Software Transactional Memory vs. Locking in a Functional Language:
A Controlled Experiment

Fernando Castor

João Paulo Oliveira

André L. M. Santos

Informatics Center – Federal University of Pernambuco – Brazil

SPLASH'2011 Workshop on Transitioning to Multicore. Portland, USA, October 23rd 2011.

As a consequence of multicore…

- New techniques are being devised, e.g.
  - Concurrent revisions (SPLASH'2010)
  - Cooperative reasoning (PPoPP'2011)
- And old ones are being rediscovered
  - Software Transactional Memory
  - Functional Programming

TIOBE Programming Community Index for August 2011

August Headline: F# enters the top 20 for the first time

Finally, a new functional programming language has hit the top 20. Most people thought that hot functional languages such as Scala (#66), Clojure (#107), Haskell (#35) or Erlang (#48) would be the ones that would be the first to compete seriously with the mother of all functional languages Lisp. But it appears to be Microsoft language F#. The recent rise in popularity of F# comes as no suprise. Apart from being a nicely designed language, F# is available in the latest version of Microsoft's Visual Studio (2010).

Source: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Source: http://www.360magazine.co.uk/wp-content/uploads/2011/06/Raiders-of-the-Lost-Ark-indiana-jones-3677988-1280-720.jpg

Observation 1: the few mainstream programming languages that include STM are functional

Observation 2: there is no evaluation of STM in a functional language with a focus on software engineering

Our goal

- Evaluate Haskell's STM
  - Compare it with another concurrency mechanism
- By means of a **controlled experiment**
  - Program with mutual exclusion (coarse-grained) and sync. requirements
- Focusing on ease of use
  - Number of errors, development time, num. of LoC

Haskell's mutable variables

```
foo :: (MVar Int) -> IO ()
foo buf =
    do b <- takeMVar buf
       ... -- do something
       putMVar buf (b-500)
       -- putMVar buf b -- deadlock
```

(Software) Transactional Memory

```
consume :: TVar Int -> TVar Int -> IO()
consume x total =
    atomically (do
          current <- readTVar x
          currTot <- readTVar total
          if (currTot < 1000000)
             then do
                if (current < 500) then retry
                else writeTVar x (current - 500)
             else return ()
    )
```

Study Setting

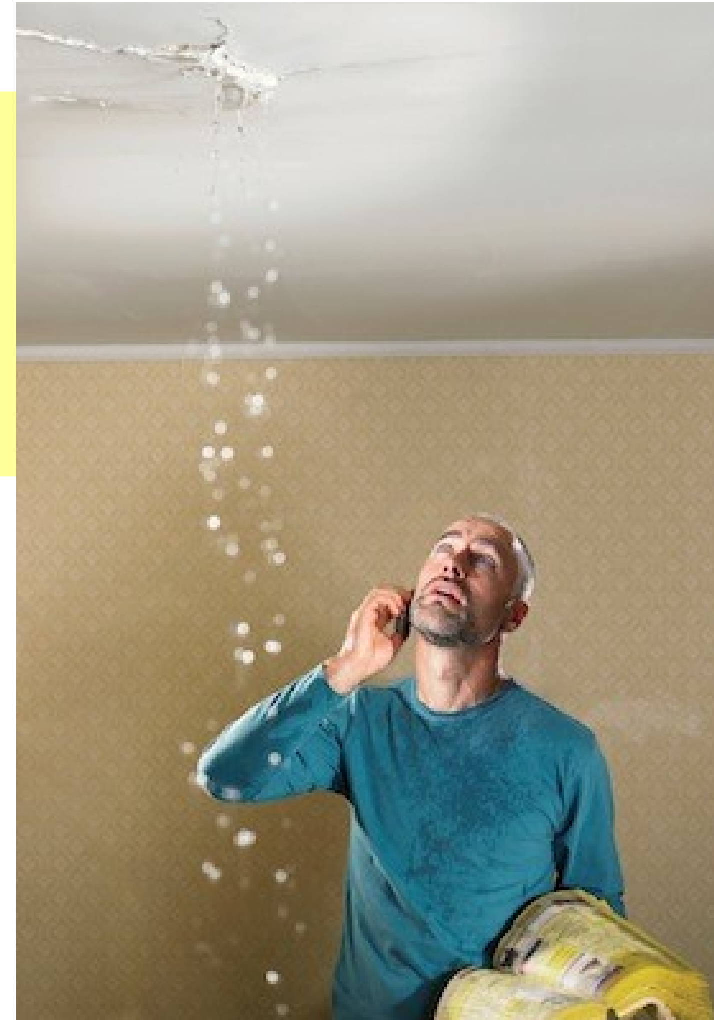
Subjects

- 51 undergraduate students
- 18+ hours training on Haskell
- 16+ hours training on concurrent programming
- Randomly assigned to one of two groups
  - STM group and MV group

Source: http://1.bp.blogspot.com/-FcubN4yc11s/TbYxSCsdi1I/AAAAAAAAABM/iYsiKDlO4Q0/s1600/lab-mice.jpg

Assignment

- Three leaks in the ceiling
  - Each leak: 10-100 drops
- One bucket per leak (5000 drops cap.)



Source: http://plumbing-fortcollins-co.info/wp-content/uploads/2011/01/emegrency-plumbing.jpg

Assignment

- Three leaks in the ceiling
  - Each leak: 10-100 drops
- One bucket per leak (5000 drops cap.)

- Man empties buckets using a mug
  - 500 drops capacity
  - Chooses a bucket and waits until it reaches 500 drops
- Simulate this situation with threads
  - Until 1.000.000 drops have fallen into the buckets

Source: http://plumbing-fortcollins-co.info/wp-content/uploads/2011/01/emegrency-plumbing.jpg

Data collection

- **Time** spent developing the program
- **Number of LoC** of the resulting programs
- **Errors** that each subject committed
  - Compilation and Logic errors
  - Concurrency errors that cause programs to hang (*hanging errors*)
  - Concurrency errors that do not cause programs to hang (*non-hanging errors*)
- **Survey** (but do not report on it in this presentation)

Hypotheses

- **Independent variable**: concurrency control technique (STM vs. MV)

- **Null hypotheses**: The average values

  – number o fLoC (NL)

  – time spent (*T*)

  – number of errors of each kind (*NE, PC, PL, PH, PN)*

  – *NLC (NL for progs. without conc. Errors)*

  – *TC* (similar to *NLC,* but applies to *T*)

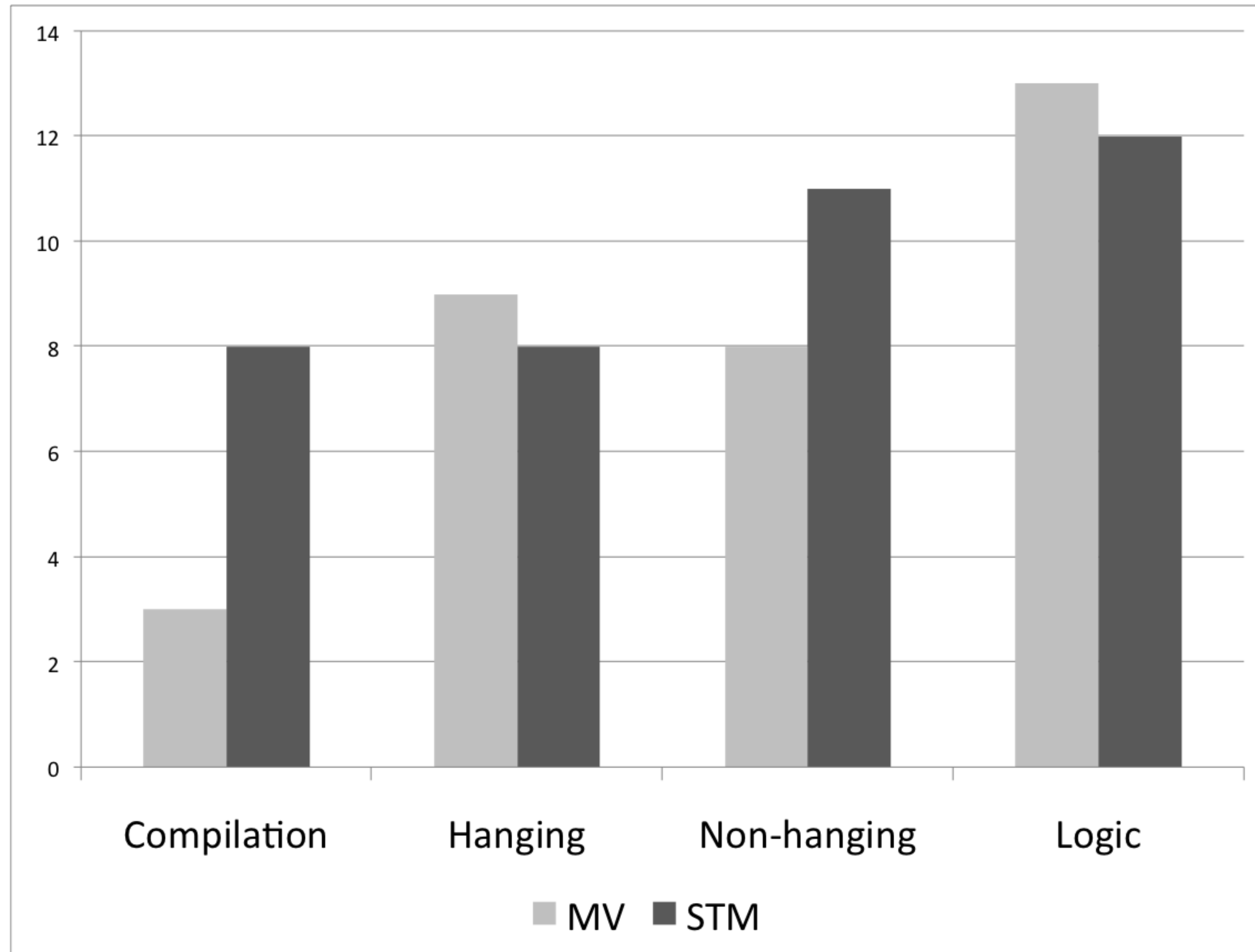
  are not significantly different for the two techniques

Results

In a nutshell…

- MV and STM not significantly different
  - Using Student’s T test with 95% confidence
  - Cannot reject the first 7 hypotheses

Overall number of errors



Compilation and non-hanging errors

```
consume :: TVar Int -> IO ()
consume bucket =
     do
        drops <- atomically(readTVar bucket)
        if (drops < 500) then
           do retry  // in Haskell, type error
           else
              do
                 atomically (
                     writeTVar balde (drops - 500))
        return ()
```

Hanging errors

- MV group: mostly mismatched `takeMVar` and `putMVar`
  - Expected result
- Main reasons for hanging errors in the STM group:
  - Race conditions that produced infinite recursion
  - Livelock due to coarse-grained transactions

Coarse-grained transaction

```
produce:: TVar Int -> TVar Int -> MVar Int -> IO()
produce bucket drops end = do ...
    atomically(
        do ...
            x <- readTVar bucket
            if(x >= 5000) then retry
            else do ...
            if(....) then
                do ...
                   produce bucket drops end
            else ...
    )
    ...
```

Avoiding the problem…

```
leak bucket drops ... =
  do ...
    atomically (do
      total <- readTVar drops;
      if (total < 1000000) then do {
        ...
        if (volBucket < 5000)
        then do ...   -- add drops to the bucket
        else retry;   -- fail now. Try again later.
        ...
      } else return (); })
    nTot <- atomically (readTVar drops);
    if (nTot < 1000000) then leak bucket drops ...
    else ...        --got to 1000000 drops. Finish.
```

Avoiding the problem…

```
leak bucket drops ... =
  do ...
    atomically (do
      total <- readTVar drops;
      if (total < 1000000) then do {
        ...
        if (volBucket < 5000)
        then do ...   -- add drops to the bucket
        else retry;   -- fail now. Try again later.
        ...
      } else return (); })
    nTot <- atomically (readTVar drops);
    if (nTot < 1000000) then leak bucket drops ...
    else ...        --got to 1000000 drops. Finish.
```

Requires **double-checking**

Time spent and number of LoC

- Very similar
  - Specially for time spent

Technique	Measure	Mean	Std. Dev.	Median	Min.	Max
MV	LoC	115.54	31.83	110	52	174
	Time	144.42	29.51	141	75	180
STM	LoC	110.24	36.63	99	65	217
	Time	138.32	33.33	134	80	180

- Was this result somewhat uniform?

Time and LoC considering programs without concurrency errors

- 21 programs
  - 12 belong to the MV group
  - 9 to the STM group

- For time, *p-value* = 0.0292
  - Statistically different with 95% confidence
- For LoC, p-value = 0.0562
  - Not enough to reject the null hypothesis, but barely

What hindered STM subjects?

- Subjects who “got it right” were considerably faster
  - Hence, maybe the **concurrency bugs** delayed them

Source: http://leonmeijer.nl/images/leonmeijer_nl/WindowsLiveWriter/TestdrivendevelopmentUni.NETwhatsallthis_D86E/sw_testing.jpg

What hindered STM subjects?

- Subjects who “got it right” were considerably faster
  - Hence, maybe the **concurrency bugs** delayed them

- How much time did they **spend debugging**?
- To answer this, we examined the survey results

Source: http://leonmeijer.nl/images/leonmeijer_nl/WindowsLiveWriter/TestdrivendevelopmentUni.NETwhatsallthis_D86E/sw_testing.jpg

Estimates about debugging time

- MV subjects: **30.6%**, on the average
- STM subjects: **42.95%**, on the average

Estimates about debugging time

- MV subjects: **30.6%**, on the average
- STM subjects: **42.95%**, on the average
- Difference is statistically significant with 95% confidence:
  - ***P-value = 0.0277***
- Previous work [Panktratius and Tabatabai, 2011] has claimed that STM *eases* debugging
  - Under specific circumstances

Concluding remarks

- First assessment of STM for a functional language
  - From a software engineering perspective
  - **Controlled experiment**
- Data from 51 subjects (including programs and surveys)
  - http://sites.google.com/a/cin.ufpe.br/castor/tmc_2011

Future work:

– Fine-grained locking
– Better organize the survey results ☺

Thank You!

Contact info: Fernando Castor (castor@cin.ufpe.br)

Slides, data, and paper at
http://twitter.com/fernandocastor

PC1 syntax error

PC2 semantic error

PH1 circular dependency

PH2 non-matching putMVar and takeMVar

PH3 infinite recursion

PH4 entire recursion running within a single transaction (livelock)

PN1 race condition with MVar (fine locking or non-locking access)

PN2 race condition due to too fine-grained transactions

PN3 busy wait instead of retry

PN4 uses a global lock even though the solution is based on transactions

PN5 nonsensical results due to unknown causes

PN6 main thread finishes before the program is done

PL1 updates the total number of drops and the number in the bucket inconsistently

PL2 does not correctly limit the maximum number of drops

PL3 does not randomly generate the number of drops

PL4 uses the wrong variable type (e.g., an MVar when should be using a TVar)

PL5 very specific logic errors

PL6 employed a monad-unsafe operation

MV group non-hanging errors

- Mainly problems with not “holding” the main thread
  - 5 occurrences for the MV group, only one for STM
  - **Probable coincidence**

Summary of results (cont.)

- Surprising number of hanging errors for STM
- Many compilation errors for STM (mostly due to monads)
- For **programs without conc. errors**, difference in time is **significant**
  - **STM subjects were faster**
  - Difference in # LoC is considerable **(STM progs. smaller)**

Promises of STM

- Ease the construction of concurrent/parallel applications
  - Simplifies fine-grained access
- Improved performance
  - For situations where there are few collisions

http://www.nydailynews.com/img/2008/04/29/alg_china_train_collision.jpg

Up until now, we believe that...

- Performance can be comparable to or even better than locking [Nakaike et al., 2010]
  - It is easy to get a lot of unwanted collisions, though
- Block-structured transactional code is not always a good thing [Zyulkyarov et al., 2009]
- Dev. time and num. of errors seem to be comparable to coarse-grained locking [Rossbach et al., 2010]
  - and better than fine-grained locking

Existing evaluations

- Some focus on performance
- Some target preexisting systems
  - Refactoring locks to STM
- Two emphasize software engineering aspects
  - **Pankratius and Tabatabai, 2011**: too few subjects, strong conclusions, uses C++
  - **Rossbach et al., 2010**: some biases (not controlled), uses Java