MULTICORE PROGRAMMING

Advanced usage of HTM

Lecture 14

Trevor Brown

LAST TIME

- Transactional memory (TM)
- Intel's restricted transactional memory (RTM / HTM / TSX-NI)
- Transactional lock elision (TLE)
  - Hash table

THIS TIME

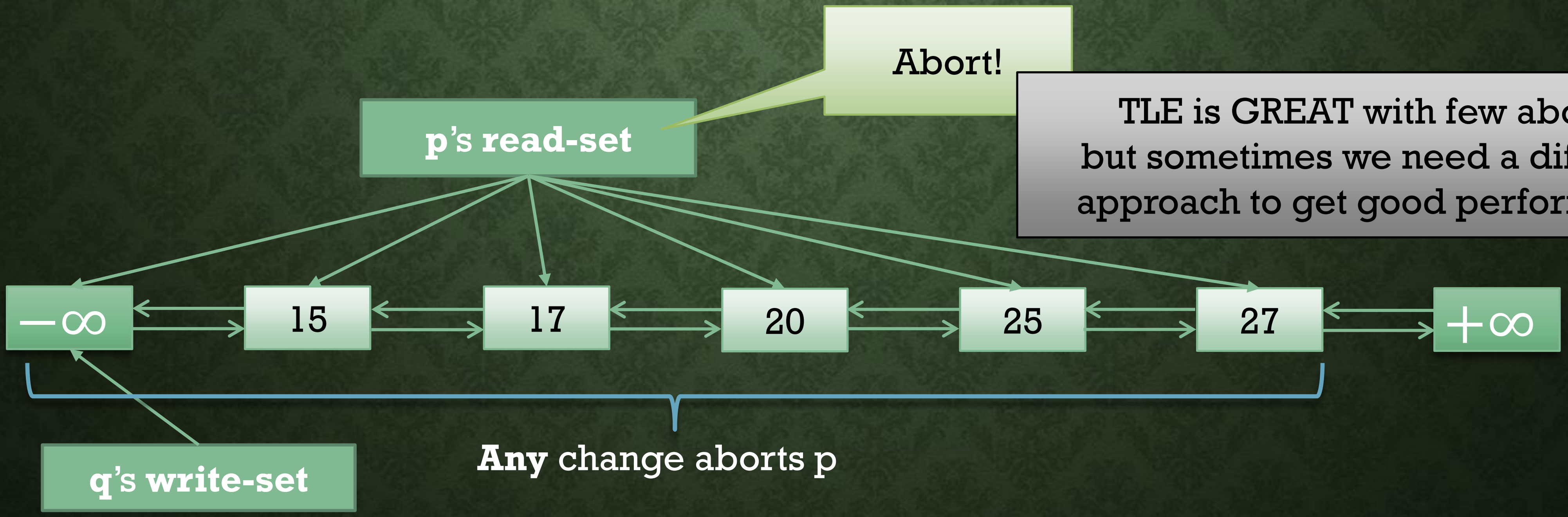
- When does TLE perform well? Poorly?
- More sophisticated uses of hardware transactional memory (HTM)
  - **Accelerating** lock-free KCAS

SOMETIMES TLE PERFORMS POORLY

- Lists: when a thread is traversing the list, **a huge prefix of the list** is in its read-set!

Thread **p**:
Search(27)

Thread **q**:
Insert(6)



TLE is GREAT with few aborts,
but sometimes we need a different
approach to get good performance

PERFORMANCE PROBLEMS WITH TLE

- Traversals are performed inside the transaction
  - Usually fine for trees and hash tables, where threads naturally spread out
  - But **terrible** for lists, where many threads follow the same path

Use another technique (such as KCAS)

- Global locking fallback path kills scalability when aborts are common
  - **Different** fallback path?
- What if we use HTM to **accelerate** existing concurrent algorithms like KCAS?
  - Does **not** help with correctness / progress arguments
  - But can obtain big performance benefits!

USING HTM TO ACCELERATE LOCK-FREE KCAS

- Goal
  - HTM-based KCAS that uses **lock-free KCAS as the fallback path**
  - Fast path transactions should be able to run **concurrently** with the fallback path!
- Approach
  - Fast path algorithm:
    - Wrap KCAS in a transaction: xstart ; KCAS ; xend
    - Now each KCAS on the fast path is atomic, just because it is in a transaction
    - Some parts of the algorithm are no longer needed because of the transaction
      - Example: DCSS is not needed – could just do two reads and a write in the transaction!
    - Get rid of parts of the algorithm that are unnecessary

STEP 1: ADDING TRANSACTIONS

Recall:
(lock-free KCAS)

```
bool KCAS_LF(addr1..., exp1..., new1...)
KCAS_desc * d = new KCAS_desc(addr1...);
d->status = Undecided;
SortRowsByAddress(d);
return KCASHelp(d);
```

```
bool KCAS(addr1..., exp1..., new1...)
int retries = 5;
retry:
if (xbegin() == XSTARTED) {
  bool result = KCAS_TXN(addr1..., exp1..., new1...);
  xend();
  return result;
} else {
  if (--retries > 0) goto retry;
  return KCAS_LF(addr1..., exp1..., new1...);
}
```

Fast path: lock-free KCAS code inside a transaction

Fallback path: lock-free KCAS code

(We still use the same old KCASRead)

KCAS_TXN is initially **the same as KCAS_LF**. We will optimize it.

Ideally we probably want some sort of **waiting** before we retry…

OPTIMIZING KCAS_TXN

```
bool KCAS_TXN(addr1..., exp1..., new1...)
KCAS_desc * d = new KCAS_desc(addr1...);
d->status = Undecided;
SortRowsByAddress(d);
return KCASHelp(d);
```

Inline this help function so we can modify it here (and not affect the lock-free KCAS code)

```
bool KCAS_TXN(addr1..., exp1..., new1...)
KCAS_desc * d = new KCAS_desc(addr1...);
d->status = Undecided;
SortRowsByAddress(d);
if (d->status == Undecided)
| int newStatus = Succeeded;
| for (int i = 0; i < d->n; i++)
| | word_t val2 = DCSS(&d->status, d->row[i].addr,
| | |                  Undecided,  d->row[i].exp,
| | |                              packKCAS(d));
| | if (val2 != d->row[i].exp)    // if DCSS failed
| | | if (isKCAS(val2))           // because of a KC
| | |   if (unpack(val2) != d)    // a DIFFERENT KCA
| | |     KCASHelp(unpack(val2));
| | |     --i; continue; // retry "locking" this ad
| | | else // addr does not contain its exp value
| | |   newStatus = Failed; break;
| CAS(&d->status, Undecided, newStatus);
bool succ = (d->status == Succeeded);
for (int i = 0; i < d->n; i++)
| val = (succ) ? d->row[i].new : d->row[i].exp;
| CAS(d->row[i].addr, packKCAS(d), val);
return succ;
```

Status is always Undecided here

DCSS: change addr from exp to my KCAS descriptor, only if my descriptor has status Undecided.

Phase 1: lock-free "locking"

Status CAS

Phase 2: completion

Can any other thread access my KCAS descriptor?

Only if I store a pointer to it and commit (xend)!

I never do that… Before I return, I always CAS each address to the new value, or back to the expected value…

If no one can see my descriptor, why create it at all?

```
bool KCAS_TXN(addr1..., exp1..., new1...)

  SortByAddress(addr1..., exp1..., new1...);

  for (int i = 0; i < d->n; i++)
  | word_t val2 = DCSS(&d->status, d->row[i].addr,
  | |                   Undecided,  d->row[i].exp,
  | |                               packKCAS(d));
  | if (val2 != d->row[i].exp)    // if DCSS failed
  | | if (isKCAS(val2))           // because of a KCAS
  | |   if (unpack(val2) != d)    // a DIFFERENT KCAS
  | |     KCASHelp(unpack(val2));
  | |     --i; continue; // retry "locking" this addr
  | | else // addr does not contain its exp value
  | |   newStatus = Failed; break;
  CAS(&d->status, Undecided, newStatus);
  bool succ = (d->status == Succeeded);
  for (int i = 0; i < d->n; i++)
  | val = (succ) ? d->row[i].new : d->row[i].exp;
  | CAS(d->row[i].addr, packKCAS(d), val);
  return succ;
```

Now that we have no descriptor pointer to store. This becomes a READ.

```
bool KCAS_TXN(addr1..., exp1..., new1...)


  SortByAddress(addr1..., exp1..., new1...);


  for (int i = 0; i < d->n; i++)
  | word_t val2 = *addri;
  |
  |
  | if (val2 != d->row[i].exp)    // if DCS
  | | if (isKCAS(val2))           // h        AS
  | |   if (unpack(val2) != d)    // a        CAS
  | |     KCASHelp(unpack(val2));
  | |     --i; continue; // retry "locking" this addr
  | | else // addr does not contain its exp value
  | |   newStatus = Failed; break;
  CAS(&d->status, Undecided, newStatus)
  bool succ = (d->status == Succeeded)
  for (int i = 0; i < d->n; i++)
  | val = (succ) ? d->row[i].new : d->row[i].exp;
  | CAS(d->row[i].addr, packKCAS(d), val);
  return succ;
```

Descriptor d does not exist.
Must fix references to it.

```
bool KCAS_TXN(addr1..., exp1..., new1...)


SortByAddress(addr1..., exp1..., new1...);


for (int i = 0; i < n; i++)
| word_t val2 = *addri;
|
|
| if (val2 != expi)                // if DC
| | if (isKCAS(val2))              // because of a KCAS
| |   if (unpack(val2) != d)       // a DIFFERENT KCAS
| |     KCASHelp(unpack(val2));
| |     --i; continue; // retry "locking" this addr
| | else // addr does not contain its exp value
| |   newStatus = Failed; break;
CAS(&d->status, Undecided, newStatus);
bool succ = (d->status == Succeeded);
for (int i = 0; i < n; i++)
| val = (succ) ? d->row[i].new : d->row[i].exp;
| CAS(d->row[i].addr, packKCAS(d), val);
return succ;
```

Wherever we got n from to put in the descriptor, we pass it to functions etc., to make it available here

Since d does not exist, this if-statement always evaluates to true! Kill it.

```
bool KCAS_TXN(addr1..., exp1..., new1...)


  SortByAddress(addr1..., exp1..., new1...);


  for (int i = 0; i < n; i++)
  | word_t val2 = *addri;
  |
  |
  | if (val2 != expi)             // if
  | | if (isKCAS(val2))           // be
  | |
  | |   KCASHelp(unpack(val2));
  | |   --i; continue; // retry "
  | | else // addr does not co     ain it
  | |   newStatus = Failed; break;
  CAS(&d->status, Undecided, newStatus);
  bool succ = (d->status == Succeeded);
  for (int i = 0; i < n; i++)
  | val = (succ) ? d->row[i].new : d->row[i].exp;
  | CAS(d->row[i].addr, packKCAS(d), val);
  return succ;
```

Small optimization: why not **abort** instead of **commit**?
We want to **return false**, and xabort will move our program counter back to the last xbegin immediately without doing any writes to shared memory…
Could we make xabort work?

If we get here, KCAS will return false.
Any further steps are simply done to roll back previous changes.
But we haven't made any changes!
Just **return false (and commit)**!

```
bool KCAS_TXN(addr1..., exp1..., new1...)



  SortByAddress(addr1..., exp1..., new1...);


  for (int i = 0; i < n; i++)
  | word_t val2 = *addri;
  |
  |
  | if (val2 != expi)                // if DCSS failed
  | | if (isKCAS(val2))              // because of a KCAS
  | |
  | |   KCASHelp(unpack(val2));
  | |   --i; continue; // retry "locking" this addr
  | | else // addr does not contain its exp value
  | |   return false;
  CAS(&d->status, Undecided, newStatus);
  bool succ = (d->status == Succeeded);
  for (int i = 0; i < n; i++)
  | val = (succ) ? d->row[i].new : d->row[i].exp;
  | CAS(d->row[i].addr, packKCAS(d), val);
  return succ;
```

d->status does not exist

```
bool KCAS_TXN(addr1..., exp1..., new1...)


  SortByAddress(addr1..., exp1..., new1...);


  for (int i = 0; i < n; i++)
  | word_t val2 = *addri;
  |
  |
  | if (val2 != expi)              // if DCSS failed
  | | if (isKCAS(val2))            // because of a
  | |
  | |   KCASHelp(unpack(val2));
  | |   --i; continue; // retry "locking" this a
  | | else // addr does not contain its exp val
  | |   return false;
  | |
  | |
  for (int i = 0; i < n; i++)
  | val = (succ) ? d->row[i].new : d->row[i].exp;
  | CAS(d->row[i].addr, packKCAS(d), val);
  return succ;
```

If we are here, we saw all of our expected values.
No need to test for success or store expected values. We haven't stored anything yet!
Just store new values!

Also fix references to **d**

```
bool KCAS_TXN(addr1..., exp1..., new1...)


  SortByAddress(addr1..., exp1..., new1...);


  for (int i = 0; i < n; i++)
  | word_t val2 = *addri;
  |
  |
  | if (val2 != expi)              // if DCSS failed
  | | if (isKCAS(val2))            // because of a KCAS
  | |
  | |   KCASHelp(unpack(val2));
  | |   --i; continue; // retry "locking" this addr
  | | else // addr does not contain its exp value
  | |   return false;
  | |
  | |
  for (int i = 0; i < n; i++)
  |
  | CAS(addri, expi, newi);
  return succ;
```

No need for CAS. We only got here because **addri** contains its expected value. If that changes, we are aborted!
We can just **write**!

```
bool KCAS_TXN(addr1..., exp1..., new1...)


  SortByAddress(addr1..., exp1..., new1...);


  for (int i = 0; i < n; i++)
  | word_t val2 = *addri;
  |
  |
  | if (val2 != expi)                   // if DCSS failed
  | | if (isKCAS(val2))                 // because of a KCAS
  | |
  | |   KCASHelp(unpack(val2));
  | |   --i; continue; // retry "locking" this addr
  | | else // addr does not contain its exp value
  | |   return false;
  | |
  | |
  for (int i = 0; i < n; i++)
  |
  | *addri = newi;
  return succ;
```

If we get here, we succeeded.
Just return true.

CLEANING UP WHITE SPACE / COMMENTS

```
bool KCAS_TXN(addr1..., exp1..., new1...)
SortByAddress(addr1..., exp1..., new1...);
for (int i = 0; i < n; i++)
| word_t val2 = *addri;
| if (val2 != expi)    // if we see a non-expected val
| | if (isKCAS(val2)) // --that is a KCAS descriptor
| |   KCASHelp(unpack(val2)); // unpack & help it
| |   --i; continue; // retry "locking" this addr
| | else // addr contain a non-expected program val
| |   return false;
for (int i = 0; i < n; i++)
| *addri = newi;
return true;
```

Seems implausible that we will get to retry “locking” this addr (by reading it). Aren’t we likely to get aborted by then?

HELPING AND TRANSACTIONS

- Helping involves touching data other threads are working on (data conflicts!!)
- Transactions that help non-transactional operations
  - If you read some data, and someone else writes to it, your transaction will abort
  - They are highly likely to write to data you've read,
    since you have found them in the middle of their operation
- Non-transactional operations helping transactions
  - If you perform a write that a transaction is trying to do also, two cases arise:
  - (a) you write after the transaction commits, and you didn't really help
  - (b) you write before the transaction commits, and it must abort
- Transactions helping transactions
  - No. Just no.

WHY DO WE HELP AT ALL?

- To guarantee lock-free progress:
  - ***Some*** operation always completes in the future
- How much helping is needed to guarantee progress in our algorithm?
  - What if transactions don't help, and we don't help them?
  - Suppose all transactions abort (so they do not make progress)
  - Then all operations go to their fallback code paths, and run lock-free code
  - This lock-free code **guarantees progress**

REMOVING TRANSACTIONAL HELPING

```
bool KCAS_TXN(addr1..., exp1..., new1...)
SortByAddress(addr1..., exp1..., new1...);
for (int i = 0; i < n; i++)
| word_t val2 = *addri;
| if (val2 != expi)    // if we see a non-expected val
| | if (isKCAS(val2)) // --that is a KCAS descriptor
| |   KCASHelp(unpack(val2)); // unpack & help it
| |   --i; continue; // retry "locking" this addr
| | else // addr contain a non-expected program val
| |   return false;
for (int i = 0; i < n; i++)
| *addri = newi;
return true;
```

Note: could even remove this sorting as a fast-path optimization!

Instead of helping, just assume we will get aborted, and issue our own explicit **xabort**. (After this we'll **try again**.)

FINAL KCAS_TXN IMPLEMENTATION

```
bool KCAS_TXN(addr1..., exp1..., new1...)
SortByAddress(addr1..., exp1..., new1...);
for (int i = 0; i < n; i++)
| word_t val2 = *addri;
| if (val2 != expi)    // if we
| | if (isKCAS(val2)) // --that
| |   xabort();        // give up
| | else // addr contain a non-e
| |   return false;
for (int i = 0; i < n; i++)
| *addri = newi;
return true;
```

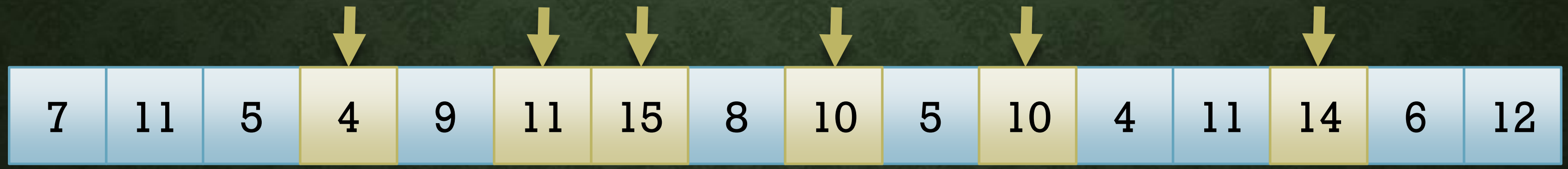
Step 1: Sort args by address

Step 2: Read all addresses and check if they contain their expected values.
If an address contains a non-expected program value, return false.
If we encounter a KCAS descriptor, abort (and retry)

Step 3: Write new values and return true

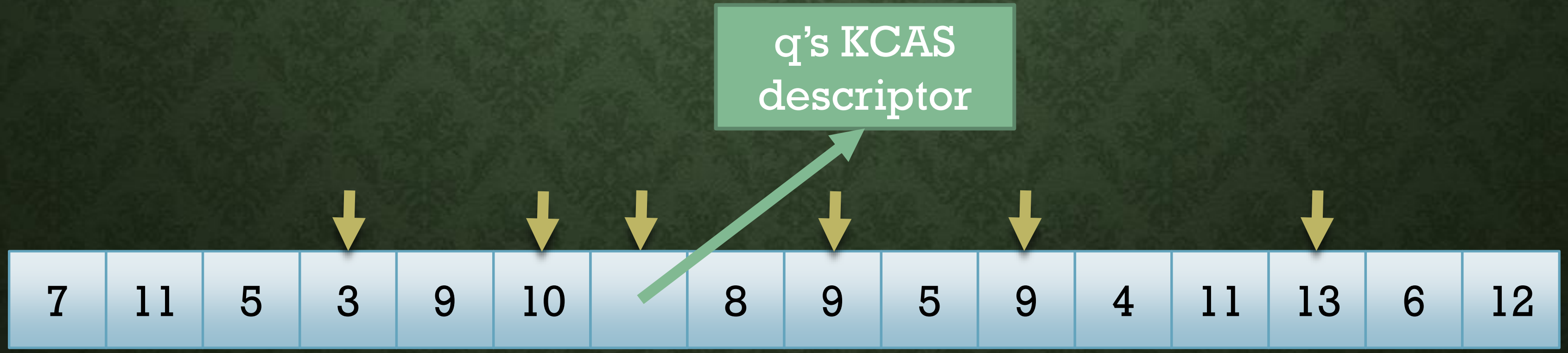
EXAMPLE EXECUTION 1

- Consider an execution where KCAS is used to increment cells in an array
- Suppose thread p runs on the fallback path, and "lock-free locks" k addresses
- Then thread q runs on the fast path and reads one of these addresses
- Thread q sees a pointer to p's KCAS descriptor and **aborts**
- Thread p then completes its KCAS
- Thread q can then retry and perform its KCAS



EXAMPLE EXECUTION 2

- Consider an execution where KCAS is used to increment cells in an array
- Suppose thread p runs on the fast path,
  reads all k addresses, and sees the expected values
- Before p commits, thread q runs on the fallback path
  and uses CAS to store a descriptor pointer in one of these addresses
- Thread p will be immediately **aborted** by the HTM system due to a **data conflict**



CORRECTNESS ARGUMENT INTUITION

- For simplicity, consider a system with two threads
- Imagine two operations running on the fallback path
  - Both behave correctly because the lock-free algorithm is correct
- Two operations on the fast path
  - Correct because both are atomic, because of transactional memory
- One operation on the fast path and one operation on the fallback path
  - Claim: the fast path operation does **not** modify addresses while they are "lock-free locked" by the fallback path operation
  - I.e., fast path **respects** the "lock-free locks" taken by the fallback path

MECHANICS OF PROVING CORRECTNESS

- Correctness of each path in isolation:
  - Fallback path is correct in isolation
  - Fast path is atomic because of transactions, and correct in isolation
- Compatibility between paths:
  - Fast path was obtained from fallback by wrapping it in a transaction
    (which makes it atomic)
    and then performing correctness-preserving transformations

To be rigorous, one option is to start with a correct lock-free algorithm, and prove that each transformation **preserves** correctness

USING HTM TO IMPLEMENT SYNCHRONIZATION PRIMITIVES LIKE KCAS

- Advantages
  - Programmer only needs to write one code path
    (fast path & fallback path are hidden in the KCAS implementation)
  - Hides the complexity of proving correctness
    for interactions between fast path & fallback path
  - Makes it practical to design / accelerate data structures with KCAS
    (should result in great performance)
  - Code still works on systems with **no** HTM (just run the fallback path)
- Disadvantages
  - Still need to prove correctness for searches
  - Minor: must use KCASRead to read

SUMMARIZING

- We can use TLE to make designing **new** data structures **easy**
- We can use advanced HTM-based techniques to make **existing** data structures **faster**
- Open question: can we make designing **new** data structures both **easy and fast**?
  - Hybrid transactional memory?
    - Combines HTM with software implementations of transactional memory to guarantee progress
    - Good algorithms have been designed, but they may be too complex to implement in compilers!
  - KCAS with some generic theory that proves searches work?
    - Some work has been done in this direction
    - "Generalized hindsight" and "Data expansion" lemmas
    - Easy proofs that searches work for data structures that satisfy some simple invariants