CS4021/4521 Tutorial 3

Q1. Implement and compare the performance of (i) a binary search tree (BST) protected by a lock (ii) a *lockless* BST using HLE and (iii) a *lockless* BST using RTM.

The test framework should randomly add and remove random keys to a BST using from 1 to 2*NCPU threads. In each case, test key ranges 16 [0 … 15], 256, 4096, 65536, 1048576.

Hand in a short report describing what you have done, your results including graphs (max four A4 pages) and a code listing.

Consult CS4021/4521 web page for deadline.

Tips gained from the Class of 2014/15

Step one: make sure that the code for a BST is working correctly for a single thread and without a lock. As the key range increases, the BST gets larger and consequently the ops/sec should decrease. Your results should show this basic characteristic.

The standard VC++ rand() function only generates a 15 bit pseudo random number (0 .. 32767). For key ranges greater than 32768, rand() is unsuitable. Alternative rand functions are provided in helper.cpp (for example UINT rand(UINT &r)). Note also that the VC++ rand() function is thread safe, but clearly the ones in helper.cpp are not. This means that the variable used to hold the by address parameter r must be thread local (for example declared in worker and not globally).

A large tree will take time to fill, consequently the ops/sec may be higher than expected because the most of the operations will occur on a smaller tree than that in steady state. It can take seconds for the BST to become *full* and the ops/sec reach a steady state. One way around this is to prefill the tree, with, for example, odd integers.

Step two: add a testAndTestAndSet lock and multiple threads and then check that the results have the correct characteristics (1) as the key range increases the ops/sec should decrease and (2) as the number of threads increase, the ops/sec should remain more or less the same or decrease due the cost of sharing data between threads. This latter effect should be more evident on smaller BSTs.

Step three: convert the testAndTestAndSet to an HLE lock. The transaction should be kept as "small" as possible. Larger read and write sets increase the probability of conflict and hence transaction aborts. In lectures, the size of the read and write sets when updating a BST were discussed. It should be apparent that it is unwise to call new and delete (malloc and free) within a transaction. At one level, we have little knowledge of what memory updates these routine perform, but it is sensible to assume they read and write many memory locations and consequently increase the probability of transaction aborts. Keep the calls to new a delete (malloc and free) outside of the transaction.

For parallel execution, it is important that items in the shared data structures are declared volatile. If not, changes may not be observed. Typically this means that the key, left and right in Node and root in BST are declared volatile.

```
class Node {
public:
    INT64 volatile key;          // NB: volatile
    Node* volatile left;         // NB: volatile
    Node* volatile right;        // NB: volatile
    …
};
```

As VS2012 doesn't recognise _InterlockedExchange_HLEAcquire(&lock, 1), an alternative is required. If _InterlockedExchangeAdd_HLEAcquire(&lock, 1) is used, note that this can keep incrementing the lock so the test for lock == 1 should be changed as follows:

```
while(_InterlockedExchangeAdd_HLEAcquire(&lock, 1))
```

```
while(lock >= 1)      // change == to >=
    _mm_pause();
```

Remember, for best performance, to store lock in its own cache line.

Step four: add the code, as outlined in the lectures, to implement RTM. Experiment with wait and the number of retries (8 works quite well).