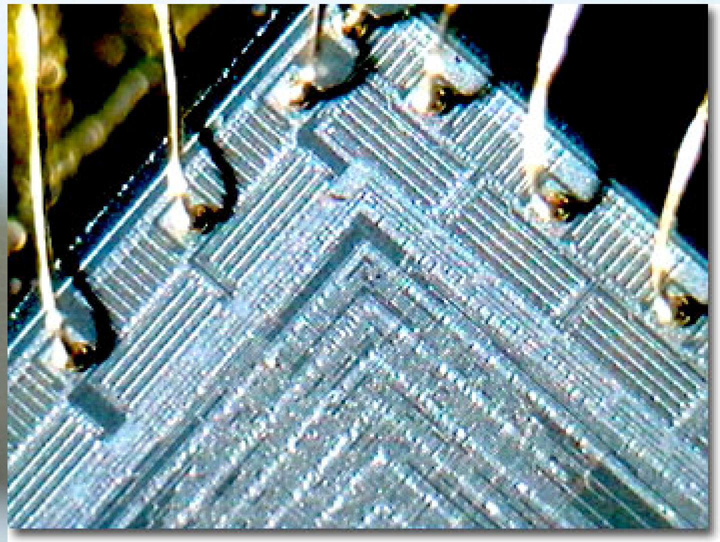
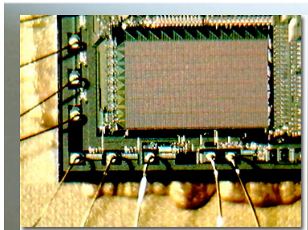


Higher Order Combinators for Join Patterns using STM

Satnam Singh, Microsoft



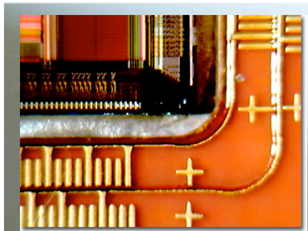
Overview

- Specifically: encoding an existing concurrency idiom with STM
  - very straightforward
  - nothing clever
- More generally: what kind of existing idioms can we sensibly encode with STM?
  - Or should we not bother?

Cω Concurrency

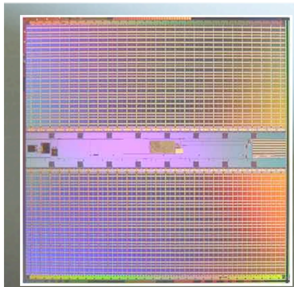
- Objects have both synchronous and *asynchronous* methods
- Values are passed by ordinary method calls:
  - If the method is synchronous, the caller blocks until the method returns some result (as usual)
  - If the method is async, the call completes at once and returns void
- A class defines a collection of *chords* (synchronization patterns), which define what happens once a particular *set* of methods have been invoked. One method may appear in several chords.
  - When pending method calls match a pattern, its body runs.
  - If there is no match, the invocations are queued up.
  - If there are several matches, an unspecified pattern is selected.
  - If a pattern containing *only* async methods fires, the body runs in a new thread.

Cω asynchronous methods



```
using System ;

public class MainProgram
{ public class ArraySummer
  { public async sumArray (int[] intArray)
    { int sum = 0 ;
      foreach (int value in intArray)
        sum += value ;
      Console.WriteLine ("Sum  = " + sum) ;
    }
  }

  static void Main()
  { Summer = new ArraySummer () ;
    Summer.sumArray (new int[] {1, 0, 6, 6, 1, 9, 6, 6}) ;
    Summer.sumArray (new int[] {3, 1, 4, 1, 5, 9, 2, 6}) ;
    Console.WriteLine ("Main method done.") ;
  }
}
```

Cω chords



```
using System ;

public class MainProgram
{ public class Buffer
  { public async Put (int value) ;
    public int Get () & Put(int value)
    { return value ; }
  }

  static void Main()
  { buf = new Buffer () ;
    buf.Put (42) ;
    buf.Put (66) ;
    Console.WriteLine (buf.Get() + " " + buf.Get()) ;
  }
}
```

Reader/Writer Locks

```
public class ReaderWriter {
  private async idle();
  private async s(int n);

  public ReaderWriter() {idle();}

  public void Exclusive() & idle() {}
  public void ReleaseExclusive() { idle(); }

  public void Shared() & idle()    { s(1);}
                       & s(int n) { s(n+1);}

  public void ReleaseShared() & s(int  n) {
    if (n == 1) idle(); else s(n-1);
  }
}
```

“STM”s in Haskell

```
-- Running STM computations
atomically :: STM a -> IO a
retry :: STM a
orElse :: STM a -> STM a -> STM a

-- Transactional variables
data TVar a
newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
newTChan :: STM (TChan a)
writeTChan :: a -> TChan a -> STM ()
readTChan :: TChan a -> STM a
```

Haskell Crash Course

```
add :: Int -> Int -> Int
add a b = a + b


add 2 4 = 6
2 `add` 4 = 6


(&) a b = a + b
2 & 4 = 6
```

Haskell Crash Course

```
inc :: Int -> Int
inc x = x + 1


twice :: (Int -> Int) -> Int -> Int
twice f v = f (f v)


twice (inc) 6
twice (\ x -> x + 1) 6
```

One-Shot Synchronous Join

```
(&) :: TChan a -> TChan b -> STM (a, b)
(&) chan1 chan2
  = do a <- readTChan chan1
       b <- readTChan chan2
       return (a, b)

(>>>) :: STM a -> (a -> IO b) -> IO b
(>>>) joinPattern handler
   = do results <- atomically joinPattern
        handler results

example chan1 chan2
  = chan1 & chan2 >>>
    \ (a, b) -> putStrLn (show (a, b))
```

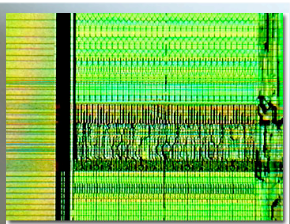
Repeating Asynchronous Join

```
(>!>) :: STM a -> (a -> IO ()) -> IO ()
(>!>) joins cont
   = do forkIO (asyncJoinLoop joins cont)
        return () -- discard thread ID

asyncJoinLoop :: (STM a) -> (a -> IO ()) -> IO ()
asyncJoinLoop joinPattern handler
  = do joinPattern >>> forkIO . handler
       asyncJoinLoop joinPattern handler

example chan1 chan2
  = chan1 & chan2 >!>
    \ (a, b) -> putStrLn (show ((a, b)))
```

Exploiting Overloading

```
class Joinable t1 t2 where
  (&) :: t1 a -> t2 b -> STM (a, b)

instance Joinable TChan TChan where
  (&) = join2

instance Joinable TChan STM where
  (&) = join2b

instance Joinable STM TChan where
  (&) a b = do (x,y) <- join2b b a
               return (y, x)

chan1 & chan2 & chan3 >>>
\ ((a, b), c) -> putStrLn (show (a,b,c))
```

Biased Synchronous Choice

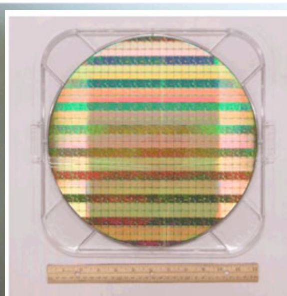


```
(|+|) :: (STM a, a -> IO c) ->
         (STM b, b -> IO c) ->
         IO c
(|+|) (joina, action1) (joinb, action2)
  = do io <- atomically
             (do a <- joina
                 return (action1 a)
                   `orElse`
                   do b <- joinb
                 return (action2 b))
       io

    (chan1 & chan2 & chan3,
        \ ((a,b),c) -> putStrLn (show (a,b,c)))
|+|
    (chan1 & chan2,
        \ (a,b) -> putStrLn (show (a,b)))
```

Conditional Joins

```
(??) :: TChan a -> (a -> Bool) -> STM a
(??) chan predicate
  = do value <- readTChan chan
       if predicate value then
          return value
         else
          retry

(chan1 ?? \x -> x > 3) & chan2 >>>
  \ (a, b) -> putStrLn (show (a, b))
```

Dynamic Joins



```
example numSensors numSensors chan1 chan2 chan3
 = if numSensors = 2 then
      chan1 & chan2 >!> \ (a, b) ->
      putStrLn (show ((a, b)))
   else
      chan1 & chan2 & chan3 >!> \ (a, (b, c))
        -> putStrLn (show ((a, b, c)))
```

Transacted Handlers

```
(>%>) :: STM a -> (a -> STM b) -> IO b
(>%>) joinPattern handler
   = atomically (do results <- joinPattern
                    handler results)
```

Non-Blocking Variants

```
nonBlockingJoin :: STM a ->
                   STM (Maybe a)
nonBlockingJoin pattern
  = (do result <- pattern
         return (Just result))
    `orElse`
    (return Nothing)
```

Summary and Questions

- Straightforward encoding of Cω join patterns using STM.
- Higher order combinators in Haskell act as powerful “glue”.
- Model for understanding join patterns in terms of STMs.
- A good literal implementation (?)
  - Parallel execution?
- Joins as statements instead of declarations.
- Q: What other concurrency idioms can be nicely modeled by STM with retry and orElse?

Puzzle

```
main :: IO ()
main
  = do chan1 <- atomically $ newTChan
       atomically $ writeTChan chan1 42
       atomically $ writeTChan chan1 74
       chan1 & chan1 >>>
         \ (a, b) -> putStrLn (show (a,b))
```

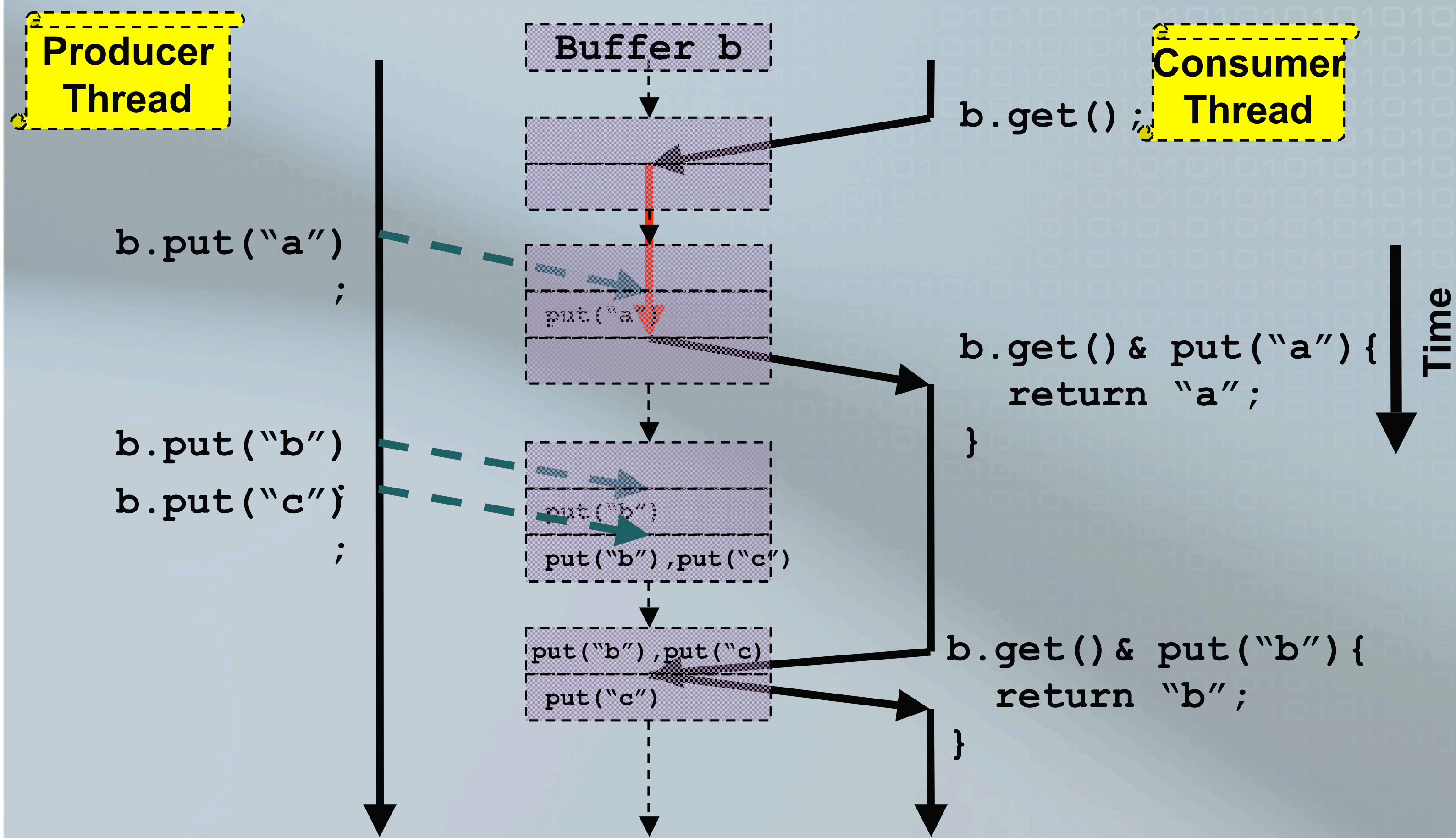
Conditional Joins

```
(?) :: TChan a -> Bool -> STM a
(?) chan predicate
  = if predicate then
      readTChan chan
    else
      retry

(chan1 ? cond) & chan2 >>>
\ (a, b) -> putStrLn (show (a, b))
```

Conditional Joins

```
(?) :: TChan a -> STM Bool -> STM a
(?) chan predicate
  = do cond <- predicate
       if cond then
          readTChan chan
        else
          retry
```

The Buffer Over Time



Backup

Backup