Lock-Free Extensible Hash Tables Implementation using STM

SHUAICHENG MA, University of Central Florida, USA

This paper discusses the lock-free extensible hash tables implementation, which is introduced by Shalev and Shavit[9]. The lock-free extensible hash table offers concurrent insert, delete, and find operations with an expected $O(1)$ cost. The lock-free hash table is build on lock-free linked list and manipulating the bit of hash key. The paper discusses how to apply the same bitwise manipulation on software transactional memory linked list and compare the performance.

Additional Key Words and Phrases: Lock-Free, Hash Table, STM

1 INTRODUCTION

Lock-free data structures allow concurrent access to the data faster than if the entire structure needs to be locked.A typical extensible hash table is a continuously resized array of buckets, each holding an expected constant number of elements, and thus requiring an expected constant time for insert, delete and find operations[1] The split-ordered list avoids this problem by keeping all nodes in one linked list and splitting the list into buckets,rather than moving nodes from bucket to bucket. This idea is achieved by manipulating bit of hash key, therefore, the same bitwise manipulation can be apply on other sorted linked list as well.

2 DATA STRUCTURE

The lock-free extensible hash table data structure consists of two inter connected substructures: A lock-free linked list[7] of nodes containing the stored items and keys, and an expanding array of pointers into the list.

2.1 Lock-Free Linked Lists

This implementation depends on the lock-free linked list of Michael[7]. The linked list is one of the most basic data structures used in program design, and the lock-free linked list serves as the basis of the split-ordered list. This implementation of the linked list is non-blocking and based on the CAS operation. The objective of the lock-free linked list is maintaining concurrency among the insert, erase and search operations using atomic instructions.Each dynamic node contains a key, data, mark bit, and the pointer to the next node. Since the mark and the next-pointer need to be compared in one atomic instruction, the mark bit and the next-pointer are placed in a single word.

The idea behind the implementation of the list operations is finding a segment of the list including the unmarked node that contains the lowest key value greater than or equal to the input key and its predecessor. The find operation is the basis of the linked list: it returns a reference to the previous pointer and the current and next node pointers. If the find operation finds a marked node while traversing the list, it puts it into the deletion list of the running thread and starts again from the beginning.

Insertion is achieved by pointing the next-pointer of the new node to the following node, and then performing a CAS operation on the next-pointer of the previous node. The atomicity of CAS ensures that the nodes on either side of the inserted node have remained adjacent; if the CAS fails, then insertion starts over again from the beginning of the bucket. Before erasing a node, no new node should be introduced between the node to be erased and its corresponding next node; otherwise the new node would be lost. This is ensured by marking the deleted node before performing the

Author's address: Shuaicheng Ma, University of Central Florida, 4000 Central Florida Blvd, Orlando, FL, 32807, USA, mshuaic@knights.ucf.edu.

CAS that swings the reference from the previous node to point to the next node. Insertion will try again from the start if the previous node is marked for deletion.

The Search operation simply returns the result of the find function.

No thread should delete a node which another thread may be looking at. This is ensured by putting all the nodes currently being accessed by the thread into the hazard pointer list of the thread. Effectively, a thread can point to only three nodes (previous, current and next node).

2.2 Split-Ordered Lists and Hash Tables

Split ordered list is an innovative idea of ordering elements in a list which can be split recursively and forms the basis for a lock-free extensible hash table. The most difficult part in a lock free extensible hash table algorithm is to ensure concurrency and avoid data loss while reordering the elements into the new buckets. It is not possible to remove an item from one bucket and insert it into another in one atomic operation. The idea of the split ordered list algorithm is to order the elements in just one linked list and include marker nodes in the list which delineate the buckets. Since the buckets are split without moving elements, the algorithm is much more efficient and there is no data loss or other related concurrency issues.

The hash table data structure in this implementation consists of two interconnected substructures: the lock free linked list containing the sorted items and keys, and an expanding array of pointers into the list. The former is based on the lock free linked list implementation as detailed in the previous section. The latter is an array pointing to segments; each segment is an array of pointers to the heads of buckets. The head of each bucket is a dummy node with no data.

In order to achieve recursive split ordering this implementation used a simple binary reversal of the hash value of the actual key to get the split-ordered key. I generated the hash value using a standard hash function on the user defined key. (This hash function could be user defined.) The split-ordered keys are of two types, regular keys and dummy keys, for regular nodes and dummy nodes respectively. The regular keys are the reversed hash values of the actual keys with their most significant bit (MSB) turned on. Dummy keys are the reversed hash values of the bucket indices with the MSB turned off. The elements in the linked list are stored in the order of the split-ordered keys.

The algorithm starts with a bucket array size 4 and repeatedly double the size based on a load factor. The load factor is the maximum average number of items per bucket. I used a load factor of 4 which means that the table size will be doubled every time the total count of elements reaches 4 times the number of buckets. The table size is thus always equal to some power 2^i, where $i >= 1$,so that the bucket index is exactly the integer represented by the keyâĂŹs i least significant bits.When an item of key k is inserted,deleted or searched for in the table, a hash function modulo the table size is used, that is, the bucket chosen for item k is $k \mod 2^i = b$, When the size is doubled, the sub-list corresponding to the bucket b will be split into two buckets: some remain in the bucket b, and others, for which $k \mod 2^{i+1} = b + 2^i$, migrate to the bucket $b + 2^i$. Bucket $b + 2^i$ points to the dummy node between first group of items and the second.

A bucket is initialized only when it is accessed for the first time, except for the bucket of index 0 which is initialized to the head pointer of the list. A bucket is initialized by making it point to a dummy node which precedes all elements in that bucket. This simplifies deletion because it ensures there is only one pointer to any data node.

The basic hash table operations insert, erase and find use the corresponding operations of the linked list implementation. Apart from the obvious, all these functions also ensure that the corresponding bucket has been initialized.

Insert() creates a new node, assigns it a regular split-ordered key, and calls the list's insert function with this key. On successful insertion, the node count is incremented and the table size is doubled if the load factor has been exceeded. CAS instructions are used to ensure synchronization. Find() calls the list_search() with the hash value of the actual key

after converting it to a regular split-ordered key. The list_search() ensures that the list is traversed only until a key is greater or equal to the find key (which could be the dummy key of the next bucket). Erase() calls list_delete() with the regular split-ordered key, and decrements the total count atomically on success.

In initialize_bucket(), a dummy node is first inserted into a parent bucket. The parent is the preceding bucket which is closest to the current bucket. The function recursively initializes the buckets until a parent bucket is found. The code ensures that even if a bucket is being initialized by more than one thread, only one of them succeeds; the other threads will use the dummy node that the first thread inserted. Dummy nodes are never deleted; as specified by Shalev and Shavit[9], there is no way to shrink a split-ordered list.

2.3 Linearization Point

Since this hash table is entirely depended on lock-free linked list[7], the linearization points are exactly same as lock-free linked list. Moreover, if one replaces the lock-free linked list to other sorted linked list, the linearization point shall be based on the replaced linked list. The following is the pseudo code of lock-free linked list[7]. If insertion fails, the linearization point is at line 4 after *return 0*; If insertion succeeds, the linearization point is at line 6 when CAS succeeds; If deletion fails, the linearization point is at line 13 after *return 0*; If deletion succeeds, the linearization point is at line 14 when CAS successfully marks the node. The linearization pointer of find operation is at line 34 after *return ckey == key*

```
1  int list_insert(MakrPtrType* head, NodeType *node){
2          key = node->key;
3      while(1){
4          if(list_find(head,key)) return 0;
5          node-><mark,next> = <0,curr>;
6          if(CAS(prev,<0,curr>,<0,node>))
7                  return 1;
8      }
9  }
10 int list_delete(MakrPtrType* head, so_key_t key){
11         while(1){
12         if(!list_find(head_key))
13                 return 0;
14         if(!CAS(&(cur-><mark,next>),<0,next>,<1,next>))
15                 continue;
16         if(CAS(prev,<0,cur>, <0,next>))
17                 delete_node(curr)
18         else list_find(head,key);
19         return 1;
20     }
21 }
22 int list_find(NodeType ** head, so_key_t key) {
23         try_again:
```

```
24      prev = head;
25      <pmark, cur> = *prev;
26      while(1){
27          if(cur = NULL) return 0;
28          <cmark, next> = cur-><mark, next>;
29          ckey = cur->keyl
30          if(*prev != <0,cur>)
31                  goto try_again;
32          if(!cmark) {
33                  if(ckey >= key)
34                  return ckey == key
35              prev = &(cur-><mark, next>);
36          }
37          else{
38                  if(CAS(prev,<0,cur>,<0,next>))
39                  delete_node(cur);
40              else goto try_again;
41          }
42          <pmark, cur> = <cmark, next>;
43      }
44  }
```

2.4 ABA problem

The original paper[9] did not discuss about ABA problem, since ABA was discovered later at same year when the original paper was published. Like all concurrent data structures that use CAS, this algorithm suffers ABA problem as well. ABA can happen at any CAS operation, but it only matter when undesired behavior leads to incorrect result. For this implementation,

Assuming the linked list looks like A->B->C. Thread0 is trying to delete B from list. It will perform CAS(A->next,B,C). Before this CAS happens, Thread1 interrupts and removes B and C. Then Thread1 adds B back to the list. Since some programing languages lack garbage collector(e.g. C/C++), node C is the memory and B's next pointer still point to C. Now Thread0 revivals and perform CAS(A->next, B, C). This CAS will succeed, since B is back to the list. Now A->next is pointing at an useless Node C and hence break the entire data structure.

To solve this ABA problem, I used a vision counter[3] to maintain reference counting for each element. Because of lacking the support of CAS2, I only use CAS. This solution only works at current architecture of AMD64 architecture and it is compiler depended(only tested for gcc and msvc). Since current AMD64 memory address implementation only uses lower 48bits, it leaves upper 16bits unused. This 16bits can be used as a counter, assuming the 16bits counter(short integer) is enough.

2.5 Memory Reclamation

The original paper didn't discuss any memory reclamation. The author of paper, Shavit, later states in his *The art of multiprocessor programming*[5] book "that it is often more efficient for a class to do its own memory management, particularly if it creates and releases many small objects.". Since I implement this data structure in C++, I have to do garbage collection anyway. Moreover, the garbage collection must be implemented in a lock-free manner.

I use thread local freelist[8] to manage memory reclamation. Every time, a new node is inserted to the hash table, it will check the thread local freelist first. If a old deleted node is in freelist, it will be reused. Simply it change old value of node to new value. If freelist is empty, new memory will be allocated. When a thread preforms physical deletion, in stead of calling *delete*, it simply appends deleted node to thread local freelist. Since this implementation use only thread local variable to store and load nodes, threads will not compete with each other and cause conflicts, therefore, it is lock-free and solves lacking of garbage collection of C++. One particular important thing to notice is ABA problem Can happen during this memory reclamation(check section2.4).

3 SOFTWARE TRANSACTIONAL MEMORY

software transactional memory (STM) is a concurrency control mechanism analogous to database transactions for controlling access to shared memory in concurrent computing.These reads and writes logically occur at a single instant in time; intermediate states are not visible to other (successful) transactions. The idea of providing hardware support for transactions originated in a 1986 paper by Tom Knight.[6] The idea was popularized by Maurice Herlihy and J. Eliot B. Moss.[2] In 1995 Nir Shavit and Dan Touitou extended this idea to software-only transactional memory (STM).[4]

3.1 Translate to STM implementation

3.1.1 Version A. Since this hash table implementation is heavily based on lock-free linked list implementation and all actual hashing relies on bitwise manipulation, the translation of STM is Simply replacing ll read operations to TM_READ, and all write operations to TM_WRITE in lock-free linked list. However, the original lock-free design is optimized in lock-free manner, so it has multiple while loops to protect CAS failure which downgrades the performance of STM version implementation.

3.1.2 Version B. Design STM list and apply the bitwise manipulation to the STM list. The Design of STM list strict follows STM design principle, which is basically a sequential version of linked list with TM_READ and TM_WRITE.

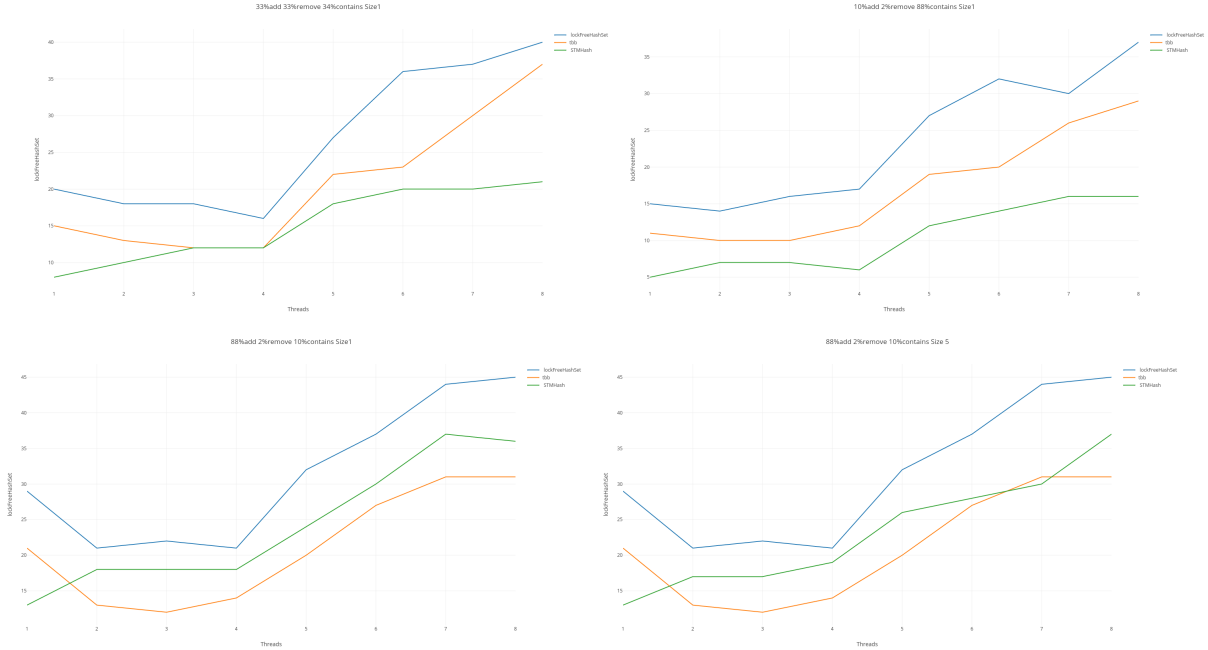
3.1.3 Performance Difference. When transaction size is 1, Version A has much less aborted operations then version B because of the while loops protection and the lock-free implementation avoids many abortions. When transaction size gets larger, the number of aborted operations of two versions gets closer.

3.2 Performance Evaluation

Setup each thread performs 50000 operations and measures execution time from 1 thread to 8 threads. I take average of 5 runs and uses three different profiles(10%add 2%remove 88%contains), (33%add 33%remove 34%contains), (88%add 2%remove 10%contains). The transaction size is either 1, 5, 10

All test runs on Ubuntu 16.04 in VMware Workstation 12 Pro

Memory: 4GB Processors: 1 cpu, 4 cores



4 COMPARISON WITH OTHER LOCK-FREE HASH TABLES

A SOURCE CODE

https://github.com/mshuaic/LockFreeExtensibleHashTable

REFERENCES

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2001. Introduction to algorithms second edition. (2001).

[2] David E Culler, Jaswinder Pal Singh, and Anoop Gupta. 1999. *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing.

[3] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. 2010. Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on*. IEEE, 185–192.

[4] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. 2003. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*. ACM, 92–101.

[5] Maurice Herlihy and Nir Shavit. 2011. *The art of multiprocessor programming*. Morgan Kaufmann.

[6] Tom Knight. 1986. An Architecture for Mostly Functional Languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming (LFP '86)*. ACM, New York, NY, USA, 105–112. https://doi.org/10.1145/319838.319854

[7] Maged M. Michael. 2002. High Performance Dynamic Lock-free Hash Tables and List-based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '02)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/564870.564881

[8] Maged M Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 73–82.

[9] Ori Shalev and Nir Shavit. 2006. Split-ordered Lists: Lock-free Extensible Hash Tables. *J. ACM* 53, 3 (May 2006), 379–405. https://doi.org/10.1145/1147954.1147958