

# DikuSTM: Transactional Memory in C++

Jesper Alf Dam

12. april 2010

## Motivation

Multithreading is hard. Lock-based synchronization isn't much help:

```
class Account {  
    void Withdraw(int amount);  
    void Deposit(int amount);  
    int Balance();  
};
```

## Motivation

Multithreading is hard. Lock-based synchronization isn't much help:

```
class Account {  
    void Withdraw(int amount);  
    void Deposit(int amount);  
    int Balance();  
};
```

```
act.Withdraw(50); // safe  
act.Deposit(50); // safe
```

## Motivation

Multithreading is hard. Lock-based synchronization isn't much help:

```
class Account {  
    void Withdraw(int amount);  
    void Deposit(int amount);  
    int Balance();  
};  
  
act.Withdraw(50); // safe  
act.Deposit(50); // safe  
Transfer(act1, act2, 50); // ???
```

## Motivation

```
void Transfer(Account act1, Account act2, int amount) {  
    act1.Withdraw(amount);  
    act2.Deposit(amount);  
}
```

## Motivation

```
void Transfer(Account act1, Account act2, int amount) {  
    act1.Withdraw(amount);  
    // Oops, where's the money now?  
    act2.Deposit(amount);  
}
```

## Motivation

```
void Transfer(Account act1, Account act2, int amount) {  
    act1.mutex.lock();  
    act2.mutex.lock();  
    act1.Withdraw(amount);  
    act2.Deposit(amount);  
    act1.mutex.unlock();  
    act2.mutex.unlock();  
}
```

## Motivation

```
void Transfer(Account act1, Account act2, int amount) {  
    act1.mutex.lock();  
    act2.mutex.lock();  
    act1.Withdraw(amount);  
    act2.Deposit(amount);  
    act1.mutex.unlock();  
    act2.mutex.unlock();  
}
```

- ▶ Breaks abstraction.

## Motivation

```
void Transfer(Account act1, Account act2, int amount) {  
    act1.mutex.lock();  
    act2.mutex.lock();  
    act1.Withdraw(amount);  
    act2.Deposit(amount);  
    act1.mutex.unlock();  
    act2.mutex.unlock();  
}
```

- ▶ Breaks abstraction.
- ▶ Lock-based code is not **composable**.

## Databases

In databases, this problem has been solved for decades. Database changes happen in transactions, and the system guarantees that every transaction is

## Databases

In databases, this problem has been solved for decades. Database changes happen in transactions, and the system guarantees that every transaction is

- ▶ **Atomic** (the results of the transaction are either all visible, or none are)
- ▶ **Consistent** (constraints on our data must not be violated — in a bank, the sum of all accounts should stay the same at any point during a transfer)
- ▶ **Isolated** (every transaction executes as if it'd been the only transaction currently executing)
- ▶ **Durable** (once committed, the changes made will stay applied.)

## Databases

In databases, this problem has been solved for decades. Database changes happen in transactions, and the system guarantees that every transaction is

- ▶ **Atomic** (the results of the transaction are either all visible, or none are)
- ▶ **Consistent** (constraints on our data must not be violated — in a bank, the sum of all accounts should stay the same at any point during a transfer)
- ▶ **Isolated** (every transaction executes as if it'd been the only transaction currently executing)
- ▶ **Durable** (once committed, the changes made will stay applied.)

A database transaction is **committed** as a single atomic operation. If a conflict occurs, the transaction is **rolled back** and retried. The programmer never needs to worry about synchronization or consistency.

Can we do something like this in our C/C++/Java/C#/Python code?

## Sounds clever. Is it plausible?

What if...?

## Sounds clever. Is it plausible?

What if...?

- ▶ Every shared object stores a timestamp recording the last time it was changed.  
Transactions record the time they started.

## Sounds clever. Is it plausible?

What if...?

- ▶ Every shared object stores a timestamp recording the last time it was changed. Transactions record the time they started.
- ▶ A transaction accessing a shared object compares the timestamps.

## Sounds clever. Is it plausible?

What if...?

- ▶ Every shared object stores a timestamp recording the last time it was changed. Transactions record the time they started.
- ▶ A transaction accessing a shared object compares the timestamps.
- ▶ A transaction takes a backup copy of each object it modifies.

## Transactional Memory

```
int val = 42;
```

```
thread1() {  
    ++val;  
}
```

```
thread2() {  
    --val;  
}
```

## Transactional Memory

```
int val = 42;
```

```
thread1() {  
  atomic {  
    ++val;  
  }  
}
```

```
thread2() {  
  atomic {  
    --val;  
  }  
}
```

## Transactional Memory

```
shared<int> val = 42;
```

```
thread1() {  
    atomic {  
        val.write(val.read() + 1);  
    }  
}
```

```
thread2() {  
    atomic {  
        val.write(val.read() - 1);  
    }  
}
```

## Transactional Memory

```
shared<int> val = 42;
```

```
thread1() {  
    atomic {  
        int& i = val.open();  
        ++i;  
    }  
}
```

```
thread2() {  
    atomic {  
        int& i = val.open();  
        --i;  
    }  
}
```

## It's been done. But it's not pretty

Simple example problem:

- ▶ Assume that we have a shared integer `i`.
- ▶ We wish to increment `i`,
- ▶ and then return the updated value of `i` to the calling code.

In a singlethreaded program:

```
int i = 42;
```

```
int result = ++i;
```

Of course, we require this to be done as a transaction.

## It's been done. But it's not pretty

```
int i = 42;

int result;
transaction tx;
transaction_state state = e_no_state;
do {
    try {
        tx.write(i).value()++;
        result = tx.read(i).value();
        state = tx.end_transaction();
    }
    catch(aborted_transaction_exception&) {
        tx.restart_transaction();
    }
}while(state != e_committed)
```

## It's been done. But it's not pretty

Or we could hide some of the complexity behind macros:

```
ATOMIC_BEGIN(tx) {  
    estm::shared<int>* i  
    = estm::shared<int>::construct<int>(tx, 42);  
} ATOMIC_END(tx);
```

```
int result;  
estm::transaction tx;  
ATOMIC_BEGIN(tx) {  
    result = ++i->openRW(tx);  
} ATOMIC_END(tx);
```

## It's been done. But it's not pretty

Or we could hide some of the complexity behind macros:

```
ATOMIC_BEGIN(tx) {  
    estm::shared<int>* i  
    = estm::shared<int>::construct<int>(tx, 42);  
} ATOMIC_END(tx);
```

```
int result;  
estm::transaction tx;  
ATOMIC_BEGIN(tx) {  
    result = ++i->openRW(tx);  
} ATOMIC_END(tx);
```

And these don't even solve the problem!

## Challenges

- ▶ Interface — What should code using STM look like?

Must be easy to get right. How many things can go wrong in the previous examples?

- ▶ Lots of boilerplate code to get wrong.
- ▶ Macros are evil.
- ▶ No dedicated scope for the transaction: what happens if I put a `break`, `continue` or `return` inside a transaction?
- ▶ (in the first example) no way to distinguish shared/transactional objects from non-shared ones, or to protect transactional objects against accidental non-transactional accesses. (In the second case) complex and inflexible mechanism for creating shared objects.

## Challenges

- ▶ Interface — What should code using STM look like?
- ▶ Semantics — How should STM code behave?

What happens if an exception is thrown inside a transaction? What if code with side effects is executed in a transaction? What if non-transactional variables are read/modified during a transaction? (`result` in the previous examples.)

Lots of open questions.

## Challenges

- ▶ Interface — What should code using STM look like?
- ▶ Semantics — How should STM code behave?
- ▶ Performance — Is it fast enough for actual real-world use?

How fast is “fast enough”? Fast in which scenarios? What is it going to be used for? How much contention should we expect in the common case? How long will typical transactions be? Will our performance-critical code even be inside a transaction?

## Interface

```
stm::shared<int> i = 42;  
  
int result = stm::atomic<int>([&](stm::transaction& tx){  
    return ++i.open_rw(tx);  
});
```

## Interface

```
stm::shared<int> i = 42;  
  
int result = stm::atomic<int>([&](stm::transaction& tx){  
    return ++i.open_rw(tx);  
});
```

Ok, fine. I cheated... A little bit.

## Interface

```
stm::shared<int> i = 42;

int txfun(stm::transaction& tx) {
    return ++i.open_rw(tx);
}

int result = stm::atomic<int>(txfun);
```

## Interface

```
stm::shared<int> i = 42;

int txfun(stm::transaction& tx) {
    return ++i.open_rw(tx);
}

int result = stm::atomic<int>(txfun);
```

- ▶ `commit` is implicit. No need to explicitly end the transaction.
- ▶ Shared objects are wrapped in the `shared` class, preventing non-transactional accesses.
- ▶ Transactions have their own scope: user code is in a function.
- ▶ Return values are assigned only when the transaction commits.

## Semantics

How should such a STM library behave? So far, there is little consensus.

- ▶ Conflict toleration

What happens if transaction A is accessing a shared object  $x$  while transaction B modifies it?

```
atomic {  
    int& i = x.open();  
    if (i == 42) {  
        // do something  
    }  
}
```

## Semantics

How should such a STM library behave? So far, there is little consensus.

- ▶ Conflict toleration

What happens if transaction A is accessing a shared object  $x$  while transaction B modifies it?

```
atomic {  
    int& i = x.open();  
    if (i == 42) {  
        while (i != 42) {}  
    }  
}
```

## Semantics

How should such a STM library behave? So far, there is little consensus.

- ▶ Conflict toleration
- ▶ User-thrown exceptions?

```
atomic {  
    // modify shared object x  
    throw exception();  
}
```

## Semantics

How should such a STM library behave? So far, there is little consensus.

- ▶ Conflict toleration
- ▶ User-thrown exceptions?
- ▶ Nested transactions

```
void foo() {  
    atomic {  
        // modify shared object x  
    }  
}  
atomic {  
    foo();  
}
```

## Semantics

How should such a STM library behave? So far, there is little consensus.

- ▶ Conflict toleration
- ▶ User-thrown exceptions?
- ▶ Nested transactions

## Semantics

How should such a STM library behave? So far, there is little consensus.

- ▶ Conflict toleration
- ▶ User-thrown exceptions?
- ▶ Nested transactions

```
void Transfer(Account act1, Account act2, int amount) {  
    atomic {  
        try {  
            act1.Withdraw(amount);  
            act2.Deposit(amount);  
        } catch (...) { abort(); }  
    }  
}
```

## Performance?

Is a bit of a mixed bag.

## Performance?

Is a bit of a mixed bag.

Depending on circumstances, performance varies from millions of commits per second to a few hundred.

Is that good or bad?

## Performance?

Is a bit of a mixed bag.

Depending on circumstances, performance varies from millions of commits per second to a few hundred.

Is that good or bad?

Fairly easy to improve the worst issues — but complicates the implementation and may hurt the best-case performance. Is it worth it?

## Performance?

Is a bit of a mixed bag.

Depending on circumstances, performance varies from millions of commits per second to a few hundred.

Is that good or bad?

Fairly easy to improve the worst issues — but complicates the implementation and may hurt the best-case performance. Is it worth it?

My gut feeling: it's not good enough. Probably. But I don't know how often the “bad” cases would occur in real-world code.

Requires more research.

## So how does it work?

Traditionally, three major approaches have been used:

- ▶ In-place direct update

## So how does it work?

Traditionally, three major approaches have been used:

- ▶ In-place direct update
- ▶ In-place deferred update

## So how does it work?

Traditionally, three major approaches have been used:

- ▶ In-place direct update
- ▶ In-place deferred update
- ▶ Indirection-based deferred update

## So how does it work?

Traditionally, three major approaches have been used:

- ▶ In-place direct update
- ▶ In-place deferred update
- ▶ Indirection-based deferred update

My solution: In-place deferred update using double-buffering.

- ▶ good locality
- ▶ (almost) no dynamic allocations
- ▶ commit requires two copies, but one can be a destructive move.

## Extras

## Extras

- ▶ Retry/orelse

## Extras

- ▶ Retry/orelse
- ▶ Detached metadata

## Extras

- ▶ Retry/orelse
- ▶ Detached metadata
- ▶ Snapshot

## The end

`http://jal.f.dk/thesis/thesis.pdf`  
`http://jal.f.dk/thesis/dikustm.zip`

## The end

`http://jal.f.dk/thesis/thesis.pdf`  
`http://jal.f.dk/thesis/dikustm.zip`  
Questions?