Transactional Memory

(Based on S. Peyton Jones, ‘Beautiful concurrency’ in *Beautiful Code*,Oram and Wilson (eds) 2007.)

The illusion of mutual exclusion.

Context

Shared memory computations

The problems with locks

Consider back account class

```
class Account() {
        int balance = 0 ;
        void withdraw( int n ) {
                this.lock() ; balance -= n ; this.unlock() ; }
        void deposit( int n ) {
                this.lock() ; balance += n ; this.unlock() ; }
}
```

This is the usual implementation of synchronized methods.

Consider client code

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

As we've seen, this creates a **race condition**.

Locks don't compose:

- We've put together two actions that are reasonable on their own but that when put together give unsafe code.

We can fix it by adding more locking code

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

(**Aside:** I am assuming that we have *re-entrant locks*, which means that a thread can acquire the same lock more than once.)

But now we have a **potential deadlock**.

Consider

---

transfer(a, b, 100); || transfer(b, a, 100) ;

We can avoid this by fixing the locking order:

```
void transfer( Account from, Account to, int amount ) {
        if( from.compareTo( to ) < 0 ) {
                from.lock() ; to.lock() ; }
        else {
                to.lock() ; from.lock() ; }
        from.withdraw( amount ) ;
        to.deposit( amount ) ;
        to.unlock() ; from.unlock() ; }
```

Of course we have to be consistent across the entire system.)

This forces us to lock all objects up front even if we don't need them.

For example if we have overdraft protection

```
void transferWithODP( Account from, Account to,
                Account other, int amount ) {
        ...Lock from, to, and other in ascending order....
```

```
if( from.balance() >= amount ) {
      from.withdraw( amount ) ;
else other.withdraw( amount ) ;
to.deposit( amount ) ;
other.unlock() ; to.unlock() ; from.unlock() ; }
```

Here we locked other, even though it is not needed in most cases.

We could avoid all these problems by just having one lock that locks the whole system of accounts, but that would prevent concurrent operations.

Coping with thread failure

Threads can fail in at least three ways.

- There may be an unexpected exception which prevents completion of a task
  - The thread must restore consistency and unlock all locked objects.
  - Recovery code often outweighs the “real” code.
  - For highly reliable software, even unexpected exceptions must be anticipated.
- The thread may die

  - If a dead thread holds locks, other threads will get stuck, causing yet other threads to stick.
  - Simply breaking locks may lead to inconsistency.
    - "WARNING: Terminating a process can cause undesired results including loss of data and system instability. The process will not be given the chance to save its state or data before it is terminated." — Windows XP

- The thread may loop or deadlock
  - Same problems as thread death.

Locks are bad

Here are some of the pitfalls that we must avoid when using locks like this.

- Taking too few locks — leads to race conditions.
- Taking too many locks — inhibits concurrency
- Locking at too coarse a level — inhibits concurrency
- Taking locks in the wrong order — leads to deadlock
- Taking the wrong locks — is easy
- Error recovery — is hard.

Transactional memory

TM is an approach to mutual exclusion that relies on

- Executing transactions in parallel. I.e. potentially conflicting transactions are allowed to execute in parallel
- Aborting and retrying transactions that can not proceed because of
  - an unmet condition
  - an actual conflict with another transaction

A Syntax for TM

(Based on Haskell's STM library, but with Java, rather than Haskell, syntax.)

We divide object fields into two camps. Ordinary fields and transactional fields.

I will use the syntax

transactional int f ;

We notate transactions as follows.

atomically{ S }

- Essentially this implements $\langle S \rangle$
- Code within a transaction can access transactional fields

and thread local variables, but not ordinary fields.

- Code outside of transactions can access ordinary fields and thread local variables but not transactional fields.
- Methods that access only transactional fields are labelled **transactional**.

Example. To transfer we do

```
atomically{ transfer( from, to, amount ) ;}
```

Here is the transfer routine

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

```
class Account() {
        transactional int balance = 0 ;
        transactional void withdraw( int n ) {
                balance -= n ; }
```

```
transactional void deposit( int n ) {
    balance += n ; } }
```

Note that the transfer, withdraw, and deposit methods can only be called from within a transaction or other transactional methods.

Nontransactional methods, on the other hand, must not be called from within transactional code

Type checking

The type checker can ensure that code ordinary code and transactional code is not mixed.

We can consider each command and expression to posses one of four flavours

- **transactional** — accesses transactional fields and methods
- **unsafe** — accesses ordinary fields and methods
- **pure** — accesses no fields or methods
- **error** — indicates an error

Some type checking rules

$$flavour(E.f) = \mathbf{transactional}$$

if f is **transactional** and $flavour(E) \in \{\mathbf{pure}, \mathbf{transactional}\}$.Otherwise it is **error** or **unsafe**.

$$flavour(E.m()) = \mathbf{transactional}$$

if m is **transactional** and $flavour(E) \in \{\mathbf{pure}, \mathbf{transactional}\}$.Otherwise it is **error** or **unsafe**.

$$flavour(\mathbf{atomically}\{S\}) = \mathbf{pure}$$

if $flavour(S) \in \{\mathbf{pure}, \mathbf{transactional}\}$. Otherwise, it is **error**.

Combining code by sequential composition (and most other composition forms) is the least flavour

$$\begin{aligned} flavour(S\ T) &= flavour(S) \curlywedge flavour(T) \\ flavour(E + F) &= flavour(E) \curlywedge flavour(F) \end{aligned}$$

where

$$\begin{aligned} x \curlywedge y &= y \curlywedge x \\ x \curlywedge x &= x \\ \mathbf{error} \curlywedge x &= \mathbf{error} \\ \mathbf{pure} \curlywedge x &= x, \text{ if } x \neq \mathbf{error} \\ \mathbf{transactional} \curlywedge \mathbf{unsafe} &= \mathbf{error} \end{aligned}$$

A Software TM Implementation

There are a variety of ways to implement software transactional memory.

Here is one. We use **speculative execution** with **optimistic concurrency control**.

- At the start of each transaction,
  - save values of thread-local variables that might be changed
  - create a thread-local log, initially empty. Each log entry is of the form

    $$(a, v, v_0) \text{ or } (a, v)$$

    where
  - a is the address of a transactional variable.
  - v is the current value of the variable for this transaction
  - v_0 is the original value of the variable, if needed.
- If the first action on a transactional field is to read it, its current value recorded in the log. (Assume the value of the field at address a of main memory is 23)

  $$\mathit{read}(a, \{\}) \rightsquigarrow (23, \{(a,23, 23)\})$$

- When a transactional field is written, we change only the

log, adding a record if needed.

$$write(a, 42, \{(a,23, 23)\}) \rightsquigarrow \{(a,42, 23)\}$$
$$write(b, 13, \{(a,42, 23)\}) \rightsquigarrow \{(b,13), (a,42, 23)\}$$

- Subsequent reads of a field obtain the current value from the log.
  $read(a, \{(b,13), (a,42, 23)\}) \rightsquigarrow (42, \{(b,13), (a,42, 23)\})$
- At the end of the transaction, commit:
  - lock all fields mentioned in the log (aborting if a field is already locked);
  - validate: For each entry (a, v, v_0) check that $m[a] == v_0$;
  - if validation succeeds
    - write back: for each entry (a, v, v_0) or (a, v) set $m[a] := v$
    - unlock all fields
  - else
    - unlock all variables;
    - clear the log;
    - restore thread local variables that may have changed;
    - restart the transaction

Other implementations are possible.

Side effects

It is crucial that the only effects within the transaction are changes to transactional fields.

Other effects can not be undone

```
atomically { if( o.a == o.b ) launchTheMissiles() ; }
```

Conditional synchronization

So far we have only implemented mutual exclusion.

What about conditional synchronization.

We can implement a statement **check**(B).

```
transactional void safeTransfer( Account from, Account
to, int amount ) {
        check( from.balance >= amount ) ;
        from.withdraw( amount ) ;
        to.deposit( amount ) ; }
```

We can implement **check** by

- wiping the log; waiting a little while; restarting the

transaction

Note that check is very much like a conditional wait in monitors.

Choice

We can introduce a **symmetric choice** between transactional statements as follows:

$$S \,\square\, T$$

means try either S or T (choosing fairly), if one fails try the other.

We could implement this sequentially or in parallel.

If implemented in parallel, the first to succeed kills the other off — this is called speculative execution.

A biased choice

$$S \;\boxtimes\; T$$

tries S first and only tries T if S fails due to a failure of a check.

```
transactional void transferWithODP( Account from,
        Account to, Account other, int amount ) {
            safeTransfer( from, to, amount ) ;
            ⊠ transfer( other, to, amount ) ; }
```

Transactional code composes via sequencing and choice.

Performance

Software transactional memory is typically far slower than lock-based approaches.

For example our optimistic concurrency control implementation requires time to consult the log.

The advantage is ease of programming and assurance that the software is free of race-conditions and deadlocks.

Wither ACID?

As we saw, in database transactions, systems attempt to provide Atomicity, Isolation, Consistancy and Durability.

TM is intended to provide

- Atomicity. Each transactions appears atomic to from the outside.
- Isolation. Depending on implementation technique, transactions may see variables in intermediate states. However such a transaction will not successfully commit.
- Consistancy. It is up to the programmer to ensure that

transactions leave the system in a consistant state.

- Durability. No attempt is made in TM to provide durability.

Lack of Isolation

How can transactions be atomic but not isolated?

Remember that transactions may be executed speculatively.

Suppose that we have a global invariant that

$$\mathsf{o.p} \mathrel{!}= \mathsf{null} \vee \mathsf{o.q} \mathrel{!}= \mathsf{null}$$

A transaction

$$T_0 : \textbf{atomically}\{\ \textbf{if}(\ \mathsf{o.p}==\textbf{null}\ )\ \mathsf{o.q.a} = 0;\ \}$$

appears unproblematic.

It would seem that it can not dereference a null pointer.

Consider executing in parallel with the following transaction that swaps o.p with o.q.

$$T_1 : \textbf{atomically}\{\ \mathsf{T\ t} = \mathsf{o.p};\ \mathsf{o.p} = \mathsf{o.q};\ \mathsf{o.q} = \mathsf{t};\ \}$$

Assume speculative execution and optimistic concurrency control.

T_0	T_1	o.p	o.q
r0:=read o.p		null	x
	swaps	x	null
r0==null ?		x	null
r1:=read o.q		x	null
dereference r1		x	null

T_1 will dereference a null pointer.

Note that T_1, having seen o.p $==$ **null** $==$ o.q, can never commit.

In a safe language like Java, the lack of isolation is not a big deal, as the worst that can happen are unexpected exceptions, which will (by default) cause the transaction to immediately abort.

In unsafe languages like C and C++, where behaviour in many situations (e.g. null pointer dereference, unitialized pointer dereference, array index out of bounds) is undefined, the consequences could be much worse.

How are we doing?

Let's look back at the issues with locks

- Taking too few locks — leads to race conditions.
  - *If all shared variables are transactional, type checking ensures no race conditions*
- Taking too many locks — inhibits concurrency
  - *Transactions proceed concurrently*
- Locking at too coarse a level — inhibits concurrency
  - *Transactions proceed concurrently*
- Taking locks in the wrong order — leads to deadlock
  - *No locks: no deadlock.*
  - *However, we can have livelock, if a thread keeps **check**ing a condition that can never become true.*
- Taking the wrong locks
  - *No locks: no wrong locks*
- Error recovery
  - *Unexpected exceptions lead to aborted transactions*
  - *Dead threads do not "lock-out" live threads.*
  - *However, a live thread may be **check**ing a condition that the dead thread was expected to make true.*

Example. Santa

(Adapted from S. Peyton Jones, Beatiful Concurrency.)

```
class Gate() {
        private transactional int permits ;
        private final int gateSize ;

        public Gate( int size ) { gateSize = size ; }

        public void pass() {
            atomically {
                check( permits > 0 ) ;
                permits -= 1 ; } }

        public void open() {
            atomically { permits = gateSize ; }
            atomically { check( permits == 0 ) ; } } }
```

```
class Group {
        private final int sz ;
        private transactional int room ;
```

```
private transactional Gate[] gates ;

public Group( int size ) {
    atomically {
        sz = size ;
        room = size ;
        gates = new Gate[]{Gate(sz),Gate(sz)} ; }
      }

public Gate[] joinGroup() {
    atomically {
        check( room > 0 ) ;
        room -= 1 ;
        return gates ; } }

public transactional Gate[] awaitGroup() {
    check( room == 0 ) ;
    Gates oldGates = gates ;
    gates = new Gate[] {Gate(sz),Gate(sz)} ;
    return oldGates ; } }
```

```
class Helper extends Thread {
    private final Group group ;

    public Helper( Group group ) {
        this.group = group ; }

    protected abstract void helperHook() ;

    public void run() {
        while( true ) {
            Gate[] gates = group.joinGroup() ;
            gates[0].pass() ;
            helperHook() ;
            gates[1].pass() ; } } }
```

```
class Elf extends Helper {

    public Elf( Group elfGroup ) {super(elfGroup) ; }

    protected void helperHook() { meet with Santa } }
```

```
class Santa extends Thread {
    private final Group elfGroup ;
    private final Group deerGroup ;

    public Santa( Group elfGroup, Group deerGroup )
  {...}

    public void run() {
        while( true ) {
            Gates gates[] ;
            Group group ;
            atomically {
               { group = deerGroup ;
                 gates = awaitGroup( group ) ; }
            ☒ { group = elfGroup ;
                 gates = awaitGroup( group ) ; } }
            gates[0].open() ;
            if( group == deerGroup ) deliver presents
            else meet with elves
            gates[1].open() ; } } }
```

Other interfaces

Above I presented a language based approach:

- I modified the Java language to support TM.

We can also take a library based approach. For example DSTM2 for Java.

(The following is based on Herlihy, Luchangco, and Moir ‘A flexible framework for implementing software transactional memory’, OOPLSA, 2006.)

Objects with transactional fields are defined via programmer defined interfaces.

```
@atomic interface INode {
        int getValue() ;
        void setValue() ;
        INode getNext() ;
        void setValue( INode ) ; }
```

(Note that only getters and setters are possible. Such objects are, in essence, records.)

Objects with transactional fields are created via factories that are synthesized at run time.

```
public class List {
    static Factory<INode> factory =
  Thread.makeFactory( INode.class ) ;

    public void insert( int value ) {
        INode newNode = factory.create() ;
        newNode.setValue( value ) ;
        newNode.setNext( this.root.getNext() ) ;
        this.root.setNext( newNode ) ; }
    ...
}
```

Note that insert is transactional. To execute it safely, we must create a transaction and execute it atomically.

The equivalent to **atomically** is a static method named Thread.doIt.

```
Thread.doIt( new Runnable() { public void run() {
list.insert(42) ; } } );
```

(**Aside**: In case you are wondering, DSTM2 provides its own Thread class. This is not the standard Thread class.)

The implementation of doIt is roughly the following

```
void doIt( Runnable runnable ) {
        while( true ) {
                beginTransaction() ;
                try { runnable.run() ; }
                catch( Exception e) {
                        e.printStackTrace();
                        throw new PanicException("Unhandled
                    exception " + e); }
                if( commitTransaction() ) { return ; }
```

DSTM2 runs above an unmodified JVM.

Hardware Transactional Memory

(Based on Herilhy and Moss, 1993, ISCA.)

Provide each processor with two primary caches.

One is for regular variables. One is for transactional variables.

The transactional cache is

- fully associative — any cache slot can represent any

location

- snoopy — observes bus transactions.

Machine instructions

LT r a — transactional load
ST r a — transactional store
commit label
abort
validate label

Each m.m. word is in one of 3 states

- Memory Only — not cached anywhere
- Exclusive — owned exclusively by one cache
- Shared — cached by one or more caches

Each processor has three states

- No transaction in progress
- Transaction valid
- Transaction invalid

Each entry in the cache is marked with one of 4 tags

- EMPTY — not in use
- NORMAL — not involved in the active transaction
- XCOMMIT — to be discarded on commit
- XABORT — to be discarded on abort

LT and ST instruction actually create 2 entries in the cache.

LT and ST instructions abort the transaction if they can not obtain access

Example: A successful transaction

NTIP

Addr	**Value**	**Tag**
—	—	EMPTY
—	—	EMPTY

LT r1 100 ⟶

Valid

Addr	**Value**	**Tag**
100	42	XCOMMIT
100	42	XABORT

ST r1 100 ⟶

Valid

Addr	**Value**	**Tag**
100	42	XCOMMIT
100	43	XABORT

COMMIT ⟶

NTIP

Addr	**Value**	**Tag**
—	—	EMPTY
100	13	NORMAL

Example of with a conflict.

Suppose that the word at location 200 is held by another cache doing another transaction.

When this cache attempts to obtain ownership of the word, it is refused and all tentative changes are discarded.

NTIP

Addr	Value	Tag
—	—	EMPTY
—	—	EMPTY

LT r1 100 ⟶

Valid

Addr	Value	Tag
100	42	XCOMMIT
100	42	XABORT

ST r1 100 ⟶

Valid

Addr	Value	Tag
100	42	XCOMMIT
100	43	XABORT

ST r2 200 ⟶

Invalid

Addr	Value	Tag
100	42	NORMAL
—	—	EMPTY

Other instructions

- commit lab
  - ∗ if mode is valid
    - · discard XCOMMIT entries

        - convert XABORT entries to NORMAL
        - mode := “No transaction in progress”
    - else
        - mode := “No transaction in progress”
        - branch to lab
- abort
    - if mode is Valid
        - discard XABORT entries
        - convert XCOMMIT entries to NORMAL
        - mode := “No transaction in progress”
    - else
        - mode := “No transaction in progress”
- validate lab
    - if mode is Invalid
        - mode := “No transaction in progress”
        - branch to lab