Fredkin, E.: Trie Memory. Communications of the ACM **3** (1962) 490–499
8. Chan, I.W., Lim, C.Y.: Parallel Implementation of the Trie Structure. In: International Conference on Parallel and Distributed Systems, IEEE Computer Society (1994) 538–543
9. McCune, W.W.: Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. Journal of Automated Reasoning **9** (1992) 147–167
10. Bachmair, L., Chen, T., Ramakrishnan, I.V.: Associative Commutative Discrimination Nets. In: International Joint Conference on Theory and Practice of Software Development. Number 668 in LNCS, Springer-Verlag (1993) 61–74
11. Graf, P.: Term Indexing. Number 1053 in LNAI. Springer-Verlag (1996)