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]
Transaction read and write set

• depends if key, left and right in same cache line
• one node per cache line? probably NOT as the performance gained by using less memory (particular for large BSTs) will probably be greater

• read set
  • all nodes traversed
  • key, left and right probably in same cache line

• write set
  • probably one or two cache lines depending on type of remove being performed
Node

```cpp
class Node {

public:

    INT64 volatile key;  // multi-threaded algorithm
    Node* volatile left;  // needs members to be
    Node* volatile right; // declared volatile

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

};
```

- key, left and right declared must be declared volatile for multi-threaded version
BST

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, return 1 if node added or 0 otherwise
    Node* remove(INT64 key); // remove key from tree, return deleted Node or NULL

};

• add - if node already in BST do nothing (count as one operation either way)
• add - node allocated outside of function to reduce size of transaction read and write sets

• remove - if node not in BST do nothing (count as one operation either way)

• add lock/HLE/RTM code to contains(), add() and remove()
BST...

• recursive or iterative code?

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

• recursion may 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 () -- iterative implementation

```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;
}
```

// add Node n to tree

// find position in tree

// go left

// go right

// return 0 if key already in tree

// next Node

// add Node n
BST::remove() -- iterative implementation

Node* BST::remove(INT64 key) {
    Node **pp = &root;
    Node *p = root;
    while (p) {
        // find key
        if (key < p->key) {
            pp = &p->left;
            // go left
        } else if (key > p->key) {
            pp = &p->right;
            // go right
        } else {
            break;
        }
    }
    p = *pp;
    // next Node

    if (p == NULL) // NOT found
        return NULL;
}
BST::remove()...

if (p->left == NULL && p->right == NULL) {
    *pp = NULL;
} else if (p->left == NULL) {
    *pp = p->right;
} else if (p->right == NULL) {
    *pp = p->left;
} else {
    Node *r = p->right;
    Node **ppr = &p->right;
    while (r->left) {
        ppr = &r->left;
        r = r->left;
    }
    p->key = r->key;
    p = r;
    *ppr = r->right;
}
return p;
Tutorial 3

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

- generate a random number
  - select one bit from random number 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]
  - alternatives random number generators in helper.cpp

- make sure ops/sec decrease as size of tree increases [basic BST characteristic]
- make sure number of operations carried out fills BST and more [could prefill tree initially]
- ops/sec from a partially filled BST will not be the same as for a "full" tree
- CHECK that BST is valid at end of test and that ALL allocated Nodes are accounted for
Tutorial 3...

• step 2 – multiple threads + protect BST with a testAndTestAndSet lock
  ▪ for best performance store lock in its own cache line
  ▪ what ops/sec results should you expect?

• step 3 - convert testAndTestAndSet into an HLE lock
  ▪ must use a PC that supports TSX (eg malbec)
    • see hleSupported() and rtmSupported() in helper.cpp
  ▪ must make sure that there is nothing that inhibits any possible parallelism
  ▪ FALSE SHARING, random generator (probably NO) and malloc/free/new/delete (probably YES)
  ▪ need some sort of 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() outside of transaction
  ▪ alternatively could try tcmalloc or equivalent
Tutorial 3...

- step 4 – convert to use RTM
  - provide a non-transactional path as outlined in the notes
  - must use a PC that supports TSX (malbec)
  - experiment with back off and number of retries
  - what ops/sec results should you expect for steps 3 and 4?
Tutorial 3...

- deadline?
- start early, get something working, think and make
- no lectures in last week of term
- 4021/4521 Exam 4-Jan-18 @ 09:30 in Goldsmith Hall
- hand in at SCSS reception BEFORE Wed 20-Dec-17 @ 4pm