On the Correctness and Efficiency of Lock-Free Expandable Tries for Tabled Logic Programs

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 improves the declarativeness and expressiveness of Prolog in dealing with recursion and redundant sub-computations. A critical component in the implementation of an efficient tabling framework is the design of the data structures and algorithms to access and manipulate tabled data. One of the most successful data structures for tabling is *tries*. In previous work, our initial approach to deal with concurrent table accesses, implemented on top of the Yap Prolog system, was to use *lock-based* trie data structures. In this work, we propose a new design based on *lock-free* data structures and, in particular, we focus our discussion on the correctness and efficiency of extending Yap's tabling framework to support lock-free expandable tries. Experimental results show that our new lock-free design can effectively reduce the execution time and scale better, when increasing the number of threads, than the original lock-based design.

Keywords: Lock-Free, Tries, Hash Tables, Tabling

1 Introduction

Tabling [3] is a refinement of Prolog's standard resolution that can reduce the search space, avoid looping and have better termination properties. 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 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. 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.

A critical component in the implementation of an efficient concurrent tabling system is the design of the data structures and algorithms to access and manipulate tabled data. One of the most successful data structures for tabling is *tries* [13], a tree-based data structure in which common prefixes are represented only once. To deal with concurrent table accesses, our initial approach, implemented on top of the Yap Prolog system [15], was to use *lock-based data*

structures [2]. However, lock-based data structures have their performance restrained by multiple problems, such as, convoying, low fault tolerance and delays occurred inside a critical region. Yap's framework supports the evaluation of tabled programs according to the semantics of SLG resolution [3]. The practical significance of this is that, in general, we know that a concurrent tabled program will only execute search and insert operations over the table space shared data structures. Yap's shared data structures are only removed when the last running thread abolishes the tables. Since no concurrent delete operations are performed, the size of the shared tries always grows monotonically during an evaluation.

The main motivation of this work is then to refine our lock-based tries in order to be as efficient as possible in the concurrent search and insert operations and to maintain an efficient average node access as the size of the tries increases, independently of the number of running threads. In order to achieve that, we propose a new design based on *lock-free data structures* and we focus our discussion on the correctness and efficiency of extending Yap's tabling framework to support lock-free expandable tries, but our new design can be easily generalized and applied to similar concurrent data structures. Lock-freedom allows individual threads to starve but guarantees system-wide throughput. As we will see, this is very important since it allows to avoid the bottlenecks and performance problems mentioned above without introducing significant overheads for multithreaded tabled evaluation.

Experimental results show that our new lock-free design can effectively reduce the execution time and scale better, when increasing the number of threads, than the original lock-based design. Several lock-free approaches do exist in the literature, such as Shalev and Shavit split-ordered lists [16] or Prokopec *et al. CTries* [12], however to the best of our knowledge none of them is specifically aimed for an environment with the characteristics of our tabling framework. By avoiding the node deletion complexity, we were able to produce a fresh and new approach to deal with concurrency inside the tries.

The remainder of the paper is organized as follows. First, we briefly introduce some background and discuss related work. Then, we describe our new lock-free expandable tries design and we present the relevant implementation details. Next, we prove the correctness of our implementation. Finally, we discuss experimental results and we end by outlining some conclusions.

2 Background

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. An essential property of the trie structure is that common prefixes are represented only once. Two terms with common prefixes will branch off from each other at the first distinguishing token. Figure 1 shows an example for the internal representation of the trie levels. For the sake of simplicity, we only show two levels (the same idea applies to all trie levels).

The first level represents a parent node P and the second level represents how the trie is adapted to the insertion of distinguish child nodes with values $V1$, $V2$, $V3$ and $V4$. Figure 1(a) shows the trie representation after the insertion of $V1$ and Fig. 1(b) shows the trie representation after the insertion of $V2$. Note that new nodes are always inserted on the head for the level. Whenever the number of nodes in a level reaches a predefined threshold value, Yap's tries are expanded with a hash mechanism. Here, for the sake of simplicity, we will use a threshold value of 2. Figure 1(c) shows the hash representation after the insertion of values $V3$ and $V4$. The parent node P now points to a special hash node H, which includes a pointer to a hash bucket array with K entries, and the insert operation is now done on the head for the bucket entry corresponding to the hash key value k, $0 \leq k < K$. Whenever the hash bucket array becomes saturated, i.e., when the number of nodes in a bucket entry exceeds the threshold value and the total number of nodes exceeds K, then the bucket array is expanded to a new one with $2 * K$ entries (we will give more details about this expansion in the following sections).

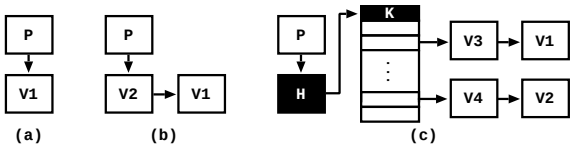


Fig. 1. Internal trie representation

To deal with concurrent table accesses, our initial approach was to use a lock-based scheme that 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 [2]. For locking, we used either a *locking field* per trie node or a *global array* of lock entries [1]. In order to reduce the lock duration, we also tried with *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, searching if the value V at hand was, in the meantime, inserted by another thread. If V is not found, the process repeats until the thread gets access to the lock or until V is found.

In this work, we are interested in taking advantage of the *CAS (Compare-and-Swap)* operation, that nowadays can be widely found on many common architectures. The CAS operation is an atomic instruction that compares the contents of a memory location to a given value and, if they are the same, modifies the contents of that memory location to a given new value. The atomicity guarantees that the new value is calculated based on up-to-date information, i.e., if the value had been updated by another thread in the meantime, the write would fail. The CAS result indicates whether it has successfully performed the substitution or not. Besides reducing the granularity of the synchronization, the CAS operation is at the heart of many *lock-free* objects [6]. An object is lock-free if it can be accessed by multiple threads concurrently without using any type of *locking mechanism*, such as spinlocks, mutexs or semaphores. For this work, we are most interested in lock-free linearizable objects as they permit greater concurrency since semantically consistent (non-interfering) operations may execute

in parallel. Further, linearizability is a local property, and is therefore independent of any underlying scheduling policy or interaction between objects. Locality improves the portability and modularity of large concurrent systems, and can simplify reasoning about concurrent objects.

3 Related Work

Despite the availability of both threads and tabling in several Prolog systems, such as Ciao, XSB and Yap, the implementation of these two features such that they work together implies complex ties to one another and to the underlying engine. To the best of our knowledge, XSB and Yap are the unique Prolog systems combining tabling with multi-threading. XSB offers two types of models for supporting multi-threaded tabling: *private tables* and *shared tables* [8]. For private tables, each thread keeps its own copy of the table space. For shared tables, each tabled subgoal is computed independently by the first thread calling it, the *generator thread*, and each generator thread is the sole responsible for fully exploiting and obtaining the complete set of answers for the subgoal. Since both XSB models avoid concurrency over the table space, Yap is thus the single Prolog system that implements and supports concurrent table accesses.

We next briefly describe some of the state-of-the-art approaches for concurrent tries and lock-free hash tables using linked lists to deal with collisions. The first practical work about a lock-free algorithm for hash tables with linked lists was presented by Michael [10]. Experimental results showed that the lock-free implementation outperformed, by significant margins, the best lock-based implementations, both under high and low contention. Another lock-free algorithm for expandable hash tables was presented by Shalev and Shavit [16]. It is based in split-ordered lists and allows the number of hash buckets to vary dynamically according to the number of nodes inserted or deleted, preserving the read-parallelism. More recently, Triplett et al. presented a set of algorithms that allow concurrent wait-free, linear scalable searches while shrinking and expanding hash tables [17]. The experimental results showed a good performance even when the hash table is under resizing.

Regarding concurrent trie data structures, Prokopec et al. presented recently the *CTries* [12]. The CTries are trees composed of internal nodes (I-Nodes) and leaves, combined with the support for a snapshot operation, where the updates on the CTries are done on the I-Nodes. The work shows how the efficiency of the CTries is directly related with the efficiency of the snapshots and how to improve the efficiency of those snapshots.

4 Lock-Free Expandable Tries

This section presents our new lock-free design to support the concurrent search, insertion, hash creation and expansion inside the trie structures. We start with Fig. 2 showing a small example that illustrates how the concurrent insertion of

nodes in the new lock-free trie structure is done. Again, for the sake of simplicity, we are only considering two levels of the trie.

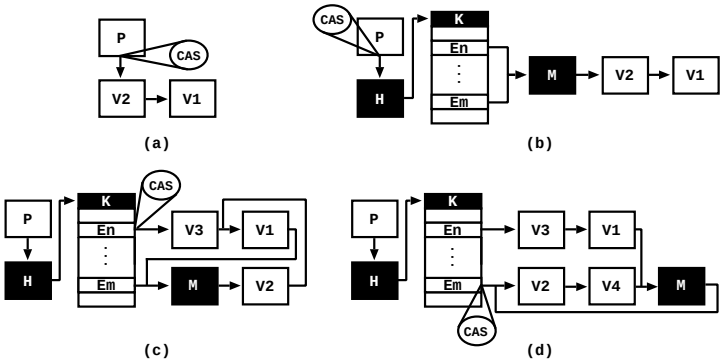


Fig. 2. Concurrent insertion of nodes in the new lock-free expandable trie structure

Figure 2(a) shows the trie configuration after the insertion of the child nodes $V1$ and $V2$ in the parent node P. At this stage, the search/insert operation for a node with a value V is straightforward. Initially, a thread follows the pointer of P to access the next level of the trie. Then, the chain of sibling nodes is searched for the value V at hand. If no such node exists, the pointer of P is used in a CAS operation to guarantee the synchronization of the insertion of V in the chain. During the search, a local counter is used to count the number of nodes on the level which, in the case of a node insertion, is then used to verify if the trie level has reached the predefined threshold value required for hash creation. For this count, no synchronization is required, since only one thread will be able to have its local counter equal to the threshold value.

Figure 2(b) then shows the trie configuration in the case where a thread has started the hash creation process for a trie level. The thread first creates the special node H, the initial bucket array with size K and initializes all entries in the bucket array pointing to a special marking node M. The node M is then used to implement a synchronization point with the first child node V of P (node $V2$ in the figure) that, whenever both are synchronized, will correspond to a successful CAS operation on P that updates V to H. This means that, from this point on, the access to the trie level will be done through the new hash node H. If a thread has accessed the trie level before the hash creation, which means that it has not seen H, in such case, when trying to insert a new node, the CAS operation on P will fail because P is now pointing to H.

In the continuation, Fig. 2(c) and Fig. 2(d) show the adjustment process of placing the child nodes in the correct bucket entries. To ensure lock-free synchronization, we need to guarantee that, at any time, all threads are able to read the correct values (starting from any bucket entry) and insert new values without any delay from the adjustment process. To guarantee both properties, we use M as a way to mark the beginning of the nodes not yet adjusted and we execute the adjustment process in reverse order. Figure 2(c) shows the case where node $V1$

is first adjusted to be in the bucket entry En and Fig. 2(d) shows the case where node $V2$ is then adjusted to be in the bucket entry Em. Concurrently with the adjustment process, other threads can be inserting nodes in the same bucket entries. In Fig. 2(c), a new node $V3$ is inserted after $V1$ in entry En and, in Fig. 2(d), a new node $V4$ is inserted before $V2$ in entry Em. To ensure that the nodes not yet adjusted (after M) can always be accessed from any bucket entry, the adjustment process may lead to cycles between the nodes. For example, in Fig. 2(c), node $V1$ is made to point to node M and since M is pointing to $V2$ and $V2$ is still pointing to $V1$, we have a temporary cycle between these nodes.

At the end of the adjustment process, all bucket entries still access M. To complete the hash creation process, the last operation is thus to remove M from all entries. For each bucket entry E, if M is on the head of E, then a CAS operation updating M to $Null$ is necessary. Otherwise, if M is not on the head of E, then we can simply mark as $Null$ the pointer of the node that is pointing to M (nodes $V1$ and $V4$ in Fig. 2(d)). This can be safely done without any CAS operation since no other thread can write on those nodes.

We complete the presentation of our new lock-free design by describing how a hash table with a bucket array of size K is expanded to a new one with size $2 * K$. The decision of performing hash expansion is similar to the hash creation process. During the search, a local counter is used to count the number of nodes on a bucket entry which, in the case of a node insertion, is then used to verify the conditions for hash expansion (please refer to Section 2). In order to ensure that only one thread gains access to the hash expansion operation, we use a CAS operation to tag a specific field on H. Figure 3 illustrates the hash expansion of Fig. 2(d) after the insertion of a new node $V5$ on the bucket entry En.

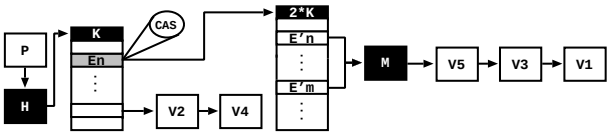


Fig. 3. Expanding the hash tables

The thread that gains access to the hash expansion operation starts by creating a new bucket array B' of size $2 * K$ entries. Next, for each old bucket entry En, it recomputes the hash function for the nodes on En and redistributes them on B' accordingly to the new hash values. In particular, for our hash function, this means that a node on the nth entry of the old bucket array B (En on Fig. 3) will be assigned to the nth or $(n+K)$th entry of B' (entries $E'n$ and $E'm$ on Fig. 3). As before, we use again a marking node M to implement a synchronization point between the old bucket entry En and the new bucket entries $E'n$ and $E'm$ that, whenever both are synchronized, will correspond to a successful CAS operation that updates En to B' (situation illustrated on Fig. 3). In the continuation, we follow the same adjustment process as before and, at the end, we remove M from $E'n$ and $E'm$. At the end, when the process of bucket expansion is completed for all K entries of B, we update H to point to the new bucket array B' (and remove simultaneously - same memory position - the tagging mark for hash expansion).

5 Implementation Details

We now present in more detail the algorithms that implement the key aspects of our new lock-free design. We start with Algorithm 1 that shows the pseudo-code for the search/insert operation of a new node N in a given bucket entry E.

Algorithm 1 $TrieSearchInsert(N, E)$

```
 1: markingNodeVisited ← False
 2: oldFirst ← Null
 3: repeat
 4:    first ← FirstNode(E)
 5:    while IsBucketArray(first) do
 6:       B ← BucketArray(first)
 7:       K ← Size(B)
 8:       E ← BucketEntry(B, Hash(K, Val(N)))
 9:       markingNodeVisited ← False
10:       oldFirst ← Null
11:       first ← FirstNode(E)
12:    chain ← first
13:    while chain ≠ oldFirst do
14:       if Val(chain) = Val(N) then
15:          return chain
16:       else if IsMarkingNode(chain) then
17:          if markingNodeVisited then
18:             break
19:          else
20:             markingNodeVisited ← True
21:       chain ← NextNode(chain)
22:    if not IsMarkingNode(first) then
23:       oldFirst ← first
24:    NextNode(N) ← first
25: until CAS(E, first, N)
26: return N
```

In a nutshell, the algorithm executes in a loop until one of the following situations occurs: (a) the search operation is successful, meaning that there is already a node in the trie level with the same value of N (lines 14–15); or (b) N is successfully inserted in the trie (lines 24–25).

In more detail, the algorithm starts by checking (lines 4–5) if the bucket entry E is referencing another bucket array (this happens when another thread is doing hash expansion). In such case, it moves to the new bucket array (variable B at line 6) and updates E (by recomputing the hash function using the value on N), $markingNodeVisited$, $oldFirst$ and $first$ accordingly (lines 8–11). The auxiliary variable $markingNodeVisited$ denotes if the marking node was already visited and the auxiliary variable $oldFirst$ marks the beginning of the chain of nodes on E that were already searched in a previous round.

On the second part of the algorithm, it then searches if there is a node with the same value of N already in the chain (lines 12–21). Note that this search is done while the nodes in the chain were not yet searched in a previous round (while condition at line 13) and while the marking node was not visited twice (lines 16–20). This second condition allows to break any potential cycle between the nodes, as a result of a hash creation/expansion operation being done by another thread. Finally, if the value of N is not found, the algorithm tries to insert N on the bucket entry E by using a CAS operation that updates $first$ to N (line 25). In case of failure, this means that the head of E has changed in the meantime, thus leading to a new round.

Next, Algorithm 2 shows the pseudo-code for the hash expansion operation given a hash node H (due to the lack of space and since it is quite similar, we

will leave aside the algorithm for hash creation). Please remember that to ensure that only one thread executes the hash expansion operation for H, we use a CAS operation to tag a specific field on H (not shown here for the sake of simplicity).

Algorithm 2 $HashExpansion(H)$

```
 1: M ← MarkingNode(H)
 2: oldB ← BucketArray(H)
 3: oldK ← Size(oldB)
 4: newK ← 2 * oldK
 5: newB ← AllocBucketArray(newK)
 6: i ← 0
 7: while i < oldK do
 8:    oldE ← BucketEntry(oldB, i)
 9:    if not CAS(oldE, Null, newB) then
10:       newE1 ← BucketEntry(newB, i)
11:       newE2 ← BucketEntry(newB, i + oldK)
12:       FirstNode(newE1) ← M
13:       FirstNode(newE2) ← M
14:       repeat
15:          NextNode(M) ← FirstNode(oldE)
16:       until CAS(oldE, NextNode(M), newB)
17:       AdjustNodes(M, newB)
18:       RemoveMarkingNode(M, newE1)
19:       RemoveMarkingNode(M, newE2)
20:    i++
21: BucketArray(H) ← newB
22: return
```

The algorithm begins by initializing a set of local variables and by allocating a new bucket array (lines 1–5). Next, for each old bucket entry $oldE$, it redistributes the chain of nodes on $oldE$ to the corresponding bucket entries on the new bucket array $newB$ (lines 7–20). At line 9, it executes a CAS operation on $oldE$ trying to update a value of $Null$ to $newB$. A successful CAS operation means that $oldE$ was empty and thus no redistribution is necessary (it just becomes a pointer to the new bucket array). An unsuccessful CAS operation means that $oldE$ has nodes to be expanded. In such case, the algorithm then computes the entries on $newB$ in which the nodes from $oldE$ will fall (entries $newE1$ and $newE2$) and initializes them to point to the marking node M (lines 10–13). The marking node M is then used to implement a synchronization point between the old bucket entry $oldE$ and the new bucket entries $newE1$ and $newE2$ that, whenever both are synchronized, will correspond to a successful CAS operation that updates $oldE$ to $newB$ (lines 14–16). In the continuation (lines 17–19), the algorithm proceeds by adjusting the nodes on the old chain (Algorithm 3 below) and by removing M from the $newE1$ and $newE2$ chains (Algorithm 4 below). At the end, when the process of bucket expansion is completed for all entries in $oldB$, H is updated to point to the new bucket array $newB$ (line 21).

Algorithm 3 $AdjustNodes(N, B)$

```
1: chain ← NextNode(N)
2: if NextNode(chain) ≠ Null then
3:    AdjustNodes(chain, B)
4: K ← Size(B)
5: E ← BucketEntry(B, Hash(K, Val(chain)))
6: repeat
7:    NextNode(chain) ← FirstNode(E)
8: until CAS(E, NextNode(chain), chain)
9: return
```

Algorithm 3 shows the pseudo-code for the process of adjusting a chain of nodes, starting from a given node N, into a given new bucket array B. One can observe that the algorithm traverses the chain of nodes recursively and that the base case for recursion is the last node on the chain

(lines 1–3). For each *chain* node, it then calculates the bucket entry E in which it will fall (lines 4–5). The bucket entry E is then used in repeated CAS operations until successfully insert the *chain* node on the head of E (lines 6–8).

Finally, Algorithm 4 shows the pseudo-code for the operation of removing a given marking node M from a given bucket entry E. Initially, it executes a CAS operation on E trying to update an expected value M to $Null$. A successful CAS operation means that no nodes were adjusted to be on E (and E just becomes a pointer to $Null$). An unsuccessful CAS operation means that at least one node was adjusted to be on E. In such case, the algorithm then follows the chain of nodes on E until reaching M and updates the node previous to M to point to $Null$ (thus removing M from the chain). This can be safely done without any CAS operation, because at this stage no other thread can be writing at this node.

Algorithm 4 $RemoveMarkingNode(M, E)$

```
1: if not CAS(E, M, Null) then
2:     chain ← FirstNode(E)
3:     next ← NextNode(chain)
4:     while (next ≠ M) do
5:         chain ← next
6:         next ← NextNode(chain)
7:     NextNode(chain) ← Null
8: return
```

6 Proof of Correctness

In this section, we discuss the correctness of our implementation.

6.1 Linearizability

Linearizability is an important correctness condition for the implementation of concurrent data structures [7]. A concurrent operation is linearizable if it appears to take effect instantaneously at some moment of time I_{time} between its invocation and response. The literature often refers to I_{time} as a *linearization point* and, for lock-free implementations, a linearization point is typically a single instant where its effects become visible to all the remaining operations. Linearizability guarantees that if all operations individually preserve an invariant, the system as a whole also will. Our new implementation is linearizable, since every trie manipulation operation takes effect in specific linearization points. The linearization points for our algorithms are the following:

- $TrieSearchInsert()$ is linearizable at successful CAS in line 25.
- $HashExpansion()$ is linearizable at successful CAS in lines 9 and 16 and at algorithms $AdjustNodes()$ and $RemoveMarkingNode()$ in lines 17–19:
  - $AdjustNodes()$ is linearizable at successful CAS in line 8.
  - $RemoveMarkingNode()$ is linearizable at successful CAS in line 1 and at line 7 when the node previous to the marking node is updated to $Null$.

Due to the lack of space, we do not show the full proof of correctness of the linearization points defined above. Instead, we focus on proving that our implementation is ABA-free.

6.2 The ABA Problem

We now discuss how we address the ABA problem. We use the fact that a memory location has not changed between two readings to assume that nothing has changed during the period of time from the first to the second reading. Although, this is a common technique when using the CAS operation, in some cases, it can lead to the ABA problem. An example of that would be: a thread T reads a value $V1$ from a memory location L, uses $V1$ to do some work, updates L to a new value $V2$ and, at the end of the work, changes the value of L again to $V1$. In such case, if another thread has read the memory location L before and after the work done by T, then it will be deceived by the fact that the memory location has not changed. In our implementation, a practical consequence of this would be to insert more than once the same value on the same level of the trie.

To address the ABA problem, several techniques already exist, such as *version tagging* [4], *hazard pointers* [11] or *value semantics* [5]. In general, these kind of techniques rely on the fact that a writing over a memory position always cause a transition from the current state of the system to a uniquely new different state. To prove that our algorithm is ABA-free, we prove that each concurrent memory location L only points once to the same value $V1$, i.e., if L is updated from $V1$ to $V2$ than L will never point to $V1$ again. Our concurrent memory locations are defined by the pointers on the parent nodes P, on the hash nodes H and on the bucket entries E as described in the previous sections.

Theorem 1. *The new implementation is ABA-free.*

Proof. Assume that P, C, H, M and E already exist in a trie T and represent, respectively, a parent node, a child node, a hash node, a marking node and a bucket entry. Assume also that NC, NH and NB represent, respectively, a new child node, a new hash node and a new bucket array.

The following writing situations may occur: (i) if a write occurs in P then a NC or NH was added to the trie T; (ii) if a write occurs in H then a NB was added to T; (iii) if a write occurs in E then a NC or NB was added to T or the node adjustment process adjusted E to $Null$ or to a child node C.

In the latter situation, if E is adjusted to $Null$ that means that before the write operation, E was pointing to a marking node M. Otherwise, if E is adjusted to a child node C, then before the write operation, E was pointing to another node, say N. N can be a new child node NC added in the meantime, a marking node M, or another child node adjusted previously. In any case, N is necessarily different from C and E will never point to N again.

Thus, all concurrent memory locations always point once to the same value whenever a write operation occurs.

6.3 Liveness

In this subsection, we prove that the insert and hash operations are lock-free and that the search operation is wait-free. For that, we begin by enumerating the following Lemmas.

Lemma 1. *If the CAS operation in $TrieSearchInsert()$ at line 25 succeeds, then a new node was inserted in the trie.*

Lemma 2. *If the CAS operation in $HashExpansion()$ at lines 9 or 16 succeeds, then the bucket entry was updated to point to a new bucket array.*

Lemma 3. *If the CAS operation in $AdjustNodes()$ at line 8 succeeds, then the bucket entry was updated to point to a node that was already in the chain of the bucket entry.*

Lemma 4. *If the CAS operation in $RemoveMarkingNode()$ at line 1 succeeds, then the bucket entry was updated to point to $Null$.*

To prove the property of lock-freedom, we prove that the insert and hash operations always lead to progress in the trie configuration. We start with Theorem 2 that proves that progress is always achieved for the insert operation. The proof is done on the point of the implementation where we try to insert new nodes in the trie, i.e., the CAS operation in $TrieSearchInsert()$ at line 25. Note that, since we are assuming that the CAS operation was executed, this means that the given node N was not found in the chain starting from $first$ (lines 12 to 21) as otherwise the *return* at line 15 would have been executed.

Theorem 2. *In $TrieSearchInsert()$, everytime a thread executes the CAS operation at line 25, then the trie configuration has made progress when compared to the time at which the thread has entered the repeat loop at line 4.*

Proof. If the CAS operation succeeds then, by Lemma 1, a new node was inserted in the trie thus leading to progress in the trie configuration.

Otherwise, if the CAS operation fails, then the value in the bucket entry E is necessarily different from the initial one, as given by $first$ (initialized at lines 4 or 11). Thus, the new value of E must be the result of one of the following situations: (i) a new node was inserted by another thread (Lemma 1); (ii) the current hash is being expanded by another thread (Lemma 2); or (iii) another thread is performing the adjustment process on E (Lemmas 3 or 4). In either one of these three cases, another thread has lead to progress in the trie configuration.

To prove that the hash creation/expansion operations progress even when other threads are inserting new nodes, we can use as sketch the proof for Theorem 2. Due to the lack of space, we are also omitting such proofs here.

Theorem 3. *The new implementation is lock-free.*

Next, we prove the wait-free property of the search operation and, for that, we show that any search operation is always completed in a bounded number of visited nodes. In particular, this bound is always lower or equal to the number of nodes in the chain being searched. Since the number of steps of the search operation is finite, the proof that the bound exists is sufficient to prove that the search operation is wait-free.

Theorem 4. *The search operation is completed within a bounded number of visited nodes.*

Proof. Assume that CN is a chain of nodes and that a search operation in CN is executed between two instants of time, I_{init} and I_{final}. This corresponds to the block of code between lines 12 and 21 in the $TrieSearchInsert()$ algorithm. Assume also that N_{init} is the number of nodes at instant I_{init}, N_{new} is the total number of new nodes inserted between I_{init} and I_{final}, and that N_{vis} is the number of nodes visited between I_{init} and I_{final}.

The variable chain represents a node to be visited, the variable $first$ represents the first node visited, and variable $oldFirst$ represents the first node that was visited on the previous search operation. On the first search operation, $oldFirst$ is always $Null$. We begin now the proof that N_{vis} is bounded for all the configurations of CN.

If $N_{init} = 0$, then $first$ and $oldFirst$ are both $Null$ and thus $N_{vis} = 0$.

If $N_{init} \neq 0$, then $first \neq Null$. Now, if $oldFirst = Null$ then two situations can occur. On the first situation, no concurrent hash expansion has interfered with the search, thus the variable chain visits all nodes until reaching $oldFirst$, and in such case $N_{vis} = N_{init}$. On the second situation, a concurrent hash expansion has interfered with the search, thus the variable chain may not visit all N_{init} nodes (some nodes may be scheduled to a different bucket entry) but a node can be visited more than once (please remember that, during the adjustment process, we may have cycles between the nodes). In any case, it stops either when reaching $oldFirst$ (line 13) or when the marking node is visited twice (line 17). Thus, $N_{vis} \leq 2 * (N_{init} + N_{new})$.

Finally, if $N_{init} \neq 0$ and $oldFirst \neq Null$, then the variable chain will not visit all N_{init} nodes (the ones after $oldFirst$) and thus $N_{vis} \leq N_{init}$.

7 Experimental Results

We now present experimental results for the new lock-free design using the set of benchmarks from [1] which includes 19 different programs in total. We choose these benchmarks because they have characteristics that cover a wide number of scenarios in terms of trie usage. The benchmarks create different trie configurations with lower and higher number of nodes and depths, and also have different demands in terms of trie traversing. The environment for our experiments was a machine with 32 Core AMD Opteron (tm) Processor 6274 using 32 GBytes of memory and running the Linux kernel 3.6.6-1.fc17.x86_64 with Yap Prolog 6.3.

To compare our new design, which we named *Lock-Freedom (FD)*, we used the four lock-based strategies from the previous design, which we named *Local Locks (LL)*, *Global Locks (GL)*, *Local Trylocks (LT)* and *Global Trylocks (GT)*. All strategies use the Pthreads implementation for lock support. The LL and LT strategies use a lock field per trie node. The GL and GT strategies use a global array of 512 lock entries with a hash function that maps trie nodes to lock entries. Through experimentation, we observed that the number of trie

nodes mapped by hash function to each lock entry shows a good balancing, thus reducing contention points. To put our results in perspective, we also make a comparison with XSB Prolog, version 3.4.0, using thread-private tables [8].

Note that our goal with these experiments is not to prove that we can speedup the execution of tabled programs, despite this is an obvious goal of having a concurrent implementation. Other works have already showed the parallel capabilities of the use of multithreaded tabling [8, 9]. Since parallelism is highly dependent on the available concurrency that programs have and on the way synchronization is done, we can easily select/construct programs where linear speedups can be achieved or, on the other hand, where no speedups exist. Here, we are more interested in evaluating the robustness of our implementation when exposed to worst case scenarios. Note that if we are able to deal well with such scenarios, we will certainly have the conditions to better support parallelism. Moreover, by doing that, we avoid the peculiarities of the program at hand and we try to focus on measuring the real value of our new design.

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 intervals of 8 threads until 64 threads (two times the number of cores in our machine). Figure 4 shows the overhead ratios, comparing the execution time with 8, 16, 24, 32, 40, 48, 56 and 64 threads against the respective execution time with one thread, for the average of five runs, when running the set of benchmarks.

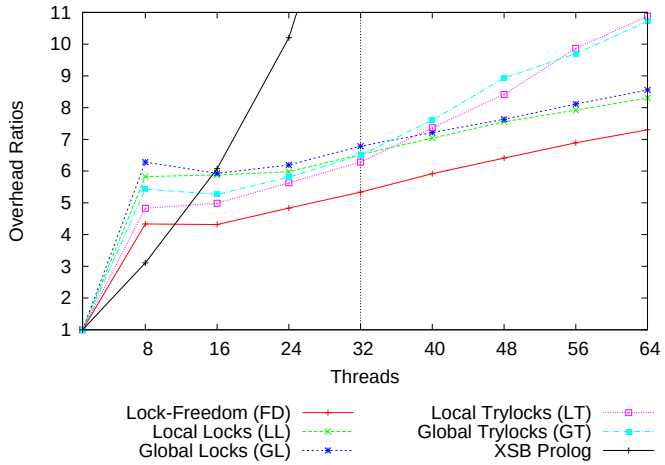


Fig. 4. Overhead ratios, comparing the execution time with 8, 16, 24, 32, 40, 48, 56 and 64 threads against the respective execution time with one thread

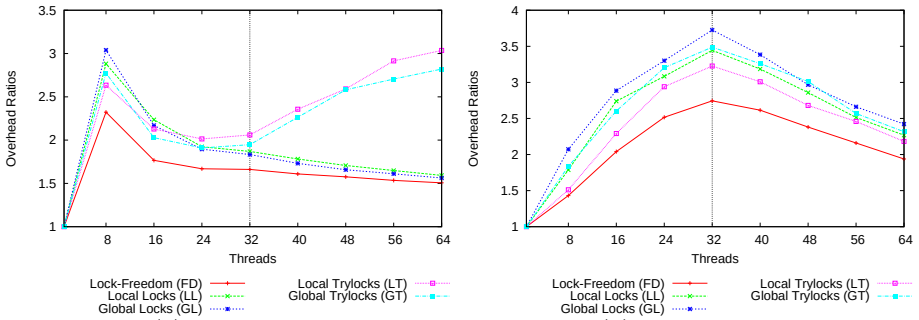
By observing Fig. 4, the results show that XSB achieves the best ratio for 8 threads but then, for more than 8 threads, XSB is noticeably worse than all Yap's strategies, showing a clear tendency to worsen as we increase the number

of threads. For the sake of presentation, we are not showing the results for more than 24 threads (for 32, 40, 48, 56 and 64 threads, XSB is respectively 17.35, 22.35, 27.21, 32.41 and 36.60 times slower than the execution with one thread). On comparison with Yap, these results are even more important since XSB shows, on average, base execution times (with one thread) higher than Yap. Regarding Yap's synchronization strategies, the results show that FD is always the best strategy of all, regardless of the number of threads. The best lock-based strategy is LT, for 8 to 32 threads, and LL, for 40 to 64 threads. In general, the differences to the corresponding GT and GL strategies is meaningless, which confirms the low contention observed for the global lock array.

Starting from 32 threads, one can also observe that the LT and GT trylock strategies start to diverge and that the LL and GL strategies keep the difference to the FD design. This is explained by the fact that, in the Pthreads implementation, when a thread fails to get a lock it falls asleep, leaving the machine resources available to the remaining threads. In particular, for the LL and GL strategies, when the number of execution threads exceeds the number of cores, this leads to an inversion on the execution priorities, which results in having the machine resources always available to the threads inside the critical regions (i.e., holding the corresponding synchronization locks).

For a number of threads smaller than 32, the LT and GT trylock strategies perform better. This is due to the fact that, for fewer threads than the number of cores, they do not have to pay the cost of resuming the threads that fall asleep when failing to get a lock but, for more threads than the number of cores, they may have to pay the cost of not having machine resources always available to the threads holding the synchronization locks. Again, since the FD strategy is immune to the availability of machine resources, and since the CAS operation was a lower synchronization overhead when compared with a lock-based design, makes our new FD design clearly the best approach for both scenarios.

To better understand these results, we next show the overhead ratios, but now comparing the average *user time* (Fig. 5(a)) and the average *system time* (Fig. 5(b)) for 8, 16, 24, 32, 40, 48, 56 and 64 threads against the respective execution time (walltime) with one thread for Yap's synchronization strategies. The results on Fig. 5(a) show us how concurrency affects, on average, the execution of a thread, i.e., how much more user code, on average, a thread has to execute when compared with the base execution with one thread. One can observe that all strategies start to pay a huge cost for eight threads (between 2.32 (FD) and 3.04 (GL) times the execution time with one thread) and then this cost decreases gradually, except for the LT and GT trylock strategies that, for more than 32 threads, start paying the cost of not having machine resources always available (as explained before). The results on Fig. 5(b) show us how much more system (synchronization) code, on average, a thread has to execute when compared with the base execution with one thread. One can observe that all strategies show a similar tendency, with FD always showing the least overhead of all, which confirms the lower synchronization overhead of the CAS operation.



(a) User time (b) System time

Fig. 5. Overhead ratios, comparing the average user/system time with 8, 16, 24, 32, 40, 48, 56 and 64 threads against the respective execution time with one thread

8 Conclusions

We have presented a novel, efficient and lock-free design for expandable trie data structures applied to the multithreaded tabled evaluation of logic programs. Our main motivation was to refine the previous lock-based design in order to be as efficient as possible in the concurrent search and insert operations and to maintain an efficient average node access as the size of the tries increases, independently of the number of running threads. We discussed the relevant implementation details and we proved the correctness of our implementation. Experimental results show that our new lock-free design can effectively reduce the execution time and scale better, when increasing the number of threads, than the original lock-based design. Further work will include extending our framework to support multithreaded mode-directed tabling [14], which includes studying how to extend our new lock-free design to allow the concurrent deletion of trie nodes.

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 LEAP (FCOMP-01-0124-FEDER-015008) and PEst (FCOMP-01-0124-FEDER-037281). Miguel Areias is funded by the FCT grant SFRH/BD/69673/2010.

References

1. Areias, M., Rocha, R.: An Efficient and Scalable Memory Allocator for Multithreaded 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. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM 43(1), 20–74 (1996)
4. Detlefs, D.L., Martin, P.A., Moir, M., Jr., G.L.S.: Lock-Free Reference Counting. In: ACM Symposium on Principles of Distributed Computing. pp. 190–199. ACM (2001)
5. Hendler, D., Shavit, N., Yerushalmi, L.: A Scalable Lock-free Stack Algorithm. In: ACM Symposium on Parallelism in Algorithms and Architectures. pp. 206–215. ACM (2004)
6. Herlihy, M., Wing, J.M.: Axioms for Concurrent Objects. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 13–26. ACM (1987)
7. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
8. 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-Verlag (2008)
9. Marques, R., Swift, T., Cunha, J.C.: A Simple and Efficient Implementation of Concurrent Local Tabling. In: International Symposium on Practical Aspects of Declarative Languages. pp. 264–278. No. 5937 in LNCS, Springer-Verlag (2010)
10. Michael, M.M.: High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In: ACM Symposium on Parallel Algorithms and Architectures. pp. 73–82. ACM (2002)
11. Michael, M.M.: Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. IEEE Transactions on Parallel and Distributed Systems 15(6), 491–504 (2004)
12. Prokopec, A., Bronson, N.G., Bagwell, P., Odersky, M.: Concurrent Tries with Efficient Non-Blocking Snapshots. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 151–160. ACM (2012)
13. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming 38(1), 31–54 (1999)
14. Santos, J., Rocha, R.: On the Efficient Implementation of Mode-Directed Tabling. In: International Symposium on Practical Aspects of Declarative Languages. LNCS, Springer-Verlag (2013)
15. Santos Costa, V., Rocha, R., Damas, L.: The YAP Prolog System. Journal of Theory and Practice of Logic Programming 12(1 & 2), 5–34 (2012)
16. Shalev, O., Shavit, N.: Split-Ordered Lists: Lock-Free Extensible Hash Tables. Journal of the ACM 53(3), 379–405 (2006)
17. Triplett, J., McKenney, P.E., Walpole, J.: Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In: USENIX Annual Technical Conference. pp. 11–11. USENIX Association (2011)