Multi-cores are coming!
- For 50 years, hardware designers delivered 40-50% increases per year in sequential program speed.
- Around 2004, this pattern failed because power and cooling issues made it impossible to increase clock frequencies.
- Now hardware designers are using the extra transistors that Moore’s law is still delivering to put more processors on a single chip.

If we want to improve program speed, concurrent programs are no longer optional.

Concurrent Programming
- Concurrent programming is essential to improve performance on a multi-core.
- Yet the state of the art in concurrent programming is 30 years old:
  locks and condition variables.
  (In Java: synchronized, wait, and notify.)
- Locks and condition variables are fundamentally flawed: it’s like building a skyscraper out of bananas.

This lecture describes significant recent progress: bricks and mortar instead of bananas.

What we want
- Libraries build layered concurrency abstractions.
- Concurrency primitives
- Hardware

What we have
- Locks and condition variables
  (a) are hard to use and
  (b) do not compose.

Idea: Replace locks with Atomic Blocks
- Atomic blocks
  (a) are easier to use and
  (b) they do compose.
- 3 primitives: atomically, retry, orElse
What’s wrong with locks?

A 30-second review:

- **Races**: forgotten locks lead to inconsistent views
- **Deadlock**: locks acquired in “wrong” order
- **Lost wakeups**: forgotten notify to condition variables
- **Diabolical error recovery**: need to restore invariants and release locks in exception handlers

- These are serious problems. But even worse...

Locks are Non-Compositional

- Consider a (correct) Java bank `Account` class:

  ```java
  class Account{
  float balance;
  synchronized void deposit(float amt) {
    balance = balance + amt;
  }
  synchronized void withdraw(float amt) {
    if (balance < amt) throw new OutOfMoneyError();
    balance -= amt;
  }
  }
  ``

- Now suppose we want to add the ability to transfer funds from one account to another.

  ```java
  void transfer_wrong1(Account other, float amt) {
    other.withdraw(amt);
    this.deposit(amt);
  }
  ```

Locks are Non-Compositional

- Simply calling `withdraw` and `deposit` to implement `transfer` causes a race condition:

  ```java
  class Account{
  float balance;
  synchronized void deposit(float amt) {
    balance += amt;
  }
  synchronized void withdraw(float amt) {
    if (balance < amt) throw new OutOfMoneyError();
    balance -= amt;
  }
  synchronized void transfer_wrong1(Account other, float amt) {
    other.withdraw(amt);
    // race condition: wrong sum of balances
    this.deposit(amt);
  }
  }
  ```

Locks are Non-Compositional

- Synchronizing `transfer` can cause deadlock:

  ```java
  class Account{
  float balance;
  synchronized void deposit(float amt) {
    balance += amt;
  }
  synchronized void withdraw(float amt) {
    if (balance < amt) throw new OutOfMoneyError();
    balance -= amt;
  }
  synchronized void transfer_wrong2(Account other, float amt) {
    this.deposit(amt);
    other.withdraw(amt);
    // can deadlock with parallel reverse-transfer
  }
  }
  ```

Locks are absurdly hard to get right

**Scalable double-ended queue: one lock per cell**

No interference if ends “far enough” apart

But watch out when the queue is 0, 1, or 2 elements long!

**Coding style** | **Difficulty of queue implementation**
--- | ---
Sequential code | Undergraduate
Atomic Memory Transactions

To a first approximation, just write the sequential code, and wrap atomicity around it.
- All-or-nothing semantics: Atomic commit.
- Atomic block executes in Isolation.
- Cannot deadlock (there are no locks!).
- Atomicity makes error recovery easy (e.g. throw exception inside sequential code).

How does it work?

atomically {... <code> ...}...

One possibility:
- Execute <code> without taking any locks.
- Log each read and write in <code> to a thread-local transaction log.
- Writes go to the log only, not to memory.
- At the end, the transaction validates the log.
  - If valid, atomicly commits changes to memory.
  - If not valid, re-runs from the beginning, discarding changes.

Realizing STM in Haskell

Why STM in Haskell?

- Logging memory effects is expensive.
- Haskell already partitions the world into
  - immutable values (zillions and zillions)
  - mutable locations (some or none)
    Only need to log the latter!
- Type system controls where I/O effects happen.
- Monad infrastructure ideal for constructing transactions & implicitly passing transaction log.
- Already paid the bill. Simply reading or writing a mutable location is expensive (involving a procedure call) so transaction overhead is not as large as in an imperative language.
Consider a simple Haskell program:

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

- Effects are explicit in the type system.
- Main program is a computation with effects.

```haskell
main :: IO ()
```

Recall that Haskell IO Monad functions `newIORef`, `readIORef`, and `writeIORef` manage mutable state.

```haskell
main = do
  r <- newIORef 0
  incR r
  s <- readIORef r
  print s
```

- Reads and writes are 100% explicit. The type system disallows `(r + 6)` because `r :: IORef Int`.

The `forkIO` function spawns a thread. It takes an IO action as its argument.

```haskell
main = do
  r <- newIORef 0
  forkIO (incR r)
  incR r
```

- A race

Introduce a type for imperative transaction variables `(TVar)` and a new Monad (`STM`) to track transactions.

```haskell
atomically :: STM a -> IO a
newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
incT :: TVar Int -> STM ()
incT r = do
  v <- readTVar r
  writeTVar r (v+1)
maint = do
  r <- atomically (newTVar 0)
  forkIO (atomically (incT r))
  atomically (incT r)
```

- Can't fiddle with `TVars` outside atomic block. [good]
- Can't do IO or manipulate regular imperative variables inside atomic block. [sad, but also good]

...and, best of all...
The type guarantees that an STM computation is always executed atomically (e.g. `incT2`).

Simply glue STMs together arbitrarily; then wrap with `atomically` to produce an IO action.

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

incT2 :: TVar Int -> STM ()
incT2 r = do { incT r; incT r }

main :: IO ()
main = ...atomically (incT2 r) ...
```

Exceptions

- The STM monad supports exceptions:
  ```haskell
  throw :: (Exception e) => e -> a
  catchSTM :: STM a -> (DoneException -> STM a) -> STM a
  ```
  - In the call `atomically s`, if `s` throws an exception and the transaction validates, the transaction is aborted with no effect and the exception is propagated to the enclosing IO code.
  - No need to restore invariants, or release locks!
  - See "Composable Memory Transactions" for details.

Composition is THE way to build big programs that work.

Three new ideas

- retry
- `orElse`
- `always`

Function `retry` means "Abort the current transaction and re-execute it from the beginning."

Implementation avoids the busy wait by using reads in the transaction log (i.e. `acc`) to wait simultaneously on all read variables.

```haskell
retry :: STM()
retry
```

No condition variables!

Retrying thread is woken up automatically when `acc` is written, so there is no danger of forgotten notifies.

No danger of forgetting to test conditions again when woken up because the transaction runs from the beginning. For example:

```haskell
atomically (do { withdraw a1 3; withdraw a2 7 })
```

What makes `retry` compositional?

- Function `retry` can appear anywhere inside an atomic block, including nested deep within a call. For example,

```haskell
atomically (do { withdraw a1 3; withdraw a2 7 })
```

waits for `a1>3 AND a2>7`, without any change to the `withdraw` function.

- Contrast:

```haskell
atomically (a1 > 3 && a2 > 7) { ...stuff... }
```

which breaks the abstraction inside "...stuff..."
Idea 2: Choice

* Suppose we want to transfer 3 dollars from either account a1 or a2 into account b.

```haskell
orElse :: STM a -> STM a -> STM a

atomically (do {
    withdraw a1 3
    `orElse`
    withdraw a2 3;
    deposit b 3 })

...and if it retries, try this

...and and then do this

orElse :: STM a -> STM a -> STM a
```

Choice is composable, too!

```haskell
transfer :: TVar Int -> TVar Int -> STM ()

transfer a1 a2 b = do {
    withdraw a1 3
    `orElse`
    withdraw a2 3;
    deposit b 3 }

atomically (transfer a1 a2 b `orElse`
    transfer a3 a4 b)
```

* The function transfer calls orElse, but calls to transfer can still be composed with orElse.

Composing Transactions

* A transaction is a value of type STM a.
* Transactions are first-class values.
* Build a big transaction by composing little transactions: in sequence, using orElse and retry, inside procedures....
* Finally seal up the transaction with

```haskell
atomically :: STM a -> IO a
```

Algebra

* STM supports nice equations for reasoning:
  * orElse is associative (but not commutative)
  * retry `orElse` s = s
  * s `orElse` retry = s

(These equations make STM an instance of the Haskell typeclass MonadPlus, a Monad with some extra operations and properties.)

Idea 3: Invariants

* The route to sanity is to establish invariants that are assumed on entry and guaranteed on exit by every atomic block.
* We want to check these guarantees. But we don't want to test every invariant after every atomic block.
* Hmm.... Only test when something read by the invariant has changed.... rather like retry.

```haskell
always :: STM Bool -> STM ()

newAccount :: STM (TVar Int)
newAccount = do { v <- newTVar 0;
    always (do { cts <- readTVar v;
           return (cts >= 0) });
    return v }
```

Any transaction that modifies the account will check the invariant (no forgotten checks). If the check fails, the transaction restarts.

~ An arbitrary boolean valued STM computation ~
What always does

```haskell
always :: STM Bool -> STM ()
```

- The function `always` adds a new invariant to a global pool of invariants.
- Conceptually, every invariant is checked as every transaction commits.
- But the implementation checks only invariants that read `TVars` that have been written by the transaction.
- ...and garbage collects invariants that are checking dead `TVars`.

What does it all mean?

- Everything so far is intuitive and arm-wavey.
- But what happens if it is raining, and you are inside an `orElse` and you throw an exception that contains a value that mentions...?
- We need a precise specification!

Haskell Implementation

- A complete, multiprocessor implementation of `STM` exists as of GHC 6.
- Experience to date: even for the most mutation-intensive program, the Haskell `STM` implementation is as fast as the previous `MVar` implementation.
  - The `MVar` version paid heavy costs for (usually unused) exception handlers.
- Need more experience using `STM` in practice, though!

STM in Mainstream Languages

- There are similar proposals for adding STM to Java and other mainstream languages.

```java
class Account {
    float balance;
    void deposit(float amt) {
        atomic { balance += amt; }
    }
    void withdraw(float amt) {
        atomic {
            if(balance < amt) throw new OutOfMoneyError();
            balance -= amt;
        }
    }
    void transfer(Account other, float amt) {
        atomic {
            // Can compose withdraw and deposit.
            other.withdraw(amt);
            this.deposit(amt);
        }
    }
}
```

Weak vs Strong Atomicity

- Unlike Haskell, type systems in mainstream languages don't control where effects occur.
- What happens if code outside a transaction conflicts with code inside a transaction?
  - **Weak Atomicity**: Non-transactional code can see inconsistent memory states. Programmer should avoid such situations by placing all accesses to shared state in transaction.
  - **Strong Atomicity**: Non-transactional code is guaranteed to see a consistent view of shared state. This guarantee may cause a performance hit.

For more information: "Enforcing Isolation and Ordering in STM"
Performance

- At first, atomic blocks look insanely expensive. A naïve implementation (c.f. databases):
  - Every load and store instruction logs information into a thread-local log.
  - A store instruction writes to the log only.
  - A load instruction consults the log first.
  - Run-time system (RTS) validates the log at the end of the atomic block.
  - If succeeds, the RTS atomically commits writes to shared memory.
  - If fails, the RTS restarts the transaction.

New Implementation Techniques

- **Direct-update STM**
  - Allows transactions to make updates in place in the heap.
  - Avoids reads needing to search the log to see earlier writes the transaction has made.
  - Makes successful commit operations faster at the cost of extra work on contention or when a transaction aborts.
- **Compiler integration**
  - Decomposes transactional memory operations into primitives.
  - Expose these primitives to compiler optimization (e.g., hoist concurrency control operations out of a loop).
- **Runtime system integration**
  - Integrates transactions with the garbage collector to scale to atomic blocks containing 100M memory accesses.

Results: Scalability

- Naïve STM implementation is hopelessly inefficient.
- There is a lot of research ongoing in the compiler and architecture communities to optimize STM.
- This work typically assumes transactions are smallish and have low contention. If these assumptions are wrong, performance can degrade drastically.
- We need more experience with “real” workloads and various optimizations before we will be able to say for sure that we can implement STM sufficiently efficiently to be useful.
Easier, But Not Easy.

- The essence of shared-memory concurrency is deciding where critical sections should begin and end. This is a hard problem.
  - Too small: application-specific data races (Eg, may see deposit but not withdraw if transfer is not atomic).
  - Too large: delay progress because deny other threads access to needed resources.

---

Starvation

- Worry: Could the system "thrash" by transactions continually having conflicts and re-executing?
- No: A transaction can be forced to re-execute only if another succeeds in committing. That gives a strong progress guarantee.
- But: A particular thread could starve:

---

A Monadic Skin

- In languages like ML or Java, the fact that the language is in the IO monad is baked in to the language. There is no need to mark anything in the type system because IO is everywhere.
- In Haskell, the programmer can choose when to live in the IO monad and when to live in the realm of pure functional programming.
- Interesting perspective: It is not Haskell that lacks imperative features, but rather the other languages that lack the ability to have a statically distinguishable pure subset.
- This separation facilitates concurrent programming.

---

The Central Challenge

Useful

Arbitrary effects

No effects

Useless

Dangerous

Safe

---

The Challenge of Effects

Useful

Arbitrary effects

Nirvana

Plan B (Haskell)

Useless

Dangerous

Safe

Plan A (everyone else)
Examples

- Regions
- Ownership types
- Vault, Spec#, Cyclone

Arbitrary effects

Default = Any effect
Plan = Add restrictions

Vault, Spec#, Cyclone

Arbitrary effects

Default = No effects
Plan = Selectively permit effects

Domain specific languages (SQL, Xquery, Google map/reduce)
Wide-spectrum functional languages + controlled effects (e.g. Haskell)

Plan A (everyone else)
Plan B (Haskell)

Ideas: e.g. Software Transactional Memory
(retry, orElse)

Plan A (everyone else)
Plan B (Haskell)

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

Conclusions

- Atomic blocks (atomic, retry, orElse) dramatically raise the level of abstraction for concurrent programming.
- It is like using a high-level language instead of assembly code. Whole classes of low-level errors are eliminated.
- Not a silver bullet:
  - You can still write buggy programs.
  - Concurrent programs are still harder than sequential ones.
  - It addresses only shared memory concurrency, not message passing.
- There is a performance hit, but it seems acceptable (and things can only get better as the research community focuses on the question.)