Lazy Software Transactional Memory

Master Thesis

Lasse Folger

05.04.2017

Motivation

```
type Account = MVar Int

transfer :: Account -> Account -> Int -> IO ()
transfer src dst am = do
  a1 <- takeMVar src
  a2 <- takeMVar dst
  putMVar src (a1 - am)
  putMVar dst (a2 + am)
```

MVar

Thread 1:

```
transfer acc1 acc2 50
```

Thread 2:

```
transfer acc2 acc1 50
```

MVar

Thread 1:

```
a1 <- takeMVar acc1
a2 <- takeMVar acc2
writeMVar acc1 (a1 - 50)
writeMVar acc2 (a2 + 50)
```

Thread 2:

```
b1 <- takeMVar acc2
b2 <- takeMVar acc1
writeMVar acc2 (b1 - 50)
writeMVar acc1 (b2 + 50)
```

MVar

Thread 1:

```
a1 <- takeMVar acc1
a2 <- takeMVar acc2
writeMVar acc1 (a1 - 50)
writeMVar acc2 (a2 + 50)
```

Thread 2:

```
b1 <- takeMVar acc2
b2 <- takeMVar acc1
writeMVar acc2 (b1 - 50)
writeMVar acc1 (b2 + 50)
```

$\Rightarrow$ Deadlock

Use Transactions

```
type Account = TVar Int

transfer :: Account -> Account -> Int -> STM ()
transfer src dst am = do
  a1 <- readTVar src
  a2 <- readTVar dst
  writeTVar src (a1 - am)
  writeTVar dst (a2 + am)
```

TVar

Thread 1:

```
atomically $
  transfer acc1 acc2 50
```

Thread 2:

```
atomically $
  transfer acc2 acc1 50
```

TVar

Thread 1:

```
atomically $
  transfer acc1 acc2 50
```

Thread 2:

```
atomically $
  transfer acc2 acc1 50
```

$\Rightarrow$ works fine, because transactions provide ACI(D) properties

Current Implementation (Control.Concurrent.STM)

- *writeTVar*, *readTVar* and *newTVar* modify TVars
- *retry* and *orElse* alter the control flow
- *atomically* executes a transaction
- composition via bind operator (or do)

Transactional Log

- one log per transaction
- three elements per log entry
  - **TVar**
  - **expectedValue**
  - **currentValue**

Modify Operations

- **newTVar**: creates a new, initialized TVar
- **writeTVar**: updates *currentValue* in log entry
- **readTVar**: reads TVar from log or actual TVar

atomically :: STM a -> IO a

1. compute the log
2. lock TVars
3. validate the log
4. if valid then commit
5. else roll back

Validation

1. compare *expectedValue* to the value in the actual TVar
2. if all values match return valid
3. else return invalid

Problem

Thread 1:

```
a1 <- readTVar acc1
a2 <- readTVar acc2
writeTVar acc1 (a1 - 50)
writeTVar acc2 (a2 + 50)
```

Thread 2:

```
b1 <- readTVar acc2
b2 <- readTVar acc1
writeTVar acc2 (b1 - 50)
writeTVar acc1 (b2 + 50)
```

Problem

Thread 1:

```
[(acc1, a1, a1-50),
 (acc2, a2, a2+50)]
```

Thread 2:

```
[(acc2, b1, b1-50),
 (acc1, b2, b2+50)]
```

Problem

Thread 1:

```
a1 <- readTVar acc1
a2 <- readTVar acc2
writeTVar acc1 (a1 - 50)
writeTVar acc2 (a2 + 50)
```

Thread 2:

```
b1 <- readTVar acc2
b2 <- readTVar acc1
writeTVar acc2 (b1 - 50)
writeTVar acc1 (b2 + 50)
```

$\Rightarrow$ either sequential or one transaction is rolled back

Critical TVar

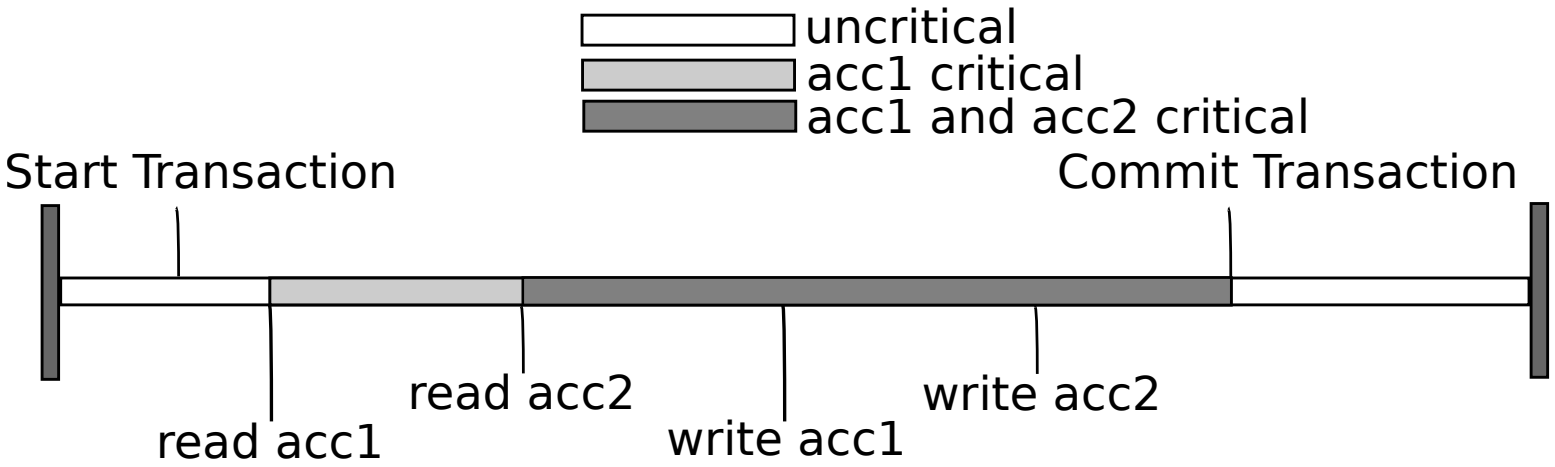
- Critical between read and commit
- modifications to critical TVars cause rollback

Critical TVar

- Critical between read and commit
- modifications to critical TVars cause rollback

$\Rightarrow$ minimize the time TVars are critical

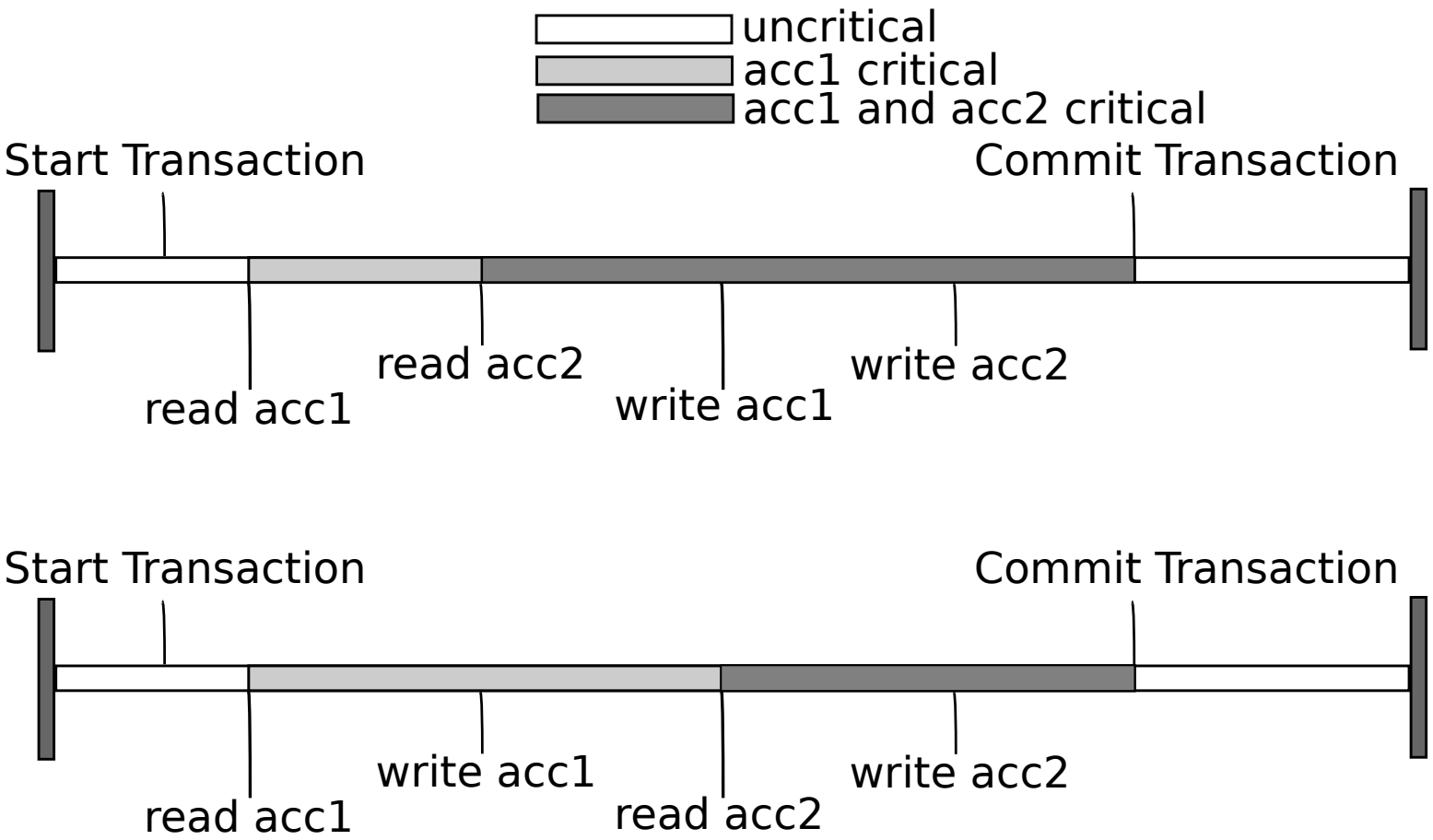
Critical TVar



Idea

```
transfer = do
  a1 <- readTVar acc2
  writeTVar acc2 (a1 - 50)
  a2 <- readTVar acc1
  writeTVar acc1 (a2 + 50)
```

Critical TVar



Idea

- delay the evaluation of **readTVar** to commit phase
- no TVars is critical at all
- **writeTVar** does not need the value in the computation phase

Idea does not work

```
limitedTransfer src dst am = do
    a1 <- readTVar src
    if a1 < am
      then return ()
      else do a2 <- readTVar  dst
              writeTVar src (a1 - am)
              writeTVar dst (a2 + am)
```

$\Rightarrow$ idea does not work because the value is needed.

Solution

- delay evaluation as far as possible
- evaluate them just before they are needed..
- ..or in the commit phase

When is a value needed?

- branch conditions
  - if-then-else
  - case
  - patternmatching
  - guards
- IO-actions $\Rightarrow$ not allowed in STM

New Combinators

- (<*>) :: STM (a -> b) -> STM a -> STM b
- (<**>) :: STM a -> STM (a -> b) -> STM b
- (**>) :: STM a -> (STM a -> STM b) -> STM b
- (>>=) :: STM a -> (a -> STM b) -> STM b
- (>>) :: STM a -> STM b -> STM b

New Transfer

```
transfer :: Account -> Account -> Int -> STM ()
transfer src dst am = do
  readTVar src <**> pure (- am) **> writeTVar src
  readTVar dst <**> pure (+ am) **> writeTVar dst
```

Problem solved

Thread 1:

```
atomically $
  transfer acc1 acc2 50
```

Thread 2:

```
atomically $
  transfer acc2 acc1 50
```

$\Rightarrow$ no more rollback

Todo

- Reduce the number of combinators
- Change writeTVars type and use ApplicativeDo
- **But** it extracts the Value from STM context
- investigate other problems:
  - Branch condition is not changed by TVar modification
  - Recomputation of values which did not change

Unnecessary Recomputation

```
transaction = do
  ImitedTransfer acc1 acc2 50
  limtedTransfer acc3 acc4 100
```

If one transfer is invalidated, both are recomputed

Questions about...

- ...Control.Concurrent.STM?
- ...rollback avoidance?
- ...unnecessary recomputation?
- ...something else?