Atomicity and Software Transactional Memory

(thanks to SPJ for many slides)
The Problem

locks/condition variables

Traditional concurrency does not compose well
deadlock/lost wakeups...
What we have

Locks and condition variables

Hardware
Idea: Replace locks with atomic blocks
What's wrong with locks?

- Races
  - forgot to lock
- Deadlock
  - used wrong order
- Lost wakeups
  - forgot to notify
- Error recovery
  - forgot to cleanup on exception
Locks are non-compositional

class Account {
    int balance;
    synchronized void deposit(int amt) {
        balance += amt;
    }
    synchronized void withdraw(int amt) {
        if (balance < amt)
            throw new OutOfMoney();
        balance -= amt;
    }
}

add fund transfer?
Locks are non-compositional

class Account {
    int balance;

    synchronized void deposit(int amt) {
        balance += amt;
    }

    synchronized void withdraw(int amt) {
        if (balance < amt)
            throw new OutOfMoney();
        balance -= amt;
    }

    void transfer_wrong1(Account o, int amt) {
        o.withdraw(amt);
        this.deposit(amt);
    }
}
Locks are non-compositional

class Account {
    int balance;
    synchronized void deposit(int amt) {
        balance += amt;
    }
    synchronized void withdraw(int amt) {
        if (balance < amt)
            throw new OutOfMoney();
        balance -= amt;
    }
    synchronized void transfer_wrong(Account o, int amt) {
        o.withdraw(amt);
        this.deposit(amt);
    }
}
Limitations of race-freedom

```java
class Ref {
    int i;
    void inc() {
        int t;
        synchronized (this) {
            t = i;
            i = t + 1;
        }
    }
}
```

Race free!
Doesn’t do what you want!

Race freedom does not eliminate concurrency errors.
(SC is still not easy!)
Limitations of race-freedom

```java
class Ref {
    int i;
    void inc() {
        int t;
        synchronized (this) {
            t = i;
            i = t + 1;
            i = t + 1;
        }
    }
    int read() { return i; }
}
Has a race!
Does what you want.
Race freedom is not necessary to eliminate errors!
```
Locks are absurdly hard to get right

<table>
<thead>
<tr>
<th>Coding style</th>
<th>Difficulty of queue implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential code</td>
<td>Undergraduate</td>
</tr>
</tbody>
</table>
Locks are absurdly hard to get right

<table>
<thead>
<tr>
<th>Coding style</th>
<th>Difficulty of queue implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential code</td>
<td>Undergraduate</td>
</tr>
<tr>
<td>Locks</td>
<td>Publishable result at PODC(^1)</td>
</tr>
</tbody>
</table>
Locks are absurdly hard to get right

<table>
<thead>
<tr>
<th>Coding style</th>
<th>Difficulty of queue implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential code</td>
<td>Undergraduate</td>
</tr>
<tr>
<td>Locks</td>
<td>Publishable result at PODC¹</td>
</tr>
<tr>
<td>Atomic blocks</td>
<td>Undergraduate</td>
</tr>
</tbody>
</table>

¹ Simple, fast, and practical non-blocking concurrent queue algorithms
Atomic: easier-to-use, harder to implement

```java
def void deposit(int x) {
    synchronized(this) {
        int tmp = balance;
        tmp += x;
        balance = tmp;
    }
}
```

semantics:
lock acquire/release

```java
def void deposit(int x) {
    atomically {
        int tmp = balance;
        tmp += x;
        balance = tmp;
    }
}
```

semantics: (behave as if)
no interleaved execution
Atomic memory transactions

atomically²  ... sequential code ... ²

- All or nothing!  (error recovery)
- Isolated!
- No deadlocks!  (no locks!)
How does it work?

atomically ...

... <code> ...

One possibility:

1. Execute <code> w/o any locks
2. Log each read/write in a thread-local transaction log
3. Writes go to log only, not memory

At the end, validate the log
- If valid, atomically commit changes to memory
- If invalid, re-run transaction from beginning
Software Transactional Memory

- Original paper/patent Tom Knight

- Many research/experimental implementations
  C/C++, C#, Java, OCaml, Python, Scala

- Haskell STM!
  Functional core simplifies STM concepts
  (Transaction access of non-transactional memory?)
Some questions

- Java has locking & Condition variables
Some questions

- Java has locking & Condition variables atomically
Some questions

- Java has locking & condition variables atomically. blocking?
Some questions

- Java has locking & condition variables atomically

- Is programming with atomic really this easy? efficiency?
STM in Haskell
Why Haskell?

Other languages:

Mutable data

Logging = Expensive
Why Haskell?

Haskell

Immutble Data

Mutable Data

only need to log here!
Why Haskell?

Pure code

Transactions here only!

IO code

Immutable Data

Mutable Data

Monads!
Why Haskell?
Recap: Effects in the type system

```haskell
main = do {
  putStrLn (reverse "yes");
  putStrLn "no";
}
```

:: String \(\triangleleft\) no effects

:: IO () \(\downarrow\) effects
Recap: Mutable State

\[
\text{main} = \text{do} \begin{cases} r \leftarrow \text{newIORef} \emptyset; \\
\quad \text{incR} \ r; \\
\quad s \leftarrow \text{readIORef} \ r; \\
\quad \text{print} \ s \end{cases}
\]

\text{incR} :: \text{IORef} \ \text{Int} \rightarrow \text{IO} ()

\text{incR} \ r = \text{do} \begin{cases} v \leftarrow \text{readIORef} \ r; \\
\quad \text{writeIORef} \ r \ (v + 1) \end{cases}

\text{r+1 disallowed!}
Recap: Concurrency in Haskell

forkIO :: IO () → IO ThreadId

main = do 
  r ← newIORef 0;
  forkIO (incR r);
  s ← readIORef r;
  print s

(could fix with IORef → MVar)
STM in Haskell

Idea: atomically :: IO a → IO a

```haskell
main = do { r ← newIORef ∅;
           forkIO (atomically (incr r));
           s ← readIORef r;
           print s }
```

Problem: shouldn't allow non-transactional access to r...
STM in Haskell

Better: atomically :: STM a → IO a
newTVar :: a → STM (TVar a)
readTVar :: TVar a → STM a
writeTVar :: TVar a → a → STM

TVar can only be modified in a transaction!
main = do { r <- atomically (newTVar φ); forkIO (atomically (incT r)); s <- atomically (readTVar r); print s }

incT :: TVar Int -> STM ()
incT r = do { v <- readTVar r; writeTVar r (v+1) }

Can't modify TVar outside of transaction

Atomically is just a function

Can't do IO inside transaction
Recap: Monads

\[ (\gg\gg) :: \text{STM} a \rightarrow (a \rightarrow \text{STM} b) \rightarrow \text{STM} b \]

STM composes
Recap: Exceptions

\[ \text{throwSTM} :: \text{Exception} \ e \Rightarrow e \rightarrow \text{STM} \ a \]

\[ \text{catchSTM} :: \text{Exception} \ e \Rightarrow \text{STM} \ a \]

\[ \rightarrow (e \rightarrow \text{STM} \ a) \]

\[ \rightarrow \text{STM} \ a \]

If s throws exception in atomically s, abort transaction! No cleanup needed.
Three new ideas:
retry orElse always
Idea 1: Compositional Blocking

withdraw :: TVar Int \rightarrow \text{Int} \rightarrow \text{STM} ()
withdraw acc n =
    do bal \leftarrow \text{readTVar acc} ;
        if bal < n then \text{retry}
        else \text{writeTVar acc (bal - n)} ;

retry :: STM ()
Abort transaction and try again from beginning. (Impl!!)
Idea 1: Compositional Blocking

withdraw :: TVar Int → Int → STM ()
withdraw acc n =
  do { bal ← readTVar acc;
       if bal < n then retry
       else writeTVar acc (bal - n) }

- No condition variables
- Retrying thread woken up automatically
- No danger of forgetting to re-test conditions
Why is retry compositional?

Retry can occur anywhere in an atomic block atomically (withdraw a1 3 » withdraw a2 7)

Non-composition alternative: declare all conditions upfront
Idea 2: Choice

Suppose we want to transfer three dollars from either account a1 or a2 to account b

atomic ( ((withdraw a1 3 'orElse' withdraw a2 3) >> deposit b 3 ))

try this

... and if it retries

... then do this.

orElse :: STM a → STM a → STM a
Choice composes too!

\[
\text{transfer2 } a_1 \ a_2 \ b \\
\text{ } \text{orElse} \\
\text{transfer2 } a_3 \ a_4 \ b
\]

It's associative! (but not commutative)
An algebra!

\[
\text{retry } \text{`orElse`} \ s \equiv s \\
\text{s } \text{`orElse`} \ \text{retry} \equiv s
\]

MonadPlus!
Idea 3: Invariants

Goal: Establish invariants which are true on entry & exit from atomic

always :: STM Bool \rightarrow STM ()

newAccount :: STM (TVar Int)
newAccount = do
  v <- newTVar \emptyset
  always (do cts <- readTVar v
           return (cts >= \emptyset))
  return v
Idea 3: Invariants

Goal: Establish invariants which are true on entry & exit from atomic

always :: STM Bool → STM ()

- Adds a new invariant to pool of invariants
- Conceptually: checked after every txn
  (Actually, only check for modified TVars; garbage collect based on dead TVars)
spec?

See Composable Memory Transactions for details!
GHC ships w/ a complete STM impl

import Control.Concurrent.STM

Microbenchmarks:
~50-80 ns  TVar read/put
~20 ns  MVar take/put
  (Mutex cost)

Worse: readTVar O(n) in number of TVars in txn
class Account {
    int balance;

    atomic void deposit(int amt) {
        balance += amt;
    }

    atomic void withdraw(int amt) {
        if (balance < amt)
            throw new OutOfMoney();
        balance -= amt;
    }

    atomic void transfer(Account o, int amt) {
        o.withdraw(amt);
        this.deposit(amt);
    }
}
STM in mainstream languages?

Trouble: type system doesn’t control effects

Weak atomicity
Non-transactional code may see inconsistent states on transactional code.

Strong atomicity
Non-transactional code guaranteed to see a consistent state (performance hit!)

Enforcing Isolation and Ordering in STM
Performance?

- Naïve STM is hopelessly inefficient (think 6x slowdown or more)

- HTM: hardware transactional memory: hardware support for "lock elision"

- Hybrid STM: use HTM support to implement STM efficiently

↑ research direction
Atomicity not a silver bullet

How big should atomic blocks be?

Thread 1

\[
\begin{align*}
\text{atomic} & \{ x = 1; \} \\
\text{atomic} & \{ \text{if } (y==\emptyset) \text{ retry;} \} \\
\end{align*}
\]

incorrect to make this a single atomic block

Thread 2

\[
\begin{align*}
\text{atomic} & \{ \text{if } (x==\emptyset) \text{ retry;} \\
y & = 1; \\
\} \\
\end{align*}
\]

Races vs. Starvation
Atomicity not a silver bullet

STM can’t deadlock, but it’s not necessarily fair
Some parting thoughts
Side-effectful IO computation

Pure effect free computation
Useful

arbitrary effects

Useless

dangerous

no effects

safe
arbitrary effects

Plan A (everyone else) -> NIRVANA

Default: any effect
Plan: add restrictions

Examples: Regions, Ownership types, Vault, Spec#, Cyclone
Default: no effects
Plan: selectively permit effects

Examples:
- Domain specific languages (SQL, XQuery, MapReduce)
- Functional languages + controlled effects (Haskell)
Plan A (everyone else) → NIRVANA

Plan B (Haskell)

useful

useless

dangerous

safe

arbitrary effects

envy

no effects
Plan A (everyone else) → NIRVANA

Plan B (Haskell)

Idea: e.g. STM, monads...

useful

useless

dangerous

safe

arbitrary effects

no effects
One of Haskell’s most significant contributions is to take purity seriously and relentlessly pursue Plan B.

Imperative languages will embody growing (and checkable) pure subsets.

— Simon Peyton Jones
Conclusion

- Atomic blocks raise the level of abstraction for concurrent programming.
  not assembly!

- Not a silver bullet
  bugs! concurrency is hard! fairness

- Performance hit, but it seems acceptable
  HTM! Transactional boosting!