IJPP manuscript No.
(will be inserted by the editor)

A Lock-Free Hash Trie Design for Concurrent Tabled Logic Programs

Miguel Areias · Ricardo Rocha

Received: date / Accepted: date

Abstract Tabling is an implementation technique that improves the declarativeness and expressiveness of Prolog systems in dealing with recursion and redundant sub-computations. A critical component in the design of a concurrent tabling system is the implementation of the table space. One of the most successful proposals for representing tables is based on a *two-level trie data structure*, where one trie level stores the tabled subgoal calls and the other stores the computed answers. In previous work, we have presented a sophisticated *lock-free* design where both levels of the tries where shared among threads in a concurrent environment. To implement lock-freedom we used the *CAS* atomic instruction that nowadays is widely found on many common architectures. *CAS* reduces the granularity of the synchronization when threads access concurrent areas, but still suffers from problems such as false sharing or cache memory effects. In this work, we present a simpler and efficient lock-free design based on *hash tries* that minimizes these problems by dispersing the concurrent areas as much as possible. Experimental results in the Yap Prolog system show that our new lock-free design can effectively reduce the execution time and scales better than previous designs.

Keywords Tabling · Concurrency · Hash Tries · Lock-Freedom · Performance

Miguel Areias
CRACS & INESC TEC, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
E-mail: miguel-areias@dcc.fc.up.pt

Ricardo Rocha
CRACS & INESC TEC, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
E-mail: ricroc@dcc.fc.up.pt

1 Introduction

Tabling [5] is a recognized and powerful implementation technique that overcomes some limitations of traditional Prolog systems in dealing with recursion and redundant sub-computations. Tabling is a refinement of Prolog's SLD resolution that stems from one simple idea: save intermediate answers for current computations, in a specific data area called the *table space*, so that they can be reused when a *similar computation* appears during the resolution process. Tabled evaluation can reduce the search space, avoid looping and have better termination properties than SLD resolution. Work on tabling proved its viability for application areas such as deductive databases [17], model checking [15], parsing [10], program analysis [6], reasoning in the semantic Web [21], among others. Currently, tabling is widely available in systems like ALS-Prolog, B-Prolog, Ciao, Mercury, XSB and Yap Prolog.

Multithreading in Prolog is the ability to perform concurrent computations, in which each thread runs independently but shares the program clauses [13]. 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. Despite the availability of both multithreading and tabling in some Prolog systems, the efficient implementation of these two features, such that they work together, implies a complex redesign of several components of the underlying engine. XSB was the first Prolog system to combine tabling with multithreading [11]. In more recent work [2], we have proposed an alternative view to XSB's approach, where each thread views its tables as private but, at the engine level, we use a *common table space*, i.e., from the thread point of view, tables are private but, from the implementation point of view, tables are shared among all threads.

A critical component in the design of an efficient concurrent tabling system is the implementation of the data structures and algorithms that manipulate tabled data. Our initial approach, implemented on top of the Yap Prolog system [18], was to use *lock-based* data structures [2]. Yap implements a two-level trie data structure, where one trie level stores the tabled subgoal calls and the other stores the computed answers [16]. More recently [3], we presented a sophisticated *lock-free* design to deal with concurrency in both trie levels. Lock-freedom allows individual threads to starve but guarantees system-wide throughput. To implement lock-freedom we took advantage of the *CAS* atomic instruction that nowadays is widely found on many common architectures. *CAS* reduces the granularity of the synchronization when threads access concurrent areas, but still suffers from contention points where synchronized operations are done on the same memory locations, leading to problems such as false sharing or cache memory ping pong effects.

In this work, we go one step further and we present a simpler and efficient lock-free design based on *hash tries* that minimizes these problems by dispersing the concurrent areas as much as possible. Hash tries (or hash array mapped tries) are a trie-based data structure with nearly ideal characteristics for the implementation of hash tables [4]. An essential property of the trie data structure is that common prefixes are stored only once [7], which in the context of hash tables allows us to efficiently solve the problems of setting the size of the initial hash table and of dynamically resizing it in order to deal with hash collisions. Several approaches exist in the literature for the implementation of lock-free hash tables, such as Shalev and Shavit split-ordered lists [19], Triplett *et al.* relativistic hash tables [20] or Prokopec *et al.* CTries [14]. However, to the best of our knowledge, none of them is specifically aimed for an environment with the characteristics of our tabling framework that does not requires deletion support[1]. The aim of our proposal is to be as effective as possible in the search and insert operations, by exploiting the full potentiality of lock-freedom on those operations, and in such a way that it minimizes the bottlenecks and performance problems mentioned above without introducing significant overheads for sequential execution.

To put our proposal in perspective, we therefore first compared it against some of the best-known currently available implementations, such as the *ConcurrentHashMap* and *ConcurrentSkipListMap* from the *Java* standard library, as well as two different version of the Ctries, and for that we used a publicly available framework, also used for benchmarking the previously mentioned CTries work. Our experiments show that our new lock-free hash-trie design can effectively reduce the execution time and scales better than all the other implementations on top of a 32 Core AMD machine. In the context of Yap's concurrent tabling support, we then evaluated our proposal against all the previously implemented lock-based and lock-free strategies. Results in the same machine show indeed that our new lock-free hash trie design can also reduce the execution time and scales better than all the previous designs.

The remainder of the paper is organized as follows. First, we introduce some background and discuss the general idea behind our proposal. Next, we describe the algorithms that support the implementation and present their proof of correctness. Then, we show experimental results obtained independently for the framework mentioned above and for concurrent tabling within Yap Prolog. We end by outlining some conclusions.

2 Background

A trie is a tree structure where each different path corresponds to a term described by the tokens labeling the nodes traversed. For example, the tokenized form of the term $p(1, f(X))$ is the sequence of 4 tokens $p/2$, 1, $f/1$ and VAR_0, where each variable is represented as a distinct VAR_i constant. Two terms with common prefixes will branch off from each other at the first distinguishing token. Consider, for example, a second term $p(1, a)$. Since the main functor and the first argument, tokens $p/2$ and 1, are common to both terms, only one

[1] In general, a tabled program is deterministic, finite and only executes search and insert operations over the table space data structures. In Yap Prolog, space is recovered when the last running thread abolishes a table. Since no delete operations are performed, the size of the tables always grows monotonically during an evaluation.

additional node will be required to fully represent this second term in the trie. Figure 1 shows the trie structure representing both terms.

Whenever the chain of child nodes for a common parent node becomes larger than a predefined threshold value, a *hash mechanism* is used to provide direct node access and therefore optimize the search. To deal with hash collisions, all previous Yap's approaches implemented a dynamic resizing of the hash tables by doubling the size of the bucket entries in the hash [2,3]. In this work, we present a simpler and efficient lock-free design based on *hash tries* to implement the hash mechanism inside the subgoal and answer tries. To put our proposal in perspective, Fig. 2 shows a schematic representation of the trie hierarchical levels we are proposing to implement Yap's table space.

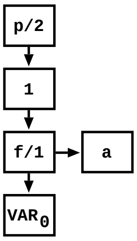


Fig. 1 Trie example

For each tabled predicate, Yap implements tables using two levels of tries together with the *table entry* and *subgoal frame* auxiliary data structures [16]. The first level, the *subgoal trie*, stores the tabled subgoal calls and the second level, the *answer trie*, stores the answers for a given call. Then, for each particular subgoal/answer trie, we have as many trie levels as the number of parent/child relationships (for example, the trie in Fig. 1 has 4 trie levels). Finally, to implement hashing inside the subgoal/answer tries, we use another trie-based data structure, the hash trie, which is the focus of the current work. In a nutshell, a hash trie is composed by *internal hash arrays* and *leaf nodes*. The leaf nodes store *key* values and the internal hash arrays implement a hierarchy of hash levels of fixed size 2^w. To map a *key* into this hierarchy, we first compute the hash value h for *key* and then use chunks of w bits from h to index the entry in the appropriate hash level. Hash collisions are solved by simply walking down the tree as we consume successive chunks of w bits from the hash value h.

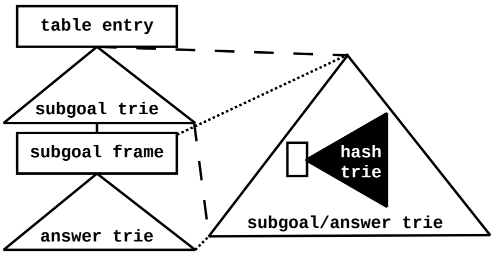


Fig. 2 Trie hierarchical levels overview

3 Our Proposal By Example

We will use two examples to illustrate the different configurations that the hash trie assumes for one and two levels (for more levels, the same idea applies). We begin with Fig. 3 showing a small example that illustrates how the concurrent insertion of nodes is done in a hash level.

Figure 3(a) shows the initial configuration for a hash level. Each hash level H_i is formed by a bucket array of 2^w entries and by a backward reference to the previous level (represented as *Prev* in the figures that follow). For the root

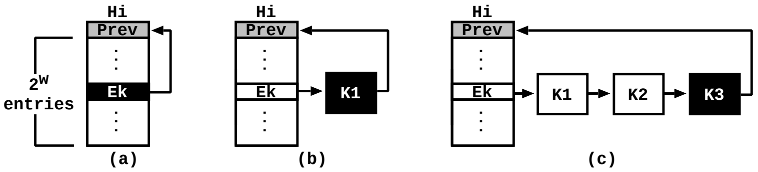


Fig. 3 Insert procedure in a hash level

level, the backward reference is *Nil*. In Fig. 3(a), E_k represents a particular bucket entry of the hash level. E_k and the remaining entries are all initialized with a reference to the current level H_i. During execution, each bucket entry stores either a reference to a hash level or a reference to a separate chaining mechanism, using a chain of internal nodes, that deals with the hash collisions for that entry. Each internal node holds a *key* value and a reference to the next-on-chain internal node. Figure 3(b) shows the hash configuration after the insertion of node K_1 on the bucket entry E_k and Fig. 3(c) shows the hash configuration after the insertion of nodes K_2 and K_3 also in E_k. Note that the insertion of new nodes is done at the end of the chain and that any new node being inserted closes the chain by referencing back the current level.

During execution, the different memory locations that form a hash trie are considered to be in one of the following states: *black*, *white* or *gray*. A black state represents a memory location that can be updated by any thread (concurrently). A white state represents a memory location that can be updated only by one (specific) thread (not concurrently). A gray state represents a memory location used only for reading purposes. As the hash trie evolves during time, a memory location can change between black and white states until reaching the gray state, where it can no longer be updated.

The initial state for E_k is black, because it represents the next synchronization point for the insertion of new nodes. After the insertion of node K_1, E_k moves to the white state and K_1 becomes the next synchronization point for the insertion of new nodes. To guarantee the property of lock-freedom, all updates to black states are done using CAS operations. Since we are using single word *CAS* operations, when inserting a new node in the chain, first we set the node with the reference to the current level and only then the *CAS* operation is executed to insert the new node in the chain.

When the number of nodes in a chain exceeds a *MAX_NODES* threshold value, then the corresponding bucket entry is expanded with a new hash level and the nodes in the chain are remapped in the new level. Thus, instead of growing a single monolithic hash table, the hash trie settles for a hierarchy of small hash tables of fixed size 2^w. To map our key values into this hierarchy, we use chunks of w bits from the hash values computed by our hash function. For example, consider a *key* value and the corresponding hash value h. For each hash level H_i, we use the bits $[w * i, w * (i + 1) - 1]$ in h to index the entry in the appropriate bucket array, i.e., we consume h one chunk at a time as we walk down the hash levels. Starting from the configuration in Fig. 3(c),

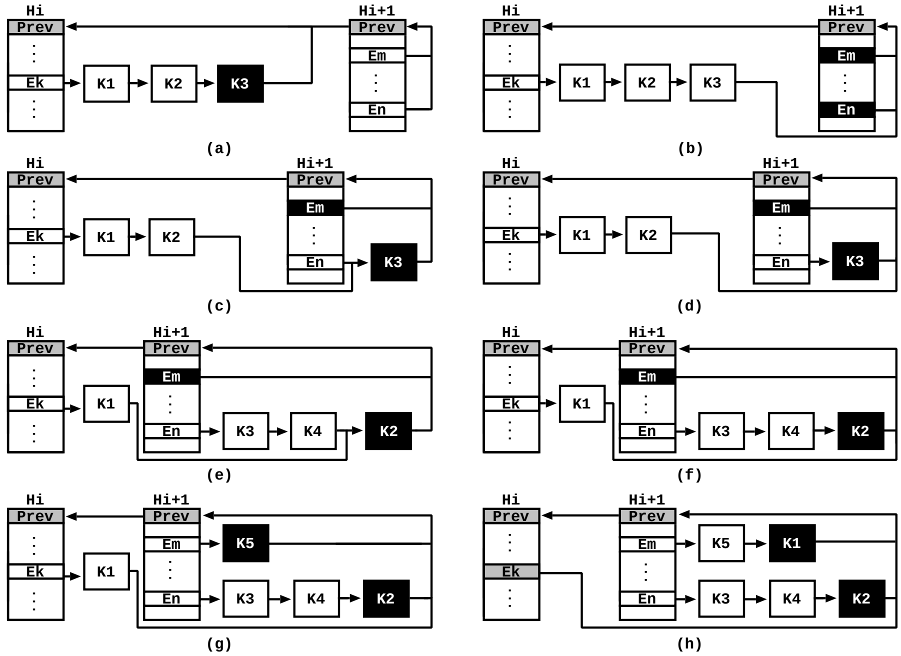


Fig. 4 Expanding a bucket entry with a second level hash

Fig. 4 illustrates the expansion mechanism with a second level hash H_{i+1} for the bucket entry E_k.

The expansion procedure is activated whenever a thread T meets the following two conditions: (i) the key at hand was not found in the chain and (ii) the number of nodes in the chain is equal to the threshold value (in what follows, we consider a threshold value of three nodes). In this case, T starts by pre-allocating a second level hash H_{i+1}, with all entries referring the respective level (Fig. 4(a)). At this stage, the bucket entries in H_{i+1} can be considered white memory locations, because the hash level is still not visible to the other threads. The new hash level is then used to implement a synchronization point with the last node on the chain (node K_3 in the figure) that will correspond to a successful *CAS* operation trying to update H_i to H_{i+1} (Fig. 4(b)). From this point on, the insertion of new nodes on E_k will be done starting from the new hash level H_{i+1}.

If the *CAS* operation fails, that means that another thread has gained access to the expansion procedure and, in such case, T aborts its expansion procedure. Otherwise, T starts the remapping process of placing the internal nodes K_1, K_2 and K_3 in the correct bucket entries in the new level. Figures 4(c) to 4(h) show the remapping sequence in detail. For simplicity of illustration, we will consider only the entries E_m and E_n on level H_{i+1} and assume that K_1, K_2 and K_3 will be remapped to entries E_m, E_n and E_n, respectively. In order to ensure lock-free synchronization, we need to guarantee that, at any time, all threads are able to read all the available nodes and insert new nodes without any delay from the remapping process. To guarantee both properties, the remapping process is thus done in reverse order, starting from the last node on the chain, initially K_3.

Figure 4(c) then shows the hash trie configuration after the successful *CAS* operation that adjusted node K_3 to entry E_n. After this step, E_n moves to the white state and K_3 becomes the next synchronization point for the insertion of new nodes on E_n. Note that the initial chain for E_k has not been affected yet, since K_2 still refers to K_3. Next, on Fig. 4(d), the chain is broken and K_2 is updated to refer to the second level hash H_{i+1}. The process then repeats for K_2 (the new last node on the chain for E_k). First, K_2 is remapped to entry E_n (Fig. 4(e)) and then it is removed from the original chain, meaning that the previous node K_1 is updated to refer to H_{i+1} (Fig. 4(f)). Finally, the same idea applies to the last node K_1. Here, K_1 is also remapped to a bucket entry on H_{i+1} (E_m in the figure) and then removed from the original chain, meaning in this case that the bucket entry E_k itself becomes a reference to the second level hash H_{i+1} (Fig. 4(h)). From now on, E_K is also a gray memory location since it will be no longer updated.

Concurrently with the remapping process, other threads can be inserting nodes in the same bucket entries for the new level. This is shown in Fig. 4(e), where a node K_4 is inserted before K_2 in E_n and, in Fig. 4(g), where a node K_5 is inserted before K_1 in E_m. As mentioned before, lock-freedom is ensured by the use of *CAS* operations when updating black state memory locations.

To ensure the correctness of the remapping process, we also need to guarantee that the nodes being remapped are not missed by any other thread traversing the hash trie. Recall that any chaining of nodes is closed by the last node referencing back the hash level for the node. Thus, if when traversing a chain of nodes, a thread T ends up in a hash level H_{i+n} different than the initial hash level H_i, this means that T has started from a bucket entry E_k being remapped, which includes the possibility that some nodes initially on E_k were not seen by T. To guarantee that no node is missed, T simply needs to move backwards and restart its traversal from H_{i+1}.

We argue that a key design decision in our approach is thus the combination of hash tries with the use of a separate chaining (with a threshold value) to resolve hash collisions (the original hash trie design expands a bucket entry when a second key is mapped to it). Also, to ensure that nodes being remapped are not missed by any other thread traversing the hash trie, any chaining of nodes is closed by the last node referencing back the hash level for the node, which allows to detect the situations where a node changes level. This is very important because it allows to implement a clean design to resolve hash collisions by simply moving nodes between the levels. In our design, updates and expansions of the hash levels are never done by using data structure replacements (i.e., create a new one to replace the old one), which also avoids the complex mechanisms necessary to support the recovering of the unused data structures. Another important design decision that minimizes the low-level synchronization problems leading to false sharing or cache memory effects, is the insertion of nodes done at the end of the separate chain. Inserting nodes at the end of the chain allows for dispersing the memory locations being updated concurrently as much as possible (the last node is always different) and, more importantly, reduces the updates for the memory locations accessed more frequently, like the bucket entries for the hash levels (each bucket entry is at most only updated twice).

4 Algorithms

This section presents the algorithms that implement our new lock-free hash trie design. We begin with Algorithms 1 and 2 that show the pseudo-code for the search/insert operation of a given key *K* in a hash level *H*. In a nutshell, the algorithms execute recursively, moving through the hierarchy of hash levels until *K* is found or inserted in a hash level *H* (for the entry call, *H* is the root level). Algorithm 1 deals with the hash level data structures and Algorithm 2 deals with the internal nodes in a separate chaining.

Algorithm 1 *SearchInsertKeyOnHash*(K, H)

```
 1: $B \leftarrow GetHashBucket(K, H, Level(H))$
 2: if $EntryRef(B) = H$ then {B is an empty bucket}
 3:    $newNode \leftarrow AllocNode()$
 4:    $Key(newNode) \leftarrow K$
 5:    $NextRef(newNode) \leftarrow H$
 6:    if $CAS(EntryRef(B), H, newNode)$ then
 7:       return $newNode$
 8:    else
 9:       $FreeNode(newNode)$
10: $R \leftarrow EntryRef(B)$
11: if $IsNode(R)$ then {start traversing the chain}
12:    return $SearchInsertKeyOnChain(K, H, R, 1)$
13: else {R references a second level hash}
14:    return $SearchInsertKeyOnHash(K, R)$
```

In more detail, Algorithm 1 starts by applying the hash function that allows obtaining the appropriate bucket entry *B* of *H* that fits *K* (line 1). Next, if *B* is empty (i.e., if *B* is referencing back the hash level *H*), then a new node *newNode* representing *K* is allocated and properly initialized (lines 3–5). Then, the algorithm tries to insert *K* on the head of *B* by using a *CAS* operation that updates *H* to *newNode* (line 6). If the operation is successful, then the node was successfully inserted and the algorithm ends by returning it (line 7). Otherwise, in case of failure, the head of *B* has changed in the meantime, so *B* is not empty (lines 10–14). Here, the algorithm then reads the reference *R* on *B* (line 10) and checks whether it references an internal node or a second hash level. If *R* is a node, then it calls Algorithm 2 to traverse the chain of nodes (line 12). Otherwise, it calls itself, but now for the second level hash represented by *R* (line 14).

Algorithm 2 shows the search/insert operation of a given key *K* in a hash level *H* starting from a node *R* at position *C* in a separate chaining (for the entry call, *C* is 1 and *R* is the head node in the chain). Initially, the algorithm simply checks if *R* holds the key *K*, in which case, it ends by returning *R* (lines 1–2). Otherwise, it checks if *R* is the last node in the chain (line 3). If so, then two situations might occur: (i) the chain is full, in which case, the expansion procedure should be activated (lines 5–13); or (ii) the chain is not full, in which case, a new node representing *K* should be inserted in the chain (lines 15–21).

Algorithm 2 *SearchInsertKeyOnChain*(K, H, R, C)

```
 1: if $Key(R) = K$ then {we have found K in the chain}
 2:    return $R$
 3: if $NextRef(R) = H$ then {R is last in the chain}
 4:    if $C = MAX\_NODES$ then {chain is full}
 5:       $newHash \leftarrow AllocInitHash(Level(H) + 1)$
 6:       $PrevHash(newHash) \leftarrow H$
 7:       if $CAS(NextRef(R), H, newHash)$ then
 8:          $B \leftarrow GetHashBucket(K, H, Level(H))$
 9:          $AdjustChainNodes(EntryRef(B), newHash)$
10:          $UPDATE(EntryRef(B), newHash)$
11:          return $SearchInsertKeyOnHash(K, newHash)$
12:       else
13:          $FreeHash(newHash)$
14:    else
15:       $newNode \leftarrow AllocNode()$
16:       $Key(newNode) \leftarrow K$
17:       $NextRef(newNode) \leftarrow H$
18:       if $CAS(NextRef(R), H, newNode)$ then
19:          return $newNode$
20:       else
21:          $FreeNode(newNode)$
22: $R \leftarrow NextRef(R)$
23: if $IsNode(R)$ then {keep traversing the chain}
24:    return $SearchInsertKeyOnChain(K, H, R, C + 1)$
25: else {R references a second level hash}
26:    while $PrevHash(R) \neq H$ do {move backwards}
27:       $R \leftarrow PrevHash(R)$
28:    return $SearchInsertKeyOnHash(K, R)$
```

For the former situation, a second level hash *newHash* is first allocated and initialized (lines 5–6) and then used to implement a synchronization point that will correspond to a *CAS* operation trying to update the next reference of *R* from *H* to *newHash* (line 7). If the *CAS* operation fails, that means that another thread has gained access to the expansion procedure. Otherwise, if successful, the algorithm starts the remapping process of adjusting the internal nodes on the separate chaining, corresponding to the bucket entry *B* at hand, to the new hash level (line 9) and, for that, it calls the *AdjustChainNodes*() procedure (see Algorithm 3 next). After that, it updates the bucket entry *B* to refer to the new level (line 10) and then Algorithm 1 is called again, this time to search/insert for *K* in the new hash level (line 11).

For the latter situation (lines 15–21), a new node representing *K* is allocated and properly initialized (lines 15-–17), and a *CAS* operation tries to insert it at the end of the chain. If successful, the reference to the new node is returned. Otherwise, this means that another thread has inserted another node in the chain in the meantime, which lead us to the situation in the last block of code (lines 22-–28), where *R* is not last in the chain.

In the last block of code, the algorithm then updates *R* to the next reference in the chain (line 22) and, as in Algorithm 1, it checks whether *R* references an internal node or a second hash level. If *R* is still a node, then the algorithm calls itself to continue traversing the chain of nodes (line 24). Otherwise, it returns to Algorithm 1, but now for the hash level after the given hash *H* (lines 26–28). Note that, if other threads are simultaneously expanding the hash tries, it might happen that we end in a hash level several levels deeper and thus incorrectly miss the node we are searching for. This is why we need to move backwards to the hash level after the given hash *H* (lines 26–27).

Algorithms 3, 4 and 5 show the pseudo-code for the remapping process of adjusting a chain of nodes to a new hash level *H*. Algorithm 3 is the entry procedure that ensures that the remapping process is done in reverse order, Algorithm 4 deals with the adjustment on hash level data structures and Algorithm 5 deals with the adjustment on a separate chaining.

In more detail, Algorithm 3 starts by traversing the nodes in the chain until reaching the last one. Then, for each node *R* in the chain (from last to first), it calls *AdjustNodeOnHash()* in order to remap *R* to the given new hash level *H*.

Algorithm 3 $AdjustChainNodes(R, H)$

```
if $NextRef(R) \neq H$ then
    $AdjustChainNodes(NextRef(R), H)$
$AdjustNodeOnHash(R, H)$
return
```

Algorithm 4 shows the pseudo-code for the process of remapping a given node *N* into a given hash *H*. It is quite similar to Algorithm 1, except for the fact that there is no need to allocate and initialize a new node with the key at hand (here, we already have the node). It starts by updating the next reference of *N* to *H* (line 1), next it applies the hash function that allows obtaining the appropriate bucket entry *B* of *H* that fits the key on *N* (line 2), and then, if *B* is empty, it tries to successfully insert *N* on the head of *B* by using a *CAS* operation (lines 3–5). Otherwise, *B* is not empty, and the same procedure as in Algorithm 1 applies (lines 6–10). The difference is that here it calls the *AdjustNodeOnChain()* and *AdjustNodeOnHash()* algorithms, instead of the *SearchInsertKeyOnChain()* and *SearchInsertKeyOnHash()* algorithms.

Algorithm 4 *AdjustNodeOnHash*(N, H)

```
$UPDATE(NextRef(N), H)$
$B \leftarrow GetHashBucket(Key(N), H, Level(H))$
if $EntryRef(B) = H$ then {B is an empty bucket}
    if $CAS(EntryRef(B), H, N)$ then
        return
$R \leftarrow EntryRef(B)$
if $IsNode(R)$ then {start traversing the chain}
    return $AdjustNodeOnChain(N, H, R, 1)$
else {R references a second level hash}
    return $AdjustNodeOnHash(N, R)$
```

Algorithm 5 then concludes the presentation. It shows the pseudo-code for the process of remapping a given node *N* into a hash level *H* starting from a node *R* at position *C* in a separate chaining. As before, Algorithm 5 also shares similarities, but now with Algorithm 2, except for the fact that there is no need to check if *N* already exists in the chain (lines 1—2 in Algorithm 2) and, as before, that there is no need to allocate and initialize a new node with the key at hand (lines 15—17 in Algorithm 2). The last block of code (lines 14–20) is also identical to Algorithm 2, except for the fact that it calls the *AdjustNodeOnChain()* and *AdjustNodeOnHash()* algorithms, instead of the *SearchInsertKeyOnChain()* and *SearchInsertKeyOnHash()* algorithms.

Algorithm 5 *AdjustNodeOnChain*(N, H, R, C)

```
 1: if $NextRef(R) = H$ then {R is last in the chain}
 2:     if $C = MAX\_NODES$ then {chain is full}
 3:         $newHash \leftarrow AllocInitHash(Level(H) + 1)$
 4:         $PrevHash(newHash) \leftarrow H$
 5:         if $CAS(NextRef(R), H, newHash)$ then
 6:             $B \leftarrow GetHashBucket(K, H, Level(H))$
 7:             $AdjustChainNodes(EntryRef(B), newHash)$
 8:             $UPDATE(EntryRef(B), newHash)$
 9:             return $AdjustNodeOnHash(N, newHash)$
10:         else
11:             $FreeHash(newHash)$
12:     else if $CAS(NextRef(R), H, N)$ then
13:         return
14: $R \leftarrow NextRef(R)$
15: if $IsNode(R)$ then {keep traversing the chain}
16:     return $AdjustNodeOnChain(N, H, R, C + 1)$
17: else {R references a second level hash}
18:     while $PrevHash(R) \neq H$ do {move backwards}
19:         $R \leftarrow PrevHash(R)$
20:     return $AdjustNodeOnHash(N, R)$
```

5 Proof of Correctness

In this section, we discuss the proof of correctness of our lock-free hash trie design. First, we prove that it is *linearizable* and then we prove that it is *lock-free*.

5.1 Linearizability

Linearizability is an important correctness condition for the implementation of concurrent data structures [9]. A operation is linearizable if it appears to take effect instantaneously at some moment of time t_{LP} between its invocation and response. The literature often refers to t_{LP} as a *Linearization Point (LP)* 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. We define now the invariants that must be *preserved* on every state of our design:

Inv_1 For every hash level H, $PrevHash(H)$ always refers to the previous hash level.

Inv_2 Given a bucket entry B belonging to a hash level H, B must comply with the following semantics: (i) its initial reference is H; (ii) after the first update, it must refer to a node N; (iii) after the second (and final) update, it must refer to a hash level H_d such that $PrevHash(H_d) = H$.

Inv_3 Given a node N in a chain of nodes starting from a bucket entry B belonging to a hash level H, N must comply with the following semantics: (i) its initial reference is H; (ii) after an update, it must refer to another node in the chain or to a hash level H_d (at least one level) deeper than H.

Inv_4 Given a chain of nodes in a bucket entry B belonging to a hash level H, the number C of nodes in the chain is always lower or equal than a predefined threshold value MAX_NODES ($MAX\_NODES \geq 1$).

Lemma 1 *In the initial configuration, all invariants hold.*

Proof Consider that H represents the root level for a hash trie (its initial configuration is the same as the one represented in Fig. 3(a)). Since H is the root level, the reference $PrevHash(H)$ is *Nil* (Inv_1), each bucket entry B is referring H (Inv_2) and the number C of nodes in any chain is 0 (Inv_3 and Inv_4). □

Lemma 2 *In Algorithm 1, if the CAS operation at line 6 succeeds (i.e., if both conditions at lines 2 and 6 are true), then all invariants hold.*

Proof After the successful execution of the *CAS* operation at line 6, the bucket entry B refers to *newNode* (Inv_2), *newNode* refers to H (Inv_3), as initialized at line 5, and $C = 1$ (Inv_4). Inv_1 is not affected. □

Lemma 3 *In Algorithm 2, if the CAS operation at line 18 succeeds (i.e., if conditions at lines 3, 4 and 18 are true, false and true, respectively), then all invariants hold.*

Proof After the successful execution of the *CAS* operation at line 18, the node *R* refers to *newNode* and *newNode* refers to *H*, as initialized at line 17 (Inv_3). Inv_4 also holds because since the condition at line 4 failed, meaning that initially $C_i < MAX\_NODES$, the insertion of a new node in the chain after *R* leads to $C_f = C_i + 1 \leq MAX\_NODES$. Inv_1 and Inv_2 are not affected. □

Lemma 4 *In Algorithm 2, if the CAS operation at line 7 succeeds (i.e., if conditions at lines 3, 4 and 7 are all true), then all invariants hold.*

Proof After the successful execution of the *CAS* operation at line 7, the node *R* refers to a deeper hash level *newHash* (Inv_3) and *PrevHash*(*newHash*) refers to the current hash level *H*, as initialized at line 6 (Inv_1). Inv_2 and Inv_4 are not affected. □

Lemma 5 *In Algorithm 2, the execution of the remapping process of adjusting a chain of nodes to a new hash level at lines 9–10, preserves the set of invariants.*

Proof At line 9, the *AdjustChainNodes()* procedure is called for the chain of nodes in the bucket entry *B* and for the deeper hash level *newHash* and, at line 10, *B* is updated to refer to *newHash* (Inv_2). The *AdjustChainNodes()* procedure then calls Algorithms 4 and 5. In Algorithm 4, at line 1, the node *N* being adjusted is made to refer to a deeper hash level (Inv_3). For the remaining parts of Algorithms 4 and 5, the proofs are similar to the proofs for Algorithms 1 and 2, as shown on the previous lemmas. Thus, the invariants still hold for Algorithms 4 and 5. □

Corollary 1 *The invariants hold on every configuration of our hash trie design due to Lemmas 1 to 5.*

Every operation that affects the configuration of the hash trie thus takes effect in specific linearization points, which are:

- *SearchInsertKeyOnHash()* is linearizable at successful *CAS* in line 6 (LP_1).
- *SearchInsertKeyOnChain()* is linearizable at successful *CAS* in lines 7 (LP_2) and 18 (LP_3) and when the bucket entry at hand is updated to refer to a deeper hash level in line 10 (LP_4).
- *AdjustNodeOnHash()* is linearizable at successful *CAS* in line 4 (LP_5) and when the node at hand is updated to refer to a deeper hash level in line 1 (LP_6).
- *AdjustNodeOnChain()* is linearizable at successful *CAS* in lines 5 (LP_7) and 12 (LP_8) and when the bucket entry at hand is updated to refer to a deeper hash level in line 8 (LP_9).

Next, we must prove that for a given key *K*, if *K* exists in the hash trie, then the algorithms are able to find it. Otherwise, if *K* does not exist, then the algorithms are able to insert it.

Lemma 6 *Consider Algorithm 1 with a given key K and a hash level H. If K exists in a chain of nodes in a hash level deeper than H, then Algorithm 1 computes the next* *hash level H_d where K can be found, and calls itself for H_d. When K exists in a chain of nodes in H, then Algorithm 1 maps K to the correct bucket B of H that holds K and calls Algorithm 2 to search for K in the separate chaining of B.*

Proof Since we are assuming that K already exists in a chain of nodes, the code between lines 2–9 can be ignored because the condition at line 2 is always false. If R is then a reference to a hash trie, the algorithm calls itself for the next hash level (as defined by Inv_2) and the process continues recursively until the condition at line 11 be true. At that stage, Algorithm 2 is called to search for K starting from the first node R in the corresponding separate chaining B of H. □

Lemma 7 *Consider Algorithm 2 with a given key K, a hash level H and a reference to the first node R in a chain of nodes. If K exists in the chain, then Algorithm 2 finds the node with K.*

Proof Since we are assuming that K already exists in a chain of nodes, the code between lines 3–21 can be ignored because the condition at line 3 is always false. If the condition at line 1 succeeds then K was found in the chain. Otherwise, if the chain is not being remapped to a second hash level, the algorithm uses the lines 22–24 to call itself recursively until it finds K at line 1. If the chain is being remapped, Inv_3 ensures that we will reach a reference to a hash level H_d which is deeper than H. Thus, at some point in the execution, the algorithm reads H_d at line 22, calling Algorithm 1 in the continuation with a hash level H_a one level deeper than H (not that H_d can be in a deeper level than H_a). The search process then continues using Lemma 6. Since K exists and was not found yet, Algorithm 2 will be called again, this time for H_a or for a deeper level and the process will be repeated until K be found in a node. □

Lemma 8 *If a given key K does not exist in the hash trie, then it will be inserted in the linearization points LP_1 or LP_3.*

Proof Since we are assuming that K does not exist in the hash trie, then the search procedure will necessarily end when it finds an empty bucket entry (line 2 in Algorithm 1) or when it reaches the last node in a chain of nodes not being remapped (line 3 in Algorithm 2). If the *CAS* operation at line 6 for Algorithm 1 (LP_1) or at line 18 for Algorithm 2 (LP_3) then succeeds, a new node with the key K was inserted in the hash trie. Otherwise, in case of *CAS* failure, the separate chaining at hand was changed by another thread T in the meantime. In particular, it could happen that T had inserted a node for K. The search process is then resumed and if K was inserted by another thread then, using Lemmas 6 and 7, Algorithm 2 will find it. Otherwise, the search process will end again in the lines mentioned above until K be successfully inserted in the hash trie. □

Corollary 2 *When a thread performs a search/insert operation for a given key K then, due to Lemmas 6 to 8, if K exists in the hash trie, then it is able to find it. Otherwise, if K does not exist, it is able to insert it.*

Theorem 1 *Our trie hash design is linearizable.*

5.2 Lock-Freedom

The lock-freedom property is very important because, although it allows individual threads to starve, it guarantees system-wide throughput [8]. To prove that our design is lock-free, we will prove that the insert and hash expansion operations always lead to *progress in the state* of the hash trie configuration. We will use the linearization points defined previously to prove that, even when the operations on those points fail, it exists at least another thread that has *updated a memory location* in the hash trie configuration.

Lemma 9 *The execution of the operations defined by the linearization points LP_4, LP_6 and LP_9 leads to progress in the state of the hash trie configuration because such operations are composed by unconditional updates.*

Lemma 10 *When a thread executes the operations defined by the linearization points LP_1, LP_2, LP_3, LP_5, LP_7 and LP_8 then the hash trie configuration has made progress.*

Proof All linearization points correspond to *CAS* operations on a given memory location M trying to update an initial reference to a hash level H with a reference R corresponding to a new node or hash level. Assuming that t_i is the instant of time where a thread T first reads H from M and that t_f is the instant of time where T executes the *CAS* operation trying to update H to R, then a successful *CAS* execution leads to progress in the state of the hash trie configuration because M was updated to R. Otherwise, if the *CAS* operation fails, that means that between instants t_i and t_f, the reference on M was changed, which means that at least another thread has changed M between the instants of time t_i and t_f, thus leading to progress in the state of the hash trie configuration. □

Corollary 3 *When a thread executes one of the linearization points LP_1 to LP_9 then, due to Lemmas 9 to 10, the hash trie configuration has made progress.*

Theorem 2 *Our trie hash design is lock-free.*

6 Performance Evaluation

Although the driving motivation for our proposal comes from our work on concurrent tabled logic programs, to put it in perspective, we first compared it against some of the best-known currently available implementations of lock-free hash tables, and for that we used a publicly available framework[2] developed to evaluate lock-free hash tables. We tested the following implementations: two CTries versions (**CT1** is the original approach and **CT2** is a second version with improved snapshots); the *ConcurrentHashMap* (**CHM**) and *ConcurrentSkipListMap* (**CSL**) from Java's standard library; and our lock-free hash trie design (**LFHT**)[3], all implemented on top of JDK version 1.7.0_25.

[2] Available at `https://github.com/axel22/Ctries`

[3] We have experimented with multiple configurations for the separate chaining and hash levels. Our best results were obtained with 4 nodes for the separate chaining and 8 entries for the hash levels, which are the current default values for the experiments that follow.

For the experiments, we used two benchmarks already available in the framework, **insert(N)** and **lookup(N)** for a numeric data-set with $N = 10^7$ different elements. The **insert(N)** benchmark starts with an empty set and inserts the N elements. The **lookup(N)** benchmark does N searches on a previously created data structure containing the same N elements. For both benchmarks, the work of inserting/searching the N elements is equally divided between the working threads[4]. In addition, we created a new benchmark, named **worst(N)**, for testing a worst case scenario where all threads fully insert the same N elements (we used a numeric data-set with $N = 2 * 10^6$ different elements). By doing this, it is expected that all threads will access the same data structures, to search/insert for elements, at similar times, thus stressing the synchronization on common memory locations, which can increase the aforementioned problems of false sharing and cache memory effects.

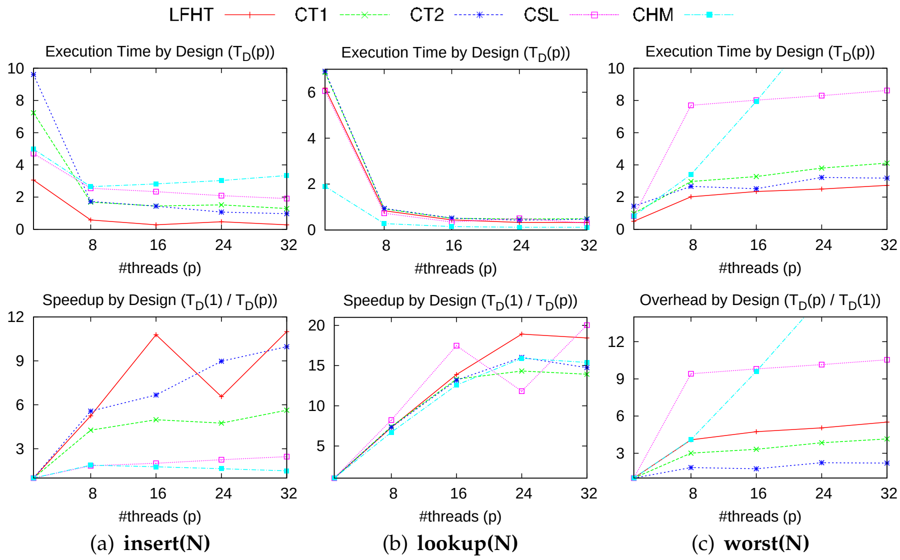
The environment for our experiments was a machine with 2x16 (32) Core AMD Opteron (TM) Processor 6274 @ 2.2 GHz with 32 GBytes of memory and running the Linux kernel 3.8.3-1.fc17.x86_64. We experimented with intervals of 8 threads up to 32 threads (the number of cores in the machine) and all results are the average of 25 runs for each benchmark. Figure 5 shows the execution time, in seconds, and the speedup/overhead, compared against the respective execution time with one thread, for the five designs when running the **insert(N)**, **lookup(N)** and **worst(N)** benchmarks.

For the **insert(N)** benchmark (Fig. 5(a)), **LFHT** has the best results for the execution time, showing a significant difference to all other designs. On average, **LFHT** is around three times faster than the second best design, which is **CT2**. Regarding the speedup, **CT2** competes with **LFHT** for the best results, but for most cases, **LFHT** still gets the best speedup. The top speedups for **LFHT** are 10.78 and 10.99 for 16 and 32 threads. For **CT2**, the top speedup is 9.96 for 32 working threads.

For the **lookup(N)** benchmark (Fig. 5(b)), **CHM** achieves the best results for the execution time followed by **CSL** and **LFHT** as third placed. When the work is split among multiple threads, **LFHT** is up to 1.5 times faster than **CT1** and **CT2**. For the speedup, **CSL** and **LFHT** show the best results. The top speedup for both designs is achieved for 32 threads with a 20.03 value for **CSL** and 18.44 for **LFHT**.

For the **worst(N)** benchmark (Fig. 5(c)), we are interested in evaluating the robustness of the implementations when exposed to worst case scenarios. As it is expected that the execution time with multiple threads will result in worst results when compared with the base execution time with one thread, we thus show the overhead (not the speedup) for comparing the execution with increasing number of threads (values close to 1.00 are thus better). For the execution time, **LFHT** shows again the best results with **CT2** being very close. For the overhead, **CT2** and **CT1** are better than **LFHT** mostly because the base execution times with one thread are significantly higher than **LFHT**

[4] Due to the lack of space, we are not including mixed insert/lookup benchmarks. Yet, we have experimented with a lot of mixed scenarios and we have verified that the results obtained are within the bounds of the results shown next for the **insert(N)** and **lookup(N)** benchmarks.



(a) **insert(N)** (b) **lookup(N)** (c) **worst(N)**

Fig. 5 Execution time, in seconds, and speedup/overhead, against the execution time with one thread, for the **insert(N)**, **lookup(N)** and **worst(N)** benchmarks with increasing number of threads

(0.50, 0.99 and 1.44 seconds, respectively, for the **LFHT**, **CT1** and **CT2** designs). The **CSL** and **CHM** designs show a poor performance for this benchmark. In particular, **CHM** has the worst results with an overhead almost linear to the number of working threads.

In summary, these experiments show that our new lock-free design clearly outperforms all the other designs for the execution times and that, in general, it also achieves the best results for the speedup/overhead in most experiments.

In the context of Yap, we then evaluated our lock-free hash trie design (**LFHT**)[5] against all the previously implemented lock-based and lock-free strategies for concurrent tabling. For the sake of simplicity, here we will only consider Yap's best lock-based strategy (**LB)** and the lock-free design (**LF**) presented in [3]. For benchmarking, we used the set of tabling 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.

Since the system's performance is highly dependent on the available concurrency that a particular program might have, our initial goal was to evaluate the robustness of our implementation when exposed to worst case scenarios. As before, we thus followed a common approach to create worst case scenarios and we ran all threads executing the same query goal for each benchmark.

[5] Note that we have separate implementations for Yap and for JDK.

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. Since, all threads are executing the same query goal, it is expected that the aforementioned problems of false sharing and cache memory effects to show up and thus penalize the less robust designs.

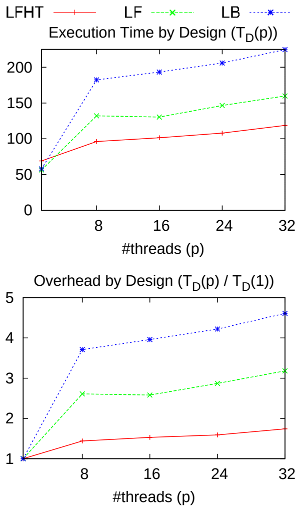


Fig. 6 Execution time, in seconds, and overhead, against the execution time with one thread, for the set of 19 tabling benchmarks with all threads executing the same query goal for each benchmark

For the experiments, we used Yap Prolog 6.3 running on top of the same 32 Core AMD machine. To put the results in perspective, we experimented with intervals of 8 threads up to 32 threads and all results are the average of 5 runs for each benchmark. Figure 6 shows the average execution time, in seconds, and the average overhead, compared against the respective execution time with one thread, for the **LFHT**, **LF** and **LB** designs when running the set of tabling benchmarks with all threads executing the same query goal for each benchmark.

The results clearly show that the new **LFHT** design achieves the best performance for both the execution time and the overhead. As expected, **LF** is the second best and **LB** is the worst. In general, our design clearly outperforms the other designs with an overhead of at most 1.74 for 32 threads. Another important observation is that both **LF** and **LB** show an initial high overhead in the execution time in most experiments, mainly when going from 1 to 8 threads, in contrast to **LFHT** that shows more smooth curves. The difference between **LFHT** and **LF**/**LB** for the overhead ratio in these benchmarks clearly shows the distinct potential of the **LFHT** design.

Besides measuring the value of our new design through the use of worst case scenarios, we conclude the paper by showing the potential of our work to speedup the execution of tabled programs. Other works have already showed the capabilities of the use of multithreaded tabling to speedup tabled execution [12]. Here, for each program, we considered a set of different queries and then we ran this set with different number of threads. To do that, we implemented a naive scheduler in Prolog code that initially launches the number of threads required and then uses a *mutex* to synchronize access to the pool of queries. We experimented with a **Path** program using a grid-like configuration and with two well-known ILP data-sets, the **Carcino**genesis and **Muta**genesis data-sets. We used the same 32 Core AMD machine, experimented with intervals of 8 threads up to 32 threads and the results that follow are the average of 5 runs. Figure 7 shows the average execution time, in seconds, and the average speedup, compared against the respective execution time with one thread, for running the naive scheduler on top of these three programs with the **LFHT** design.

The results show that our design has potential to speedup the execution of tabled programs. For the **Path** benchmark, the speedup increases up to 10.24 with 16 threads, but then it starts to slow down. We believe that this behavior is related to the large number of tabled dependencies in the program. For the **Carcino** and **Muta** benchmarks, the speedup increases up to a value of 16.68 and 18.84 for 32 threads, respectively. Note that our goal with these experiments was not to achieve maximum speedup because this would require to take into account the peculiarities of each program and eventually develop specialized schedulers, which is orthogonal to the focus of this work.

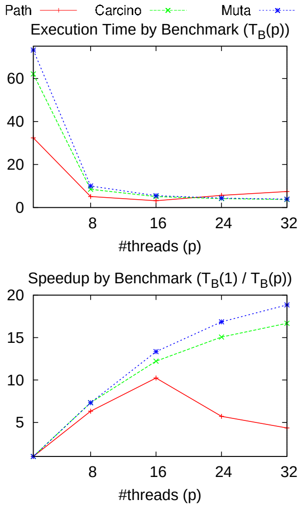


Fig. 7 Execution time, in seconds, and speedup, against the execution time with one thread, for running the naive scheduler program with the **LFHT** design

7 Conclusions

We have presented a novel, simple and efficient lock-free design for concurrent tabling environments based on hash tries. Our main motivation was to refine the previous designs in order to be as effective as possible in the concurrent search and insert operations over the table space data structures. We discussed the relevant details of the proposal, described the main algorithms and proved the correctness of our implementation. We based our discussion on Yap's concurrent tabling environment, but our design can be applied to general purpose applications, such as word counting, compilers, language run-times and some components of game development, that only require search and insert operations on their hash mapping mechanisms.

A key design decision in our approach is the combination of hash tries with the use of a separate chaining closed by the last node referencing back the hash level for the node. This allows us to implement a clean design to solve hash collisions by simply moving nodes between the levels. In our design, updates and expansions of the hash levels are never done by using data structure replacements (i.e., create a new one to replace the old one), which also avoids the need for memory recovery mechanisms. Another key design decision that minimizes the bottlenecks leading to false sharing or cache memory effects, is the insertion of nodes done at the end of the separate chain. This allows for dispersing the memory locations being updated concurrently as much as possible and, more importantly, reduces the updates for the memory locations accessed more frequently, like the bucket entries for the hash levels.

Experimental results obtained independently (i.e., not within Yap) show that our new lock-free design can effectively reduce the execution time and scales better than some of the best-known currently available lock-free hashing implementations. In the context of Yap's concurrent tabling support, our design clearly achieved the best results for the execution time, speedup and overhead ratios. In particular, for worst case scenarios, our design clearly outperformed the previous designs with a superb overhead always below 1.74 for 32 threads or less. We thus argue that our design is the best proposal to support concurrency in general purpose multithreaded tabling applications.

Acknowledgements 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 SIBILA (NORTE-07-0124-FEDER-000059) 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. Areias, M., Rocha, R.: On the Correctness and Efficiency of Lock-Free Expandable Tries for Tabled Logic Programs. In: International Symposium on Practical Aspects of Declarative Languages, no. 8324 in LNCS, pp. 168–183. Springer (2014)
4. Bagwell, P.: Ideal Hash Trees. Es Grands Champs **1195** (2001)
5. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43**(1), 20–74 (1996)
6. Dawson, S., Ramakrishnan, C.R., Warren, D.S.: Practical Program Analysis Using General Purpose Logic Programming Systems – A Case Study. In: ACM Conference on Programming Language Design and Implementation, pp. 117–126. ACM (1996)
7. Fredkin, E.: Trie Memory. Communications of the ACM **3**, 490–499 (1962)
8. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
9. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems **12**(3), 463–492 (1990)
10. Johnson, M.: Memoization in top-down parsing. Computational Linguistics **21**(3), 405–417 (1995)
11. Marques, R., Swift, T.: Concurrent and Local Evaluation of Normal Programs. In: International Conference on Logic Programming, no. 5366 in LNCS, pp. 206–222. Springer (2008)
12. 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, no. 5937 in LNCS, pp. 264–278. Springer (2010)
13. Moura, P.: ISO/IEC DTR 13211–5:2007 Prolog Multi-threading Predicates (2008). URL `http://logtalk.org/plstd/threads.pdf`
14. Prokopec, A., Bronson, N.G., Bagwell, P., Odersky, M.: Concurrent Tries with Efficient Non-Blocking Snapshots. In: ACM Symposium on Principles and Practice of Parallel Programming, pp. 151–160. ACM (2012)
15. Ramakrishna, Y.S., Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S.A., Swift, T., Warren, D.S.: Efficient Model Checking Using Tabled Resolution. In: Computer Aided Verification, no. 1254 in LNCS, pp. 143–154. Springer (1997)
16. Rocha, R., Silva, F., Santos Costa, V.: On applying or-parallelism and tabling to logic programs. Theory and Practice of Logic Programming **5**(1 & 2), 161–205 (2005)
17. Sagonas, K., Swift, T., Warren, D.S.: XSB as an Efficient Deductive Database Engine. In: ACM International Conference on the Management of Data, pp. 442–453. ACM (1994)
18. 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)
19. Shalev, O., Shavit, N.: Split-Ordered Lists: Lock-Free Extensible Hash Tables. Journal of the ACM **53**(3), 379–405 (2006)

20. 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)
21. Zou, Y., Finin, T.W., Chen, H.: F-OWL: An Inference Engine for Semantic Web. In: International Workshop on Formal Approaches to Agent-Based Systems, *LNCS*, vol. 3228, pp. 238–248. Springer (2004)