Atomicity and Software Transactional Memory
John Mitchell

Reading
“Beautiful Concurrency”,
“The Transactional Memory / Garbage Collection Analogy”

Thanks to Simon Peyton Jones for many slides.

Multicore is here!
• Moore’s Law -> multicore
  – Higher chip density increases computing power
  – Since ~2004, increasing power and cooling requirements limit increases in clock frequency
  – Extra transistors allow more processors per chip
• Multicore architectures require concurrent programming

The Problem
• Traditional concurrency control uses
  – Locks: Java synchronize methods (or blocks)
  – Condition variables: wait/notify
• These mechanisms do not compose well
  – It is difficult to build complex thread-safe classes out of simpler thread-safe classes
• Are there promising alternatives?

What we want

What we have

Idea: Replace locks with atomic blocks

Atomic blocks are much easier to use, and do compose
Atomic blocks
3 primitives: atomic, retry, orElse
What’s wrong with locks?

- Races: forgotten locks lead to inconsistent views
- Deadlock: locks acquired in “wrong” order
- Lost wakeups: forget notify to condition variables
- 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:

```
class Account{
    float balance;
    synchronized void deposit(float amt) {
        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:

```
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(Acct other, float amt) {
        other.withdraw(amt);
        // race condition: wrong sum of balances
        this.deposit(amt);
    }
}
```

Locks are Non-Compositional

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

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

Locks are Non-Compositional

- Synchronizing transfer can cause deadlock:

```
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(Acct other, float amt) {
        // can deadlock with parallel reverse-transfer
        this.deposit(amt);
        other.withdraw(amt);
    }
}
```

Another problem:

Limitations of Race-Freedom

- `Ref.inc()`
  - race-free
  - behaves incorrectly in a multithreaded context

Race freedom *does not* prevent errors due to unexpected interactions between threads

- `Ref.read()`
  - has a race condition
  - behaves correctly in a multithreaded context

Race freedom *is not necessary* to prevent errors due to unexpected threads interactions
Locks are 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 and condition variables</td>
<td>Publishable result at international conference¹</td>
</tr>
</tbody>
</table>

¹ Simple, fast, and practical non-blocking and blocking concurrent queue algorithms.

Atomic

- Easier-to-use and harder-to-implement

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

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

Atomic semantics: (behave as if) no interleaved execution

Atomic memory transactions

- To a first approximation, just write the sequential code, and wrap atomic 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?

```
atomic {... <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, atomically commits changes to memory.
  - If not valid, re-runs from the beginning, discarding changes.
Implementing atomic

- Key pieces:
  - Execution of an atomic block logs writes
  - If scheduler pre-empts a thread in atomic, rollback the thread
  - Duplicate code so non-atomic code is not slowed by logging
  - Smooth interaction with GC

Some questions

- In Java, we have both
  - Locking (synchronized blocks and methods)
  - Condition variables (wait, notify)
- In software transactional memory
  - *atomic* seems to be replacement for locking
  - What about condition variables?
    - Do we need additional primitives?
    - Or can producer/consumer be programmed using *atomic*?

More questions

- Tools and methods exist for locking
  - Race condition detection tools, ...
  - Choose: synchronization, immutability, thread-safe wrappers
- Is programming with *atomic* that easy?
  - Or is there a catch?
  - How to efficiently achieving intended semantics?

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."
  - Reading or writing a mutable location is expensive
    - (involving a procedure call)
  - Transaction overhead is not as large as in imperative language

Status: Software Transactional Memory

- Original concept
  - Research paper and patent by Tom Knight
- Research and experimental implementations
  - C/C++, C#, Java, Ocaml, Python, Scala, ...
- Best illustration of concept
  - Haskell STM
  - Functional core of Haskell makes some concepts simpler ...
    - Example: big problem in other languages if non-transactional code can access transactional memory

Realizing STM in Haskell
Tracking Effects with Types

• Consider a simple Haskell program:
  ```haskell
  main = do { putStr (reverse "yes"); putStr "no" }
  ```

• Effects are explicit in the type system
  ```haskell
  (reverse "yes") :: String  -- No effects
  (putStr "no") :: IO ()   -- Effects okay
  ```

• Main program is a computation with effects.
  ```haskell
  main :: IO ()
  ```

Mutable State

• Haskell uses newIORef, readIORef, and writeIORef functions from IO Monad to manage mutable state
  ```haskell
  main = do { r <- newIORef 0;
  incR r;
  s <- readIORef r;
  print s }
  ```

Concurrency in Haskell

• The fork function spawns a thread

```haskell
fork :: IO a -> IO ThreadId
```  

```haskell
main = do { r <- newIORef 0;
  fork (incR r);
  incR r;
  ... }
```  

```haskell
incR :: IORef Int -> IO ()
incR r = do { v <- readIORef r;
  writeIORef r (v+1) }
```  

A Better Type for Atomic

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

```haskell
atomic :: STM a -> IO a
newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```  

Atomic Blocks in Haskell

• Idea: add a function atomic that executes its argument computation atomically.
  ```haskell
  atomic :: IO a -> IO a -- almost
  ```

```haskell
main = do { r <- newIORef 0;
  fork (atomic (incR r));
  atomic (incR r);
  ... }
```  

STM in Haskell

• Notice that:
  - Can’t modify TVars outside atomic block [good]
  - Can’t do IO or manipulate regular imperative variables inside atomic block [sad, but also good]

```haskell
atomic (if x<y then launchMissiles)
```  

...and, best of all...
STM Computations Compose (unlike locks)

- The type guarantees that an STM computation is always executed atomically (e.g. incT2).
- Simply glue STMs together arbitrarily; then wrap with atomic 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 }

foo :: IO ()
foo = ...atomic (incT2 r)...```

Exceptions

- The STM monad supports exceptions:

```haskell
throw :: Exception -> STM a
throw e = return $ E.throw e

catch :: STM a -> (Exception -> STM a) -> STM a
catch s f = do { x <- s;
                 case x of { E.Exception e -> f e; otherwise -> return x } }
```

In the call (atomic s), if s throws an exception, 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 more information.

Idea 1: Compositional Blocking

```haskell
withdraw :: TVar Int -> Int -> STM ()
withdraw acc n =
  do { bal <- readTVar acc;
       if bal < n then retry;
              writeTVar acc (bal-n) }

retry :: STM ()
retry =
  do { bal <- readTVar acc;
       if bal < n then retry;
              writeTVar acc (bal-n) }
```

- 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.

Compositional Blocking

```haskell
withdraw :: TVar Int -> Int -> STM ()
withdraw acc n =
  do { bal <- readTVar acc;
       if bal < n then retry;
              writeTVar acc (bal-n) }
```

- 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: atomic (do { withdraw a1 3;
                 withdraw a2 7 })

What makes Retry Compositional?

- retry can appear anywhere inside an atomic block, including nested deep within a call.
- For example,
  ```haskell
  atomic (do { withdraw a1 3;
              withdraw a2 7 })
  ```
  waits for a1>3 AND a2>7, without any change to withdraw function.
- Contrast:
  ```haskell
  atomic (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
atomic (do {
    withdraw a1 3
    `orElse`
    withdraw a2 3;
    deposit b 3 })
```

**Choice is composable, too!**

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

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

- 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 value
- Build a big transaction by composing little transactions: in sequence, using orElse and retry, inside procedures....
- Finally seal up the transaction with

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

Invariant: One New Primitive

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

```haskell
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.
What always does

\[
\text{always :: STM Bool} \rightarrow \text{STM ()}
\]

- 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
  - garbage collects invariants that are checking dead TVars

What does it all mean?

- Everything so far is intuitive and arm-wavey
- Like Java memory model experience, 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!
- You can play with it. The reading assignment contains a complete STM program.

STM in Mainstream Languages

- Proposals for adding STM to Java, others

```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 expensive
- A naive implementation (c.f. databases):
  - Every load and store instruction logs information into a thread-local log
  - A store instruction writes the log only
  - A load instruction consults the log first
  - Validate the log at the end of the block
    • If succeeds, atomically commit to shared memory
    • If fails, restart the transaction

State of the Art Circa 2003

Results: Concurrency Control Overhead

Results: Scalability

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 that the transaction has made
  - Makes successful commit operations faster at the cost of extra work on contention or when a transaction aborts
- Compiler integration
  - Decompose transactional memory operations into primitives
  - Expose these primitives to compiler optimization (e.g. to hoist concurrance control operations out of a loop)
- Runtime system integration
  - Integrates transactions with the garbage collector to scale to atomic blocks containing 100M memory accesses

Performance, Summary

- Naive STM implementation is hopelessly inefficient.
- There is a lot of research going on 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.

Results:
Concurrency Control Overhead

Scalability

Scaling to multicores

- See “Optimizing Memory Transactions” for more information.
Atomic: 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.

Still Not Easy, Example

- Consider the following program:

```plaintext
Initially, x = y = 0
Thread 1
// atomic {                  //A0
atomic { x = 1; }      //A1
atomic { if (y==0) abort; } //A2
}
Thread 2
atomic {      //A3
if (x==0) abort;
y = 1;
}
```

- Successful completion requires A3 to run after A1 but before A2.
- So adding a critical section (by uncommenting A0) changes the behavior of the program (from terminating to non-terminating).

Starvation

- Worry: Could the system “thrash” by continually colliding 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:

```
Thread 1
Thread 2
Thread 3
```

Less is More

- 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 A (everyone else)

Plan B (Haskell)

Useless

No effects

Dangerous Safe
```
Two Basic Approaches: Plan A

- Arbitrary effects
  - Default = Any effect
  - Plan = Add restrictions
  - Examples
    - Regions
    - Ownership types
    - Vault, Spec#, Cyclone

Two Basic Approaches: Plan B

- Default = No effects
- Plan = Selectively permit effects

- Types play a major role

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

Lots of Cross Over

- Arbitrary effects
- Plan A (everyone else)
- Plan B (Haskell)
- Envy
- Useful
- No effects
- Dangerous
- Safe
- Useless

Lots of Cross Over

- Arbitrary effects
- Plan A (everyone else)
- Plan B (Haskell)
- No effects
- Useful
- Dangerous
- Safe
- Useless

An Assessment and a Prediction

- 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
  - aimed only at 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.)