Software Transactional Memory

Madhavan Mukund

Chennai Mathematical Institute
http://www.cmi.ac.in/~madhavan

Formal Methods Update Meeting
TRDDC, Pune
19 July 2008

The challenge of concurrent programming

- Concurrent programming is difficult
- Over the years, semaphores, monitors . . . to lock and unlock shared data

The challenge of concurrent programming

- Concurrent programming is difficult
- Over the years, semaphores, monitors . . . to lock and unlock shared data
- Thesis
  - Lock based programming is difficult to design and maintain
  - Lock based programs do not compose well

The challenge of concurrent programming

- Concurrent programming is difficult
- Over the years, semaphores, monitors ... to lock and unlock shared data
- Thesis
  - Lock based programming is difficult to design and maintain
  - Lock based programs do not compose well
- With multicore architectures, concurrent programming will become more ubiquitous
- Goal
  - Design a new mechanism for reliable, modular concurrent programming with shared data

The challenge of concurrent programming

- Concurrent programming is difficult
- Over the years, semaphores, monitors ... to lock and unlock shared data
- Thesis
  - Lock based programming is difficult to design and maintain
  - Lock based programs do not compose well
- With multicore architectures, concurrent programming will become more ubiquitous
- Goal
  - Design a new mechanism for reliable, modular concurrent programming with shared data
  - Software Transactional Memory!

The problem with locks

A bank account class

```
class Account {
  Int balance;

  synchronized void withdraw( int n ) {
    balance = balance - n;
  }

  synchronized void deposit( int n ) {
    withdraw( -n );
  }
}
```

- Each object has a lock
- synchronized methods acquire and release locks

The problem with locks . . .

How do we transfer money from one account to another?

The problem with locks . . .

How do we transfer money from one account to another?

```
void transfer( Account from,
               Account to, Int amount ) {
  from.withdraw( amount );
  to.deposit( amount );
}
```

The problem with locks . . .

How do we transfer money from one account to another?

```
void transfer( Account from,
               Account to, Int amount ) {
  from.withdraw( amount );
  to.deposit( amount );
}
```

Is there a problem?

The problem with locks . . .

How do we transfer money from one account to another?

```
void transfer( Account from,
               Account to, Int amount ) {
  from.withdraw( amount );
  to.deposit( amount );
}
```

Is there a problem?

- Intermediate state when money has left `from` and not been deposited in `to` should not be visible!
- Having `withdraw` and `deposit` synchronized does not help

The problem with locks . . .

To fix this, we can add more locks

```
void transfer( Account from,
               Account to, Int amount ) {
  from.lock(); to.lock();
  from.withdraw( amount );
  to.deposit( amount );
  from.unlock(); to.unlock();
}
```

The problem with locks ...

To fix this, we can add more locks

```
void transfer( Account from,
               Account to, Int amount ) {
  from.lock(); to.lock();
  from.withdraw( amount );
  to.deposit( amount );
  from.unlock(); to.unlock();
}
```

Is there a problem?

The problem with locks . . .

To fix this, we can add more locks

```
void transfer( Account from,
               Account to, Int amount ) {
  from.lock(); to.lock();
  from.withdraw( amount );
  to.deposit( amount );
  from.unlock(); to.unlock();
}
```

Is there a problem?

- Two concurrent transfers in opposite directions between accounts `i` and `j` can deadlock!

The problem with locks . . .

Order the locks

```
void transfer( Account from,
               Account to, Int amount ) {
  if (from < to)
      then {from.lock(); to.lock(); }
      else {to.lock(); from.lock(); }
  from.withdraw( amount );
  to.deposit( amount );
  from.unlock(); to.unlock();
}
```

The problem with locks . . .

Order the locks

```
void transfer( Account from,
               Account to, Int amount ) {
  if (from < to)
      then {from.lock(); to.lock(); }
      else {to.lock(); from.lock(); }
  from.withdraw( amount );
  to.deposit( amount );
  from.unlock(); to.unlock();
}
```

Is there a problem?

The problem with locks ...

Order the locks

```
void transfer( Account from,
               Account to, Int amount ) {
  if (from < to)
      then {from.lock(); to.lock(); }
      else {to.lock(); from.lock(); }
  from.withdraw( amount );
  to.deposit( amount );
  from.unlock(); to.unlock();
}
```

Is there a problem?

- Need to know all possible locks in advance

The problem with locks . . .

```
void transfer( Account from,
               Account to, Int amount ) {
  if (from < to)
      then {from.lock(); to.lock(); }
      else {to.lock(); from.lock(); }
  from.withdraw( amount );
  to.deposit( amount );
  from.unlock(); to.unlock();
}
```

- What if `from` is a Super Savings Account in which most of the money is in a medium term fixed deposit `fromFD`?

The problem with locks ...

```
void transfer( Account from,
               Account to, Int amount ) {
  if (from < to)
      then {from.lock(); to.lock(); }
      else {to.lock(); from.lock(); }
  from.withdraw( amount );
  to.deposit( amount );
  from.unlock(); to.unlock();
}
```

- What if `from` is a Super Savings Account in which most of the money is in a medium term fixed deposit `fromFD`?
- `from.withdraw(amt)` may require an additional transfer from `fromFD` to `from`
  - `transfer` may not know anything about `fromFD`
  - Even if it did, it has to acquire a third lock

The problem with locks . . .

```
void transfer( Account from,
               Account to, Int amount ) {
  if (from < to)
      then {from.lock(); to.lock(); }
      else {to.lock(); from.lock(); }
  from.withdraw( amount );
  to.deposit( amount );
  from.unlock(); to.unlock();
}
```

- What if `transfer` can block in case of insufficient funds?
  - Wait on a condition variable (monitor queue)
  - Becomes more complex as number of locks increase

The problem with locks . . .

- Take too few locks — data integrity is compromised
- Take too many locks — deadlocks, lack of concurrency
- Take wrong locks, or in wrong order — connection between lock and data it protects is informal
- Error recovery — how to recover from errors without leaving system in an inconsistent state?
- Lost wake-ups, erroneous retries — Easy to forget to signal a waiting thread, recheck condition after wake-up

The problem with locks . . .

- Take too few locks — data integrity is compromised
- Take too many locks — deadlocks, lack of concurrency
- Take wrong locks, or in wrong order — connection between lock and data it protects is informal
- Error recovery — how to recover from errors without leaving system in an inconsistent state?
- Lost wake-ups, erroneous retries — Easy to forget to signal a waiting thread, recheck condition after wake-up

Lack of modularity

Cannot easily make use of smaller programs to build larger ones

- Combining `withdraw` and `deposit` to create `transfer` requires exposing locks

Transactions

- Import idea of transactions from databases
  - Hardware support for transactions in memory
    [Herlihy,Moss 1993]
- Instead, move transaction support to run time software
  - Software Transactional Memory [Shavit,Touitou 1995]
- An implementation in Haskell
  [Harris, Marlow, Peyton Jones, Herlihy 2005]
  - Tutorial presentation
    Simon Peyton Jones: Beautiful concurrency,
    in *Beautiful code*, ed. Greg Wilson, OReilly (2007)

Transactions . . .

- A transaction is an indivisible unit
- Execute a transaction as though it was running sequentially

Transactions . . .

- A transaction is an indivisible unit
- Execute a transaction as though it was running sequentially
- Check at the end of the transaction if any shared variables touched by the transaction have changed (due to external updates)

Transactions . . .

- A transaction is an indivisible unit
- Execute a transaction as though it was running sequentially
- Check at the end of the transaction if any shared variables touched by the transaction have changed (due to external updates)
  - Maintain a transaction log for each transaction, noting down values that were written and read
  - If a value is written in a transaction and read later, look it up in the log
  - At the end of the transaction, use log to check consistency

Transactions . . .

- A transaction is an indivisible unit
- Execute a transaction as though it was running sequentially
- Check at the end of the transaction if any shared variables touched by the transaction have changed (due to external updates)
  - Maintain a transaction log for each transaction, noting down values that were written and read
  - If a value is written in a transaction and read later, look it up in the log
  - At the end of the transaction, use log to check consistency
- If no inconsistency was seen, commit the transaction
- Otherwise, roll back and retry

Transactions . . .

Use `atomic` to indicate scope of transactions

```
void withdraw( int n ) {
  atomic{ balance = balance - n; }
}

void deposit( int n ) {
  atomic{ withdraw( -n ); }
}
```

Transactions . . .

Use `atomic` to indicate scope of transactions

```
void withdraw( int n ) {
  atomic{ balance = balance - n; }
}

void deposit( int n ) {
  atomic{ withdraw( -n ); }
}
```

Now, building a correct version of `transfer` is not difficult

```
void transfer( Account from, Account to, Int amount ) {
  atomic { from.withdraw( amount );
           to.deposit( amount ); }
}
```

Transaction interference

Independent transactions updating the same object

```
atomic{                                 // Transaction 1
  if a.getName().equals("B")
     s.setVal(8);
}

atomic{                                 // Transaction 2
  int previous = a.getVal();
  a.setVal(previous+1);
}
```

Transaction interference

Independent transactions updating the same object

```
atomic{                                  // Transaction 1
  if a.getName().equals("B")
     s.setVal(8);
}

atomic{                                  // Transaction 2
  int previous = a.getVal();
  a.setVal(previous+1);
}
```

- If Transaction 1 executes between first and second instruction of Transation 2, transaction log shows that value of `previous` is inconsistent
- Transaction 2 should roll back and reexecute

Transactions . . .

What else do we need?

Transactions . . .

What else do we need?

- Blocking
  - If amount to be withdrawn is more than current balance, wait

```
void transfer( Account from, Account to, Int amount ) {
  atomic {
    if (amount < from.balance) retry;
    from.withdraw ( amount );
    to.deposit( amount );
  }
}
```

Transactions . . .

What else do we need?

- Blocking
  - If amount to be withdrawn is more than current balance, wait

```
void transfer( Account from, Account to, Int amount ) {
  atomic {
    if (amount < from.balance) retry;
    from.withdraw ( amount );
    to.deposit( amount );
  }
}
```

- `retry` suspends transaction without any partial, inconsistent side-effects
- Transaction log indicates possible variables that forced `retry`
- Wait till one of these variables changes before attempting to rerun transaction from scratch

Transactions . . .

What else do we need?

Transactions . . .

What else do we need?

- Nested `atomic` allows sequential composition
- How about choosing between transactions with alternatives
  - If amount to be withdrawn is more than current balance, move money from linked fixed deposit

Transactions . . .

What else do we need?

- Nested `atomic` allows sequential composition
- How about choosing between transactions with alternatives
  - If amount to be withdrawn is more than current balance, move money from linked fixed deposit

```
void transfer( Account from, Account to, Int amount ) {
  atomic {
    atomic{ from.withdraw ( amount ); }
    orElse
    atomic{ LinkedFD[from].withdraw ( amount ); }

    to.deposit( amount );
  }
}
```

What could go wrong?

```
void b( Account from, Account to, Int amount ) {
  atomic {
    x = a.getVal();
    y = b.getVal();
    if (x > y){ launchMissiles(); }
    ...
  }
}
```

What could go wrong?

```
void b( Account from, Account to, Int amount ) {
  atomic {
    x = a.getVal();
    y = b.getVal();
    if (x > y){ launchMissiles(); }
    ...
  }
}
```

- If an inconsistency is found later, the transaction should roll back and retry
- How do we recall the missiles that have been launched?
- Need a strong type system to ensure that transactions affect only transactional memory

Dealing with exceptions

```
atomic{
   a = q1.extract();
   q2.insert(a);
}
```

Dealing with exceptions

```
atomic{
   a = q1.extract();
   q2.insert(a);
}
```

- Suppose `q2.insert(a)` fails because `q2` is full

Dealing with exceptions

```
atomic{
   a = q1.extract();
   q2.insert(a);
}
```

- Suppose `q2.insert(a)` fails because `q2` is full
- Reasonable to expect that value in `a` is pushed back into `q1`.

Dealing with exceptions

```
atomic{
   a = q1.extract();
   q2.insert(a);
}
```

- Suppose `q2.insert(a)` fails because `q2` is full
- Reasonable to expect that value in `a` is pushed back into `q1`.

How about

```
try { atomic{
        a = q1.extract();  q2.insert(a);
    }
catch (QueueFullException e) { a = q3.extract() } ;
```

- What is the state of `q1`?

STM summary

- Mechanism for delimiting transactions (`atomic`)
  - Programmer writes “sequential” code
  - Implementation determines granularity of concurrency — e.g. using transaction logs

STM summary

- Mechanism for delimiting transactions (`atomic`)
  - Programmer writes “sequential” code
  - Implementation determines granularity of concurrency — e.g. using transaction logs
- Transactions can be sequentially composed — nesting of transactions

STM summary

- Mechanism for delimiting transactions (`atomic`)
  - Programmer writes "sequential" code
  - Implementation determines granularity of concurrency — e.g. using transaction logs
- Transactions can be sequentially composed — nesting of transactions
- Transactions can block — `retry`

STM summary

- Mechanism for delimiting transactions (`atomic`)
  - Programmer writes “sequential” code
  - Implementation determines granularity of concurrency — e.g. using transaction logs
- Transactions can be sequentially composed — nesting of transactions
- Transactions can block — `retry`
- Choice between transactions – `orElse`

STM summary

- Mechanism for delimiting transactions (`atomic`)
  - Programmer writes "sequential" code
  - Implementation determines granularity of concurrency — e.g. using transaction logs
- Transactions can be sequentially composed — nesting of transactions
- Transactions can block — `retry`
- Choice between transactions – `orElse`
- Need to restrict what transactions can encompass — `LaunchMissiles()`

STM summary

- Mechanism for delimiting transactions (`atomic`)
  - Programmer writes "sequential" code
  - Implementation determines granularity of concurrency — e.g. using transaction logs
- Transactions can be sequentially composed — nesting of transactions
- Transactions can block — `retry`
- Choice between transactions – `orElse`
- Need to restrict what transactions can encompass — `LaunchMissiles()`
- Exceptions and transactions interact in a complex manner

STMs in Haskell

- Haskell clearly separates functions (pure, no side effects) from actions (with side effects)

STMs in Haskell

- Haskell clearly separates functions (pure, no side effects) from actions (with side effects)
  - Consider the difference between

    ```
    (f x) - (g x)
    ```

    and

    ```
    (read x) - (read x)
    ```

STMs in Haskell

- Haskell clearly separates functions (pure, no side effects) from actions (with side effects)
  - Consider the difference between
    ```
    (f x) - (g x)
    ```
    and
    ```
    (read x) - (read x)
    ```
- `IO` actions can be combined in an imperative style

```
incRef var = do { val <- readIORef var
                ; writeIORef var (val+1) }
```

STMs in Haskell

- Haskell clearly separates functions (pure, no side effects) from actions (with side effects)
  - Consider the difference between

    ```
    (f x) - (g x)
    ```

    and

    ```
    (read x) - (read x)
    ```

- `IO` actions can be combined in an imperative style

```
incRef var = do { val <- readIORef var
                ; writeIORef var (val+1) }
```

- STM implementation adds `STM` actions

```
withdraw acc amount =
        do { bal <- readTVar acc
           ; writeTVar acc (bal - amount) }

deposit acc amount = withdraw acc (- amount)
```

STMs in Haskell . . .

- Can combine STM actions into transactions

```
transfer from to amount
   = atomically (do { deposit to amount
                    ; withdraw from amount })
```

STMs in Haskell . . .

- Can combine STM actions into transactions

```
transfer from to amount
   = atomically (do { deposit to amount
                    ; withdraw from amount })
```

- Cannot mix IO actions and STM actions

```
bad acc n = do { putStr "Withdrawing..." -- IO
               ; withdraw acc n }        -- STM
```

STMs in Haskell . . .

- Can combine STM actions into transactions

```
transfer from to amount
   = atomically (do { deposit to amount
                    ; withdraw from amount })
```

- Cannot mix IO actions and STM actions

```
bad acc n = do { putStr "Withdrawing..." -- IO
               ; withdraw acc n }        -- STM
```

- . . . but `atomically` promotes STM actions to IO actions

```
ok acc n = do { putStr "Withdrawing..."
              ; atomically (withdraw acc n) }
```

- Strong type restriction for transactions

STMs in Haskell . . .

- Blocking works as expected — `retry`

```
limitedWithdraw acc amount
   = do { bal <- readTVar acc
        ; if amount > 0 && amount > bal
          then retry
          else writeTVar acc (bal - amount) }
```

STMs in Haskell . . .

- Blocking works as expected — `retry`

```
limitedWithdraw acc amount
   = do { bal <- readTVar acc
        ; if amount > 0 && amount > bal
          then retry
          else writeTVar acc (bal - amount) }
```

- Choice is also implemented as expected — `orElse`

```
limitedWithdraw2 acc1 acc2 amt
   = orElse (limitedWithdraw acc1 amt)
            (limitedWithdraw acc2 amt)
```

  - Withdraws `amt` from `acc1`, if `acc1` has enough money, otherwise from `acc2`.
  - If neither has enough, it retries.

STMs in Haskell . . .

- Strong typing avoids some STM pitfalls

```
atomically (do { x <- readTVar xv
               ; y <- readTVar yv
               ; if x>y then launchMissiles
                        else return () })
```

STMs in Haskell . . .

- Strong typing avoids some STM pitfalls

```
atomically (do { x <- readTVar xv
               ; y <- readTVar yv
               ; if x>y then launchMissiles
                        else return () })
```

- Unless `launchMissiles` is an STM action, this sequence of actions cannot be combined together

STMs in Haskell . . .

- Strong typing avoids some STM pitfalls

```
atomically (do { x <- readTVar xv
               ; y <- readTVar yv
               ; if x>y then launchMissiles
                        else return () })
```

- Unless `launchMissiles` is an STM action, this sequence of actions cannot be combined together
- STM roll back has been integrated with Haskell's built in exception handling mechanism (`catch`)

A case study

The Santa Claus problem

Santa Claus sleeps at the North pole until awakened by either all of the nine reindeer, or by a group of three out of ten elves. He performs one of two indivisible actions:

- If awakened by the group of reindeer, Santa harnesses them to a sleigh, delivers toys, and finally unharnesses the reindeer who then go on vacation.
- If awakened by a group of elves, Santa shows them into his office, consults with them on toy R&D, and finally shows them out so they can return to work constructing toys.

A waiting group of reindeer must be served by Santa before a waiting group of elves. Since Santas time is extremely valuable, marshalling the reindeer or elves into a group must not be done by Santa.

The Santa Claus problem

- Formulated by John Trono [Trono, SIGCSE Bulletin, 1994]
  - (Incorrect) solution with ten semaphores and two global variables
  - Can be fixed with two more semaphores
- Solutions based on semaphores, monitors are prone to race conditions
  - *Cannot be solved neatly using low level lock based primitives* [Ben-Ari, 1997]

Santa with STMs in Haskell

- Create datatypes `Group` and `Gate`
- Elves and reindeer try to assemble in respective `Group`s

Santa with STMs in Haskell

- Create datatypes `Group` and `Gate`
- Elves and reindeer try to assemble in respective `Group`s
- `Group`
  - Has an in `Gate` and out `Gate`
  - `joinGroup` atomically increments capacity if not full and returns current in and out `Gate` to elf/reindeer
  - `awaitGroup` checks if group is full, returns current in and out `Gate` to Santa, creates fresh in and out `Gate` for next group to assemble
- `Gate`
  - Has a capacity and counts how many elves/reindeer can go through before it closes
  - `passGate` atomically decrements count
  - `operateGate` initializes `Gate` count to full and waits for it to become zero

Santa with STMs in Haskell . . .

- Elves and reindeer are in infinite loop
  - `joinGroup` — returns `in_gate`, `out_gate`
  - `passGate in_gate`
  - Do appropriate business with Santa
  - `passGate out_gate`

Santa with STMs in Haskell . . .

- Elves and reindeer are in infinite loop
  - `joinGroup` — returns `in_gate`, `out_gate`
  - `passGate in_gate`
  - Do appropriate business with Santa
  - `passGate out_gate`
- Santa does the following
  - `orElse (awaitGroup rein_gp) (awaitGroup elf_gp)`
  - `awaitGroup` returns `in_gate`, `out_gate` for that group
  - `operateGate in_gate`
  - `operateGate out_gate`

Santa with STMs in Haskell ...

- Elves and reindeer are in infinite loop
  - `joinGroup` — returns `in_gate`, `out_gate`
  - `passGate in_gate`
  - Do appropriate business with Santa
  - `passGate out_gate`
- Santa does the following
  - `orElse (awaitGroup rein_gp) (awaitGroup elf_gp)`
  - `awaitGroup` returns `in_gate`, `out_gate` for that group
  - `operateGate in_gate`
  - `operateGate out_gate`
- Main program calls Santa in an infinite loop

Santa with STMs in Haskell . . .

- Elves and reindeer are in infinite loop
  - `joinGroup` — returns `in_gate`, `out_gate`
  - `passGate in_gate`
  - Do appropriate business with Santa
  - `passGate out_gate`
- Santa does the following
  - `orElse (awaitGroup rein_gp) (awaitGroup elf_gp)`
  - `awaitGroup` returns `in_gate`, `out_gate` for that group
  - `operateGate in_gate`
  - `operateGate out_gate`
- Main program calls Santa in an infinite loop
- About 100 lines of Haskell code
- Glasgow Haskell Compiler, `ghc`, has STM implementation built in

Summary

- Programming concurrent systems is hard
- Multicore technology will make concurrent programming more ubiquitous
- Existing lock based techniques do not scale up
- STMs provide a modular framework for coordinating shared data
- Not a magic bullet, but moving up from low level locks to more abstract concepts allow us to focus on coordination issues at higher level
- Implementations in other languages (e.g., Java) are being developed