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

On the Implementation of a Lock-Free Atom Table in a Prolog System

Pedro Moreno · Miguel Areias · Ricardo Rocha · Vítor Santos Costa

Received: date / Accepted: date

Abstract Prolog systems rely on an atom table for symbol management, which is usually implemented as a dynamically resizeable hash table. This is ideal for single threaded execution, but can become a bottleneck in a multi-threaded scenario. In this work, we replace the original atom table implementation in the Yet Another Prolog (YAP) system with a lock-free hash-based data structure, named Lock-free Hash Tries (LFHT), in order to provide efficient and scalable symbol management. Being lock-free, the new implementation also provides better guarantees, namely, immunity to priority inversion, to deadlocks and to livelocks. Performance results show that the new lock-free LFHT implementation has better results in single threaded execution and much better scalability than the original lock based dynamically resizing hash table.

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

1 Introduction

The initial programming languages were designed to abstract the computer hardware where, to achieve reasonable performance, a developer would have to learn first how to express the algorithmic problems in machine-oriented terms. Higher-level languages were created to allow developers to program algorithmic resolutions in terms closer to the problem's conceptualization. It is believed that higher-level languages are particularly helpful in developing

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020. Pedro Moreno is funded by the FCT grant SFRH/BD/143261/2019.

Pedro Moreno · Miguel Areias · Ricardo Rocha · Vítor Santos Costa
CRACS/INESC TEC and Dept. of Computer Science, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
E-mail: {pmoreno, miguel-areias, ricroc, vsc}@dcc.fc.up.pt

succinct and correct programs that are easy to write and also easy to understand. Logic programming languages, together with functional programming languages, form a major class of such languages, called *declarative languages*, and because logic programming languages are based on the predicate calculus, they have a strong mathematical basis.

Prolog is the most popular and powerful logic programming language. Prolog gained its popularity mostly because of the success of the sophisticated compilation technique and abstract machine known as the Warren's Abstract Machine (WAM) presented by David H.D. Warren in 1983 [23]. Nowadays, it is widely used in multiple domains, such as, machine learning [14], program analysis [7], natural language analysis [13], bioinformatics [12] and semantic web [8]. Prolog systems represent data as terms, that can be number, strings, or atoms, or a composition of terms. Prolog atoms are particularly important, as they are both used as symbols and as a convenient representation of strings. In this work, we focus on the *Atom Table* used for atom management and we investigate whether the traditional design can still be a good solution for recent challenges Prolog systems face.

One such challenge is to take best advantage of multi-core/multi-threaded architectures, arguably one of the most popular and impactful recent hardware developments. This type of architectures allow greater performance, but resources must be properly managed and exploited. Many languages and systems were not originally designed for multi-processing, which required them to be later extended to support this type of architectures, and Prolog systems were no exception.

Multi-threading in Prolog is the ability to perform concurrent computations, in which each thread runs independently but shares the program clauses [11]. Almost all Prolog systems support some sort of multi-threading. In particular, the Yet Another Prolog (YAP) multi-threading library [17] can be seen as a high-level interface to the POSIX threads library, where each thread runs on a separate data area but shares access to the global data structures (code area, atom table and predicate table). As each thread operates its own execution stack, it is natural to associate each thread with an independent computation that can run in parallel as threads already include all the machinery to support shared access and updates to the global data structures and input/output structures.

In this paper, we replace the original atom table implementation in the YAP system with a lock-free hash-based data structure, named Lock-free Hash Tries (LFHT), in order to investigate whether an efficient and scalable symbol management, can make a difference in a multi-threaded environment. Performance results show that the new implementation shows better results both in single threaded execution and much better scalability than the original atom table.

The remainder of the paper is organized as follows. First, we introduce relevant background and present the main ideas of our design. Next, we describe in detail the points required to easily reproduce our implementation. Then, we present a set of experiments comparing the new atom table against

the original one. At the end, we present conclusions and draw further work directions.

2 Background

In this section, we describe the context of our work with particular focus on the YAP system, the concurrent access to the atom table, and the LFHT design.

2.1 The YAP System

Yet Another Prolog (YAP) is a Prolog system originally developed in the mid-eighties and that has been under almost constant development since then [18].

Figure 1 presents a high-level picture of the YAP system. The system is written in C and Prolog. Interaction with the system always starts through the top-level Prolog library. Eventually, the top-level refers to the core C libraries. The main functionality of the core C libraries includes starting the Prolog engine, calling the Prolog clause compiler, and maintaining the Prolog internal database. The engine may also call the just-in-time indexer (JITI) [19]. Both the compiler and the JITI rely on an assembler to generate code that is stored in the internal database. The C-core libraries further include the parser and several built-ins (not shown in Fig. 1). An SWI-Prolog compatible threads library [24] provides support to thread creation and termination, and access to locking.

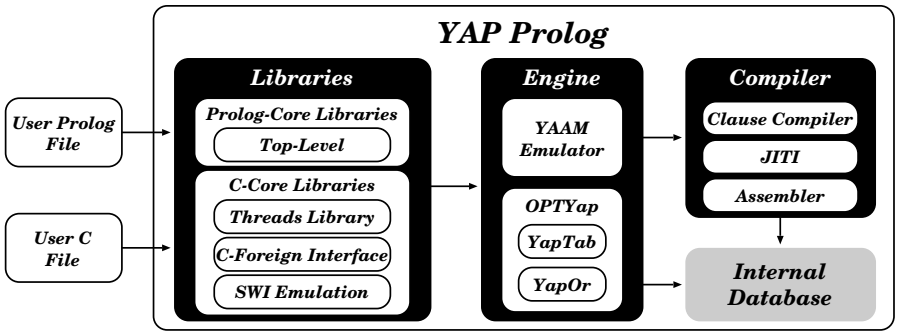

Fig. 1 The YAP system

YAP includes two main components, the *Engine* and the *Database*. The Engine maintains the abstract machine internal state, such as abstract registers, stack pointers, and active exceptions. The Database maintains the root pointers to the internal database, which includes the *Atom Table* and the *Predicate Table*. In order to support multi-threading, YAP's data structures are organized as follows:

– the GLOBAL structure is available to all threads and references the global data structures; locks should protect access to these data structures.
– the LOCAL structure is a per-thread array referencing the thread's local data structures, e.g., the engine abstract registers, internal exceptions, and thread specific predicates. The data is accessible through the thread's LOCAL structure, whose address is available from thread-local storage.

Figure 2 presents in more detail YAP's internal data structures with particular emphasis on the atom table. It assumes support for two threads, hence it requires two LOCAL structures, each containing a copy of the corresponding WAM registers.

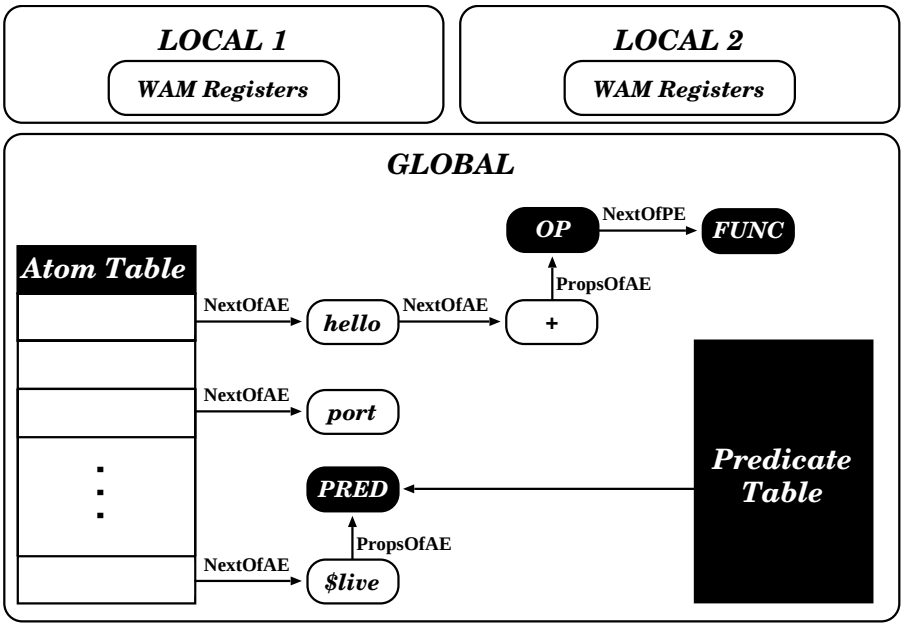

Fig. 2 YAP's internal data structures

The main structure inside GLOBAL is the *Atom Table*, which contains objects of the abstract type Atom. As discussed above, atoms are used to represent symbols and text. The latter usage stems because the same text can appear in different parts of a program. Storing text as atoms can save both space and time, once to compare two segments one just has to compare atoms, e.g., two text segments match if and only if they are the same atom, that is, if they have the same entry in the atom table. At the implementation level, the atoms are stored in a linked-list and each node within that linked-list has a reference to a secondary linked-list, that holds the properties of the atoms. Predicates with atoms as name are also stored in the atom table. Predicates are also often present in a Prolog program and there might exist several predicates with the

same name (but with a different arity or belonging to different modules), and in such situations, there is a direct hash-table for them.

The abstract type `Atom` has a single concrete type, `AtomEntry`. Thus, the atom table is implemented as a single-level bucket array hash table with a separate chaining mechanism, implemented as linked lists, to support collisions among `AtomEntry` objects. Once the bucket array data structure is saturated, the hash table duplicates its size, and the `AtomEntry` objects are placed in the newly created data structure. Each `AtomEntry` contains

1. `StrOfAE`: a `C` representation of the atom's string;
2. `NextOfAE`: a pointer to the next atom in the linked list for this hash entry;
3. `PropsOfAE`: a pointer to a linked list of atom properties;
4. `ARWLock`: a reader-writer lock that serializes access to the atom.

The `Prop` type abstracts objects that we refer to by the atom's name. Example subtypes of `Prop` include functors, modules, operators, global variables, blackboard entries, and predicates. All of them are available by looking up an atom and following the linked list of `Prop` objects.

Figure 2 shows an atom table with four atoms: `hello`, `+`, `port`, and `$live`. Notice that only `+` and `$live` have associated properties. In practice, most atoms do not have properties. Every concrete type of `Prop` implements two fields:

1. `KindOfPE` gives the type of property;
2. `NextOfPE` allows organizing properties for the same atom as a linked list.

Each property extends the abstract property in its own way. As an example, *functors* add three extra fields: a back pointer to the atom, the functor's arity, and a list of predicates that share the same name and arity, but belong to different modules.

This design is based on LISP implementations, and has been remarkably stable throughout the history of the system. Main optimizations and extensions include:

1. Older versions of YAP support two atom tables: one groups all ISO-Latin-1 atoms, where each character code c is such that $0 < c < 255$, and the other stores atoms that need to be represented as wide strings. Recent versions of YAP use UTF-8 internally.
2. As discussed above, functors have their own `Prop` objects, namely, predicates and internal database keys with that functor. This was implemented to improve performance of meta-calls.
3. The case where we have predicates with the same functor but belonging to different modules is addressed by a *predicate hash-table*, which allows direct access to a predicate from a functor-module key. A typical example is Machine Learning where each example is a module containing different versions of the same predicate.

In Fig. 2, the atom `+` has two properties: one of the type `op` and another of type `functor`. The atom `$live` has a property of type `predicate`.

2.2 Lock-Free Hash Tries

YAP's atom table uses single-level hash buckets that doubles size once they are saturated. Concurrent accesses to the atom table are serialized by the use of reader-writer locks.

Lock-freedom is an alternative to lock based data structures that allows individual threads to starve but guarantees system-wide throughput. Lock-free data structures offer several advantages over their lock-based counterparts, such as, being immune to deadlocks, lock convoying and priority inversion, and being preemption tolerant, which ensures similar performance regardless of the thread scheduling policy. Lock-freedom takes advantage of the *Compare-And-Swap (CAS)* atomic primitive 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.

Hash tries [6] minimize 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. An essential property of the trie data structure is that common prefixes are stored only once [9], 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 [20], Triplett *et al.* relativistic hash tables [22] or Prokopec *et al.* CTries [16].

The Lock-Free Hash Tries (LFHT) design, as originally proposed by Areias and Rocha [1,2], is a tree based data structure implementing two types of nodes: *hash nodes*, used to represent the hierarchy of hash levels where keys are indexed; and *leaf nodes*, used to store the key-value pairs. Figure 3 shows the general architecture of the LFHT design.

Each key is used to compute a hash h, which is then used to map the corresponding key-value pair in the LFHT hierarchy. For that, it uses chunks of w bits from h to index the entry in the appropriate hash level, i.e., for each hash level H_i, it uses the i^{th} group of w bits of h to index the entry in the appropriate bucket array of H_i. All bucket entries in a hash node are initialized with a reference to the hash node itself. During execution, each bucket entry stores either a reference to a hash node (itself or a deeper hash node) or a reference to a separate chaining mechanism of leaf nodes, that deals with the hash collisions for that entry. Intermediate leaf nodes hold a reference to the next-on-chain leaf node.

To find the value associated with a given key, it begins by computing the corresponding hash value. Then, nodes are searched in the LFHT structure by following the path given by the hash value. If the key exists, it will be found in a leaf node and the corresponding associated value is returned.

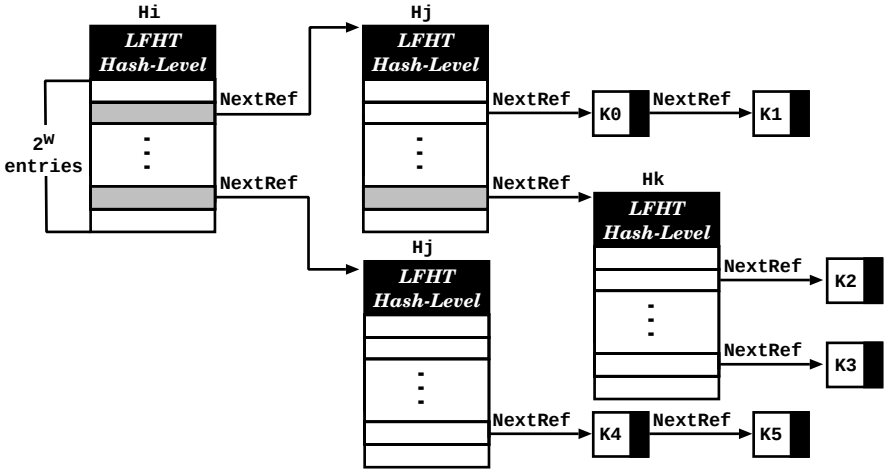

Fig. 3 General architecture of the LFHT design

The original LFHT design was implemented in `C` and proposed in the context of YAP's concurrent tabling engine [1]. In a nutshell, tabling is a refinement of Prolog's standard 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. This means that in a traditional tabling environment, only concurrent search and insert operations are executed. The authors of LFHT took advantage of this fact to create a table space design that would be as efficient as possible in these two operations. Since no remove operations were executed concurrently, no emphasis was given to memory reclamation. All memory used to represent the table space would remain valid during the execution of a concurrent tabled logic program. Only at the end, when running in single-threaded mode, could memory resources be released to the operating system.

As LFHT obtained interesting results, the authors consider the possibility of extending it to support the remove operation in order to make LFHT available as a standalone data structure. However, supporting removals implied that the design would have to support some sort of memory reclamation or garbage collection mechanism or, alternatively, to be implemented on top of a framework that would do that by default. The authors decided to exploit the advantages of the *Java Virtual Machine (JVM)* and re-implemented the design from scratch in `Java`, adding the support for the remove operation [3].

To maintain LFHT's lock-freedom property, the remove operation was implemented in three stages. On the first stage, the memory is logically removed, i.e., the block of memory m being removed is marked with some sort of tag in such a way that all other threads know that the information in m is no longer valid. On the second stage, all the memory references to m stored in other structures are deleted, meaning that, from a given instant

of time, threads entering the LFHT data structure no longer see or reach the memory *m*. Finally, on the third stage, the memory is physically released, i.e., the memory *m* can be reused in other context or freed to the host operating system. In this three stages scenario, JVM's garbage collector is very useful as it already implements the third stage by default, leaving the focus on the implementation of the first and second stages. In 2021, the LFHT design was dully formalized to prove its correctness, in particular the expand operation that handles with key collisions [4], and more recently, the design evolved as a standalone Java application with new features and operations, such as, the compress operation that is able to free unused hash levels [5].

At the same time, and starting from the ideas in the Java implementation with the remove operation, the LFHT implementation in `C` was adapted and extended to support a memory reclamation scheme that could fully support the three stages described above without losing the lock-freedom property [15], meaning that the design could finally meet the goal of being used as a standalone data structure and application. Experimental results showed that such a design is very competitive and scalable, when compared against the *Concurrent Hash-Map* implementation used in the Intel's *Thread Building Blocks (TBB)* library. More recently, the LFHT design was improved even further with a compression based design that would improve throughput [10].

3 Our Proposal

This section describes our proposal to improve the performance of YAP's atom table in concurrent environments. For that, we replaced the original version of the atom table, based in single level hashing, by the LFHT design in such a way that, instead of having a specialized version of a concurrent hash table implementing the atom table, we can simply use the general purpose LFHT design and allow it to manage everything, which goes from managing the concurrent accesses, to indexing the atoms for a faster access and handling atom collisions through a highly efficient chaining mechanisms. Moreover, to free memory from the atom table, we also take advantage of LFHT's memory reclamation mechanism, which will automatically handle the physical removal of atoms and corresponding internal data structures.

In what follows, we show in more detail how the LFHT data structure was integrated into the YAP system. To make the integration as smooth as possible, we need to understand all the details regarding YAP's internal database and how it is accessible from all internal and external libraries and data structures. Figure 4 presents the new organization of YAP's internal data structures based in the LFHT design (for comparison with Fig. 2, we left in gray the parts that were not changed from the original design). For the sake of presentation, the LFHT hash levels shown at the left of the figure are presented in a compact way as a single level, representing the initial configuration, which will be expanded during executing to multiples levels as described in the previous section.

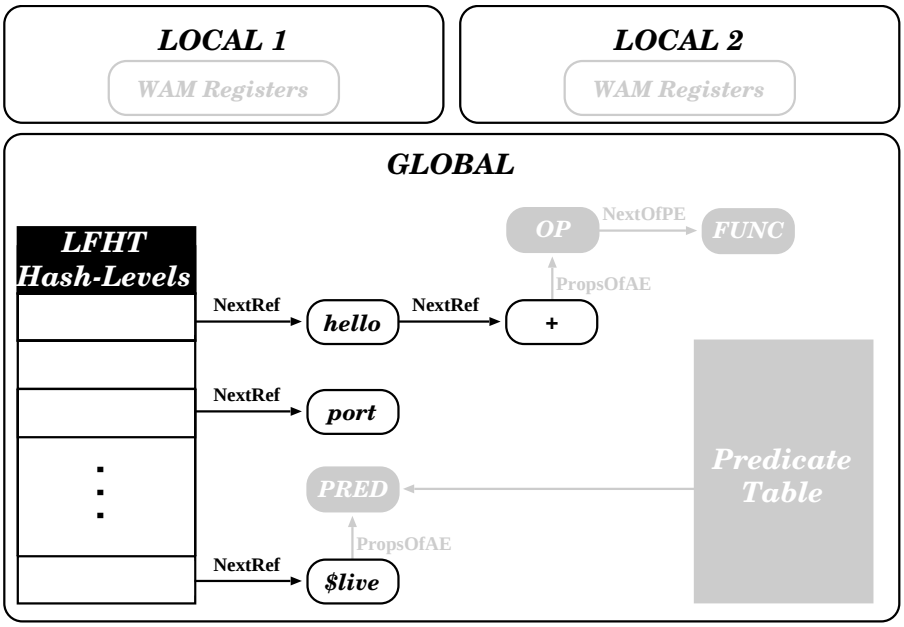

Fig. 4 The new organization of YAP's internal data structures

When comparing the new organization in Fig. 4 with the previous one in Fig. 2, one can observe two main modifications. The original `NextOfAE` field was removed, since the chaining mechanism will be managed by LFHT's design, and the read-writer lock `ARWLock`, used to serialize the access to the atoms in the original version of the atom table, was also removed, since now the LFHT design only uses CAS operations.

Using CAS operations instead of read-writer locks has some advantages. It can potentially reduce significantly the number of write operations done in memory during the execution of a program. At the implementation level, a read-write lock, requires writing operations even in when threads are only reading information from a protected memory region. This happens because read-write locks need to keep track of the number of threads that are in a protected memory region and, to do so, they use standard atomic counters. Moreover, these writing operations require also memory barriers to ensure the consistency of memory operations. These memory barriers have a considerable cost in the performance of a system, since they apply an ordering constraint between all memory operations that occur before and after the memory barrier, affecting this way all running threads.

Note that LFHT does not completely avoid memory barriers, as the CAS operation also uses them when executing a write operation. The gain comes from the fact that the design is lock-free, which means that reading operations do not require any write operations.

The remaining data structures and references are unchanged. This is the case of the `PropsOfAE` pointer to the atom's *properties* and the `StrOfAE` representation of the atom's string, therefore allowing the other YAP's data structures, such as the Predicate Table, to still access the atoms' information as they do in the original design.

In order to fully replace YAP's atom table with LFHT's design, some additional extensions were required to ensure full compatibility with the original design. These extensions include: (i) support for arbitrary keys and full-hashing collisions; and (ii) an iteration mechanism capable of traversing all keys stored in the atom table in a given instant of time. In the following subsections, we discuss how these extensions were implemented.

3.1 Arbitrary Keys

By default, the LFHT implementation assumes that the hash function is good enough to avoid key collisions, meaning that it relies only on the generated hash value to find a key, thus not considering the case of two keys generating the same hash value. To also consider this situation, when searching for a key K, we still use the hash value h to move through the hash levels but, when a node N corresponding to h is found, we need to confirm that N holds K. And, if this is not the case, we keep searching for the next node corresponding to h that may hold K.

YAP's atom table uses strings as keys, and although we could add support for strings to LFHT's design, we decided to implement a more general solution independently of the type of the key. During LFHT's initialization, now we must give the following parameters: (i) a key comparison function; (ii) a hash function; and (iii) a key destructor function. The key comparison function should implement the comparison of keys to be used in the hash value searching mechanism. The hash function allows to simplify the API, since now we only need the key as argument to the LFHT operations instead of both the key and the hash value. The key destructor functions allows to free memory used by the key when we remove a node. We also allow for any of these parameters to be undefined, and in such case we disable the associated feature. For example, if no hash function is defined, we assume that the given key is the hash itself, if no key comparison function is passed, we assume that the user knows that hash values will not collide, and if no key destructor is passed, we assume that the key will never be deleted during the execution. Figure 5 shows the new `C` language high-level API of the LFHT data structure.

3.2 The Iteration Procedure

During the execution of a program, a Prolog system might be required to iterate over all atoms present in the atom table. YAP is no exception, thus LFHT data structure was extended to support this additional operation. In

```
// Initializes the data structure and returns a handler
struct lfht_head *init_lfht(size_t (*hash_func)(void *),
       int (*key_cmp)(void *, void *), void (*key_free)(void *));

// Returns the value associated with the key if it exists
void *lfht_search(struct lfht_head *head, void *key);

// Returns the value associated with the key if it exists,
// otherwise inserts the key with the provided value
void *lfht_insert(struct lfht_head *head, void *key, void *value);

// Removes the key and returns the associated value
void *lfht_remove(struct lfht_head *head, void *key);

// Returns the next key in hash/key order
void *lfht_next_key(struct lfht_head *head, void *key);
```

Fig. 5 C language high-level API of the LFHT data structure

a nutshell, the iterator of LFHT data structure presents atoms by the natural order that their hash value appears in the data structure for collision free atoms, otherwise, the LFHT data structure consumes the atoms by the natural order of their keys.

At the implementation level, the iterator begins by presenting the atom with the lowest hash value. And then, to present the next atom it uses the previously presented atom, and the process continues until there are no more atoms to be presented. If there are atoms with the same hash value, it presents the next smallest key with the same hash value. Otherwise, returns the smallest key of the next available smallest hash. By iterating this way, it ensures that iteration is done over all keys that were present when the iteration began and that were not removed during the iteration process. Keys that are inserted concurrently during an iteration might not be presented, this will happen if the iterator is iterating over a hash value which is higher than the hash value of the key that was inserted.

Algorithm 1 shows how the iteration process is done over the hash nodes, in order to find the next key. Note that we use the hash value from the most significant bits to the least significant bits from the first level to the last level, so that we can have the property that nodes in a bucket $B[i]$ always have smaller hash values than nodes in a bucket $B[k]$ in the same hash node (for $i < k$). To find the first key we pass the *Null* key to the *Iterator* function which lets us start at the bucket entry corresponding to the hash with value 0. Otherwise, we compute the hash value from the key and start iterating from the corresponding bucket. We begin in the root hash node and, if in the corresponding bucket we find a new hash node, we try to recursively find a next key in such hash node. If the bucket contains leaf nodes we call the *IterateChain()* function described in Algorithm 2 in order to find a next key in the chain. In both situations, if we find such a key we return it, otherwise we

continue searching in the next bucket. If we reach the end of the hash node without finding a key, we return *Null* in order to indicate no key was found.

Algorithm 1 *Iterate*(*Key K, Node Hn*)

```
1:  if K = Null then
2:    H ⇐ 0
3:  else
4:    H ⇐ Hash(K)
5:  for i ⇐ Index(Hn, H) to Hn.size do
6:    if Hn.array[i].type = HASHNODE and Hn.array[i] ≠ Hn then
7:      R ⇐ Iterate(K, Hn.array[i])
8:    else if Hn.array[i].type = LEAFNODE then
9:      R ⇐ IterateChain(K, H, Hn.array[i])
10:   if R ≠ Null then
11:     return R
12: return Null
```

Algorithm 2 shows how we find the next node in a chain. We need to iterate over the whole chain as the nodes are unordered in the chain. We start by filtering the nodes that are actually ordered after the key provided, then we start by assigning the 1st node to *N* and replace it if we find a node that is ordered before it[1].

Algorithm 2 *IterateChain*(*Key K, Hash H, Node Ln*)

```
1: N ⇐ Null
2: while Ln.type = LEAFNODE do
3:   if Ln.hash > H or (Ln.hash = H and (K = Null or Ln.key > K)) then
4:     if N = Null or Ln.hash < N.hash or (Ln.hash = N.hash and
       Ln.key < N.key) then
5:       N ⇐ Ln
6:   Ln ⇐ Ln.next
7: return N
```

4 Experimental Results

In order to evaluate the impact of our proposal, we next show experimental results comparing the original and new versions of YAP's atom table. To put the results in perspective, we also compare both YAP's implementations with SWI-Prolog, a well-known and popular Prolog system that also implements concurrent support for the atom table in a lock-free fashion [25]. SWI-Prolog uses a single-level hash design to implement the atom table with lock-free operations, except for the resizing of the hash table, which is not lock-free

[1] Note that, for the sake of simplicity, we are omitting how the iterator proceeds when a concurrent expansion of hash nodes occurs.

because it uses a standard read-writer locking scheme. This happens because while the resize is in progress, the next pointers linking atoms in the same bucket are generally incorrect, and dealing with this incorrectness is not a trivial task, which is solved with a standard read-writer lock.

The hardware used was a machine with 4 AMD Opteron(TM) Processor 8425 HE with 6 cores each, 64KiB of L1 cache per core, 512KiB of L2 cache per core and 5MiB of usable shared L3 cache per CPU. It had a total of 128GiB of DDR3 memory. The machine was running the Ubuntu 22.04 operating system with Linux kernel version 5.15.0-69.

4.1 Benchmark

We describe next the benchmark used to evaluate the performance of our implementation. In a nutshell, the benchmark will generate a huge stress over the Prolog's atom table, by inserting an enormous amount of atoms in a multi-threaded fashion. Although it is an artificial benchmark, it is designed to expose all the potential bottlenecks in the atom table, allowing a deeper study about using the LFHT design in YAP. Next, we show the pipeline of predicates used in the benchmark.

```
% compile the generation sequences
:- compile('seq.pl').

% top query call
benchmark(W0, T):-
   atom_dataset(DS),
   % mark the inital time
   statistics(walltime,[InitTime,_]),
   % create and join threads
   findall(Id, (between(1, T,_),
                thread_create(worker(DS, W0),Id)), Ids),
   forall(member(I,Ids), thread_join(I,_)),
   % mark the final time
   statistics(walltime,[EndTime,_]),
   Time is EndTime - InitTime,
   % show the execution time
   write('Time: '), write(Time).
```

Fig. 6 Initial setup and top query call

We begin with Fig. 6 showing the Prolog code for the initial setup of the benchmark and the *benchmark/2* predicate, which is the top predicate to be called. We start by compiling an initial set (file `seq.pl`) of 240,000 different sequences that will be used as base sequences to generate a combination of multiple atoms to be inserted in the atom table. The *benchmark/2* predicate is then used to mark the initial and final times, create and join threads, and to

show the execution time. It receives two arguments, the worker offset *WO*, used to batch a set of sequences from the initial set that will be used to create the combination of atoms, and the total number of threads *T* to be executed. For this benchmark, we used a batch of 2,000 sequences of work to be done.

```
% setup scheduler
:- dynamic qsize/1.
:- mutex_create(qlock).
qsize(0).

% manage the working queue
worker(DS, WO) :-
   mutex_lock(qlock),
   % get work from queue
   retract(qsize(I)),
   (I =< DS -> % thread got work W
      % setup next work
      IL is I + WO, assert(qsize(IL)),
      mutex_unlock(qlock),
      % compute work W
      compute(I, IL),
      % get more work
      worker(DS, WO)
   ; % no more work to be done
      assert(qsize(I)),
      mutex_unlock(qlock)).
```

Fig. 7 The naive parallel scheduler

The second stage of the pipeline is the scheduler. Figure 7 shows the code that implements the naive parallel scheduler used in the benchmark. It uses a dynamic predicate *qsize/1* to mark the number of the next sequence from the initial set that is available to be used for the generation of atoms and a standard lock named *qlock* to synchronize threads when they are getting work. To get work, a thread *T* begins by gaining access to the lock, then it reads the next sequence *I* stored in *qsize/1* and, if there is work to be done, *T* prepares the queue with the next available sequence *IL*, releases the lock and goes to executing work. Otherwise, there is no more work to be done, thus *T* keeps *qsize/1* in the same state, releases the lock, and proceeds to the thread join predicate.

The third and final stage of the pipeline implements the process of generating atoms to be inserted and stored in the atom table. Figure 8 shows both *compute/2* and *combine_atoms/2* predicates. For each batch of work, a thread uses the *compute/2* predicate to get the corresponding sequences from the initial set, and, for each sequence, it calls the *combine_atoms/2* predicate to generate all possible combination of atoms from the sequence. Each generated atom is then automatically inserted by the Prolog system in the atom table.

```
% compute the sequences
compute(I, I) :- !.
compute(I, IL) :-
   atom_seq(I, AS),
   (combine_atoms(AS, _), fail; true),
   I1 is I + 1,
   compute(I1, IL).

% generation of atoms
combine_atoms(AS, R) :-
   atom_concat(A1, A2, AS),
   atom_concat(A2, A1, R).
```

Fig. 8 Generation of the atoms to be inserted in the atom table

4.2 Results

The results shown in the following figures were obtained by taking the mean of 10 benchmark runs. Figure 9 shows the speedup obtained by YAP with the atom table replaced by the LFHT data structure against YAP's original implementation for every combination of 1 to 24 threads. The results show that, on average, we can achieve a minimum speedup of 1.8 with a single thread and a maximum speedup around 3.4 with 23 threads. The speedup for 24 threads is slightly worse than for 23 threads because, as the LFHT version has better CPU utilization, it is more affected by background/operation system processes when all cores are in use.

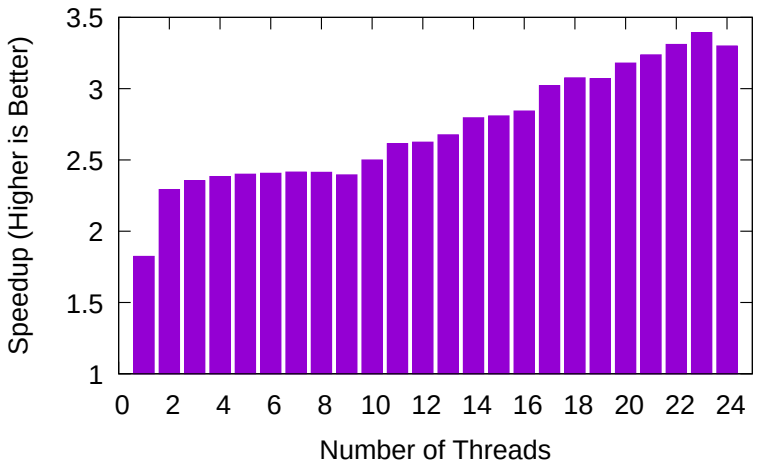

Fig. 9 Speedup of YAP's LFHT version against YAP's original implementation

These results show that we can achieve not only better overall performance, but also much better scalability. In particular, the readers-writer locks present in the original atom table can be a significant bottleneck that the LFHT data structure is able to avoid.

To put the results in perspective, we also compared the YAP results with SWI-Prolog. Figure 10 shows the throughput of sequences that are computed per second in both the YAP (original and LFHT-based atom tables) and SWI-Prolog implementations. As one can observe, the original YAP implementation already provides much better performance and scalability than SWI-Prolog, and the LFHT-based atom table is able to provide a considerable improvement on top of it. For example, with 24 threads, our LFHT-based implementation is able to achieve 24.6 times the throughput of SWI-Prolog.

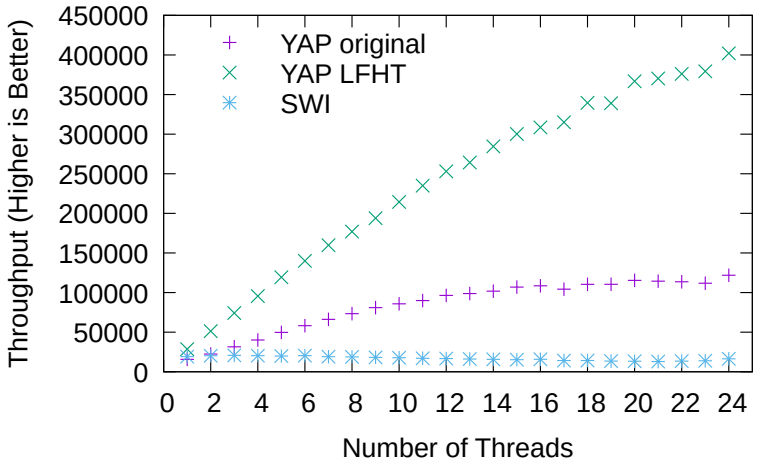

Fig. 10 Throughput for YAP and SWI-Prolog

5 Conclusions and Future Work

We have presented an approach to replace the original atom table implementation in the YAP system with a lock-free hash-based data structure, named LFHT. Our main motivation was to refine the previous atom table design in order to be as effective as possible in the concurrent search and insert operations over the atom table. We discussed the relevant details of the approach and described the main algorithms. We based our discussion on YAP's concurrent atom table data structure, but our approach can be applied to other Prolog systems or to other generic systems that need to use similar concurrent atom tables.

A key design decision in our approach was to adapt the LFHT design to work as a fully standalone `C` application, allowing the hash function to be defined by the user, and implementing a new iterate operator. This facilitated the migration from the old lock-based atom table to the new lock-free atom table, where threads do not block when accessing the data structure. Experimental results showed that our approach can effectively reduce the execution time and scale better than the previous design.

As future work, we plan to test our approach on real world Prolog applications widely-used in the community, such as, the Aleph Machine Learning system [21] and the ClioPatria Semantic Web system[2].

References

1. 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)
2. Areias, M., Rocha, R.: A lock-free hash trie design for concurrent tabled logic programs. International Journal of Parallel Programming **44**(3), 386–406 (2016)
3. Areias, M., Rocha, R.: Towards a Lock-Free, Fixed Size and Persistent Hash Map Design. In: M. Valero, A. Melo (eds.) Proceedings of the International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2017), pp. 145–152. IEEE Computer Society, Campinas, Brazil (2017)
4. Areias, M., Rocha, R.: On the Correctness and Efficiency of a Novel Lock-Free Hash Trie Map Design. Journal of Parallel and Distributed Computing **150**, 184–195 (2021). DOI https://doi.org/10.1016/j.jpdc.2021.01.001
5. Areias, M., Rocha, R.: On the Correctness of a Lock-Free Compression-based Elastic Mechanism for a Hash Trie Design. Computing (2022). DOI https://doi.org/10.1007/s00607-022-01085-2
6. Bagwell, P.: Ideal Hash Trees. Es Grands Champs **1195** (2001)
7. Benton, W.C., Fischer, C.N.: Interactive, scalable, declarative program analysis: from prototype to implementation. In: M. Leuschel, A. Podelski (eds.) Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 14-16, 2007, Wroclaw, Poland, pp. 13–24. ACM (2007)
8. Devitt, S., Roo, J.D., Chen, H.: Desirable features of rule based systems for medical knowledge. In: W3C Workshop on Rule Languages for Interoperability, 27-28 April 2005, Washington, DC, USA. W3C (2005)
9. Fredkin, E.: Trie Memory. Communications of the ACM **3**, 490–499 (1962)
10. Moreno, P., Areias, M., Rocha, R.: A Compression-Based Design for Higher Throughput in a Lock-Free Hash Map. In: M. Malawski, K. Rzadca (eds.) Proceedings of the 26th International European Conference on Parallel and Distributed Computing (Euro-Par 2020), LNCS, pp. 458–473. Springer International Publishing, Warsaw, Poland (2020)
11. Moura, P.: ISO/IEC DTR 13211–5:2007 Prolog Multi-threading Predicates (2008). URL `http://logtalk.org/plstd/threads.pdf`
12. Mungall, C.: Experiences using logic programming in bioinformatics. In: P.M. Hill, D.S. Warren (eds.) Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings, *Lecture Notes in Computer Science*, vol. 5649, pp. 1–21. Springer (2009)
13. Nugues, P.M.: An Introduction to Language Processing with Perl and Prolog: An Outline of Theories, Implementation, and Application with Special Consideration of English, French, and German (Cognitive Technologies). Springer-Verlag New York, Inc., Secaucus, NJ, USA (2006)

[2] `http://cliopatria.swi-prolog.org`

14. Page, D., Srinivasan, A.: Ilp: A short look back and a longer look forward. Journal of Machine Learning Research **4**, 415–430 (2003)
15. P.Moreno, Areias, M., Rocha, R.: On the Implementation of Memory Reclamation Methods in a Lock-Free Hash Trie Design. Journal of Parallel and Distributed Computing **155**, 1–13 (2021). DOI https://doi.org/10.1016/j.jpdc.2021.04.007
16. 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)
17. Santos Costa, V.: On Supporting Parallelism in a Logic Programming System. In: Workshop on Declarative Aspects of Multicore Programming, pp. 77–91 (2008)
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. Santos Costa, V., Sagonas, K., Lopes, R.: Demand-Driven Indexing of Prolog Clauses. In: V. Dahl, I. Niemelä (eds.) Proceedings of the 23rd International Conference on Logic Programming, *Lecture Notes in Computer Science*, vol. 4670, pp. 305–409. Springer (2007)
20. Shalev, O., Shavit, N.: Split-Ordered Lists: Lock-Free Extensible Hash Tables. Journal of the ACM **53**(3), 379–405 (2006)
21. Srinivasan, A.: The Aleph Manual (2004). URL `http://www.cs.ox.ac.uk/activities/machlearn/Aleph`
22. Triplett, J., McKenney, P.E., Walpole, J.: Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In: USENIX Annual Technical Conference, p. 11. USENIX Association (2011)
23. Warren, D.H.D.: An Abstract Prolog Instruction Set. Technical Note 309, SRI International (1983)
24. Wielemaker, J.: Native Preemptive Threads in SWI-Prolog. In: International Conference on Logic Programming, no. 2916 in LNCS, pp. 331–345. Springer (2003)
25. Wielemaker, J., Harris, K.: Lock-free atom garbage collection for multithreaded prolog. Theory and Practice of Logic Programming **16**(5-6), 950–965 (2016)