MULTICORE PROGRAMMING

Hardware transactional memory, and TLE

Lecture 13

Trevor Brown

ANNOUNCEMENTS

- Memory reclamation: last time was hopefully **enough detail for A5**, but maybe **not** enough to fully understand the limitations of EBR in complex scenarios
- Would like to talk more about memory reclamation, but a bit later…

USEFULNESS OF KCAS

- KCAS is an awesome tool, but it doesn't solve **everything**
- It makes it **easy** to change multiple addresses atomically (and with lock-free progress)
  - Locks do this too (but without lock-freedom)
  - Implement KCAS with locks, if you like (can also be fast). surprisingly tricky to get right, though… more later…
- It **does not** make it trivial to argue searches work
  - And searches are part of updates!
  - So, we still need some ad-hoc correctness arguments for both searches and updates!
- Question: how to get **fast** data structures with easy/trivial proofs for searches (as well as the search/traversal part of updates)?

Lock-based algorithms that **do not** lock while searching also have this **same challenge**: proving correctness for searches is hard.

This is why I think lock-free algorithms and fast lock-based ones are somewhat similar…

THIS TIME

- A technology that can help with correctness arguments for searches
  - Implemented in some modern hardware
    - Many recent Intel CPUs
    - IBM POWER8+ (IMO, not as good as Intel’s implementation)
    - ARMv8 (haven’t used it yet…)
  - Can also be used to greatly accelerate some algorithms such as KCAS
    - And can even accelerate lock-based algorithms

TRANSACTIONAL MEMORY (TM)

- Allows a programmer to perform arbitrary blocks of code atomically
- Note: locks also do this (just not always efficiently or easily)

```
bool transfer(int *src, int *dst, int amt)
    bool result = false;
    atomic {
        if (*src > amt) {
            *src -= amt;
            *dst += amt;
            result = true;
        }
    }
    return result;
```

DEFINITIONS

- Each transaction **commits** or **aborts**
  - **Commit:** as if the entire transaction happened atomically
  - **Abort:** as if the transaction never happened at all
- **Read-set (at time t)**: the set of all addresses read by a transaction (up to time t)
- **Write-set (at time t):** the set of all addresses written by a transaction (up to time t)
- **Data-set**: (read-set) + (write-set)
- **Data conflicts**: two concurrent transactions have a **data conflict** if the write-set of one intersects the data-set of the other (examples soon)

TRANSACTIONAL OPERATIONS

- Studying Intel's hardware implementation of TM
- xbegin: start a new transaction and return XSTARTED
- xend: try to commit the transaction (might abort instead)
- xabort: abort the transaction
- read *addr: Read & add addr to the transaction's read-set (in L3 cache)
- write *addr = val: Write & add addr to the transaction's write-set (in L1 cache)

Note: **xbegin**, **xend**, **xabort** are actual x86/64 assembly instructions
Instruction set: TSX-NI / RTM (provided in several modern Intel chips)

HIGH LEVEL IDEA

- Transaction works sort of like a lock, but can abort

```
bool transfer(int *src, int *dst, int amt)
1   bool result = false;
2   xbegin();
3       if (*src > amt) {
4           *src -= amt;
5           *dst += amt;
6           result = true;
7       }
8   xend();
9   return result;
```

Work might **not** be done because of an abort

Must **handle** aborts somehow

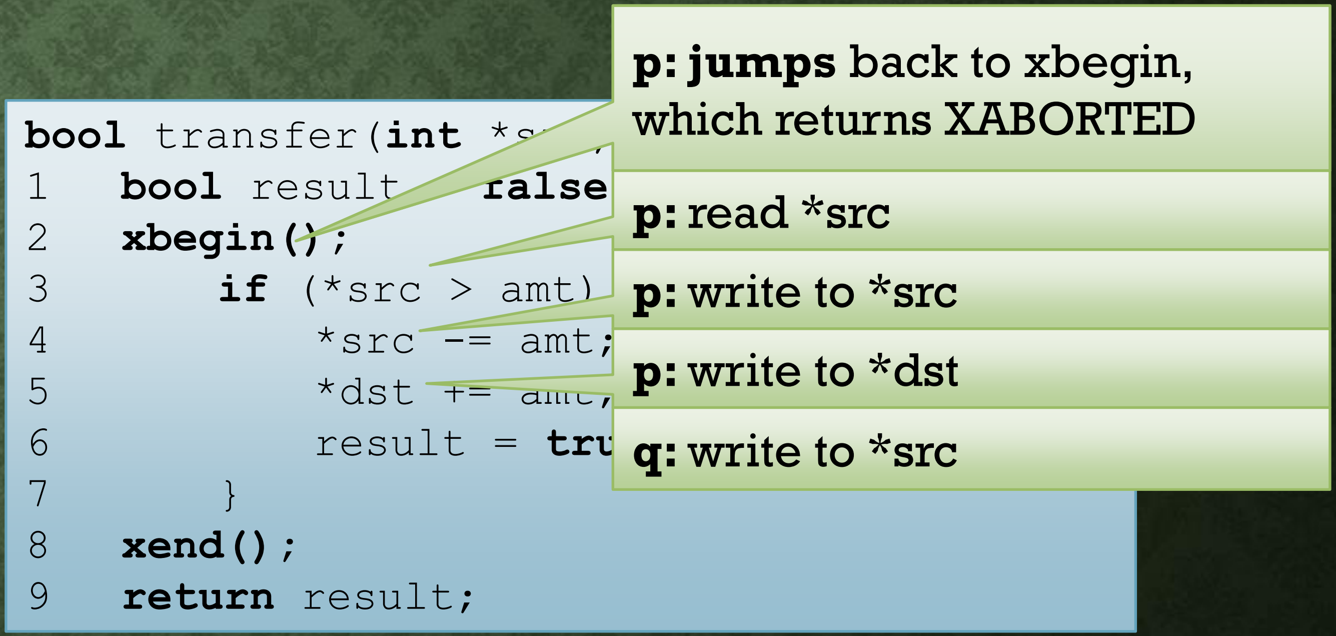
- Suppose thread p reads *src at line 3,
  then thread q subsequently modifies *src
  - This causes a **data conflict**, which will cause p’s transaction to **abort** (since its view of memory is no longer atomic)

A BIT MORE DETAIL ON INTEL'S HARDWARE TM (HTM)

- Threads can execute **transactions** that read/write/CAS/F&A any addresses
- They can also read/write/CAS/F&A addresses **non-transactionally** (as usual)
- Transactions abort as soon as there is a data conflict
  - Data conflicts can be between two transactions,
    or between a transaction and a thread that performs **non-transactional** accesses
  - Suppose a transaction T **accesses** (read/write/CAS/F&A) an address,
    and then before T commits, a different thread p **modifies** (write/CAS/F&A) that address.
    Whether p's modification is inside a transaction or not, **T will abort immediately.**
  - Suppose a transaction T **modifies** an address, and then before T commits,
    a different thread p **reads** that address.
    If p's read is not performed inside a transaction, then **T will abort immediately.**
- **Moreover, transactions *can* abort at any time, for any reason!**

WHAT HAPPENS WHEN A TRANSACTION ABORTS

- When a transaction aborts, the thread **jumps** to its last **xbegin**, and this **xbegin** returns **XABORTED**



```
bool transfer(int *s
  bool result = false
  xbegin();
    if (*src > amt)
      *src -= amt;
      *dst += amt;
      result = tru
    }
  xend();
  return result;
```

p: jumps back to xbegin, which returns XABORTED

p: read *src

p: write to *src

p: write to *dst

q: write to *src

Note: in practice, **p** will even **abort** if **q** writes to the same **cache line** (or even sometimes the **adjacent cache line**, i.e., the other member of a 128b aligned pair of cache lines); very careful padding is advised!

HANDLING ABORTS

- **Branch** based on the return value of **xbegin**
- Handle abort in else case
  - Useful to record # of aborts, debug, change code behaviour, etc.
- Usually desirable to retry aborted transactions
  - Often want to **wait a bit** before retrying…

```
bool transfer(int *src, int *dst, int amt)
    bool result = false;
    retry:
    if (xbegin() == XSTARTED) {
        if (*src > amt) {
            *src -= amt;
            *dst += amt;
            result = true;
        }
        xend();
    } else { // we aborted
        handleTheAbort();
        goto retry;
    }
    return result;
```

FIRST ATTEMPT: TRANSACTIONAL HASH TABLE

```
int sequentialInsert(int key)
  int h = hash(key);
  for (int i=0;i<capacity;++i) {
  |int index = (h+i) % capacity;
  |int found = data[index];
  |if (found == key) {
  |  return false;
  |} else if (found == NULL) {
  |  data[index] = key;
  |  return true;
  |}
  }
  return FULL;
```

```
int insert(int key)
  retry:
  if (xbegin() == XSTARTED) {
   int result = sequentialInsert(key);
   xend();
   return result;
  } else {
   // transaction aborted
   goto retry;
  }
```

But there’s a problem with this implementation…

THE PROBLEM WITH HTM

- Transactions can abort for any reason
  - No progress guarantee!
- Not hard to write code in which **all** transactions abort forever
  - Example: if a transaction causes a page fault, it will execute the page fault handler inside a transaction, and this tends to abort, reverting any progress towards loading the page!
  - So, the transaction retries, and aborts again! And again! …
- Need to provide a **fallback code path** (for example, using locks) to run when a transaction aborts too many times
  - Two code paths: fast path (using HTM), fallback path

TRANSACTIONAL LOCK ELISION (TLE)

- TLE uses the simplest and most common choice of fallback code path:
  - **Acquire a global lock** then execute the transaction's code (**without** xbegin/xend)
- Transactions on the fast path should **not** run while the global lock is held
  - This prevents transactions from changing data that the global lock is supposed to protect
  - So, on the fast path, each transaction **reads** the **lock state**
  - If it is locked, the transaction aborts, and the thread waits until the lock is free to **try again**
  - If it is not locked, the transaction proceeds
    - If the lock is acquired at **any time** during the transaction, this will be a **data conflict**, and the transaction will abort!

Crucial point: transactions only need to **read** the lock to ensure that the operation succeeds only if **no one else** holds the lock.

Without HTM, you would need to **acquire** the lock to guarantee no one else holds it when you change the data structure.

EXAMPLE: TLE-BASED HASH TABLE

```
int insert(int key)
  int retriesLeft = 5;
  retry:
  if (xbegin() == XSTARTED) {
    if (locked) xabort();
    int result = sequentialInsert(key);
    xend();
    return result;
  } else {
    // transaction aborted
    while (locked) { /* wait */ }
    if (--retriesLeft > 0) goto retry;
    acquire(&locked);
    int result = sequentialInsert(key);
    release(&locked);
    return result;
  }
```

Fast path (transactions)

Fallback path (global lock)

Why does TLE work?

What do we know about a (fast path) transaction that commits?

If it read an address, and then that address was later changed, the txn would have aborted.

So, the transaction's behaviour is the **same** as it would be if it had actually **acquired** the global lock (since it observed no changes during its execution)!

What about the fallback path?

We hold the global lock, so no one else can access anything. Equivalent to running in a single threaded system.