Tutorial 3

• implement and compare the performance of

  1) a binary search tree (BST) protected by a lock

  1) a lockless BST using HLE

  2) a lockless BST using RTM

• test framework should

  ▪ randomly add or remove random keys

  ▪ test 1 to 2*NCPU threads

  ▪ test key ranges 16 [0..15], 256, 4096, 65536, 1048576

  ▪ if key range 0..N-1, BST will contain, on average, N/2 nodes [keys]
BST Revision

- add(key)
  - always adds to a leaf node
  - add(22)
  - add(40)

- nodes can be added concurrently if they do not conflict with each other
Binary Search Tree Revision...

- remove(key)
  - depends on number of children
    - NO children
    - ONE child
    - TWO children
  - NO children
    - remove(45)
    - simply remove node
  - ONE child
    - remove(25)
    - parent points to child
Binary Search Tree Revision...

• remove(key)

• TWO children
  ▪ remove (20)
  ▪ find node 20
  ▪ find smallest key in its right sub tree [22]
  ▪ overwrite key 20 with 22
  ▪ remove old node 22 [will have ONE or zero children]
Node

class Node {

public:

    INT64 volatile key;
    Node* volatile left;
    Node* volatile right;

    Node() {key = 0; right = left = NULL;} // default constructor

};

• key, left and right declared volatile [needed for multi-threaded version]
BST

```cpp
class BST {

public:

    Node* volatile root;   // root of BST, initially NULL

    BST();                 // constructor
    int contains(INT64 key); // return 1 if key in tree
    int add(Node *nn);     // add node to tree
    Node* remove(INT64 key); // remove key from tree

};
```

- add - if node already in BST do nothing
- remove – if node not in BST do nothing
- add lock/HLE/RTM code in contains, add and remove
- add(Node*) - node allocated outside of function to reduce size of transaction read and write sets
BST...

• recursive or iterative code?

• which will minimise the read and write sets of a transaction?

• recursion could lead to more stack frames [larger read and write sets], but a compiler could easily use tail recursion optimisation to convert recursive into iterative code

• play safe, use iterative code
BST::add ()

```cpp
int BST::add (Node *n) {
    Node **pp = &root;
    Node *p = root;
    while (p) {
        if (n->key < p->key) {
            pp = &p->left;
        } else if (n->key > p->key) {
            pp = &p->right;
        } else {
            return 0;
        }
        p = *pp;
    }
    *pp = n;
    return 1;
}
```
BST::remove()

Node* BST::remove(INT64 key) {
    Node **pp = &root;
    Node *p = root;
    while (p) {
        if (key < p->key) {
            pp = &p->left;
        } else if (key > p->key) {
            pp = &p->right;
        } else {
            break;
        }
        p = *pp;
    }
    if (p == NULL)
        return NULL;
BST::remove()...

```cpp
if (p->left == NULL && p->right == NULL) {
    *pp = NULL; // NO children
} else if (p->left == NULL) {
    *pp = p->right; // ONE child
} else if (p->right == NULL) {
    *pp = p->left; // ONE child
} else {
    Node *r = p->right;
    Node **ppr = &p->right;
    while (r->left) {
        ppr = &r->left;
        r = r->left;
    }
    p->key = r->key; // could move...
    p = r; // node instead
    *ppr = r->right; // return removed node
}
return p;
```

© 2015 jones@scss.tcd.ie 9-Dec-16
School of Computer Science and Statistics, Trinity College Dublin
Tutorial 3

• step 1
  • start with existing framework
  • add code for BST and test with single thread

• generate a random number
  ▪ select one bit to determine if an add or remove operation
  ▪ select bits for key
  ▪ perform operation
  ▪ make sure random number generator generates enough bits [default VC++ rand() only generates pseudo random numbers in range 0..32767]
  ▪ make sure random generator not protected by a lock so it can be executed concurrently [otherwise the random generator will become a bottleneck]

• make sure ops/sec decrease as size of tree increases [basic characteristic]
• make sure number of operations carried out fills BST and more [could add prefill code]
• CHECK that BST is valid at end of test and that ALL allocated Nodes are accounted for
Tutorial 3...

• step 2 - protect BST with a testAndTestAndSet lock
  ▪ for best performance store lock in its own cache line
  ▪ what characteristics should the ops/sec results have?

• step 3 - convert testAndTestAndSet into an HLE lock
  ▪ use a PC that supports TSX (malbec)
  ▪ what characteristics should the ops/sec results have?
  ▪ need a concurrent memory allocator
  ▪ create per thread reuseQ
  ▪ place removed nodes on per thread reuseQ
  ▪ when allocating a node, take node from per thread reuseQ if NOT empty otherwise call malloc()
  ▪ alternatively could try tcmalloc
Tutorial 3...

• step 4 – convert to use RTM
  ▪ provide a non-transactional path as outlined in notes
  ▪ use a PC that supports TSX (malbec)
  ▪ experiment with back off and number of retries
  ▪ what ops/sec characteristic as expected?