SOFT3410: Concurrency for Software Developers

Transactional memory

Lecturer: Martin McGrane
School of Computer Science

THE UNIVERSITY OF SYDNEY

Concurrent Programming is Hard

- Hard to make **safe** and **live**
  - Safe: *where nothing bad (e.g., crash) happens*
  - Live: *where something good (that we want) eventually happens*

- The human mind is usually trained to be sequential
  - Concurrent specifications
  - Non-deterministic executions

Safety

```
public class BankAccount {
  int balance = 0;

  void deposit(int amount)  {

   balance += amount;

  }
}
```

Safety

```
public class BankAccount {
  int balance = 0;

  void deposit(int amount)  {

   balance += amount;

  }
}
```

No concurrency control: race!

Liveness

› Problem

```
public class BankAccount {
   int balance = 0;

   void deposit(int amount)   {

    balance += amount;

   }
}
```

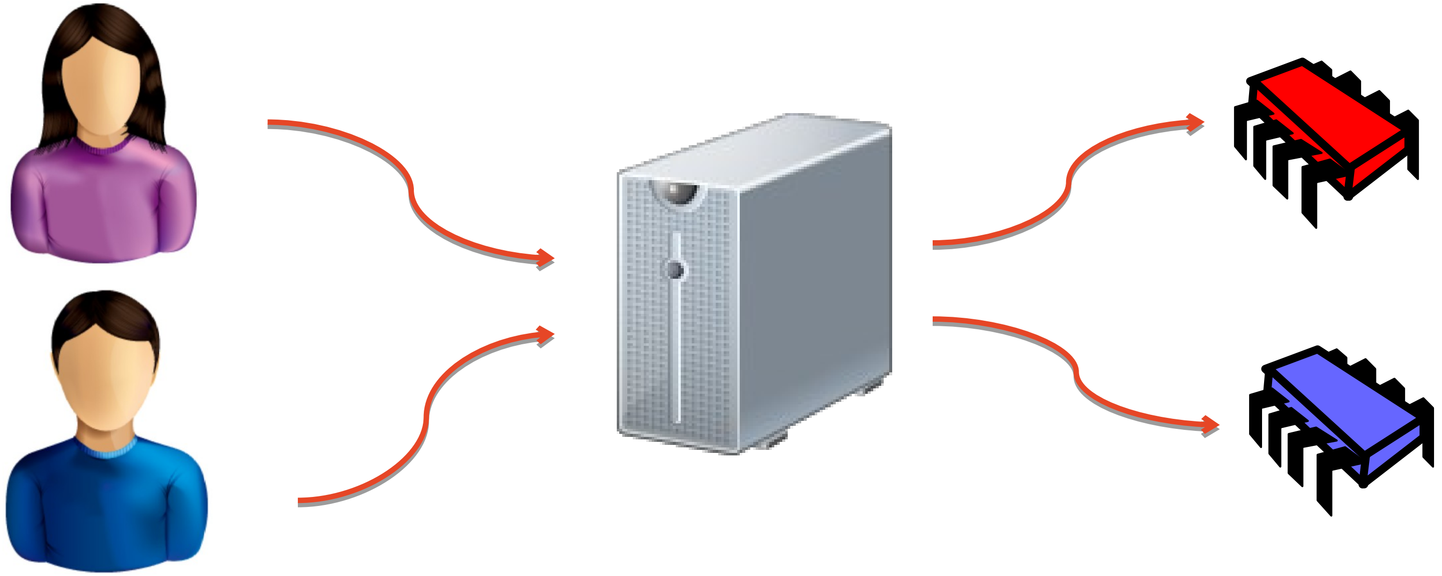
```
temp = balance;
balance = temp + amount;
```

Inconsistency

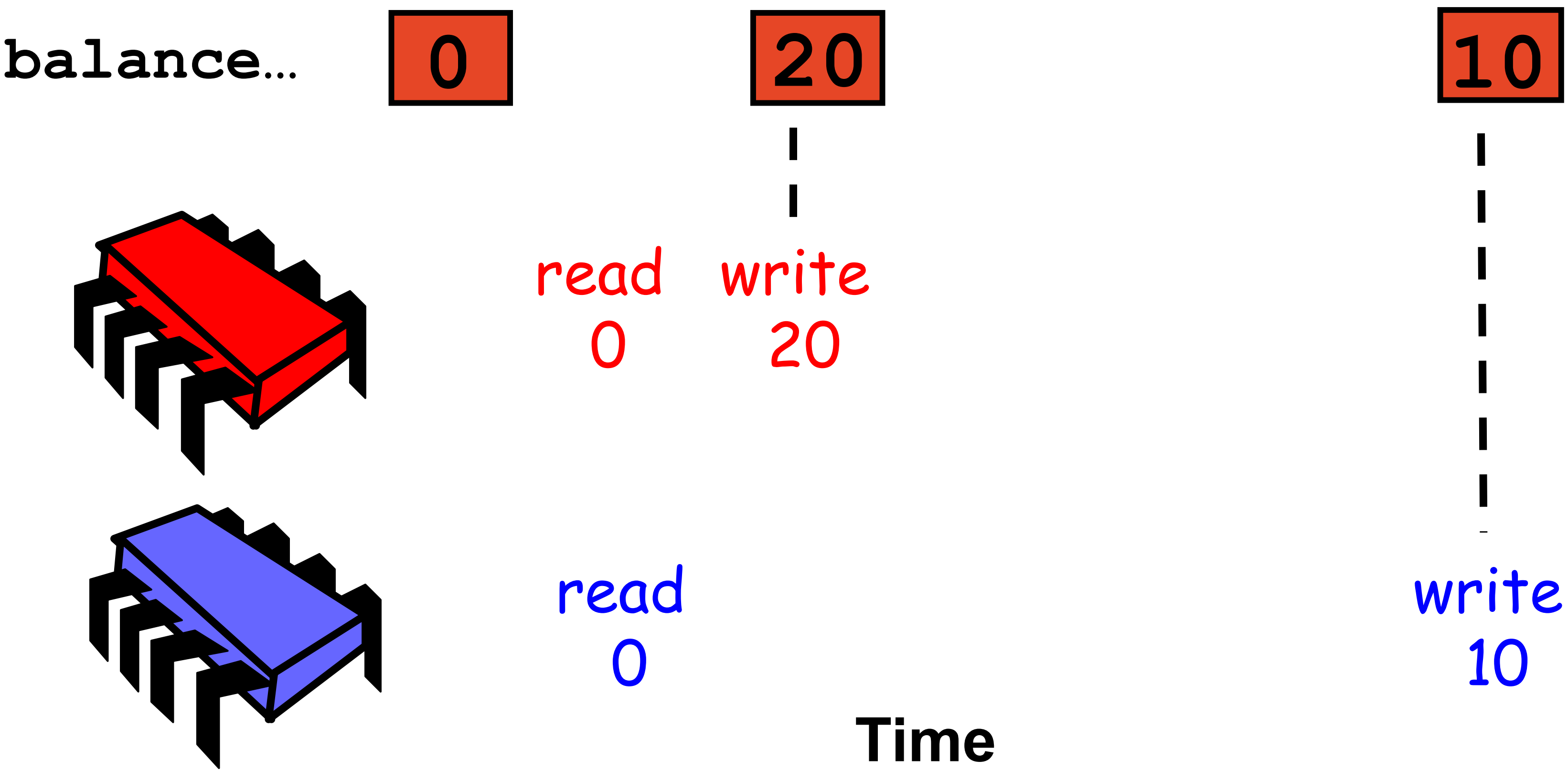
› Let Alice and Bob deposit on the same account **a** concurrently:

`a.deposit(20AUD);` // `a.deposit(10AUD);`

› The multi-threaded server handles the two requests concurrently:



Not so good…



Safety

- Challenge

```
public class BankAccount {
   int balance = 0;

   void deposit(int amount)   {

    temp = balance;
    balance = temp + amount;

   }
}
```

Make these steps indivisible, linearisable

Safety

- Hardware solution

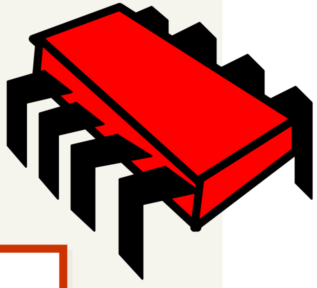
```
public class BankAccount {
   int balance = 0;

   void deposit(int amount)   {

    balance += amount;

   }
}
```

Read Modify Write Instruction



Safety

- Software solution?

```
public class BankAccount {
  int balance = 0;

  void deposit(int amount)  {
    synchronized(this) {
     temp = balance;
     balance = temp + amount;
    }
  }
}
```

Safety

- Software solution?

```
public class BankAccount {
  int balance = 0;

  void deposit(int amount)  {
    synchronized(this) {
      temp = balance;
      balance = temp + amount;
    }
  }
}
```

Synchronised block

Safety

- Software solution?

```
public class BankAccount {
   int balance = 0;

   void deposit(int amount)  {
     synchronized(this) {
      temp = balance;
      balance = temp + amount;
     }
   }
}
```

Mutual Exclusion

Safety

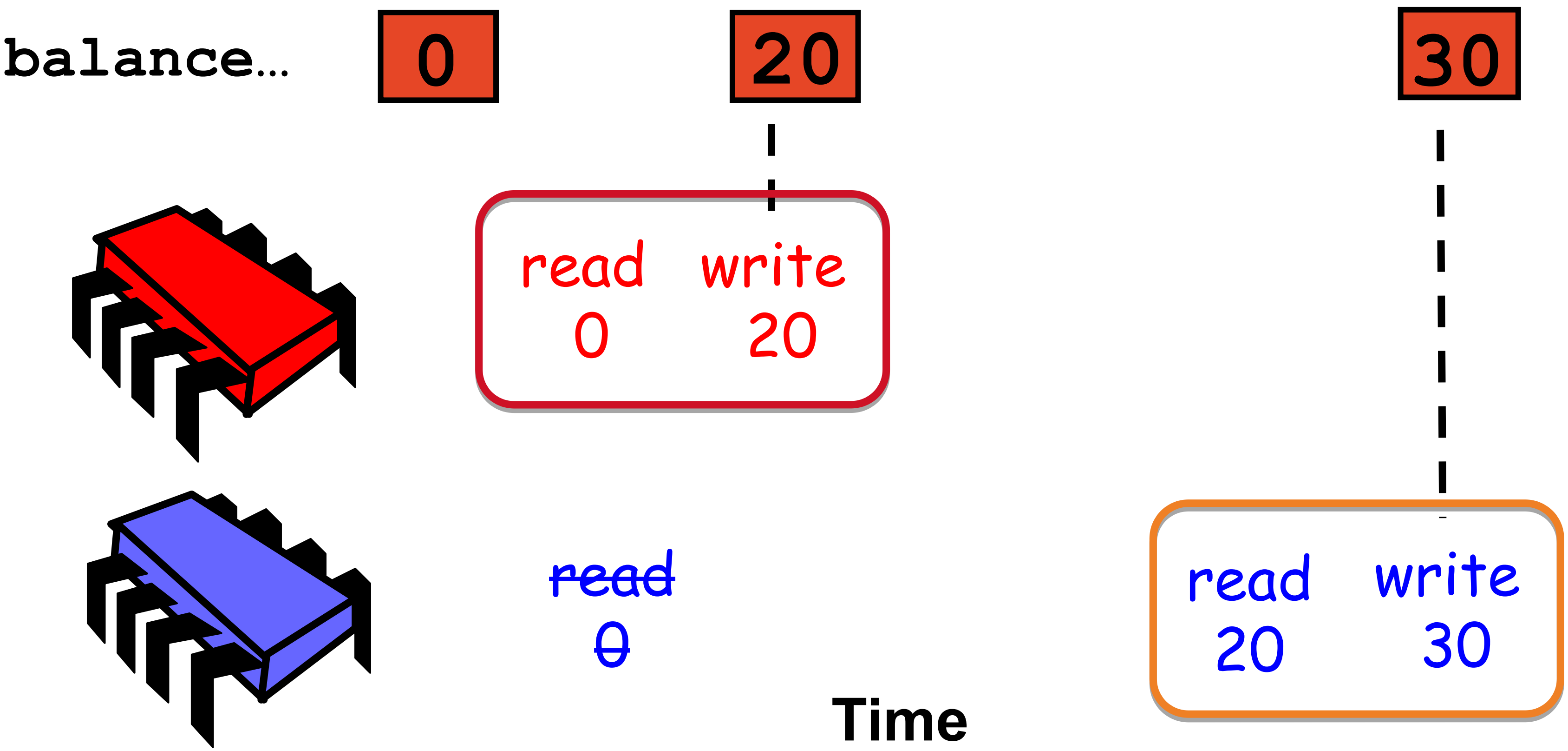
- Multiple threads cannot enter a synchronised block with the same argument at the same time. The argument represents the resource that can be used by a single thread at a time.

```
a.deposit(x);
```

//

```
a.deposit(y);
```

Not so bad…



Concurrent Programming is Hard

- Hard to make **safe** and **live**
  - Safe: *where nothing bad (e.g., crash) happens*
  - Live: *where something good (that we want) eventually happens*

We obtained a safe program

- The human mind is usually trained to be sequential
  - Concurrent specifications
  - Non-deterministic executions

Concurrent Programming is Hard

- Hard to make **safe** <u>and</u> **live**
  - Safe: *where nothing bad (e.g., crash) happens*
  - Live: *where something good (that we want) eventually happens*

- The human mind tends to be sequential
  - Concurrent specifications
  - Non-deterministic executions

But is the program also live?

Liveness

› A slightly more complex problem…

```
void deposit(…)  { synchronized(this) { … } }
void withdraw(…) { synchronized(this) { … } }
int  balance(…)  { synchronized(this) { … } }

void transfer(account from, int amount) {


   if (from.balance() >= amount) {
     from.withdraw(amount);
     this.deposit(amount);
   }


}
```

No concurrency control: race!

Liveness

```
void deposit(…)  { synchronized(this) { … } }
void withdraw(…) { synchronized(this) { … } }
int  balance(…)  { synchronized(this) { … } }

void transfer(account from, int amount) {

  synchronized(this) {
   if (from.balance() >= amount) {
     from.withdraw(amount);
     this.deposit(amount);
   }
  }

}
```

Liveness

```
void deposit(…)  { synchronized(this) { … } }
void withdraw(…) { synchronized(this) { … } }
int  balance(…)  { synchronized(this) { … } }

void transfer(account from, int amount) {

  synchronized(this) {
   if (from.balance() >= amount) {
     from.withdraw(amount);
     this.deposit(amount);
   }
  }

}
```

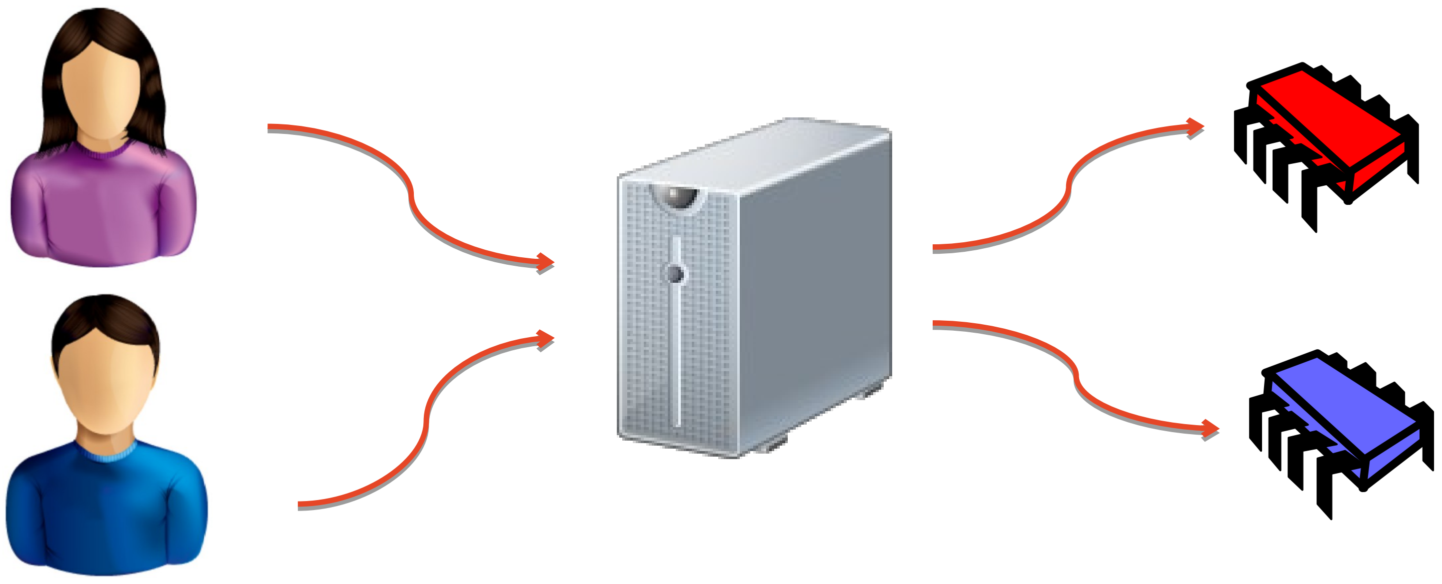
Race!

Inconsistency

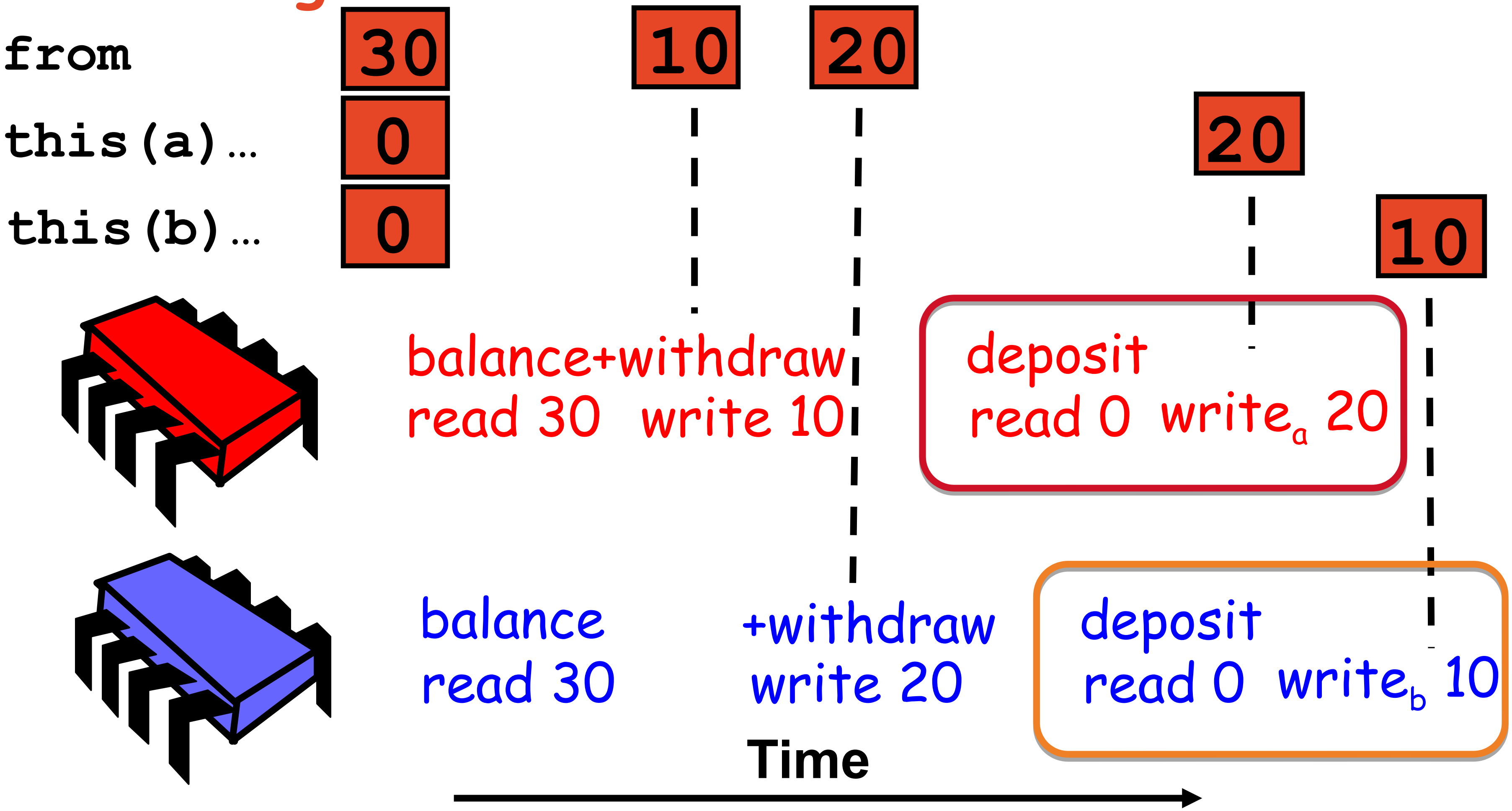
› Let Alice and Bob between the same accounts concurrently:

`a.transfer(b, 20AUD);` // `a.transfer(b, 10AUD);`

› The multi-threaded server handles the two requests concurrently:



Not so good…



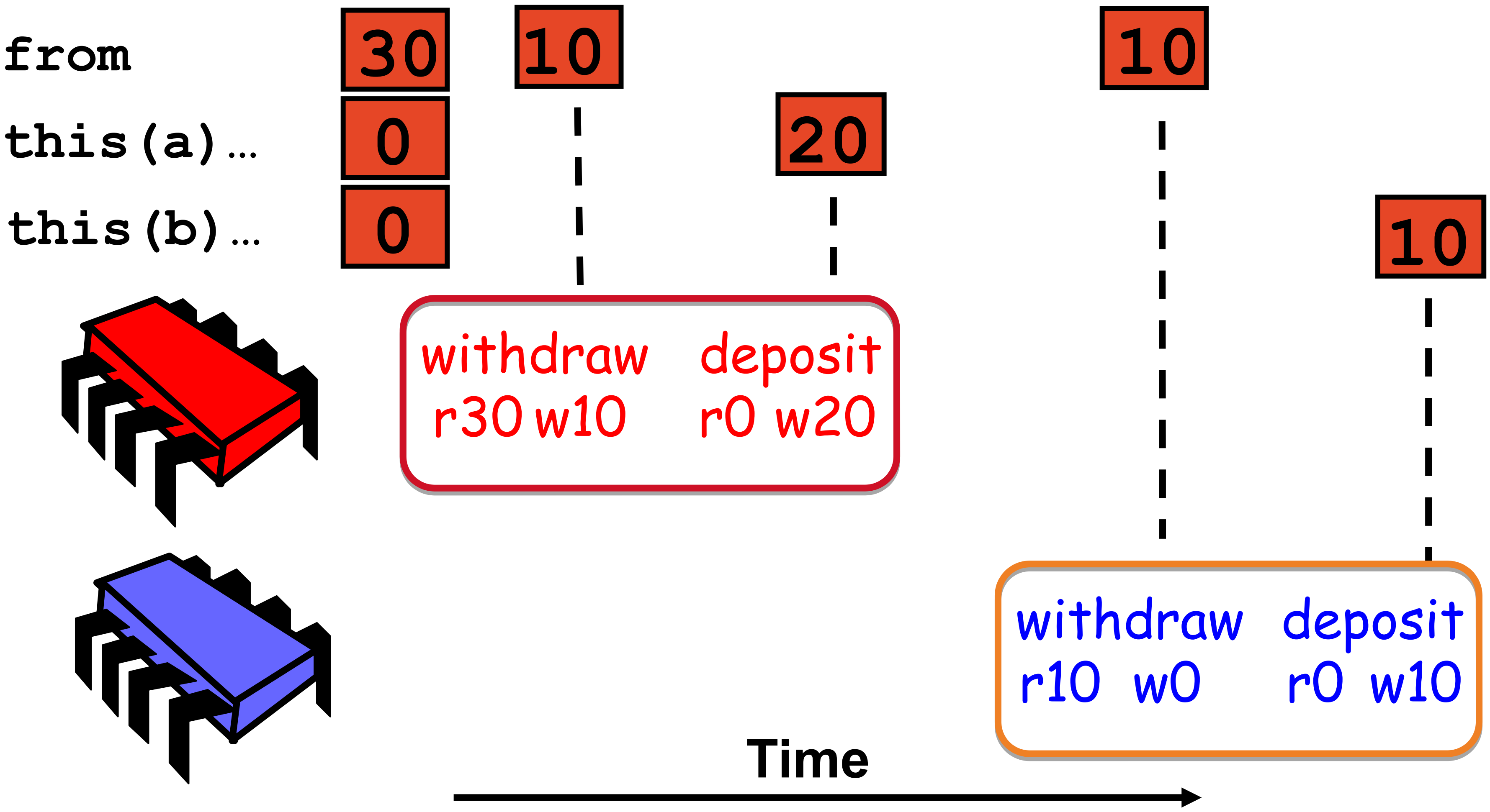
from
this(a)…
this(b)…
30
0
0
10
20
20
10
balance+withdraw
read 30 write 10
deposit
read 0 write$_a$ 20
balance
read 30
+withdraw
write 20
deposit
read 0 write$_b$ 10
Time

Liveness

```
void deposit(…)  { synchronized(this) { … } }
void withdraw(…) { synchronized(this) { … } }
int  balance(…)  { synchronized(this) { … } }

void transfer(account from, int amount) {
 synchronized(this) {
  synchronized(from) {
   if (from.balance() >= amount) {
     from.withdraw(amount);
     this.deposit(amount);
   }
  }
 }
}
```

Better…



from
this(a)…
this(b)…
30
0
0
10
20
10
10
withdraw
r30 w10
deposit
r0 w20
withdraw
r10 w0
deposit
r0 w10
Time

Liveness

```
void deposit(…)  { synchronized(this) { … } }
void withdraw(…) { synchronized(this) { … } }
int  balance(…)  { synchronized(this) { … } }

void transfer(account from, int amount) {
 synchronized(this) {
  synchronized(from) {
   if (from.balance() >= amount) {
     from.withdraw(amount);
     this.deposit(amount);
   }
  }
 }
}
```

Deadlock!

Deadlock

- Let two threads transfer between accounts **`a`** and **`b`** in opposite order:

```
a.transfer(b, x);
```

//

```
b.transfer(a, y);
```

Deadlock

› Let two threads transfer between accounts **a** and **b** in opposite order:

```
a.transfer(b, x);
```

//

```
b.transfer(a, y);
```

› Each thread may successfully acquire the first lock

```
...
synchronized(a) {
  synchronized(b) {
   ...
```

```
...
synchronized(b) {
  synchronized(a) {
   ...
```

› If so, they will never be able to acquire the second lock!

Deadlock

Deadlock

Deadlock

Consequences of deadlocks: examples in operating systems

- Google FS:
  - To make a file rename atomic when moving a file from directory `a` to directory `b`, locks must be acquired on `a` and `b`.
  - Concurrent renames moving from `a` to `b` and from `b` to `a` may deadlock.
  - Avoided by requiring lexicographical ordering for lock acquisition

- Linux kernel:
  - `linux/mm/filemap.c` starts with 50 lines of commented code to explain the order in which locks are acquired.
  - A kernel programmer must first understand this before locking anything, otherwise risking to deadlock.

Deadlock

- Multiple locks acquired in arbitrary order may lead to deadlock

- Multiple locks acquired in a single shared order are safe from deadlock
  - Lexicographical order
  - Memory address order

Transactional Memory

THE UNIVERSITY OF SYDNEY

Transactional memory

An appealing alternative:

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

Lock acquire/release

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

(As if) no interleaved computation

Easier-to-use primitive
(but harder to implement)

Transactional memory

Compiler-based instrumentation:

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

```
void deposit(int x) {
   tx-start();
   int tmp = tx-read(balance);
   tmp += x;
   tx-write(balance, tmp);
   tx-commit();
}
```

A safe and live (deadlock-free) program

Exploits the TM wrappers

A compiler supporting TM takes care of the instrumentation

Transactional memory

```
void deposit(…)  { atomic { … } }
void withdraw(…) { atomic { … } }
int  balance(…)  { atomic { … } }

void transfer(account from, int amount) {

  if (from.balance() >= amount) {
    from.withdraw(amount);
    this.deposit(amount);
  }

}
```

No concurrency control: race!

Transactional memory

```
void deposit(…)  { atomic { … } }
void withdraw(…) { atomic { … } }
int  balance(…)  { atomic { … } }

void transfer(account from, int amount) {

  atomic {
   if (from.balance() >= amount) {
     from.withdraw(amount);
     this.deposit(amount);
   }
  }

}
```

Correct and enables parallelism!

Transactional memory

Ensures safety (atomicity) and liveness (deadlock-freedom)

- Transactions: a simple paradigm

A sequence of instructions, executed atomically

- Software transactions are good for:
  - Software engineering (simple programming, avoid races & deadlocks, composability)
  - Performance (when no conflict, high parallelism and no locking overhead)

A "universal" synchronization construct

Don't care how transactions are implemented!

Transactional memory

Implementation example:

- **`tx-read(x)/tx-write(x,v)`**:
  - if unlocked, then acquire lock of x
  - else **`tx-abort()`**

- **`tx-commit()`**:
  - release all locks
  - cleanup

Transactional memory

Implementation example (continued):

```
tx-write(x,v) {
  if (!<x,v'> in rSet U wSet)
    while(!CAS(lock(x),unlocked,locked))
      tx-abort()
    wLog = wLog U {<x,store(x,v)>}
    wSet = wSet U {<x,v>}
  return ok
}
```

Transactional memory

Implementation example (continued):

```
tx-read(x) {
  if (!<x,v> in rSet U wSet)
    while(!CAS(lock(x),unlocked,locked))
      tx-abort()
    v = load(x)
    rSet = rSet U {<x,v>}
  return v
}
```

Transactional memory

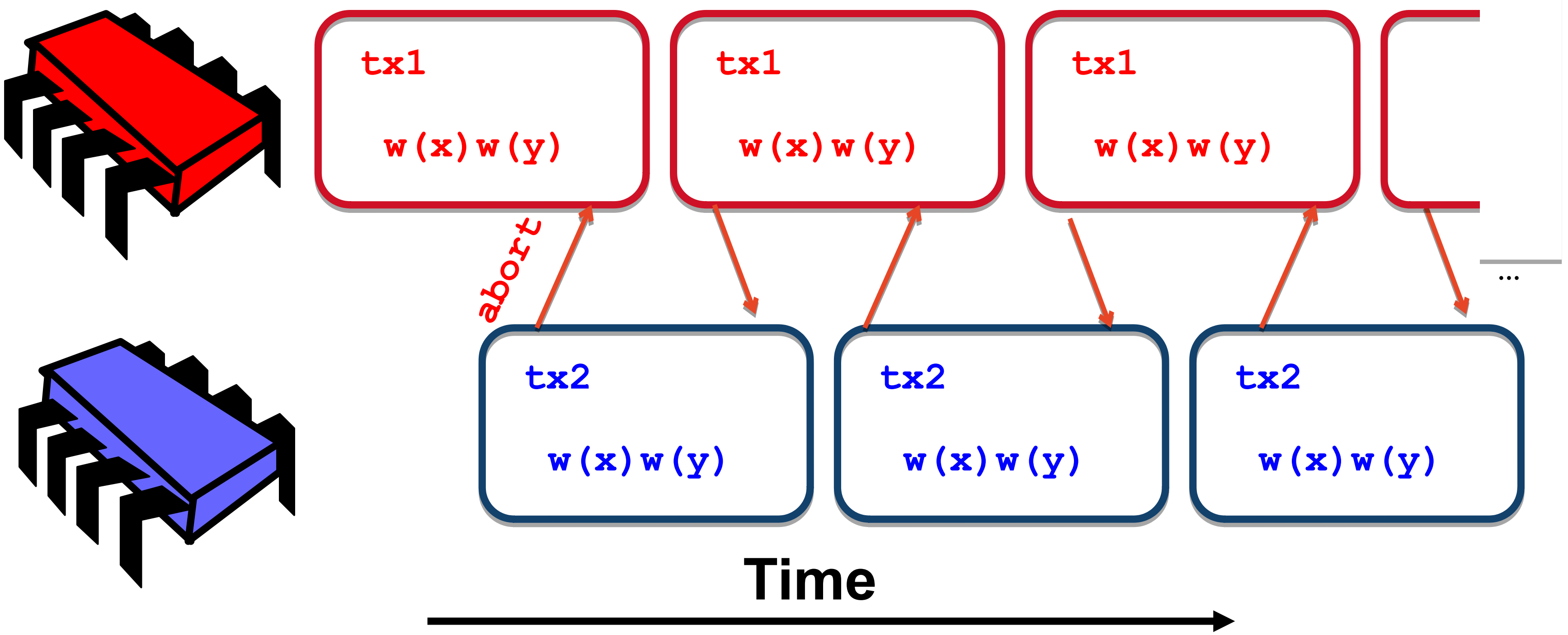
Implementation example (continued):

```
tx-commit() {
  for (x in rSet U wSet) unlock(x)
  empty rSet and wSet
}
```

```
tx-abort() {
  rollback(wLog)
  for (x in rSet U wSet) unlock(x)
  empty rSet and wSet
}
```

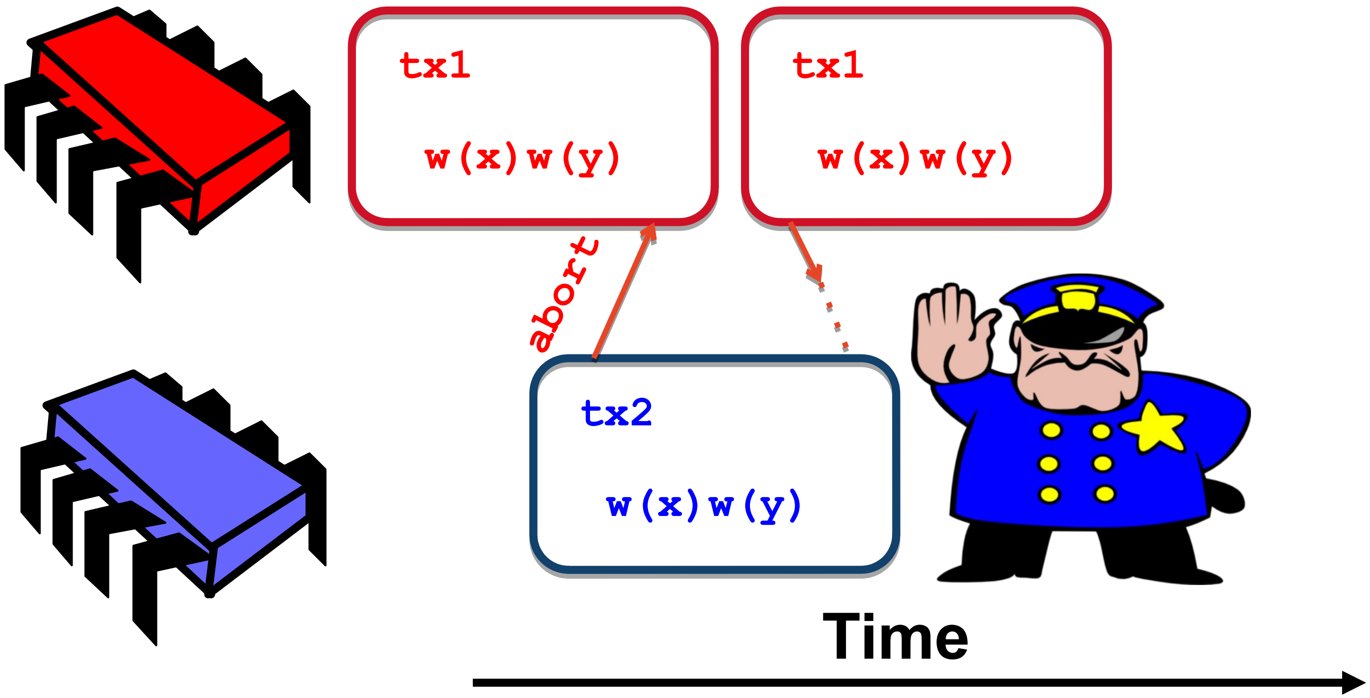
Starvation freedom

– Starvation: may happen if transactions keep aborting each other



Starvation freedom

– Contention manager: an external module called by the transactional memory to arbitrate between conflicting transactions.



Irrevocability

- An action is *irrevocable* if once it has been invoked it cannot be rolled back.
- As usual transactions execute speculatively, may abort and restart, they cannot execute irrevocable actions.

```
atomic {
    …
    fire-missile();
    …
}
```

- *Privatisation* ensures that a transaction is the only one to access data, thus guaranteeing that the transaction will commit. With privatisation, transactions can execute legacy code (and irrevocable actions) safely.

Expressiveness

Expressiveness limitations

- Transactions use a balanced open-close block whereas mutual exclusions (i.e., locks) can be ``interleaved''.

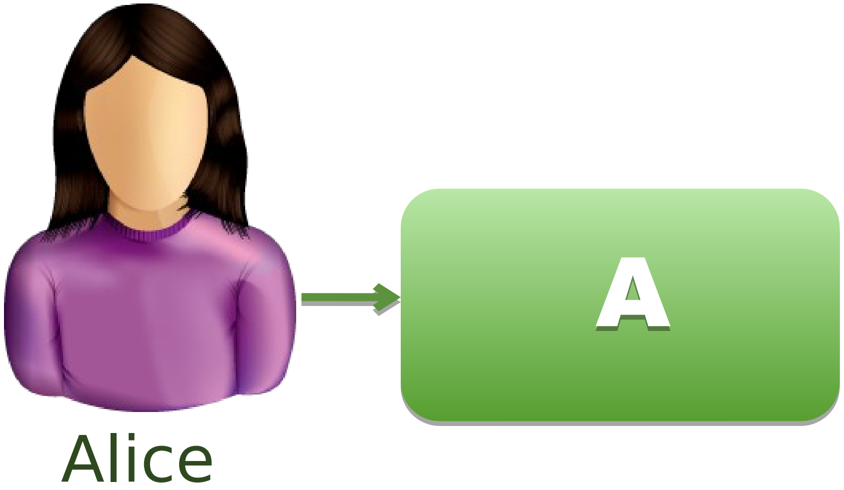
```
atomic {
    read(x)
    read(y)
    read(z)
}
```

```
lock(x)
    read(x)
lock(y)
unlock(x)
    read(y)
lock(z)
unlock(y)
    read(z)
unlock(z)
```

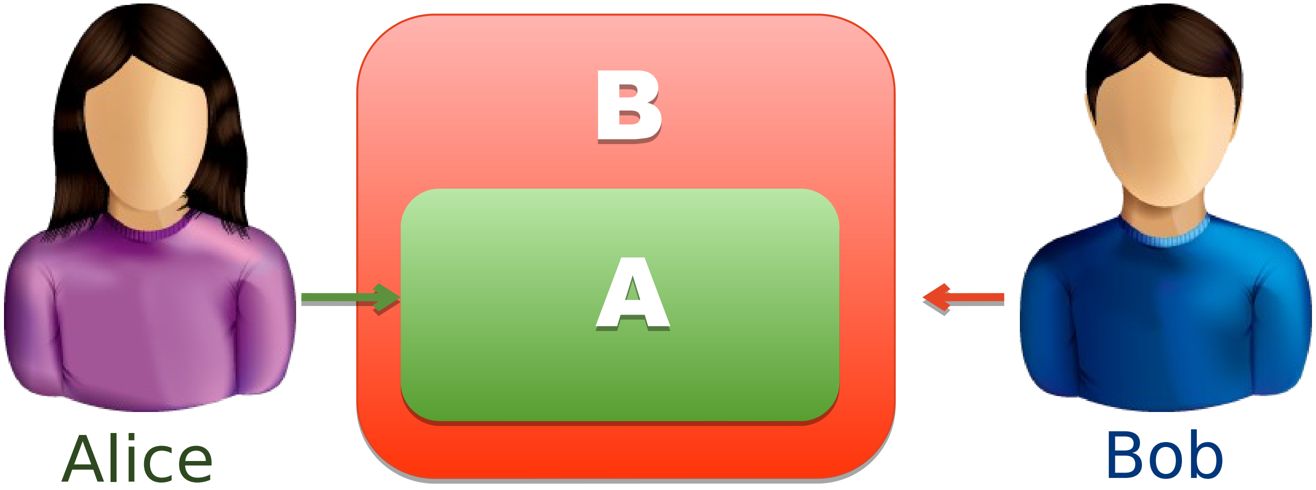
- *Relaxed transactions* help bridge the expressiveness gap between regular ones and locks.

Reusability

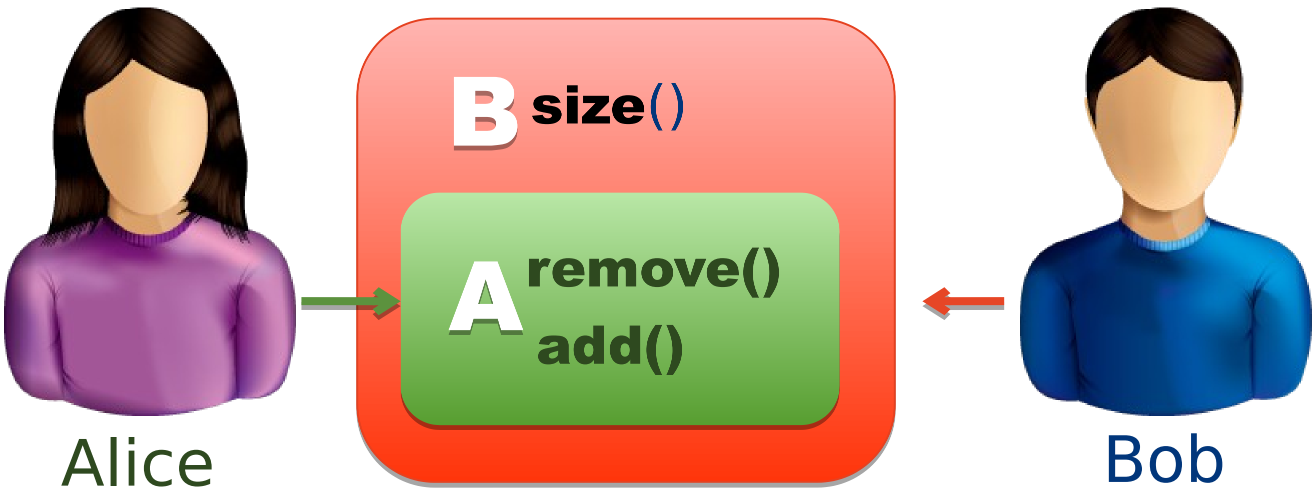
Reusability simplifies programming



Reusability simplifies programming

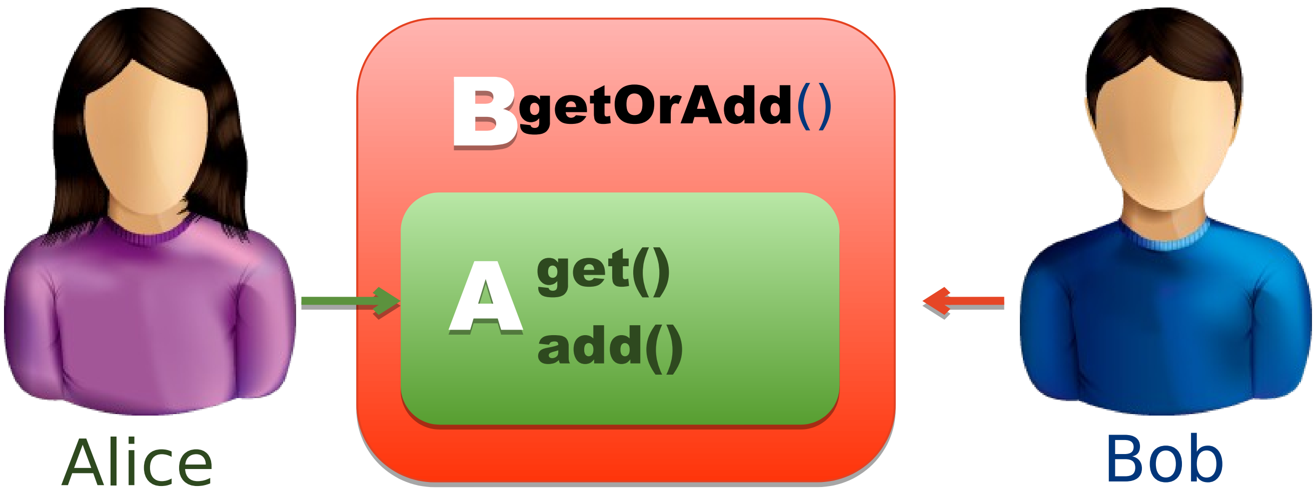


Reusability simplifies programming



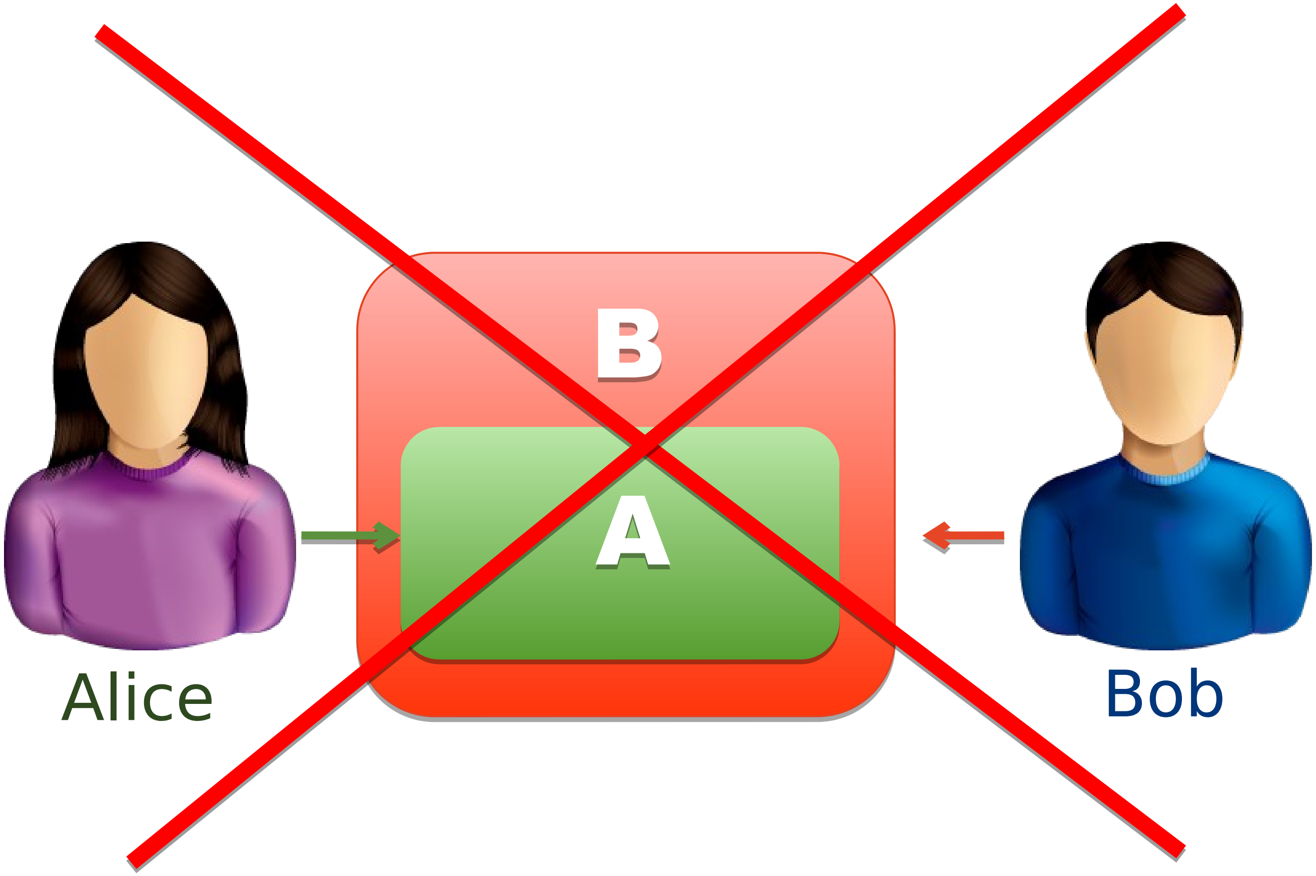
...through **extension** (adding a new method)

Reusability simplifies programming



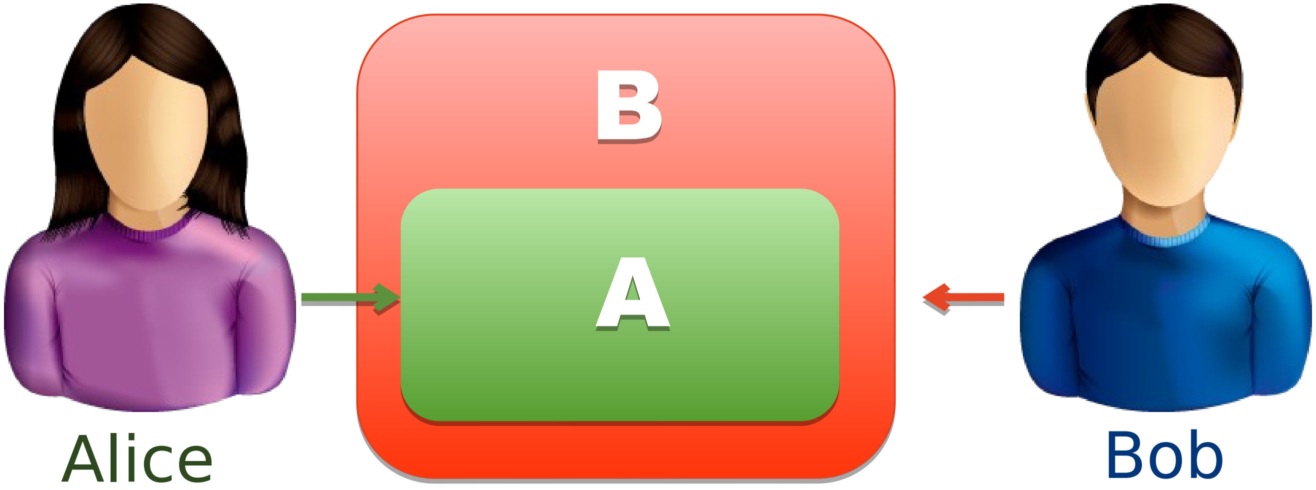
and through **composition** (composing methods into a new one)

Reusability simplifies programming



Not in a traditional concurrent environment

Reusability simplifies programming



Works with transactional memory

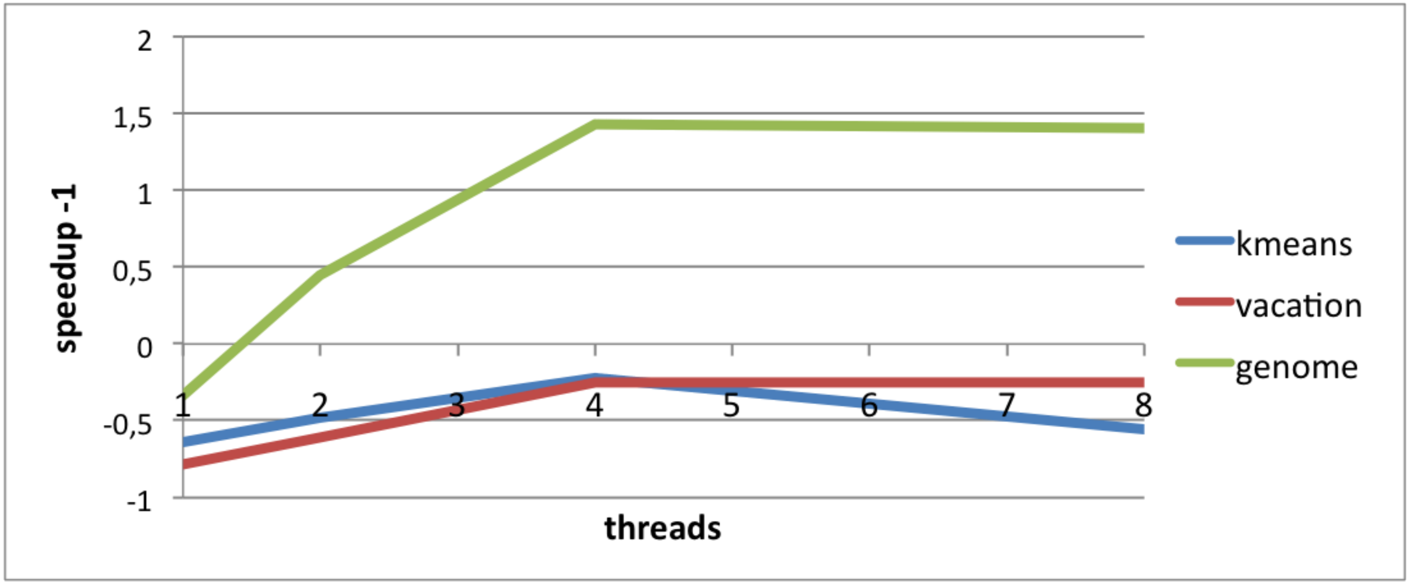
Software and Hardware TMs

Overhead of Software TM (STM)

6 researchers from IBM claimed that *"STM is only a research toy"* [IBM, CACM 2008] because:

Overhead of Software TM (STM)

6 researchers from IBM claimed that *“STM is only a research toy”* [IBM, CACM 2008] because:



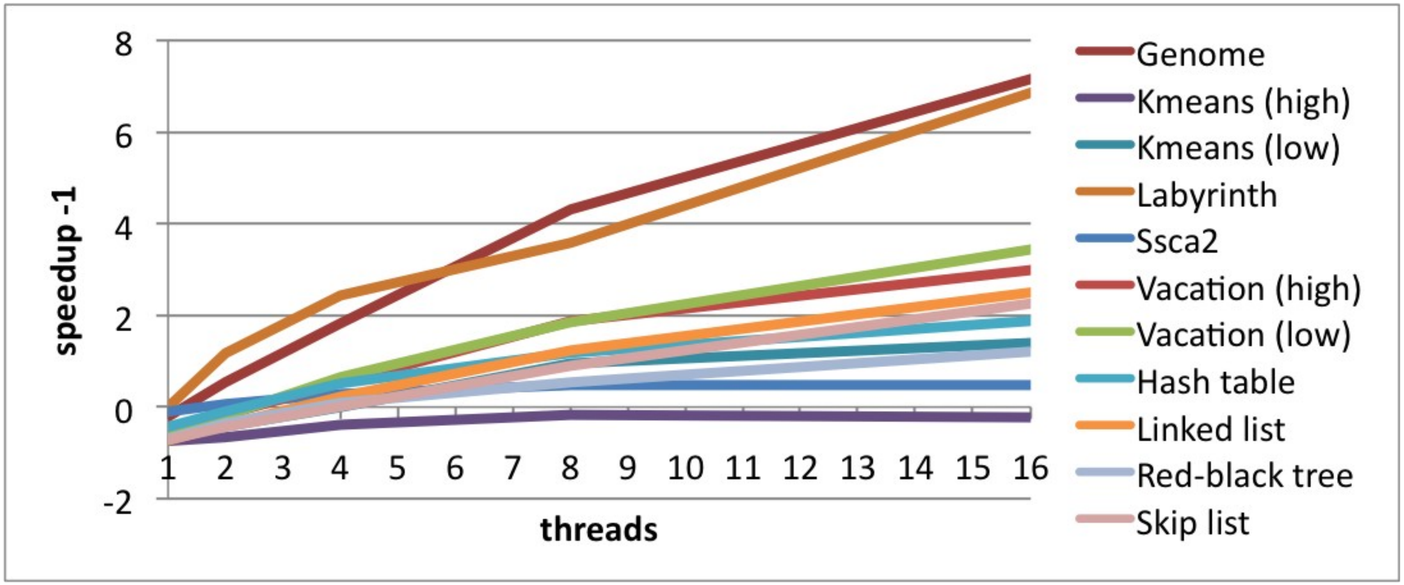
– STM overhead was too high to speed up sequential performance
– Performance was not even scaling to 8 threads

Overhead of Software TM (STM)

- We did a *thorough* analysis (benchmarks, STMs, architectures, compiler, privatisation, improvements)

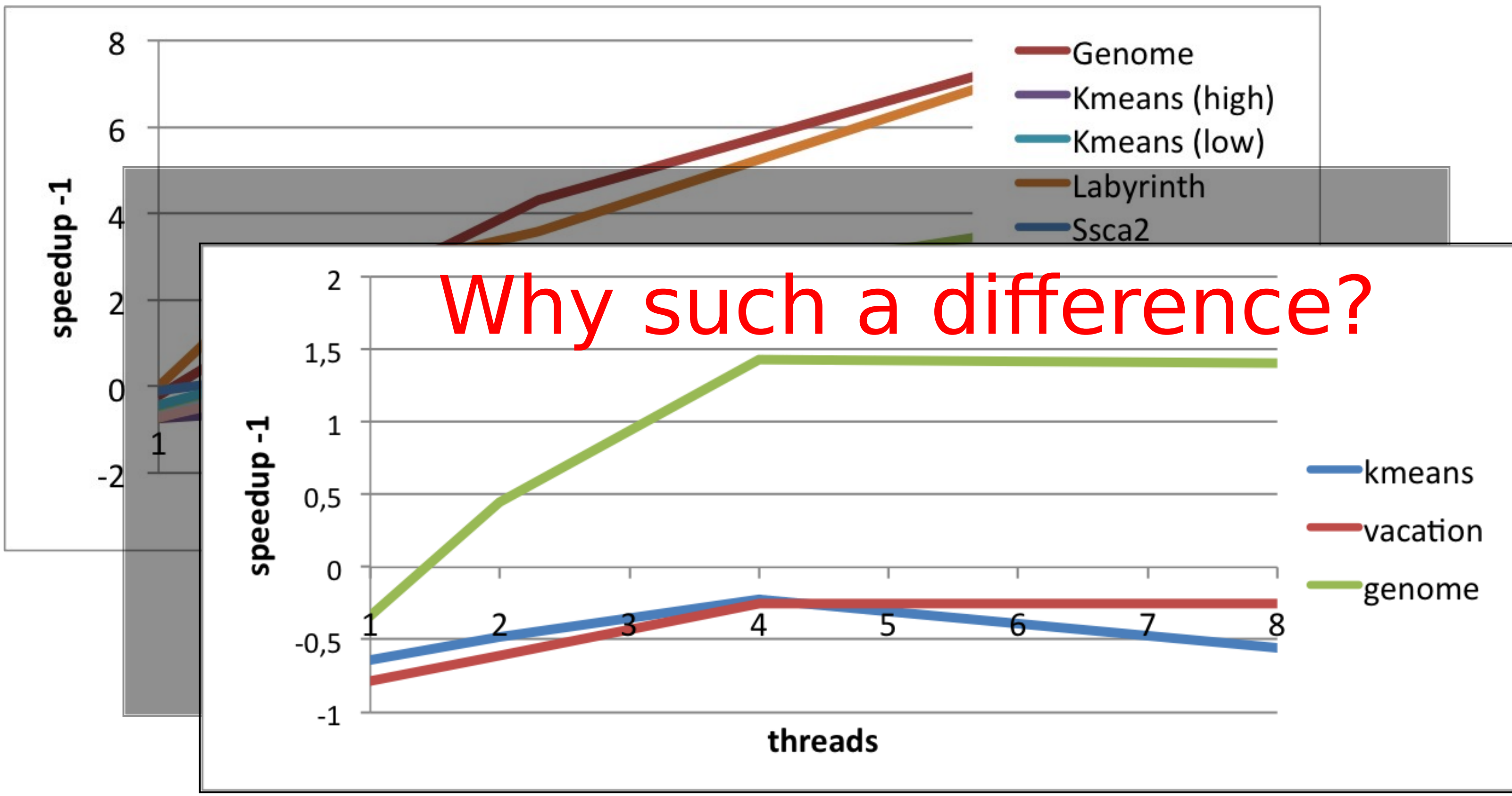
Overhead of Software TM (STM)

- We did a *thorough* analysis (benchmarks, STMs, architectures, compiler, privatization, improvements)



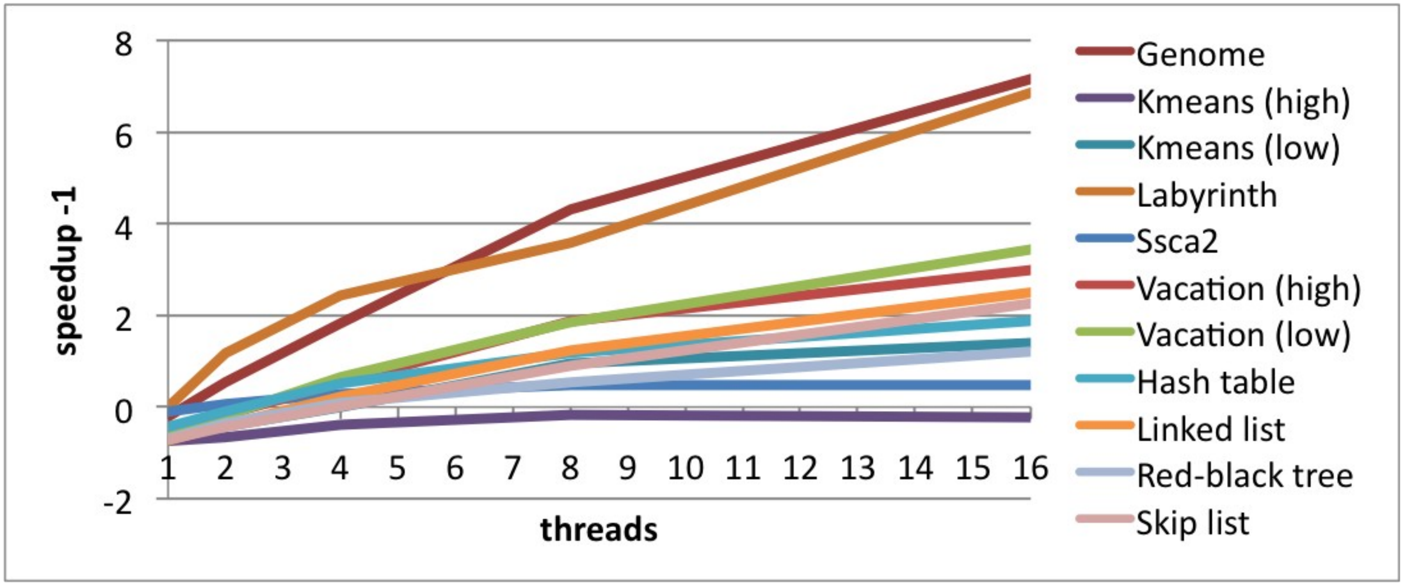
Overhead of Software TM (STM)

– We did a *thorough* analysis (benchmarks, STMs, architectures, compiler, privatization, improvements)



Overhead of Software TM (STM)

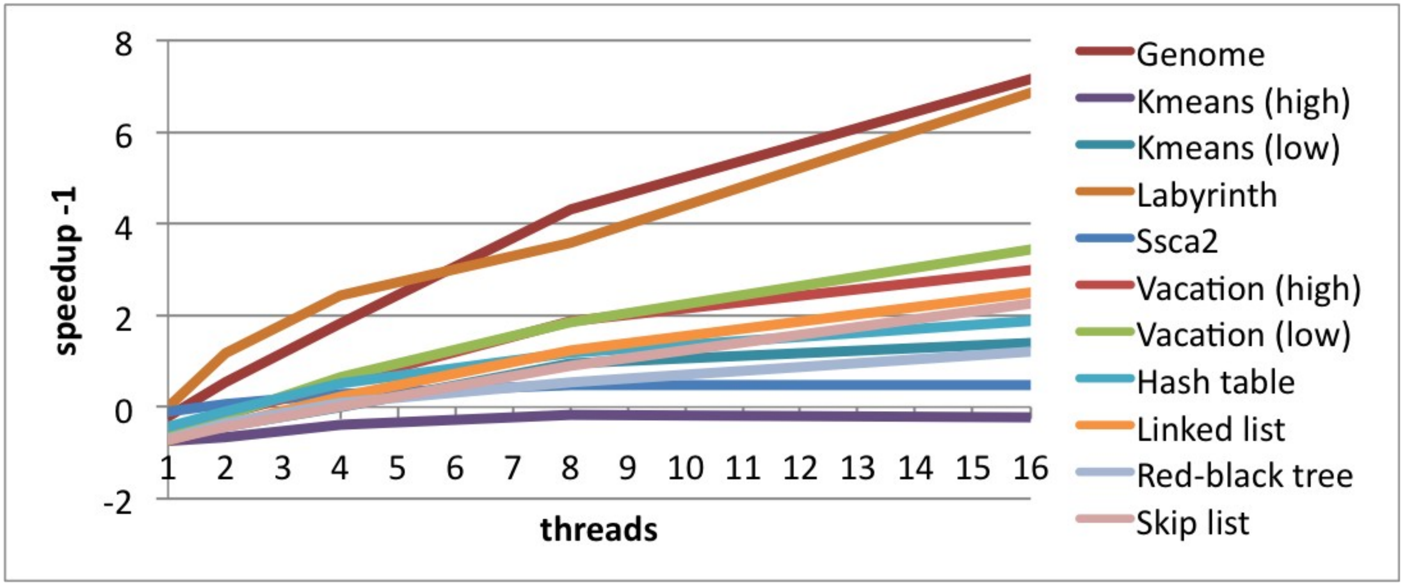
- We did a *thorough* analysis (benchmarks, STMs, architectures, compiler, privatization, improvements)



- <u>Problem</u>: 4 hyperthreaded cores ≠ 8 cores

Overhead of Software TM (STM)

- We did a *thorough* analysis (benchmarks, STMs, architectures, compiler, privatization, improvements)



- Problem: 4 hyperthreaded cores ≠ 8 cores
- Our rebuttal, *"Why STM can be more than a research toy"*, was published [CACM '11]

Hardware TM (HTM)

- February 2012: Transactional Synchronization Extensions (TSX-NI).
  - Hardware Lock Elision (HLE)
  - Restricted Transactional Memory (RTM)

- August 2014: Intel announced a bug in the TSX implementation on current steppings of Haswell, Haswell-E, Haswell-EP and early Broadwell CPUs

- November 2014: Bug fixed, in the vPro-enabled Core M-5Y70 Broadwell CPU in November 2014.

- 2015: IBM Power8 supports Hardware Transactional Memory

- 2015: Intel Skylake has TSX re-enabled.

Why should we care?

- It should be easy to make concurrent programs **safe** and **live:**
  - Safe: it ensures atomicity of transactions
  - Live: it generally ensures that a transaction commits if executing in isolation for long enough and sometimes that every transaction eventually commits

- Reusability/Composability:
  - A transaction can encapsulate multiple ones so that no knowledge is a priori necessary
  - Which makes concurrent programming modular

- Empirical studies indicate that concurrent programming with TM is:
  - Simpler
  - Safer
  - Faster ...than with existing synchronisation techniques.