The aim of this assessment was to perform a comparison between the performance of:

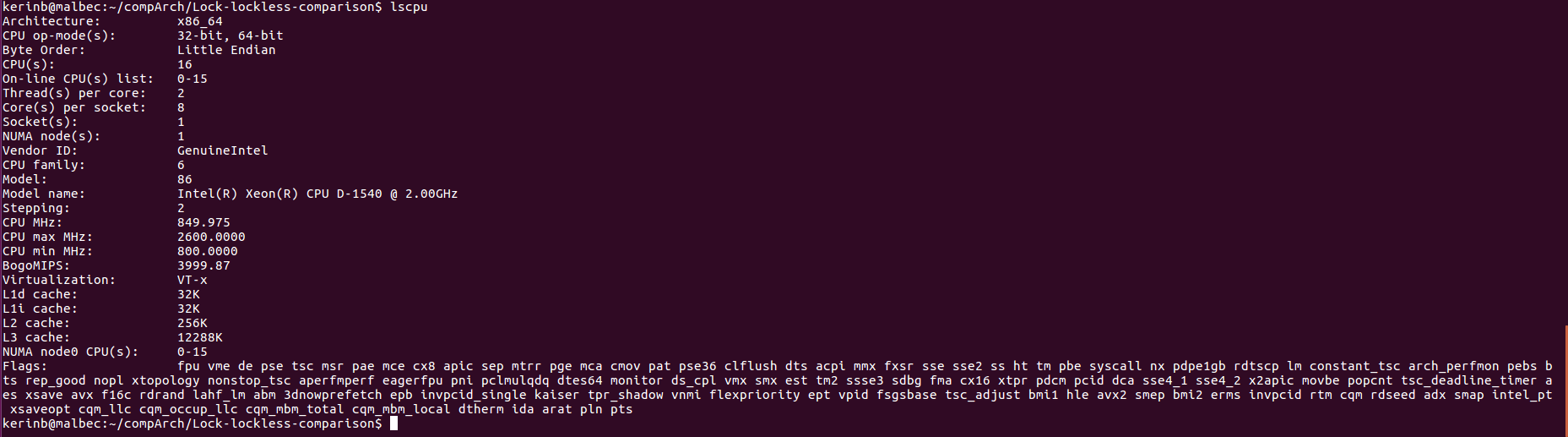
* A Test-And-Test-And-Set lock
* Hardware-Lock-Elision (HLE)
* Restricted Transactional Memory (RTM)

Which performs concurrent updates to a Binary Search Tree (BST) implemented in C.

The code, screenshots and report can all be viewed on my github account: <https://github.com/kerinb/CS4521---Advanced-Computer-Architecture/tree/master/Lock-lockless-comparison>

Hardware Parameters:

In order to run this project, it is required to run on a system that provides the support for HLE and RTM - As I am not in possession of such a machine, I used a machine provided by the SCSS in TCD: namely Malbec - whose CPU specifications are shown below:

* From the above image, the number of CPUs present on the Malbec machine is 16 – hence, the update mechanisms were tested on 5 different key ranges, for up to 2 \* 16 = 32 threads.
* The number of operations measured for each of the tests run was equal to the number of operations attempted, and not necessarily associated with a successful insertion/deletion of a node in the binary search tree. This is because in one of these update operations, the effort really goes into searching the tree to find the insertion/deletion position: the actual update itself is trivial in comparison.

Random Key Generation:

The three update mechanisms used (TATAS lock, HLE and RTM) were tested for the following key ranges:

1. 16 -> [0 - 15]
2. 256 -> [0 - 255]
3. 4,096 -> [0 – 4095]
4. 65,536 -> [0 – 65535]
5. 1,048,576 -> [0-1048575]

The Random seed was generated using the c intrinsic function *std::\_rdrand64\_step.* This seed then underwent some binary operations - namely bit left and right shifts as well as OR and multiplication:

r ^= r >> 12; // a

r ^= r << 25; // b

r ^= r >> 27; // c

r \* 2685821657736338717LL;

Optimisation

In order to ensure a high level of accurate measurement for the performance of each mechanism implemented in performing the concurrent updates, some optimisations were required to be applied so that any work that wasn’t directly involved in the update of the BST itself was kept outside the body of the operation.

The following optimisation measures were implemented in the system:

* The BST lock used by the TATAS and HLE algorithms was declared to be aligned on a 64-byte cache boundary:   
   *ALIGN(64) volatile long lock;*
* The BST was pre-filled to half-capacity before running any updates on the structure (i.e. adding and removing nodes in the BST). This was necessary to ensure that the BST was in the approximate steady-state before performing any operations using the update mechanism (lock, HLE, RTM) on it. There is no point taking performance measurements when the tree is filling up, since it won’t reflect the steady-state behaviour of updates to the BST.
* Because of the issues that arise from concurrently calling the malloc() and free() methods to allocate/deallocate memory for nodes, an alternative method of memory management was also implemented. This involved use of a variable RECYCLENODES, with which nodes are added to a tmp list when they are removed from the tree and from which nodes are reused (if any are available) when they are being added to the tree. This means that malloc() doesn’t need to be called to allocate a node for instance, unless there are no spare nodes. All of these calls are done outside of the update operation itself.
* The read and write sets for the transaction are reduced by implementing iterative rather than recursive methods for the BST (reducing the number of stack frames generated).

Implementations:

1. No Threading and TATAS:
   1. This was pre-implemented by Prof. Jeremy Jones in the source code provided on his web site. Hence it was not re-implemented
2. HLE
   1. When the global variable METHOD == 2, the system is to update the BST structure using HLE TATAS implementation. The code for obtaining the *‘lock’* is attached below, and as can be seen from the code, it was the pessimistic TATAS that was implemented - meaning it assumes it won’t receive the lock first time trying. The code for freeing the *‘lock’* is also attached.
3. RTM
   1. When the global variable METHOD == 3, the system is to update the BST structure using RTM and if a maximum amount of aborts occur, the system will use the HLE TATAS implementation instead to obtain a lock and try make progress. The code for obtaining the *‘lock’* is attached below, and as can be seen from the code, it uses intrinsic function such as \_xbegin, \_xabort and \_xend to start, stop and abort the transaction. The lock implemented in the event of maxAborts being reached, is the pessimistic HLE TATAS lock.

Results:

No Lock - 1 Thread

* The results here are that:
  + As the key range increases, The number of operations and number of operations per second decreased: For example image M1 attached shows that for maxKey = 16, ops/sec = 22’450’274, while for maxKey = 1’048’576, ops/sec = 1’743’576. This is due to the fact that for a larger tree, there are more nodes to search to find the one on which we want to operate – hence, each operation takes longer to perform.

Test And Test And Set (TATAS) Lock

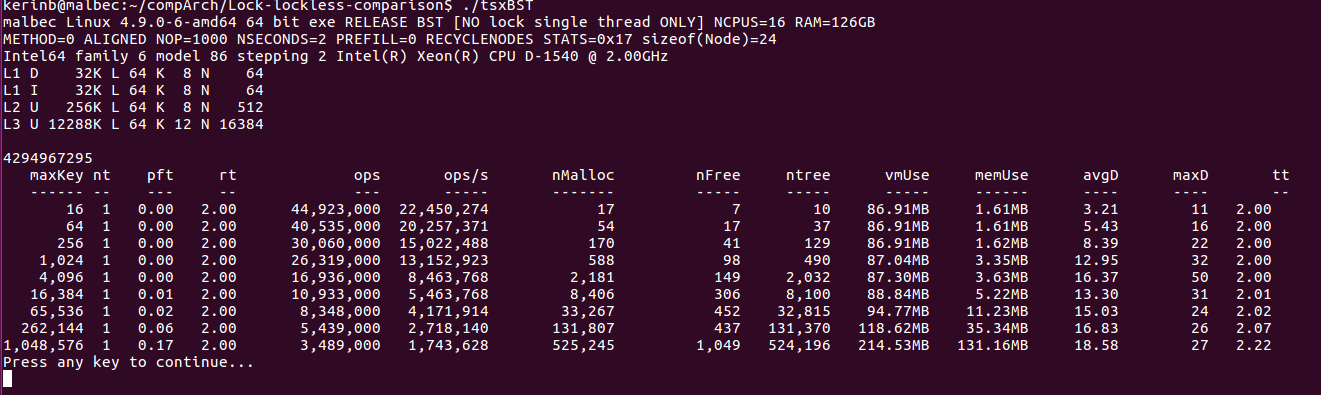
* The results here are that:
  + As the key range increases, the number of operations per second are seen to decrease. For instance, for 2 threads operating on key range 16 there are 17’349’000 operations performed per second. In contrast, 2 threads operating on key range 1,048,576 complete 3’689’000 operations per second. This is due to the fact that for a larger tree, there are more nodes to search to find the one on which we want to operate coupled with the fact that there will be a significant amount of contention for the lock to access the BST– hence, each operation takes longer to perform.
  + As the number of threads increase, the number of operations per second decrease roughly proportionally. This was expected for the TATAS lock, since it is expected that there would be significant contention for the lock resulting in fewer operations being performed for more threads.
  + Once the number of threads exceeds the number of logical CPUs, the number of operations per second performed is seen to decrease even further.

Hardware Lock Elision (HLE)

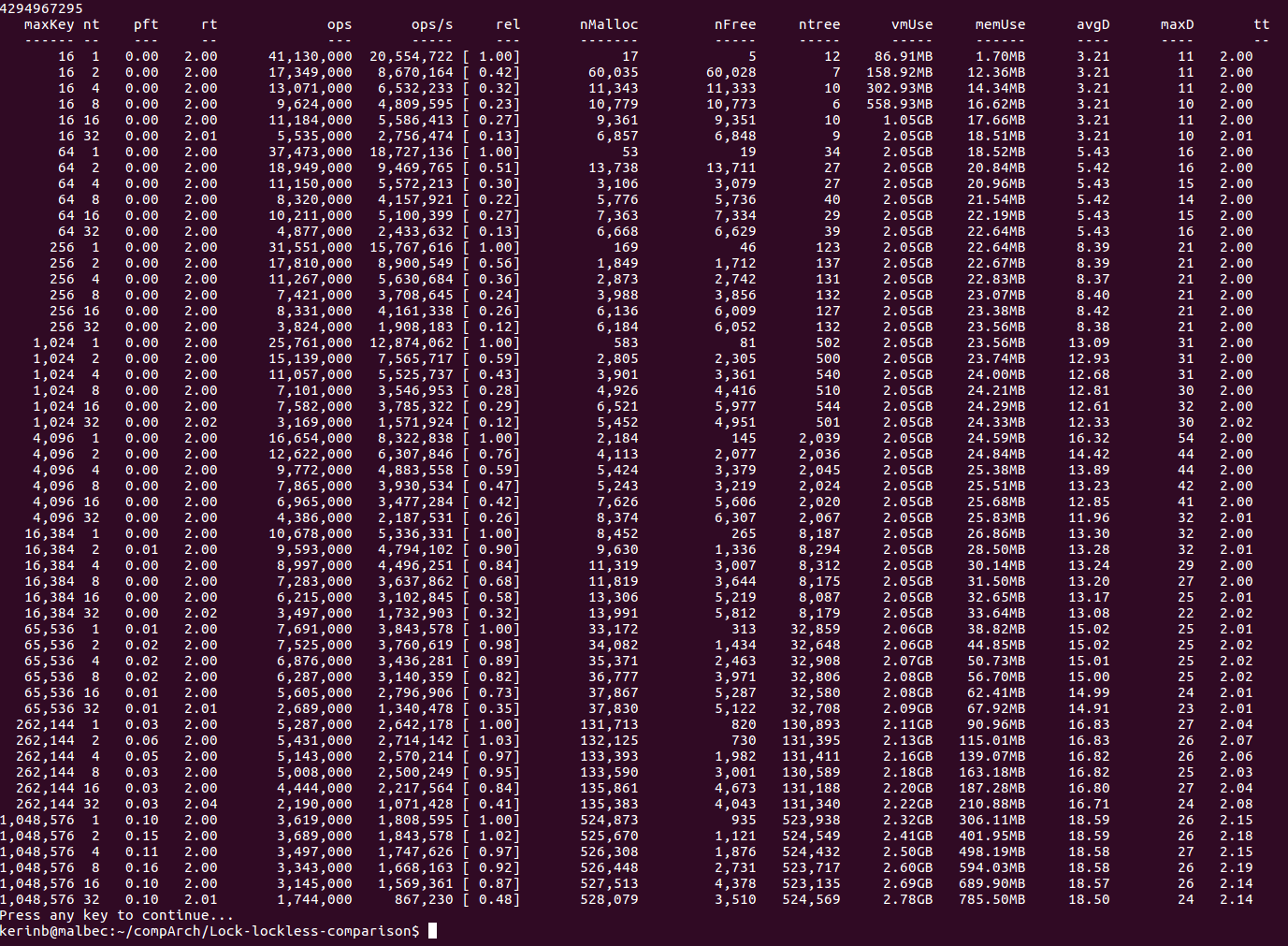
* The results here are that:
  + As the number of threads is increased, the number of operations per second decreases roughly proportionally - This is to be expected as for the increase in the number of threads, there is an increases in the amount of contention for the lock which will inevitably stall the progress made by the overall system.
  + As the keyrange increases, the number of operations per second performed decreases.
  + In comparison to the Basic TATAS lock implementation, the HLE TATAS implementation drastically outperformed the previous method - for a maxKey of 1’048’576 and 32 threads, HLE TATAS performed 6’832’667 ops/s, while basic TATAS only performed 867,320 ops/s - which has approximately 8x times more throughput in operations.

Restricted Transactional Memory (RTM)

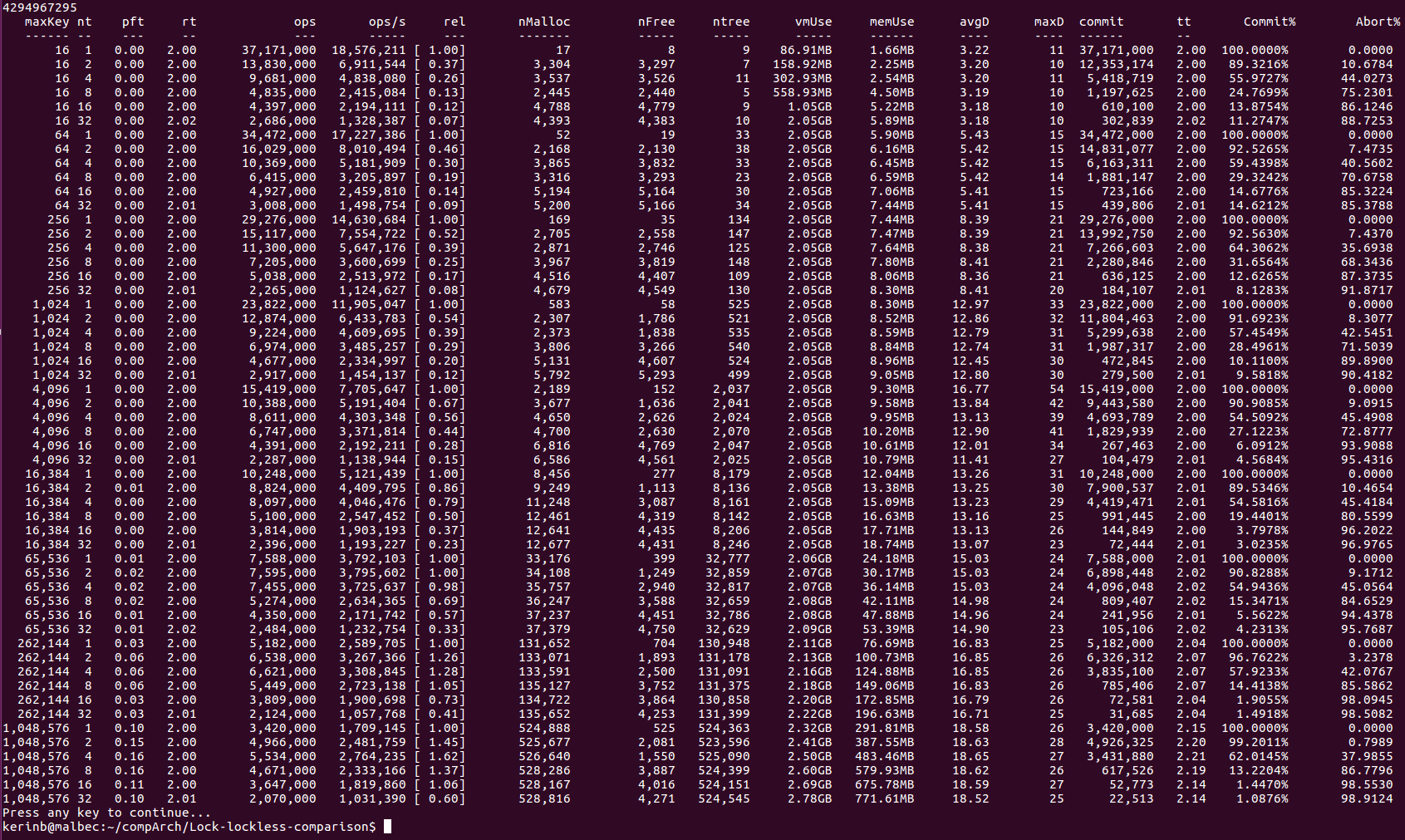
* The results here are that:
  + As the number of threads increases, the number of operations per second decreases roughly proportionally - much like above, this is expected as there will be an increase in contention for the lock since there will be more threads actively seeking the lock more often. Which will reduce the performance of the system.
  + Again, similar to the above other methods, as the key range increases, the number of operations per second performed decreases.
  + One thing that was expected from the RTM was that it would outperform all of the other methods, however it was outperformed by the HLE implementation if TATAS lock by the number of ops/s.

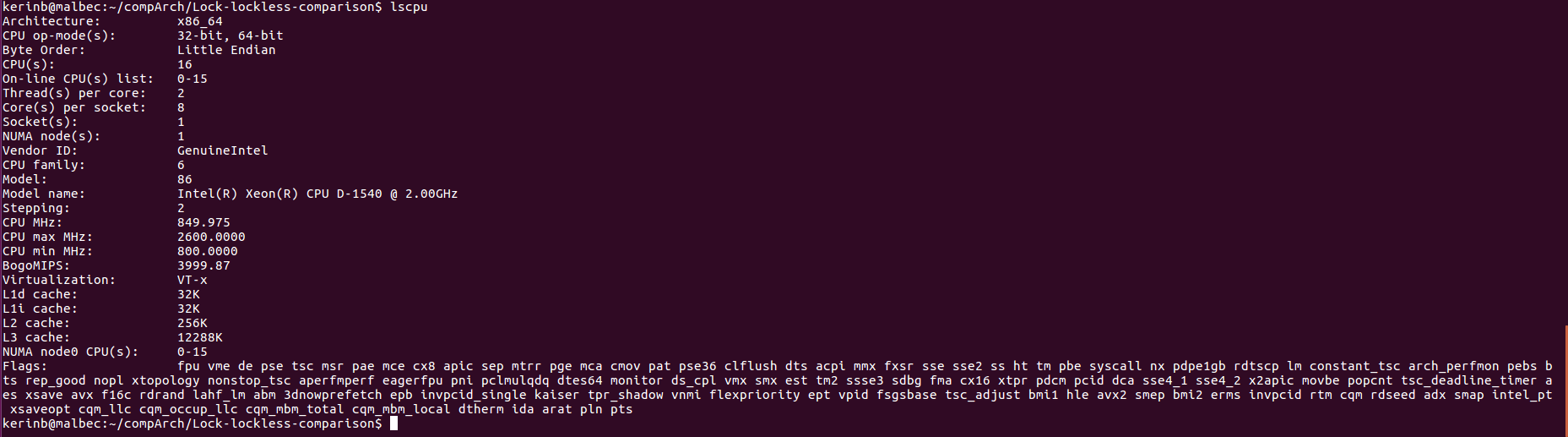
Method = 1 - No Threads

Method = 2 - TATAS Lock



Method = 3 - TATAS HLE

Hardware Specifications for Malbec Server



//

// tsxBST.cpp

//

// Copyright (C) 2013 - 2018 jones@scss.tcd.ie

//

#include "helper.h"

#include <iostream>

#include <sstream>

#include <iomanip>

using namespace std;

#define K 1024LL

#define M (K\*K)

#define G (K\*K\*K)

//

// METHOD 0: no lock single thread

// METHOD 1: testAndTestAndSet lock

// METHOD 2: HLE testAndTestAndSet lock

//

// TODO TODO TODO TODO TODO TODO TODO TODO TODO

//

// METHOD 3: RTM

//

int T = 0;

#define METHOD 3

#define PREFILL 0 // pre-fill with odd integers 0 .. maxKey-1 => 0: perfect 1: right list 2: left list

#define TRANSACTION 0

#define LOCK 1

#define MAXATTEMPT 5

#define MINKEY (16)

#define MAXKEY (1\*M)

#define SCALEKEY (4)

#define MINNT 1 // min number of threads

#define NSECONDS 2

#define STATS 0x17 // STATS bit MASK 1:commit 2:abort 4:depth 8:tsxStatus 16:times

#define ALIGNED

#define RECYCLENODES

//#define CONTAINS 100 // % contains (100-contains)/2 adds and removes

#define NOP 1000 // number of operations between tests for exceeding runtime

#if CONTAINS < 0 || CONTAINS > 100

#error CONTAINS sould be in the range 0 to 100

#endif

//

// useful MACROs

//

#if STATS & 1 //

#define STAT1(x) x //

#else

#define STAT1(x) //

#endif

#if STATS & 2 //

#define STAT2(x) x //

#else

#define STAT2(x) //

#endif

#if STATS & 4 //

#define STAT4(x) x //

#else

#define STAT4(x) //

#endif

#if STATS & 8 //

#define STAT8(x) x //

#else

#define STAT8(x) //

#endif

#if STATS & 16 //

#define STAT16(x) x //

#else

#define STAT16(x) //

#endif

#define DSUM pt->avgD += d; \

if (d > pt->maxD) \

pt->maxD = d //

#define PT(bst, thread) ((PerThreadData\*) ((BYTE\*) ((bst)->perThreadData) + (thread)\*ptDataSz))

//

// global variables

//

UINT ntMin; //

UINT nt; // # threads

UINT maxThread; //

UINT lineSz; // cache line sz

UINT ptDataSz; // per thread data size

UINT64 tStart; // start time

UINT64 t0; // start time of test

INT64 maxKey; // 0 .. keyMax-1

UINT64 totalOps = 0; // cumulative ops

THREADH \*threadH; // thread handles

TLSINDEX tlsPtIndx; // {joj 25/11/15}

//

// NB: ALL results

//

typedef struct {

INT64 maxKey; // 0 .. maxKey

UINT nt; // # threads

UINT64 pft; // pre fill time (ms)

UINT64 rt; // test run time (ms)

UINT64 nop; // nop

UINT64 nmalloc; // nmalloc

UINT64 nfree; // nfree

UINT64 avgD; // used to calculate average search depth of tree

UINT64 maxD; // max depth of tree

size\_t vmUse; // vmUse

size\_t memUse; // memUse

UINT64 ntree; // nodes in tree

UINT64 tt; // total time (ms) [fill time] + test run time + free memory time

UINT64 aborts;

UINT64 commits;

} Result;

Result \*r, \*ravg; // for results

UINT rindx; // results index

UINT64 errs = 0; // errors

class Node; // forward declaration

//

// PerThreadData

//

typedef struct {

UINT thread; // thread #

UINT64 nop; // nop

UINT64 nmalloc; // nmalloc

UINT64 nfree; // nfree

UINT64 avgD; // average tree depth

UINT64 maxD; // max tree depth

Node \*free; // head of free node list (RECYLENODES)

} PerThreadData;

//

// derive from ALIGNEDMA for aligned memory allocation

//

class ALIGNEDMA {

public:

void\* operator new(size\_t); // override new

void operator delete(void\*); // override delete

};

//

// new

//

void\* ALIGNEDMA::operator new(size\_t sz) {

return AMALLOC(sz, lineSz); // allocate on a lineSz boundary

}

//

// delete

//

void ALIGNEDMA::operator delete(void \*p) {

AFREE(p); // free memory

}

//

// Node

//

// Why are the Node members declared volatile? Consider the following code sequence:

//

// .. = n->key (1)

//

// while(1)

//

// UNIT status = \_xbegin();

//

// if (status == \_XBEGIN\_STARTED) {

//

// .. = n->key; (2)

//

// .. = n->key (3)

//

// \_xend();

//

// } else {

//

// // abort handler

//

// }

//

// }

//

// It is important that (2) reads the key value from memory as its address needs to be added

// to the transaction read set. A compiler may optimise away the memory read (2) because it

// has already been read by (1). Interestingly, it doesn't matter if (3) is read from memory

// or not as long as (2) is a memory read. A write to the key by another thread will be caught

// by the hardware and the transaction aborted. Clearly, it is more efficient to read the key

// in (3) from a register copy.

//

#if defined(ALIGNED)

class Node : public ALIGNEDMA {

#else

class Node {

#endif

public:

INT64 volatile key; // volatile

Node\* volatile left; // volatile

Node\* volatile right; // volatile

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

Node(INT64 k) {key = k; right = left = NULL;} // constructor

};

//

// BST

//

class BST : public ALIGNEDMA {

public:

PerThreadData \*perThreadData; // per thread data

Node\* volatile root; //

volatile UINT64 abortNum;

volatile UINT64 commitNum;

BST(UINT); // constructor

~BST(); // destructor

int contains(INT64); // return 1 if key in tree {joj 25/11/15}

int add(INT64); // add key to tree {joj 25/11/15}

int remove(INT64); // remove key from tree {joj 25/11/15}

INT64 checkBST(Node\*, UINT64& errBST); // {joj 13/12/15}

#ifdef PREFILL

void preFill(); //

#endif

private: // private

ALIGN(64) volatile long lock; // lock

int addTSX(Node\*); // add key into tree {joj 25/11/15}

Node\* removeTSX(INT64); // remove key from tree {joj 25/11/15}

#if defined(RECYCLENODES)

Node\* alloc(INT64, PerThreadData\*); // alloc

void dealloc(Node\*, PerThreadData\*); // dealloc

#endif

INT64 checkHelper(Node\*, INT64, INT64, UINT64&); // {joj 13/12/15}

};

//

// binary search tree

//

BST \*bst; // binary search tree

//

// BST constructor

//

BST::BST(UINT nt) {

perThreadData = (PerThreadData\*) AMALLOC(nt\*ptDataSz, lineSz); // for per thread data

memset(perThreadData, 0, nt\*ptDataSz); // zero

for (UINT thread = 0; thread < nt; thread++) //

PT(this, thread)->thread = thread; //

root = NULL; //

lock = 0;

abortNum = 0;

commitNum = 0;

}

#if defined(RECYCLENODES)

//

// alloc

//

inline Node\* BST::alloc(INT64 key, PerThreadData \*pt) {

Node \*n;

if (pt->free) {

n = pt->free;

pt->free = n->right;

n->key = key;

n->left = n->right = NULL;

} else {

n = new Node(key);

pt->nmalloc++;

}

return n;

}

//

// dealloc

//

inline void BST::dealloc(Node \*n, PerThreadData \*pt) {

n->right = pt->free;

pt->free = n;

}

#endif

//

// checkHelper

//

// will not work if key VINTMIN or VINTMAX added

//

INT64 BST:: checkHelper(Node \*n, INT64 min, INT64 max, UINT64& errBST) {

if (n == NULL)

return 0;

if (n->key <= min || n->key >= max)

errBST += 1;

return checkHelper(n->left, min, n->key, errBST) + checkHelper(n->right, n->key, max, errBST) + 1;

}

//

//

// checkBST

//

// check tree structure

// in-order traversal of binary tree making sure keys in correct range

// return number of nodes in BST

// count errors

//

INT64 BST::checkBST(Node\* n, UINT64& errBST) {

errBST = 0;

return checkHelper(n, INT64MIN, INT64MAX, errBST);

}

//

// BST destructor

//

BST::~BST() {

AFREE(perThreadData);

}

//

// contains - search for key in tree

//

// METHOD 0: NO lock single thread

// METHOD 1: testAndTestAndSet

// METHOD 2: HLE testAndTestAndSet

//

// return 1 if key in tree

//

int BST::contains(INT64 key) {

PerThreadData \*pt = (PerThreadData\*)TLSGETVALUE(tlsPtIndx);

#if METHOD == 1

while (\_InterlockedExchange(&lock, 1)) {

do {

\_mm\_pause();

} while (lock);

}

#endif

#if METHOD == 2

while (\_\_atomic\_exchange\_n(&lock, 1, \_\_ATOMIC\_ACQUIRE | \_\_ATOMIC\_HLE\_ACQUIRE)){

do {

\_mm\_pause();

} while (lock == 1);

}

#endif

#if METHOD == 3

int state = TRANSACTION;

int attempt = 1;

while(1){ // while I dont have a lock/ ability to commit transaction

UINT status = \_XBEGIN\_STARTED;

if (state == TRANSACTION){ // If I can transact

status = \_xbegin();

} else { // otherwise, grab a lock

while (\_\_atomic\_exchange\_n(&lock, 1, \_\_ATOMIC\_ACQUIRE | \_\_ATOMIC\_HLE\_ACQUIRE)){

abortNum++;

do {

\_mm\_pause();

} while (lock == 1);

}

}

cout << "test " << endl;

if (status == \_XBEGIN\_STARTED){ // If I can transact

cout << "transaction begin " << endl;

if(state == TRANSACTION && lock){

\_xabort(0xA0); // abort if lock is already set

} else {

break;

}

} else {

cout << "transaction fail " << endl;

// the transaction aborted

if(lock){

do{

\_mm\_pause();

} while(lock);

} else {

volatile UINT64 wait = attempt << 4; // initialise wait and delay by ...

while (wait--);

}

if (++attempt >= MAXATTEMPT) {

cout << "attempt " << attempt << endl;

state = LOCK; // execute non transactionally by obtaining lock

}

}

}

#endif

Node \*p = root;

STAT4(UINT64 d = 0);

while (p) {

STAT4(d++);

if (key < p->key) {

p = p->left;

} else if (key > p->key) {

p = p->right;

} else {

#if METHOD == 1

lock = 0;

#endif

#if METHOD == 2

\_\_atomic\_store\_n(&lock, 0, \_\_ATOMIC\_RELEASE | \_\_ATOMIC\_HLE\_RELEASE);

#endif

#if METHOD == 3

if(state == TRANSACTION){

commitNum++;

\_xend();

} else {

\_\_atomic\_store\_n(&lock, 0, \_\_ATOMIC\_RELEASE | \_\_ATOMIC\_HLE\_RELEASE);

}

#endif

STAT4(DSUM);

return 1;

}

}

#if METHOD == 1

lock = 0;

#endif

#if METHOD == 2

\_\_atomic\_store\_n(&lock, 0, \_\_ATOMIC\_RELEASE | \_\_ATOMIC\_HLE\_RELEASE);

#endif

#if METHOD == 3

if(state == TRANSACTION){

commitNum++;

\_xend();

} else {

\_\_atomic\_store\_n(&lock, 0, \_\_ATOMIC\_RELEASE | \_\_ATOMIC\_HLE\_RELEASE);

}

#endif

STAT4(DSUM);

return 0;

}

//

// addTSX - add key to tree

//

// METHOD 0: NO lock single thread

// METHOD 1: testAndTestAndSet

// METHOD 2: HLE testAndTestAndSet

//

// return 1 if key found

//

int BST::addTSX(Node \*n) {

PerThreadData \*pt = (PerThreadData\*)TLSGETVALUE(tlsPtIndx);

#if METHOD == 1

while (\_InterlockedExchange(&lock, 1)) {

do {

\_mm\_pause();

} while (lock);

}

#endif

#if METHOD == 2

while (\_\_atomic\_exchange\_n(&lock, 1, \_\_ATOMIC\_ACQUIRE | \_\_ATOMIC\_HLE\_ACQUIRE)){

do {

\_mm\_pause();

} while (lock == 1);

}

#endif

#if METHOD == 3

int state = TRANSACTION;

int attempt = 1;

while(1){ // while I dont have a lock/ ability to commit transaction

UINT status = \_XBEGIN\_STARTED;

if (state == TRANSACTION){ // If I can transact

status = \_xbegin();

} else { // otherwise, grab a lock

while (\_\_atomic\_exchange\_n(&lock, 1, \_\_ATOMIC\_ACQUIRE | \_\_ATOMIC\_HLE\_ACQUIRE)){

abortNum++;

do {

\_mm\_pause();

} while (lock == 1);

}

}

if (status == \_XBEGIN\_STARTED){ // If I can transact

if(state == TRANSACTION && lock){

\_xabort(0xA0); // abort if lock is already set

} else {

break;

}

} else {

// the transaction aborted

if(lock){

do{

\_mm\_pause();

} while(lock);

} else {

volatile UINT64 wait = attempt << 4; // initialise wait and delay by ...

while (wait--);

}

if (++attempt >= MAXATTEMPT) {

state = LOCK; // execute non transactionally by obtaining lock

}

}

}

#endif

Node\* volatile \*pp = &root;

Node \*p = root;

STAT4(UINT64 d = 0);

while (p) {

STAT4(d++);

if (n->key < p->key) {

pp = &p->left;

} else if (n->key > p->key) {

pp = &p->right;

} else {

#if METHOD == 1

lock = 0;

#endif

#if METHOD == 2

\_\_atomic\_store\_n(&lock, 0, \_\_ATOMIC\_RELEASE | \_\_ATOMIC\_HLE\_RELEASE);

#endif

#if METHOD == 3

if(state == TRANSACTION){

commitNum++;

\_xend();

} else {

\_\_atomic\_store\_n(&lock, 0, \_\_ATOMIC\_RELEASE | \_\_ATOMIC\_HLE\_RELEASE);

}

#endif

STAT4(DSUM);

return 0;

}

p = \*pp;

}

\*pp = n;

#if METHOD == 1

lock = 0;

#endif

#if METHOD == 2

\_\_atomic\_store\_n(&lock, 0, \_\_ATOMIC\_RELEASE | \_\_ATOMIC\_HLE\_RELEASE);

#endif

#if METHOD == 3

if(state == TRANSACTION){

commitNum++;

\_xend();

} else {

\_\_atomic\_store\_n(&lock, 0, \_\_ATOMIC\_RELEASE | \_\_ATOMIC\_HLE\_RELEASE);

}

#endif

STAT4(DSUM);

return 1;

}

//

// removeTSX - remove key from tree

//

// METHOD 0: NO lock single thread

// METHOD 1: testAndTestAndSet

//

// return pointer to removed node, otherwise NULL

//

Node\* BST::removeTSX(INT64 key) {

PerThreadData \*pt = (PerThreadData\*)TLSGETVALUE(tlsPtIndx);

#if METHOD == 1

while (\_InterlockedExchange(&lock, 1)) {

do {

\_mm\_pause();

} while (lock);

}

#endif

#if METHOD == 2

while (\_\_atomic\_exchange\_n(&lock, 1, \_\_ATOMIC\_ACQUIRE | \_\_ATOMIC\_HLE\_ACQUIRE)){

do {

\_mm\_pause();

} while (lock == 1);

}

#endif

#if METHOD == 3

int state = TRANSACTION;

int attempt = 1;

while(1){ // while I dont have a lock/ ability to commit transaction

UINT status = \_XBEGIN\_STARTED;

if (state == TRANSACTION){ // If I can transact

status = \_xbegin();

} else { // otherwise, grab a lock

while (\_\_atomic\_exchange\_n(&lock, 1, \_\_ATOMIC\_ACQUIRE | \_\_ATOMIC\_HLE\_ACQUIRE)){

abortNum++;

do {

\_mm\_pause();

} while (lock == 1);

}

}

if (status == \_XBEGIN\_STARTED){ // If I can transact

if(state == TRANSACTION && lock){

\_xabort(0xA0); // abort if lock is already set

} else {

break;

}

} else {

// the transaction aborted

if(lock){

do{

\_mm\_pause();

} while(lock);

} else {

volatile UINT64 wait = attempt << 4; // initialise wait and delay by ...

while (wait--);

}

if (++attempt >= MAXATTEMPT) {

state = LOCK; // execute non transactionally by obtaining lock

}

}

}

#endif

Node\* volatile \*pp = &root;

Node \*p = root;

STAT4(UINT64 d = 0);

while (p) {

STAT4(d++);

if (key < p->key) {

pp = &p->left;

} else if (key > p->key) {

pp = &p->right;

} else {

break;

}

p = \*pp;

}

if (p == NULL) {

#if METHOD == 1

lock = 0;

#endif

#if METHOD == 2

\_\_atomic\_store\_n(&lock, 0, \_\_ATOMIC\_RELEASE | \_\_ATOMIC\_HLE\_RELEASE);

#endif

#if METHOD == 3

if(state == TRANSACTION){

commitNum++;

\_xend();

} else {

\_\_atomic\_store\_n(&lock, 0, \_\_ATOMIC\_RELEASE | \_\_ATOMIC\_HLE\_RELEASE);

}

#endif

STAT4(DSUM);

return NULL;

}

Node \*left = p->left;

Node \*right = p->right;

if (left == NULL && right == NULL) {

\*pp = NULL;

} else if (left == NULL) {

\*pp = right;

} else if (right == NULL) {

\*pp = left;

} else {

Node\* volatile \*ppr = &p->right;

Node \*r = right;

while (r->left) {

ppr = &r->left;

r = r->left;

}

#ifdef MOVENODE

\*ppr = r->right;

r->left = p->left;

r->right = p->right;

\*pp = r;

#else

p->key = r->key;

p = r;

\*ppr = r->right;

#endif

}

#if METHOD == 1

lock = 0;

#endif

#if METHOD == 2

\_\_atomic\_store\_n(&lock, 0, \_\_ATOMIC\_RELEASE | \_\_ATOMIC\_HLE\_RELEASE);

#endif

#if METHOD == 3

if(state == TRANSACTION){

commitNum++;

\_xend();

} else {

\_\_atomic\_store\_n(&lock, 0, \_\_ATOMIC\_RELEASE | \_\_ATOMIC\_HLE\_RELEASE);

}

#endif

STAT4(DSUM);

return p;

}

//

// add

//

// add key to tree

//

int BST::add(INT64 key) {

#if defined(RECYCLENODES)

PerThreadData \*pt = (PerThreadData\*)TLSGETVALUE(tlsPtIndx); // {joj 6/11/18}

Node \*n = alloc(key, pt);

#else

Node \*n = new Node(key);

pt->nmalloc++;

#endif

if (addTSX(n) == 0) {

#if defined(RECYCLENODES)

dealloc(n, pt);

#else

delete n;

pt->nfree++;

#endif

return 0;

}

return 1;

}

//

// remove

//

// remove key from tree

//

int BST::remove(INT64 key) {

if (Node volatile \*n = removeTSX(key)) {

#if defined(RECYCLENODES)

PerThreadData \*pt = (PerThreadData\*)TLSGETVALUE(tlsPtIndx); // {joj 6/11/18}

dealloc((Node \*) n, pt);

#else

delete n;

pt->nfree++;

#endif

return 1;

}

return 0;

}

#if defined(PREFILL) && PREFILL == 0

//

// preFillHelper

//

void preFillHelper(Node\* volatile &root, INT64 minKey, INT64 maxKey, INT64 diff, PerThreadData \*pt) {

if (maxKey - minKey <= diff)

return;

INT64 key = minKey + (maxKey - minKey) / 2;

root = new Node(key);

pt->nmalloc++;

preFillHelper(root->left, minKey, key, diff, pt);

preFillHelper(root->right, key, maxKey, diff, pt);

}

//

// preFillWorker

//

// NB: use thread to set up values for preFillHelper

//

WORKER preFillWorker(void\* vthread) {

UINT64 thread = (UINT64) vthread;

INT64 minK = 0;

INT64 maxK = maxKey - 1;

INT64 key = 0;

for (UINT64 n = ncpu / 2; n; n /= 2) {

key = minK + (maxK - minK) / 2;

if (thread & n) {

minK = key;

} else {

maxK = key;

}

}

Node \*np = bst->root;

while (np) {

if (key < np->key) {

np = np->left;

} else if (key > np->key) {

np = np->right;

} else {

break;

}

}

PerThreadData \*pt = PT(bst, thread);

preFillHelper((key == maxK) ? np->left : np->right, minK, maxK, 2, pt);

return 0;

}

//

// preFill

//

// NB: single thread for small trees, ALL threads for large trees

//

void BST::preFill() {

//

// for small lists use a single thread

//

if (maxKey <= 1\*M) {

preFillHelper(bst->root, 0, maxKey - 1, 2, PT(this, 0));

return;

}

//

// fill top ncpu-1 tree nodes

//

preFillHelper(bst->root, 0, maxKey - 1, maxKey / ncpu, PT(this, 0));

//

// create worker threads and wait to finish

//

for (UINT thread = 0; thread < ncpu; thread++)

createThread(&threadH[thread], preFillWorker, (void\*) (UINT64) thread);

waitForThreadsToFinish(ncpu, threadH);

for (UINT thread = 0; thread < ncpu; thread++)

closeThread(threadH[thread]);

//

// accumulate nmalloc stats on thread 0 for threads >= nt

//

for (UINT cpu = 1; cpu < ncpu; cpu++) {

if (cpu >= nt) {

PT(this, 0)->nmalloc += PT(this, cpu)->nmalloc;

PT(this, cpu)->nmalloc = 0;

}

}

}

#elif defined(PREFILL) && (PREFILL == 1 || PREFILL == 2)

struct PARAMS {

PerThreadData \*pt;

INT64 minKey;

INT64 maxKey;

Node \*first;

Node \*last;

};

//

// preFillWorker

//

// NB: worker for single and multi-threaded prefill

// NB: makes an ordered list containing the odd numbers in the range params->minKey to params->maxKey

// NB: assumes params->minKey and params->maxKey are both even

// NB: assumes params->first is NULL

//

WORKER preFillWorker(void \*\_params) {

PARAMS \*params = (PARAMS\*) \_params; // coercion

INT64 key = params->minKey + 1; // insert odd integers

while (key < params->maxKey) {

Node \*nn = new Node(key);

params->pt->nmalloc++;

#if PREFILL == 1

if (params->first == NULL) { // in ascending order

params->first = nn;

} else {

params->last->right = nn;

}

params->last = nn;

#else

if (params->first == NULL) { // in descending order

params->last = nn;

} else {

nn->left = params->first;

}

params->first = nn;

#endif

key += 2;

}

return 0;

}

//

// preFill

//

// NB: single thread for small trees, ALL threads for large trees

//

void BST::preFill() {

//

// for small lists use a single thread

//

if (maxKey <= 1\*M) {

PARAMS params;

params.pt = PT(this, 0); // use thread 0

params.minKey = 0;

params.maxKey = maxKey;

params.first = NULL;

params.last = NULL;

preFillWorker(&params);

root = params.first;

//params.last->left = new (PT(this, 0)->thread) Node(3); // error

return;

}

//

// following code is future proofing!

// on current CPUs, searching very long ordered lists is slow

//

// set up parameters

//

PARAMS \*params = new PARAMS[ncpu];

for (UINT cpu = 0; cpu < ncpu; cpu++) {

params[cpu].pt = PT(this, cpu);

params[cpu].minKey = maxKey / ncpu \* cpu;

params[cpu].maxKey = maxKey / ncpu \* (cpu + 1);

params[cpu].first = NULL;

params[cpu].last = NULL;

}

//

// create worker threads and wait to finish

//

for (UINT cpu = 0; cpu < ncpu; cpu++)

createThread(&threadH[cpu], preFillWorker, (void\*) &params[cpu]);

waitForThreadsToFinish(ncpu, threadH);

//

// concatenate individual lists in ascending or descending order

//

for (UINT cpu = 0; cpu < ncpu; cpu++) {

#if PREFILL == 1

if (cpu == 0) {

root = params[0].first;

} else {

params[cpu-1].last->right = params[cpu].first;

}

#else

if (cpu == ncpu - 1) {

root = params[cpu].first;

} else {

params[cpu+1].last->left = params[cpu].first;

}

#endif

}

//

// tidy up

//

for (UINT cpu = 0; cpu < ncpu; cpu++)

closeThread(threadH[cpu]);

delete params;

//

// accumulate nmalloc stats on thread 0 for threads >= nt

//

for (UINT cpu = 1; cpu < ncpu; cpu++) {

if (cpu >= nt) {

PT(this, 0)->nmalloc += PT(this, cpu)->nmalloc;

PT(this, cpu)->nmalloc = 0;

}

}

}

#endif

//

// worker thread

//

WORKER worker(void\* vthread) {

PerThreadData \*pt = PT(bst, (UINT64)vthread); // {joj 25/11/15}

TLSSETVALUE(tlsPtIndx, pt); // {joj 25/11/15}

UINT64 r; // local variable for pseudo random number generator

while (\_rdrand64\_step(&r) == 0); // random seed for rand

while (1) {

//

// do some work

//

for (int i = 0; i < NOP; i++) {

UINT64 k = rand(r);

#if CONTAINS == 0

INT64 key = k & (maxKey - 1);

if (k >> 63) {

bst->add(key);

} else {

bst->remove(key);

}

#else

UINT op = k % 100;

INT64 key = (k / 100) & (maxKey - 1);

if (op < CONTAINS) {

bst->contains(key);

} else if (op < CONTAINS + (100 - CONTAINS) / 2) {

bst->add(key);

} else {

bst->remove(key);

}

#endif

}

pt->nop += NOP;

//

// check if runtime exceeded

//

if (getWallClockMS() - t0 > NSECONDS\*1000)

break;

}

return 0;

}

//

// header

//

void header() {

char date[256];

getDateAndTime(date, sizeof(date));

cout << getHostName() << " " << getOSName() << (is64bitExe() ? " 64" : " 32") << " bit exe";

#ifdef \_DEBUG

cout << " DEBUG";

#else

cout << " RELEASE";

#endif

#if METHOD == 0

cout << " BST [NO lock single thread ONLY]";

#elif METHOD == 1

cout << " BST [testAndTestAndSet lock]";

#endif

#if METHOD == 2

cout << "BST [HLE testAndTestAndSet lock]";

#endif

#if METHOD == 3

cout << "BST [Hardware Transactional Memory]";

#endif

cout << " NCPUS=" << ncpu << " RAM=" << (getPhysicalMemSz() + G - 1) / G << "GB ";

cout << endl;

cout << "METHOD=" << METHOD << " ";

#ifdef ALIGNED

cout << "ALIGNED ";

#endif

#ifdef CONTAINS

cout << "CONTAINS=" << CONTAINS << "% ADD=" << (100 - CONTAINS) / 2 << "% REMOVE=" << (100 - CONTAINS) / 2 << "% ";

#endif

#ifdef MOVENODE

cout << "MOVENODE ";

#endif

cout << "NOP=" << NOP << " ";

cout << "NSECONDS=" << NSECONDS << " ";

#ifdef PREFILL

cout << "PREFILL=" << PREFILL << " ";

#endif

#if defined(RECYCLENODES)

cout << "RECYCLENODES ";

#endif

#ifdef STATS

cout << "STATS=0x" << setfill('0') << hex << setw(2) << STATS << dec << setfill(' ') << " ";

#endif

cout << "sizeof(Node)=" << sizeof(Node) << endl;;

cout << "Intel" << (cpu64bit() ? "64" : "32") << " family " << cpuFamily() << " model " << cpuModel() << " stepping " << cpuStepping() << " " << cpuBrandString() << endl;

}

//

// main

//

int main(int argc, char\* argv[]) {

ncpu = getNumberOfCPUs(); // get number of CPUs

if (ncpu > 32) // {joj 6/11/18}

ncpu = 32; //

maxThread = 2 \* ncpu; //

TLSALLOC(tlsPtIndx); // {joj 25/11/15}

header(); // output header

tStart = time(NULL); // start time

lineSz = getCacheLineSz(); // get cache line size and output cache organisation

ptDataSz = (UINT) (sizeof(PerThreadData) + lineSz - 1) / lineSz \* lineSz; // {joj 12/2/18}

//cout << "sizeof(PerThreadData)=" << sizeof(PerThreadData) << " ptDataSz=" << ptDataSz << " sizeof(Result)=" << sizeof(Result);

cout << endl;

cout << \_XBEGIN\_STARTED << endl;

#if METHOD == 2

//

// check if HLE supported

//

if (!hleSupported()) {

cout << "HLE (hardware lock elision) NOT supported by this CPU" << endl;

quit();

return 1;

} else {

cout << "HLE Support by this CPU" << endl;

}

#endif

#if METHOD == 3

//

// check if RTM supported

//

if (!rtmSupported()) {

cout << "RTM (restricted transactional memory) NOT supported by this CPU" << endl;

quit();

return 1;

}

#endif

#ifdef WIN32

SetPriorityClass(GetCurrentProcess(), ABOVE\_NORMAL\_PRIORITY\_CLASS);

SetThreadPriority(GetCurrentThread(), THREAD\_PRIORITY\_ABOVE\_NORMAL);

#endif

INT64 keyMin = MINKEY; // set min key

INT64 keyMax = MAXKEY; // set max key

#if METHOD == 0

ntMin = 1;

UINT ntMax = 1;

#else

ntMin = MINNT;

UINT ntMax = maxThread; //

#endif

int c0 = 1, c1 = 1; //

for (maxKey = keyMin; maxKey < keyMax; maxKey \*= SCALEKEY) //

c0++; //

for (nt = ntMin; nt < ntMax; nt \*= 2) //

c1++; //

threadH = (THREADH\*) AMALLOC(maxThread\*sizeof(THREADH), lineSz); // thread handles

r = (Result\*) AMALLOC(c0\*c1\*sizeof(Result), lineSz); // for results

ravg = (Result\*) AMALLOC(c0\*c1\*sizeof(Result), lineSz); // for averages

memset(ravg, 0, c0\*c1\*sizeof(Result)); // clear results

//

// results

//

setCommaLocale();

int keyw = (int) log10((double) keyMax) + (int) (log10((double) keyMax) / log10(1000)) + 2;

keyw = (keyw < 7) ? 7 : keyw;

cout << setw(keyw - 1) << "maxKey" << setw(3) << "nt";

#ifdef PREFILL

STAT16(cout << setw(7) << "pft");

#endif

cout << setw(7) << "rt";

cout << setw(16) << "ops" << setw(12) << "ops/s";

#if METHOD > 0

if (ntMin == 1)

cout << setw(8) << "rel";

#endif

cout << setw(14) << "nMalloc" << setw(14) << "nFree";

cout << setw(keyw) << "ntree";

cout << setw(11) << "vmUse" << setw(11) << "memUse";

STAT4(cout << setw(keyw) << "avgD" << setw(keyw) << "maxD");

#if METHOD > 1

STAT1(cout << setw(8) << "commit");

#endif

STAT16(cout << setw(7) << "tt");

#if METHOD == 3

cout << setw(14) << fixed << "No Abort %";

#endif

cout << endl;

cout << setw(keyw - 1) << "------" << setw(3) << "--"; // maxKey nt

#ifdef PREFILL

STAT16(cout << setw(7) << "---"); // pft

#endif

cout << setw(7) << "--"; // rt

cout << setw(16) << "---" << setw(12) << "-----"; // ops ops/s

#if METHOD > 0

if (ntMin == 1)

cout << setw(8) << "---"; // rel

#endif

cout << setw(14) << "-------" << setw(14) << "-----"; // nMalloc nFree

cout << setw(keyw) << "-----"; // ntree

cout << setw(11) << "-----" << setw(11) << "------"; // vmUse memUse

STAT4(cout << setw(keyw) << "----" << setw(keyw) << "----"); // avgD maxD

#if METHOD > 1

STAT1(cout << setw(8) << "------");

#endif

STAT16(cout << setw(7) << "--"); // tt

cout << endl;

rindx = 0; // zero results index

for (maxKey = keyMin; maxKey <= keyMax; maxKey \*= SCALEKEY) {

#if METHOD > 0

double noppersec1 = 1;

#endif

for (nt = ntMin; nt <= ntMax; nt \*= 2) {

bst = new BST(maxThread); // create an empty binary search tree

t0 = getWallClockMS(); // get start time

#ifdef PREFILL

bst->preFill();

UINT64 pft = getWallClockMS() - t0;

t0 = getWallClockMS(); // get start time

#else

UINT64 pft = 0; //

#endif

//

// create worker threads

//

for (UINT thread = 0; thread < nt; thread++)

createThread(&threadH[thread], worker, (void\*) (UINT64) thread);

//

// wait for ALL worker threads to finish

//

waitForThreadsToFinish(nt, threadH);

UINT64 rt = getWallClockMS() - t0;

//

// calculate results

//

UINT64 nop = 0, nmalloc = 0, nfree = 0, ntree;

UINT64 avgD = 0, maxD = 0;

size\_t vmUse, memUse;

for (UINT thread = 0; thread < nt; thread++) {

PerThreadData \*pt = PT(bst, thread);

nop += pt->nop;

nmalloc += pt->nmalloc;

#if defined(RECYCLENODES)

while (pt->free) {

Node \*tmp = pt->free->right;

delete pt->free;

pt->free = tmp;

pt->nfree++;

}

#endif

nfree += pt->nfree;

avgD += pt->avgD;

maxD = (pt->maxD > maxD) ? pt->maxD : maxD;

}

#if METHOD > 0

if (nt == 1)

noppersec1 = (double) nop / (double) rt / 1000; // {joj 12/1/18}

#endif

totalOps += nop;

//

// save results

//

r[rindx].maxKey = maxKey;

r[rindx].nt = nt;

r[rindx].pft = pft;

r[rindx].rt = rt;

r[rindx].nop = nop;

r[rindx].nmalloc = nmalloc;

r[rindx].nfree = nfree;

r[rindx].avgD = avgD;

r[rindx].maxD = maxD;

r[rindx].aborts = bst->abortNum;

r[rindx].commits = bst->commitNum;

//

// get vmUse and memUse before deleting BST

//

vmUse = r[rindx].vmUse = getVMUse();

memUse = r[rindx].memUse = getMemUse();

//

// output results

//

cout << setw(keyw - 1) << maxKey << setw(3) << nt;

#ifdef PREFILL

STAT16(cout << setw(7) << fixed << setprecision(pft < 100\*1000 ? 2 : 0) << (double) pft / 1000);

#endif

cout << setw(7) << fixed << setprecision(rt < 100\*1000 ? 2 : 0) << (double) rt / 1000;

cout << setw(16) << nop << setw(12) << nop \* 1000 / rt;

#if METHOD > 0

if (ntMin == 1)

cout << " [" << fixed << setprecision(2) << setw(5) << (double) nop / (double) rt / 1000 / noppersec1 << "]"; // {joj 12/1/18}

#endif

cout << setw(14) << nmalloc << setw(14) << nfree;

cout << flush; // useful on linux

UINT64 errBST;

ntree = bst->checkBST(bst->root, errBST);

r[rindx].ntree = ntree;

cout << setw(keyw) << ntree;

delete bst;

if (vmUse / G) {

cout << fixed << setprecision(2) << setw(9) << (double) vmUse / G << "GB";

} else {

cout << fixed << setprecision(2) << setw(9) << (double) vmUse / M << "MB";

}

if (memUse / G) {

cout << fixed << setprecision(2) << setw(9) << (double) memUse / G << "GB";

} else {

cout << fixed << setprecision(2) << setw(9) << (double) memUse / M << "MB";

}

STAT4(double davgD = (double) avgD / (double) nop); // {joj 12/1/18}

STAT4(cout << fixed << setprecision(2) << setw(keyw) << setprecision(davgD < 1000 ? 2 : 0) << davgD << setw(keyw) << maxD);

#if METHOD == 3

cout << setw(7) << fixed << r[rindx].commits;

#endif

UINT64 tt = getWallClockMS() - t0;

#ifdef PREFILL

tt += pft;

#endif

STAT16(cout << setw(7) << fixed << setprecision(tt < 100\*1000 ? 2 : 0) << (double) tt / 1000);

#if METHOD == 3

cout << setw(10) << fixed << setprecision(4) << 100.00\*((double) r[rindx].nop - (double) r[rindx].aborts)/(double)r[rindx].nop << "% " << setw(7) << fixed << r[rindx].aborts;

#endif

//

// tidy up

//

for (UINT thread = 0; thread < nt; thread++)

closeThread(threadH[thread]);

//

// check for errors

//

if (errBST || (nmalloc != ntree + nfree)) {

cout << " ERROR:";

//

// BST NOT correct

//

if (errBST) {

errs++;

cout << " BST inccorrect";

}

//

// nodes missing

//

if (nmalloc != ntree + nfree) {

errs++;

cout << " diff= " << nmalloc - ntree - nfree;

}

cout << " [errs=" << errs << "]";

}

//

// accumlate averages

//

ravg[rindx].pft += r[rindx].pft;

ravg[rindx].rt += r[rindx].rt;

ravg[rindx].nop += r[rindx].nop;

ravg[rindx].nmalloc += r[rindx].nmalloc;

ravg[rindx].nfree += r[rindx].nfree;

ravg[rindx].avgD += r[rindx].avgD;

ravg[rindx].maxD += r[rindx].maxD;

ravg[rindx].vmUse += r[rindx].vmUse;

ravg[rindx].memUse += r[rindx].memUse;

ravg[rindx].ntree += r[rindx].ntree;

ravg[rindx].tt += tt;

rindx++;

cout << endl;

} // nt

} // maxkey

pressKeyToContinue();

return 0;

}

// eof

//

// helper.cpp

//

// Copyright (C) 2011 - 2018 jones@scss.tcd.ie

//

//

// NB: gcc needs flags -mrtm -mrdrnd

//

#include <iostream> // cout

#include <iomanip> // setprecision

#include "helper.h" //

#ifdef WIN32

#include <conio.h> // \_getch()

#include <psapi.h> // GetProcessMemoryInfo

#elif \_\_linux\_\_

#include <termios.h> //

#include <unistd.h> //

#include <limits.h> // HOST\_NAME\_MAX

#include <sys/utsname.h> //

#include <fcntl.h> // O\_RDWR

#endif

using namespace std; // cout. ...

//

// for data returned by cpuid instruction

//

struct \_cd {

UINT eax;

UINT ebx;

UINT ecx;

UINT edx;

} cd;

UINT ncpu; // # logical CPUs

char \*hostName = NULL; // host name

char \*osName = NULL; // os name

char \*brandString = NULL; // cpu brand string

//

// getDateAndTime

//

void getDateAndTime(char \*dateAndTime, int sz, time\_t t) {

t = (t == 0) ? time(NULL) : 0;

#ifdef WIN32

struct tm now;

localtime\_s(&now, &t);

strftime(dateAndTime, sz, "%d-%b-%Y %H:%M:%S", &now);

#elif \_\_linux\_\_

struct tm \*now = localtime(&t);

strftime(dateAndTime, sz, "%d-%b-%Y %H:%M:%S", now);

#endif

}

//

// getHostName

//

char\* getHostName() {

if (hostName == NULL) {

#ifdef WIN32

DWORD sz = (MAX\_COMPUTERNAME\_LENGTH + 1) \* sizeof(char);

hostName = (char\*) malloc(sz);

GetComputerNameA(hostName, &sz);

#elif \_\_linux\_\_

size\_t sz = (HOST\_NAME\_MAX + 1) \* sizeof(char);

hostName = (char\*) malloc(sz);

gethostname(hostName, sz);

#endif

}

return hostName;

}

//

// getOSName

//

char\* getOSName() {

if (osName == NULL) {

osName = (char\*) malloc(256); // should be large enough

#ifdef WIN32

DWORD sz = 256;

RegGetValueA(HKEY\_LOCAL\_MACHINE, "Software\\Microsoft\\Windows NT\\CurrentVersion", "ProductName", RRF\_RT\_ANY, NULL, (LPBYTE) osName, &sz);

#ifdef \_WIN64

strcat\_s(osName, 256, " (64 bit)");

#else

int win64;

IsWow64Process(GetCurrentProcess(), &win64);

strcat\_s(osName, 256, win64 ? " (64 bit)" : " (32 bit)");

#endif

#elif \_\_linux\_\_

struct utsname utsName;

uname(&utsName);

strcpy(osName, utsName.sysname);

strcat(osName, " ");

strcat(osName, utsName.release);

#endif

}

return osName;

}

//

// is64bitExe

//

// return 1 if a 64 bit .exe

// return 0 if a 32 bit .exe

//

int is64bitExe() {

return sizeof(size\_t) == 8;

}

//

// getPhysicalMemSz

//

UINT64 getPhysicalMemSz() {

#ifdef WIN32

UINT64 v;

GetPhysicallyInstalledSystemMemory(&v); // returns KB

return v \* 1024; // now bytes

#elif \_\_linux\_\_

return (UINT64) sysconf(\_SC\_PHYS\_PAGES)\* sysconf(\_SC\_PAGESIZE); // NB: returns bytes

#endif

}

//

// getNumberOfCPUs

//

int getNumberOfCPUs() {

#ifdef WIN32

SYSTEM\_INFO sysinfo;

GetSystemInfo(&sysinfo );

return sysinfo.dwNumberOfProcessors;

#elif \_\_linux\_\_

return (int) sysconf(\_SC\_NPROCESSORS\_ONLN); // {joj 12/1/18}

#endif

}

//

// cpu64bit

//

int cpu64bit() {

CPUID(cd, 0x80000001);

return (cd.edx >> 29) & 0x01;

}

//

// cpuFamily

//

int cpuFamily() {

CPUID(cd, 0x01);

return (cd.eax >> 8) & 0xff;

}

//

// cpuModel

//

int cpuModel() {

CPUID(cd, 0x01);

if (((cd.eax >> 8) & 0xff) == 0x06)

return (cd.eax >> 12 & 0xf0) + ((cd.eax >> 4) & 0x0f);

return (cd.eax >> 4) & 0x0f;

}

//

// cpuStepping

//

int cpuStepping() {

CPUID(cd, 0x01);

return cd.eax & 0x0f;

}

//

// cpuBrandString

//

char \*cpuBrandString() {

if (brandString)

return brandString;

brandString = (char\*) calloc(16\*3, sizeof(char));

CPUID(cd, 0x80000000);

if (cd.eax < 0x80000004) {

strcpy\_s(brandString, 16\*3, "unknown");

return brandString;

}

for (int i = 0; i < 3; i++) {

CPUID(cd, 0x80000002 + i);

UINT \*p = &cd.eax;

for (int j = 0; j < 4; j++, p++) {

for (int k = 0; k < 4; k++ ) {

brandString[i\*16 + j\*4 + k] = (char) ((\*p >> (k \* 8)) & 0xff); // {joj 12/1/18}

}

}

}

return brandString;

}

//

// rtmSupported (restricted transactional memory)

//

// NB: VirtualBox returns 0 even if CPU supports RTM?

//

int rtmSupported() {

CPUIDEX(cd, 0x07, 0);

return (cd.ebx >> 11) & 1; // test bit 11 in ebx

}

//

// hleSupported (hardware lock elision)

//

// NB: VirtualBox returns 0 even if CPU supports HLE??

//

int hleSupported() {

CPUIDEX(cd, 0x07, 0);

return (cd.ebx >> 4) & 1; // test bit 4 in ebx

}

//

// look for L1 cache line size (see Intel Application note on CPUID instruction)

//

int lookForL1DataCacheInfo(int v) {

if (v & 0x80000000)

return 0;

for (int i = 0; i < 4; i++) {

switch (v & 0xff) {

case 0x0a:

case 0x0c:

case 0x10:

return 32;

case 0x0e:

case 0x2c:

case 0x60:

case 0x66:

case 0x67:

case 0x68:

return 64;

}

v >>= 8;

}

return 0;

}

//

// getL1DataCacheInfo

//

int getL1DataCacheInfo() {

CPUID(cd, 2);

if ((cd.eax & 0xff) != 1) {

cout << "unrecognised cache type: default L 64" << endl;

return 64;

}

int sz;

if ((sz = lookForL1DataCacheInfo(cd.eax & ~0xff)))

return sz;

if ((sz = lookForL1DataCacheInfo(cd.ebx)))

return sz;

if ((sz = lookForL1DataCacheInfo(cd.ecx)))

return sz;

if ((sz = lookForL1DataCacheInfo(cd.edx)))

return sz;

cout << "unrecognised cache type: default L 64" << endl;

return 64;

}

//

// getCacheInfo

//

int getCacheInfo(int level, int data, int &l, int &k, int&n) {

CPUID(cd, 0x00);

if (cd.eax < 4)

return 0;

int i = 0;

while (1) {

CPUIDEX(cd, 0x04, i);

int type = cd.eax & 0x1f;

if (type == 0)

return 0;

int lev = ((cd.eax >> 5) & 0x07);

if ((lev == level) && (((data == 0) && (type = 2)) || ((data == 1) && (type == 1))))

break;

i++;

}

k = ((cd.ebx >> 22) & 0x03ff) + 1;

int partitions = ((cd.ebx) >> 12 & 0x03ff) + 1;

n = cd.ecx + 1;

l = (cd.ebx & 0x0fff) + 1;

return partitions == 1;

}

//

// getDeterministicCacheInfo

//

int getDeterministicCacheInfo() {

int type, ways, partitions, lineSz = 0, sets;

int i = 0;

while (1) {

CPUIDEX(cd, 0x04, i);

type = cd.eax & 0x1f;

if (type == 0)

break;

cout << "L" << ((cd.eax >> 5) & 0x07);

cout << ((type == 1) ? " D" : (type == 2) ? " I" : " U");

ways = ((cd.ebx >> 22) & 0x03ff) + 1;

partitions = ((cd.ebx) >> 12 & 0x03ff) + 1;

sets = cd.ecx + 1;

lineSz = (cd.ebx & 0x0fff) + 1;

cout << " " << setw(5) << ways\*partitions\*lineSz\*sets/1024 << "K" << " L" << setw(3) << lineSz << " K" << setw(3) << ways << " N" << setw(6) << sets;

cout << endl;

i++;

}

return lineSz;

}

//

// getCacheLineSz

//

int getCacheLineSz() {

CPUID(cd, 0x00);

if (cd.eax >= 4)

return getDeterministicCacheInfo();

return getL1DataCacheInfo();

}

//

// getPageSz

//

UINT getPageSz() {

#ifdef WIN32

SYSTEM\_INFO si;

GetSystemInfo(&si);

return si.dwPageSize;

#elif \_\_linux\_\_

return (UINT) sysconf(\_SC\_PAGESIZE); // {joj 12/1/18}

#endif

}

//

// getWallClockMS

//

UINT64 getWallClockMS() {

#ifdef WIN32

return (UINT64) clock() \* 1000 / CLOCKS\_PER\_SEC;

#elif \_\_linux\_\_

struct timespec t;

clock\_gettime(CLOCK\_MONOTONIC, &t);

return t.tv\_sec\*1000 + t.tv\_nsec / 1000000;

#endif

}

//

// setThreadCPU

//

void createThread(THREADH \*threadH, WORKERF, void \*arg) {

#ifdef WIN32

\*threadH = CreateThread(NULL, 0, worker, arg, 0, NULL);

#elif \_\_linux\_\_

pthread\_create(threadH, NULL, worker, arg);

#endif

}

//

// runThreadOnCPU

//

void runThreadOnCPU(UINT cpu) {

#ifdef WIN32

SetThreadAffinityMask(GetCurrentThread(), 1ULL << cpu);

#elif \_\_linux\_\_

cpu\_set\_t cpuset;

CPU\_ZERO(&cpuset);

CPU\_SET(cpu, &cpuset);

pthread\_setaffinity\_np(pthread\_self(), sizeof(cpu\_set\_t), &cpuset);

#endif

}

//

// closeThread

//

void closeThread(THREADH threadH) {

#ifdef WIN32

CloseHandle(threadH);

#elif \_\_linux\_\_

// nothing to do

#endif

}

//

// waitForThreadsToFinish

//

void waitForThreadsToFinish(UINT nt, THREADH \*threadH) {

#ifdef WIN32

WaitForMultipleObjects(nt, threadH, true, INFINITE);

#elif \_\_linux\_\_

for (UINT thread = 0; thread < nt; thread++)

pthread\_join(threadH[thread], NULL);

#endif

}

//

// pauseIfKeyPressed

//

void pauseIfKeyPressed() {

#ifdef WIN32

if (\_kbhit()) {

if (\_getch() == ' ') {

cout << endl << endl << "PAUSED - press key to continue";

\_getch();

cout << endl;

}

}

#elif \_\_linux\_\_

#endif

};

//

// pressKeyToContinue

//

void pressKeyToContinue() {

#ifdef WIN32

cout << endl << "Press any key to continue...";

\_getch();

#elif \_\_linux\_\_

termios old, input;

tcgetattr(fileno(stdin), &old); // save settings

input = old; // make new settings same as old settings

input.c\_lflag &= ~(ICANON | ECHO); // disable buffered i/o and echo

tcsetattr(fileno(stdin), TCSANOW, &input); // use these new terminal i/o settings now

puts("Press any key to continue...");

getchar();

tcsetattr(fileno(stdin), TCSANOW, &old);

#endif

}

//

// quit

//

void quit(int r) {

#ifdef WIN32

cout << endl << "Press key to quit...";

\_getch(); // stop DOS window disappearing prematurely

#endif

exit(r);

}

//

// rand

//

// due to George Marsaglia (google "xorshift wiki")

// NB: initial seed must NOT be 0

//

UINT64 rand(UINT64 &r) {

r ^= r >> 12; // a

r ^= r << 25; // b

r ^= r >> 27; // c

return r \* 2685821657736338717LL;

}

locale \*commaLocale = NULL;

//

// setCommaLocale

//

void setCommaLocale() {

if (commaLocale == NULL)

commaLocale = new locale(locale(), new CommaLocale());

cout.imbue(\*commaLocale);

}

//

// setLocale

//

void setLocale() {

cout.imbue(locale());

}

//

// getVMUse

//

size\_t getVMUse() {

size\_t r = 0;

#ifdef WIN32

HANDLE hProcess;

PROCESS\_MEMORY\_COUNTERS pmc;

if ((hProcess = OpenProcess(PROCESS\_QUERY\_INFORMATION | PROCESS\_VM\_READ, FALSE, GetCurrentProcessId()))) {

if (GetProcessMemoryInfo(hProcess, &pmc, sizeof(pmc)))

r = pmc.PagefileUsage;

CloseHandle(hProcess);

}

#elif \_\_linux\_\_

UINT64 vmuse;

FILE\* fp;

if ((fp = fopen("/proc/self/statm", "r")) != NULL) {

if (fscanf(fp, "%llu", &vmuse) == 1)

r = vmuse \* sysconf(\_SC\_PAGESIZE);

fclose(fp);

}

#endif

return r;

}

//

// getMemUse

//

size\_t getMemUse() {

size\_t r = 0;

#ifdef WIN32

HANDLE hProcess;

PROCESS\_MEMORY\_COUNTERS pmc;

if ((hProcess = OpenProcess(PROCESS\_QUERY\_INFORMATION | PROCESS\_VM\_READ, FALSE, GetCurrentProcessId()))) {

if (GetProcessMemoryInfo(hProcess, &pmc, sizeof(pmc)))

r = pmc.WorkingSetSize;

CloseHandle(hProcess);

}

#elif \_\_linux\_\_

UINT64 memuse;

FILE\* fp;

if ((fp = fopen("/proc/self/statm", "r")) != NULL) {

if (fscanf(fp, "%\*s%llu", &memuse) == 1)

r = memuse \* sysconf(\_SC\_PAGESIZE);

fclose(fp);

}

#endif

return r;

}

// eof

//

// helper.h

//

// Copyright (C) 2011 - 2018 jones@scss.tcd.ie

//

#pragma once

#ifdef WIN32

#include <tchar.h> //

#include <Windows.h> //

#include <intrin.h> // intrinsics

#endif

#if \_\_linux\_\_

#include <math.h> // log10

#include <cpuid.h> // cpuid

#include <string.h> // strcpy

#include <pthread.h> // pthread\_create

#include <x86intrin.h> // need to specify gcc flags -mrtm -mrdrnd

#include <limits.h> //s

#endif

#include <iomanip> //

#include <locale> //

//

// INT64 and UINT64

//

typedef long long INT64;

typedef unsigned long long UINT64;

#define INT64MIN LLONG\_MIN

#define INT64MAX LLONG\_MAX

#define UINT64MAX ULLONG\_MAX

#define STRTOVINT(s, c, r) \_strtoi64(s, c, r)

#define STRTOVUINT(s, c, r) \_strtoui64(s, c, r)

#define AMALLOC(sz, align) \_aligned\_malloc(((sz) + (align)-1) / (align) \* (align), align)

#define AFREE(p) \_aligned\_free(p)

#ifdef WIN32

#define CPUID(cd, v) \_\_cpuid((int\*) &cd, v);

#define CPUIDEX(cd, v0, v1) \_\_cpuidex((int\*) &cd, v0, v1)

#define THREADH HANDLE

#define WORKERF DWORD (WINAPI \*worker) (void\*)

#define WORKER DWORD WINAPI

#define ALIGN(n) \_\_declspec(align(n))

#define TLSINDEX DWORD

#define TLSALLOC(key) key = TlsAlloc()

#define TLSSETVALUE(tlsIndex, v) TlsSetValue(tlsIndex, v) // {joj 24/11/15}

#define TLSGETVALUE(tlsIndex) TlsGetValue(tlsIndex) // {joj 24/11/15}

#define thread\_local \_\_declspec(thread)

#elif \_\_linux\_\_

#define BYTE unsigned char

#define UINT unsigned int

#define INT64 long long

#define UINT64 unsigned long long

#define LONG64 signed long long

#define PVOID void\*

#define MAXINT INT\_MAX

#define MAXUINT UINT\_MAX

#define MAXUINT64 ((UINT64)~((UINT64)0))

#define MAXINT64 ((INT64)(MAXUINT64 >> 1))

#define MININT64 ((INT64)~MAXINT64)

#define CPUID(cd, v) \_\_cpuid(v, cd.eax, cd.ebx, cd.ecx, cd.edx);

#define CPUIDEX(cd, v0, v1) \_\_cpuid\_count(v0, v1, cd.eax, cd.ebx, cd.ecx, cd.edx)

#define THREADH pthread\_t

#define GetCurrentProcessorNumber() sched\_getcpu()

#define WORKER void\*

#define WORKERF void\* (\*worker) (void\*)

#define ALIGN(n) \_\_attribute\_\_ ((aligned (n)))

#define \_aligned\_malloc(sz, align) aligned\_alloc(align, ((sz)+(align)-1)/(align)\*(align))

#define \_aligned\_free(p) free(p)

#define \_alloca alloca

#define strcpy\_s(dst, sz, src) strcpy(dst, src)

#define \_strtoi64(str, end, base) strtoll(str, end, base)

#define \_strtoui64(str, end, base) strtoull(str, end, base)

#define \_InterlockedIncrement(addr) \_\_sync\_fetch\_and\_add(addr, 1)

#define \_InterlockedIncrement64(addr) \_\_sync\_fetch\_and\_add(addr, 1)

#define \_InterlockedExchange(addr, v) \_\_sync\_lock\_test\_and\_set(addr, v)

#define \_InterlockedExchangePointer(addr, v) \_\_sync\_lock\_test\_and\_set(addr, v)

#define \_InterlockedExchangeAdd(addr, v) \_\_sync\_fetch\_and\_add(addr, v)

#define \_InterlockedExchangeAdd64(addr, v) \_\_sync\_fetch\_and\_add(addr, v)

#define \_InterlockedCompareExchange(addr, newv, oldv) \_\_sync\_val\_compare\_and\_swap(addr, oldv, newv)

#define \_InterlockedCompareExchange64(addr, newv, oldv) \_\_sync\_val\_compare\_and\_swap(addr, oldv, newv)

#define \_InterlockedCompareExchange64\_HLERelease(addr, newv, oldv) \_\_sync\_val\_compare\_and\_swap(addr, oldv, newv, \_\_ATOMIC\_ACQUIRE | \_\_ATOMIC\_HLE\_ACQUIRE) // {joj 12/1/18}

#define \_InterlockedCompareExchangePointer(addr, newv, oldv) \_\_sync\_val\_compare\_and\_swap(addr, oldv, newv)

#define \_InterlockedExchange\_HLEAcquire(addr, val) \_\_atomic\_exchange\_n(addr, val, \_\_ATOMIC\_ACQUIRE | \_\_ATOMIC\_HLE\_ACQUIRE)

#define \_InterlockedExchangeAdd64\_HLEAcquire(addr, val) \_\_atomic\_exchange\_n(addr, val, \_\_ATOMIC\_ACQUIRE | \_\_ATOMIC\_HLE\_ACQUIRE)

#define \_Store\_HLERelease(addr, v) \_\_atomic\_store\_n(addr, v, \_\_ATOMIC\_RELEASE | \_\_ATOMIC\_HLE\_RELEASE)

#define \_Store64\_HLERelease(addr, v) \_\_atomic\_store\_n(addr, v, \_\_ATOMIC\_RELEASE | \_\_ATOMIC\_HLE\_RELEASE)

#define \_mm\_pause() \_\_builtin\_ia32\_pause()

#define \_mm\_mfence() \_\_builtin\_ia32\_mfence()

#define TLSINDEX pthread\_key\_t

#define TLSALLOC(key) pthread\_key\_create(&key, NULL)

#define TLSSETVALUE(key, v) pthread\_setspecific(key, v)

#define TLSGETVALUE(key) (size\_t) pthread\_getspecific(key)

#define thread\_local \_\_thread //

#define Sleep(ms) usleep((ms)\*1000) //

#endif

extern UINT ncpu; // # logical CPUs

extern void getDateAndTime(char\*, int, time\_t = 0); // getDateAndTime

extern char\* getHostName(); // get host name

extern char\* getOSName(); // get OS name

extern int getNumberOfCPUs(); // get number of CPUs

extern UINT64 getPhysicalMemSz(); // get RAM sz in bytes

extern int is64bitExe(); // return 1 if 64 bit .exe

extern size\_t getMemUse(); // get working set size

extern size\_t getVMUse(); // get page file usage

extern UINT64 getWallClockMS(); // get wall clock in milliseconds from some epoch

extern void createThread(THREADH\*, WORKERF, void\*); //

extern void runThreadOnCPU(UINT); // run thread on CPU

extern void waitForThreadsToFinish(UINT, THREADH\*); //

extern void closeThread(THREADH); //

extern UINT64 rand(UINT64&); //

#define RDRANDSTEP(r) \_rdrand64\_step(r) // {joj 28/11/15}

extern int cpu64bit(); // return 1 if CPU is 64 bit

extern int cpuFamily(); // CPU family

extern int cpuModel(); // CPU model

extern int cpuStepping(); // CPU stepping

extern char \*cpuBrandString(); // CPU brand string

extern int rtmSupported(); // return 1 if RTM supported (restricted transactional memory)

extern int hleSupported(); // return 1 if HLE supported (hardware lock elision)

extern int getCacheInfo(int, int, int &, int &, int&); // getCacheInfo

extern int getCacheLineSz(); // get cache line sz

extern UINT getPageSz(); // get page size

extern void pauseIfKeyPressed(); // pause if key pressed

extern void pressKeyToContinue(); // press key to continue

extern void quit(int = 0); // quit

//

// CommaLocale

//

// VS2015 BUG doesn't output comma thousands separator in RELEASE mode {joj 25/11/15}

//

class CommaLocale : public std::numpunct<char> {

protected:

virtual char do\_thousands\_sep() const { return ','; }

virtual std::string do\_grouping() const {return "\03"; }

};

extern void setCommaLocale();

extern void setLocale();

// eof