A Concurrent Skip List Implementation with RTM and HLE

Fan Gao

May 14, 2014

Semester Performed: Spring, 2014
Instructor: Maurice Herlihy

1 Background

The main idea of my project is to implement a skip list described in [1] using Intel TSX(Transactional Synchronization Extensions, it includes RTM–Restricted Transactional Memory and HLE–Hardware Lock Elision) which is supported by the newly released Haswell. I also do some tests on my implementation to gather information about TSX performance. I would like to divide the following reports into 3 parts:

(1) Skip List Introduction and Implementation
(2) RTM Introduction and Performance
(3) HLE Introduction and Performance

2 Skip List Introduction & Implementation

Skip List[1] is a some kind of linked list whose node is sorted according to its value. In a skip list, each node is assigned to a random height, and at each height, it has one successor and one predecessor. In the sense that it is some kind of linked list, Every node has a list of pointers to its successors whose number equals to its height. The Figure 1 shows a skip list.

The Figure 1 shows a simple skip list whose maximum height is 5. The number below each node is its belonging key and the number of layers in each node is their random height. This skip list has left and right sentinel nodes whose value is negative maximum and positive maximum respectively. This design of sentinel nodes can avoid a lot of edge case in real implementation.

My implementation of the skip list follows [2]. As the authors mentioned in [2], their algorithms has two significant features. First of all, it is an optimistic design which means that the finding method will traverse the whole list while

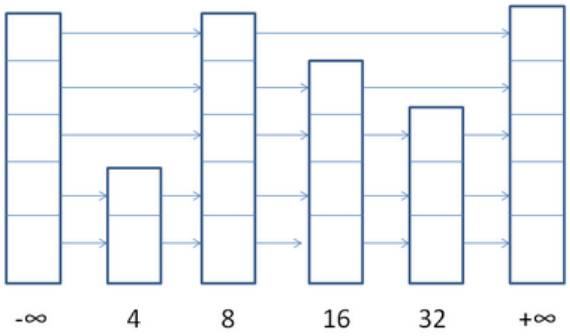

Figure 1: A simple Skiplist.

avoid using any locks. When a target node is found during the traverse, it will first lock its predecessors(which are the only nodes that matter) and verify if everything(mainly concerns about the target node, its predecessors and successors) keeps unchanged. If so, it will continue to the next move, otherwise, it will abort or retry depending on different situation. Secondly, there are two phases related to removing a node. A node is logically removed if it is marked and this is the linearer point of removal. A node is physically removed if it has been unlinked from all its predecessors.

This algorithm in [2] has two main advantage over others like the ConcurrentSkipListMap in Java SE which is the best known concurrent skip list. First of all, this implementation is simple to both implement and prove it correctness. Secondly, it can preserve the structure and feature of a skip list all the time during any operation.

I would like to first explain my definition of the node in skip list.

```
class node{
private:
        int key ;
        int top_layer;
        node ** nexts;
        volatile int marked;
        bool fully_linked;
        hle_mutex* private_mutex;

public:
        node(int k , int layer ){
                key = k ;
                top_layer = layer -1;
```

```
            nexts = new node*[layer];
            marked = 0;
            fully_linked = false;
            private_mutex = new hle_mutex();
        }

        friend class skiplist;
};
```

In the definition, top_layer is the assigned random height, nexts is an array of pointers to its successors, marked is a flag identifying logically remove, fully_linked shows whether the node have been linked at all levels, and private_mutex is the locks associated with each node. hlemutex is my own implementation of a mutex using Intel HLE. It provides normal lock/release and hle_lock/hle_release. I will explain its detailed implementation in HLE part.

The skiplist class has several important functions that I would like to introduce.

a. int find_node(int v , node* preds[] , node* succs[])
   This function traverse the whole list to find if a node with the target value v exists in the list. If such node exists, it will return the target nodes height and store all its successors and predecessors in the corresponding array succs and preds. If no such node exists, the function will simply return -1.

b. int add (int v)
   As its name suggested, this function aims to add a new node with value v to the skip list. It will first call find_node function to check whether a node with the same value exists in the skip list. If so, it will return a predefined showing NODE_ALREADY_EXIST. If no such node exist, it will first try to add the node in a transactional manner using RTM interface. I would talk about the detailed RTM things in the following part. If such add succeed(it will succeed if there is no concurrent interleaving thread), the function will return RTM_ADD_SUCCESS. If owing to concurrent contention, transaction failed, it will go to fallback path which will employ hle_mutex or normal_mutex. Basically, it will try to acquire all locks of the predecessors before if actually link the new node to the list. After that, it will mark the new nodes fully_linked flag to finish operation and return MUTEX_ADD_SUCCESS.

c. int remove (int v)
   This function tries to remove a node with target value. As add function, it will first call find_node function to determine whether a node with the target value exists in the skip list. The function will only continue when such node exists. The working flow of remove function is pretty the same

at add. However, there is one major difference that we need to lock the node_to_delete as well. After acquiring the lock, I first marked the node as logically removed. Then try to do the unlink on each level.

During the operation with locks, one point that I want to point out is that we need to be careful about releasing all locks that we previously acquired. We should always remember to release in every different code branch. This mistake costed me a lot of time to debug.

3 RTM Introduction & Performance

RTM stands for Restricted Transactional Memory. And it is a set of new instructions that is comprised of the XBEGIN, XEND, and XABORT instructions. It give programmers the freedom to define a set of codes to be conducted as a transaction although hardware can not guarantee the transactional features. More specifically, Intex RTM(as well as HLE which will be mentioned in the following parts) make it possible for the processor to determine dynamically whether a lock acquire/release is really need when performing some critical section instructions.

Since RTM transactions are best effort, it also requires the programmer to provide an alternate code path for when the transactional execution is not successful.

```
#include <immintrin.h>

if (_xbegin() == _XBEGIN_START) {
/* transaction */
    _xend();
}
else {
    /* fallback path -- take lock */
}
```

Basically, the code will work like this. One thread will first try the code between _xbegin() and _xend(). If there are interference with all other threads, it will successfully come to the _xend() clause, then the whole transaction is finished successfully.

However, if there are some other threads performing some interfer conductions like writing to some shared memory in the critical section, all the transactions of interfering threads will abort, and all changes will be reset. And the thread will return to the _xbegin() line with the return value to be some other values other than _XBEGIN_START. This value may be used to identified reason for the abortion. For example, we can intentionally abort one transaction in the middle and the behavior will be exactly the same except that the return value

of _xbegin() will be different.

If the transaction is aborted, the code will go to the else clause which is usually called as fallback path. The fallback path can be as simple as a lock protected critical section. This is because RTM does not guarantee the transaction to be succeed, programmer need to make the code make progress eventually.

I added RTM in the original skiplist and use the C++11 mutex as the fallback path. The test mainly concerns about the percentage of successful transactions among all conductions. And the following table shows test results.

	4	8	16
1000 nodes	13.50%	25.20%	17.90%
10000 nodes	5.40%	6.90%	6.80%

Figure 2: Mutex percentage of RTM perfromance.

The first line of Figure 2 shows the number of threads in each test and the first column shows different number of nodes. The table is about percentage of mutex operation(namely percentage of fallback times) of among all operations. I will discuss the table in 2 aspects.

First of all, we can see regardless the number of nodes, 4 threads has the best performance while 16 threaded test is a little bit better than 8 threaded one. This is mainly because my test machine is has 4 cores so that 4 threads make fully use of CPUs when there is little other running programs. For 8 threads or 16 threads, they are the so-called hyper-threaded, the overhead is relatively high.

Secondly, we can see that, regardless the number of threads, 10000 nodes outperform 1000 nodes test. This is because, as the size of memory increase, the contention between different threads get lower and lower and lower. The chance of two threads conducting on the same memory location is getting lower and lower. So more and more transaction can succeed.

4 HLE Introduction & Performance

HLE is short for Hardware Lock Elision, as Intel described, HLE provides a legacy compatible set interface for programmer to do transactional execution. With HLE, if multiple threads execute critical sections protected by the same lock but they do not perform any conflicting operations on each others data, then the threads can execute concurrently and without serialization. Even though

the software uses lock acquisition operations on a common lock, the hardware recognizes this, elides the lock, and executes the critical sections on the two threads without requiring any communication through the lock if such communication was dynamically unnecessary.[3]

For programmers, it provides 2 new instruction prefix hints: XACQUIRE and XRELEASE. To use HLE, programmer need to enable it on Haswell machine which is supported on GCC 4.8+ using -mhle. I created a simple spin lock using both HLE and common spin pattern. The code for this simple lock is as follows.

```
class hle_mutex {

    private:
        volatile int mutex_val;

    public:
        hle_mutex() {
            mutex_val = 0 ;
        }

        bool hle_lock() {
            while (__atomic_exchange_n( &mutex_val, 1,
            __ATOMIC_ACQUIRE| __ATOMIC_HLE_ACQUIRE ))
                _mm_pause();
                return true;
            }

        bool hle_release() {
            __atomic_store_n( &mutex_val, 0,
            __ATOMIC_RELEASE|__ATOMIC_HLE_RELEASE);
            return true;
        }

        bool normal_lock(){
            while (__atomic_exchange_n( &mutex_val, 1, __ATOMIC_SEQ_CST))
                _mm_pause();
                return true;
        }

        bool normal_release(){
            __atomic_store_n( &mutex_val, 0, __ATOMIC_SEQ_CST);
            return true;
        }

        bool is_locked() {
```

```
            return mutex_val == 0 ? false : true ;
        }
};
```

I replaced the C++11 mutex that is originally used in the skiplist with this new implementation. I compared the performance of HLE, RTM and C++11 mutex(use Mutex) in different tests where 4, 8 and 16 threads are used to examine multi-thread performance. And the results are shown in Figure 3.

4 threads	RTM	HLE	MUTEX
100 nodes	100	100	110
1000 nodes	110	120	120
10000 nodes	600	610	620
8 threads	RTM	HLE	MUTEX
100 nodes	220	200	200
1000 nodes	220	220	210
10000 nodes	660	660	660
16 threads	RTM	HLE	MUTEX
100 nodes	20328	84036	81252
1000 nodes	9441	22136	20192
10000 nodes	2242	3874	3752

Figure 3: Running time of HLE, RTM and MUTEX

Figure 3 shows running time of the skiplist when conducting 100000 add or remove operations using RTM, HLE or Mutex with 4, 8 and 16 threads respectively. There are three kinds of different situations when the skiplist has 100, 1000 or 10000 nodes. This kind of design is aimed to create different competition situation. Specifically, 100 nodes will create the most competitive sitution. And 10000 is the least competitive one. I will explain the results in 3 different way.

As we can see, the running time of different lock types is almost the same for 4 and 8 threads. This is mainly because our running machine has 4 cores. So for threads, it is amost assign one thread to one core when there is not much load. And for 8 threads, it employes hyper-threading technology thus improving performances. Another fact is that, the test actually runs slower when it comes to 10000 nodes. I think this is because the find method will take longer time to locate a value in the skip list. And every add or remove action calls find method at least once.

First of all, compare RTM with Mutex. We can see that as the memory competition become lower and lower, the running time for both RTM and Mutex

decrease. But the speedup of using RTM is also decreasing from over 4 times to about 1.5 times. This can be seen from my implementation of the mutex. The mutex simply uses CompareAndSet instruction which is supported by the hardware and the failed thread will pause for a short time trying to reduce competition. So when the competition is high, most of the time is spent on thread wakeup and some other overhead. In this case, if one thread is fortunate enough to finish a transaction using RTM, it can gain much more speedup that those who eventually finish operations after several failed acquire. The overall cumulative speedup is high in this case. When there is less competition, a thread can acquire the lock at the first time and finish the critical section right away. In this case, the difference between a transaction and a code section with acquiring and releasing lock has became less significant.

Secondly, compare HLE with RTM and Mutex. In the manual[3], Intel describes Haswells HLE like this:

```
"If multiple threads execute critical sections
protected by the same lock but they do not perform
any conflicting operations on eachother's data,
then the threads can execute concurrently and
without serialization. Even though the software
uses lock acquisition operations on a common lock,
the hardware is allowed to recognize this, elide
the lock, and execute the critical sections on
the two threads without requiring any communication
through the lock if such communication was dynamically
unnecessary."
```

This statement is interesting because it leaves out single thread which is important in our skiplist. First, for our skiplist, if multiple threads are waiting on the same lock, they are sure to modify the same value thus to be interfered with each other. So HLE cannot help with this case. What if only one thread is waiting on the lock? It seems that when it came to single thread, HLE performs even worse than common Mutex. To test this conclusion, I write a simple test. The HLE can achieve a speedup of 8 using 16 threads but it is 1.5 times slower than common Mutex when there is only one thread conducting the same code. In Marc Brooker's Blog[4], he found the similar results and stated some detailed findings that may help to explain this. Unfortunately, he can only blame this to Intels architecture.

5 Reference

[1]PUGH, W. Skip lists: A probabilistic alternative to balanced trees. Communications of the ACM 33, 6 (June 1990), 668-676

[2]M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A provably correct scalable concurrent skip list. In OPODIS '06: Proceedings of the 10th International Conference On Principles Of Distributed Systems, 2006.
[3]Intel. Intel Architecture Instruction Set Extensions Programming Reference. February 2012, 8-1 - 8-23
[4]Marc Brooker's Blog. http://brooker.co.za/blog/