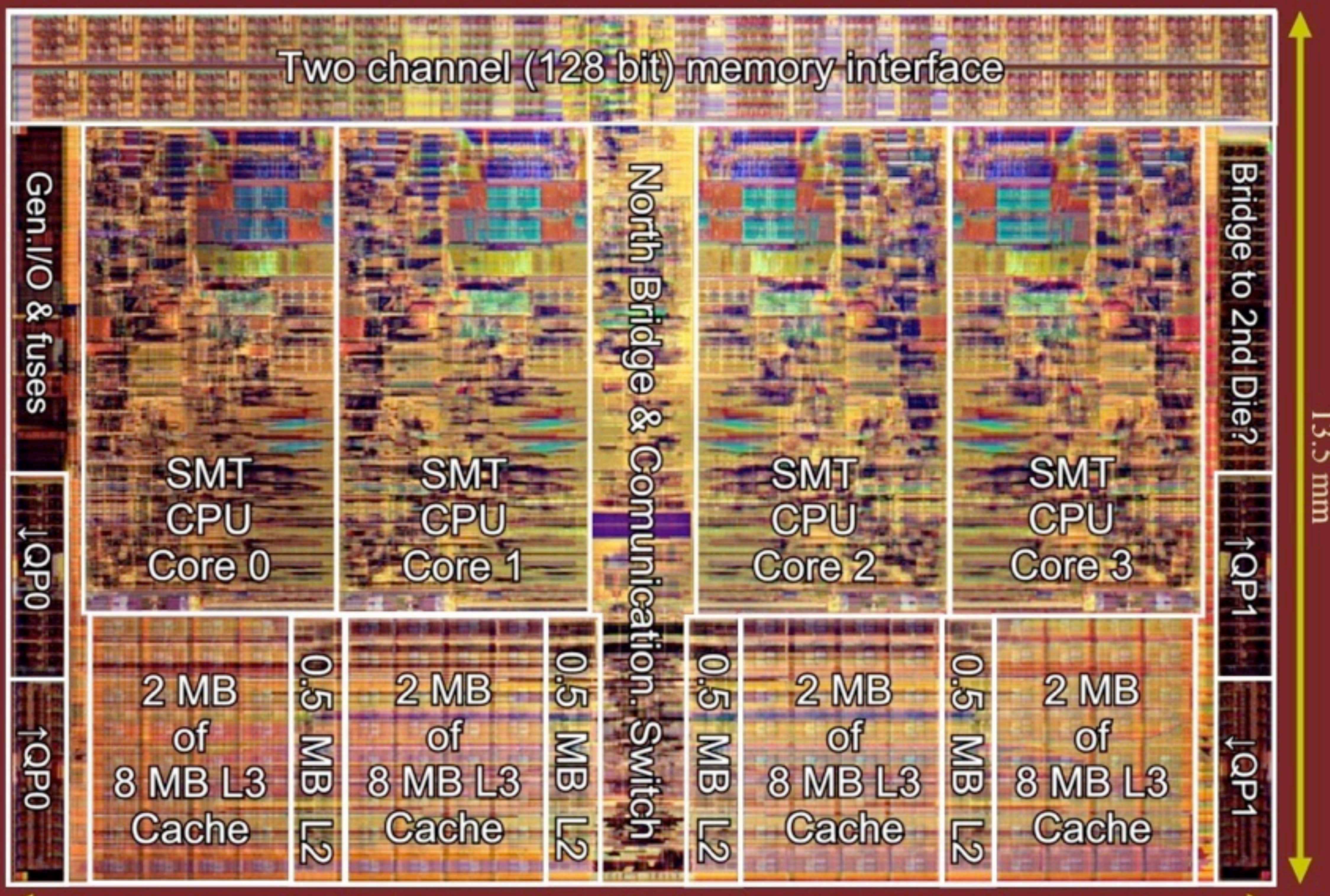
O'REILLY OSCON™ Open Source Convention

A Survey of Concurrency Constructs

Ted Leung
Sun Microsystems
ted.leung@sun.com
@twleung

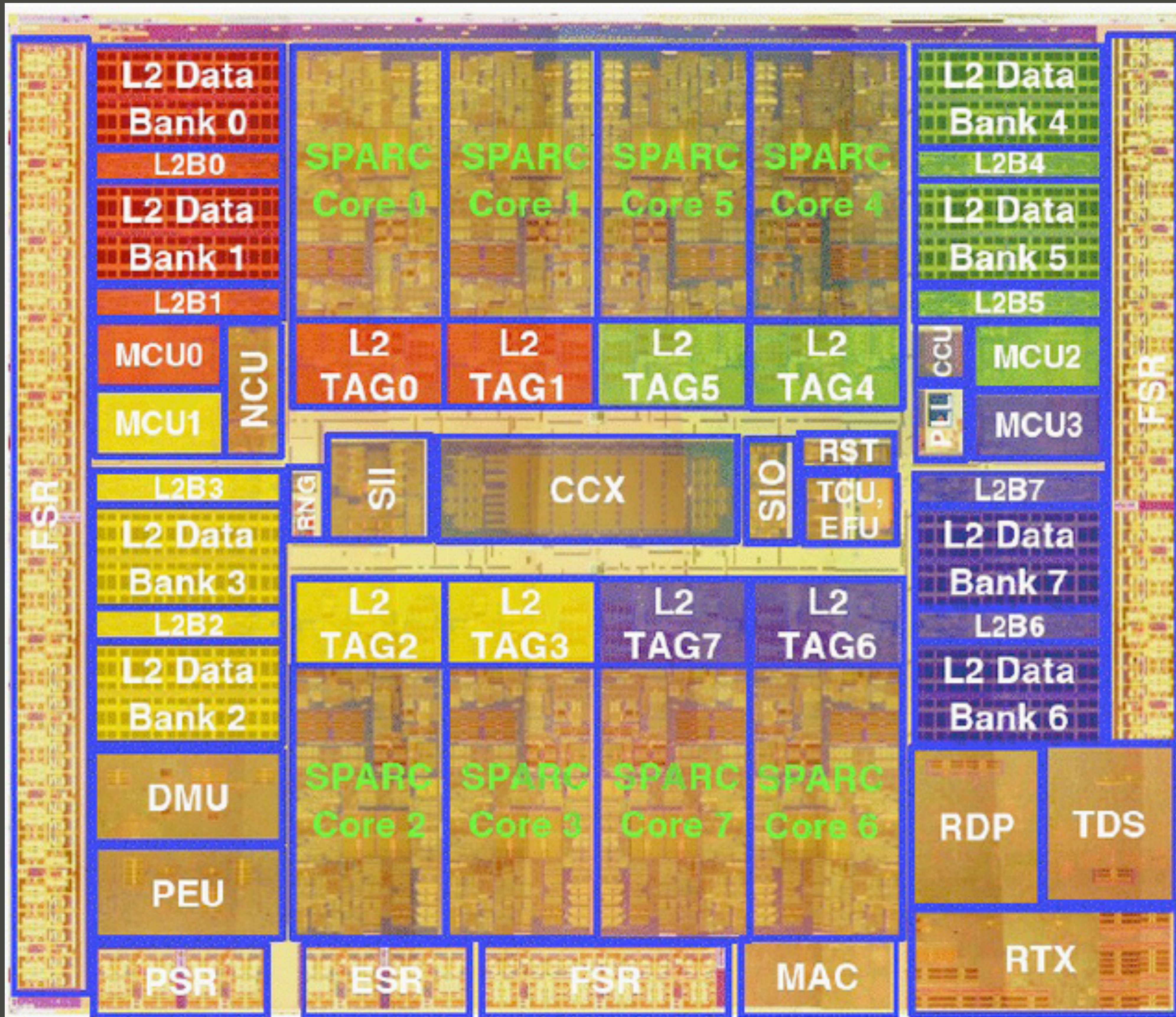
Intel Quad Core Nehalem

Die size 265 mm2



16 threads

O'REILLY OSCON Open Source Convention



128 threads

Today's model

- Threads
  - Program counter
  - Own stack
  - Shared Memory
- Locks

O'REILLY OSCON Open Source Convention

Some of the problems

- Locks
  - manually lock and unlock
  - lock ordering is a big problem
  - locks are not compositional
- How do we decide what is concurrent?
- Need to pre-design, but now we have to retrofit concurrency via new requirements

Design Goals/Space

- Mutual Exclusion
- Serialization / Ordering
- Inherent / Implicit vs Explicit
- Fine / Medium / Coarse grained
- Composability

A good solution

- Is substantially less error prone
- Makes it much easier to identify concurrency
- Runs on today's (and future) parallel hardware
  - Works if you keep adding cores/threads

O'REILLY OSCON Open Source Convention

Theoretical Models

- Actors
- CSP
- CCS
- petri-nets
- pi-calculus
- join-calculus
- Functional Programming

~~Theoretical Models~~

- Actors
- CSP
- CCS
- petri-nets
- pi-calculus
- join-calculus
- Functional Programming

O'REILLY OSCON™ Open Source Convention

Implementation matters

- Threads are not free
- Message sending is not free
- Context/thread switching is not free
- Lock acquire/release is not free

O'REILLY OSCON Open Source Convention

The models

- Transactional Memory
  - Persistent data structures
- Actors
- Dataflow
- Tuple spaces

Transactional Memory

- Original paper on STM 1995
- Idea goes as far back as 1986
  - Tom Knight (Hardware Transactional Memory)
- First appearance in a programming language
  - Concurrent Haskell 2005

The Model

- Use transactions on items in memory
- Enclose code in begin/end blocks
- Variations
  - specify manual abort/retry
  - specify an alternate path (way of controlling manual abort)

Example

```
(defn deposit [account amount]
  (dosync
    (let [owner (account :owner)
          balance-ref (account :balance-ref)]
      (do
        (alter balance-ref + amount)
        (println “depositing” amount (account :owner)))))))
```

STM Design Space

- STM Algorithms / Strategies
  - Granularity
    - word vs block
  - Locks vs Optimistic concurrency
  - Conflict detection
    - eager vs lazy
  - Contention management

STM Problems

- Non transactional access to STM cells
- Non abortable operations
  - I/O
- STM Overhead
  - read/write barrier elimination
- Where to place transaction boundaries?
- Still need condition variables
  - ordering problems are important
    - 1/3 of non-deadlock problems in one study

Implementations

- Haskell/GHC
  - Use logs and aborts txns
- Clojure STM - via Refs
  - based on ML Refs to confine changes, but ML Refs have no automatic (i.e. STM) concurrency semantics
  - only for Refs to aggregates
  - Implementation uses MVCC
  - Persistent data structures enable MVCC allowing decoupling of readers/writers (readers don't wait)

O'REILLY OSCON Open Source Convention

Persistent Data Structures

- Original formulation circa 1981
- Formalization 1986 Sarnoff
- Popularized by Clojure

The model

- Upon “update”, previous versions are still available
  - preserve functionalness
  - both versions meet O(x) characteristics
- In Clojure, combined with STM
  - Motivated by copy on write
  - hash-map, vector, sorted map

Available data structures

- Lists, Vectors, Maps
- hash list based on VLists
- VDList - deques based on VLists
- red-black trees

Available data structures

- Real Time Queues and Deques
- deques, output-restricted deques
- binary random access lists
- binomial heaps
- skew binary random access lists
- skew binomial heaps
- catenable lists
- heaps with efficient merging
- catenable deques

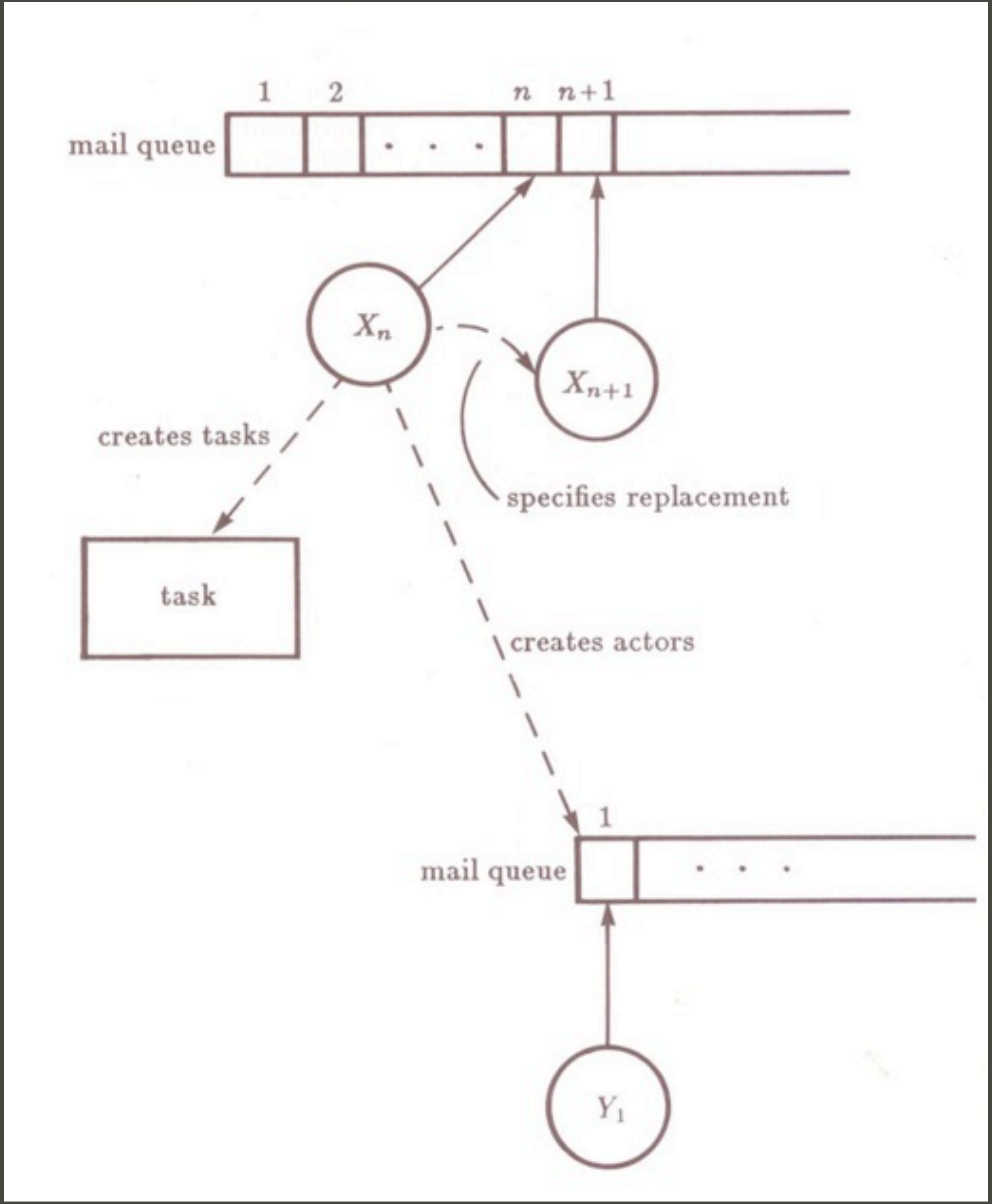
Problems

- Not really a full model
- Oriented towards functional programming

Actors

- Invented by Carl Hewitt at MIT (1973)
  - Formal Model
  - Programming languages
  - Hardware
  - Led to continuations, Scheme
- Recently revived by Erlang
  - Erlang's model is not derived explicitly from Actors

The Model



1
2
n
n+1
mail queue
X_n
X_{n+1}
creates tasks
specifies replacement
task
creates actors
1
mail queue
Y_1

Example

```
object account extends Actor {

  private var balance = 0

  def act() {
    loop {
      react {
        case Withdraw(amount) =>
           balance -= amount
           sender ! Balance(balance)
        case Deposit(amount) =>
           balance += amount
           sender ! Balance(balance)
        case BalanceRequest =>
           sender ! Balance(balance)
        case TerminateRequest =>
      }
  }

}
```

Problems with actors

- DOS of the actor mail queue
- Multiple actor coordination
  - reinvent transactions?
- Actors can still deadlock and starve
- Programmer defines granularity
  - by choosing what is an actor

Actor Implementations

- Scala
  - Scala Actors
  - Lift Actors
- Erlang
- CLR
  - F# / Axum

Java

- kilim
  - http://www.malhar.net/sriram/kilim/
- Actor Foundry
  - http://osl.cs.uiuc.edu/af/
- actorom
  - http://code.google.com/p/actorom/
- Actors Guild
  - http://actorsguildframework.org/

Measuring performance

- actor creation?
- message passing?
- memory usage?

Erlang vs JVM

- Erlang
  - per process GC heap
  - tail call
  - distributed
- JVM
  - per JVM heap
  - no tail call (fixed in JSR-292?)
  - not distributed
  - 2 kinds of actors (Scala)

Actor variants

- Kamaelia
  - messages are sent to named boxes
  - coordination language connects outboxes to inboxes
  - box size is explicitly controllable

Actor variants

- Clojure Agents
  - Designed for loosely coupled stuff
  - Code/actions sent to agents
  - Code is queued when it hits the agent
  - Agent framework guarantees serialization
  - State of agent is always available for read (unlike actors which could be busy processing when you send a read message)
  - not in favor of transparent distribution
  - Clojure agents can operate in an ‘open world’ - actors answer a specific set of messages

O'REILLY OSCON Open Source Convention

Last thoughts on Actors

- Actors are an assembly language
- OTP type stuff and beyond
- Akka - Jonas Boner
  - http://github.com/jboner/akka

O'REILLY OSCON Open Source Convention

Dataflow

- Bill Ackerman's PhD Thesis at MIT (1984)
- Declarative Concurrency in functional languages
- Research in the 1980's and 90's
- Inherent concurrency
  - Turns out to be very difficult to implement
- Interest in declarative concurrency is slowly returning

O'REILLY OSCON Open Source Convention

The model

- Dataflow Variables
  - create variable
  - bind value
  - read value or block
- Threads
- Dataflow Streams
  - List whose tail is an unbound dataflow variable
- Deterministic computation!

Example: Variables 1

```
object Test5 extends Application {
  import DataFlow._

  val x, y, z = new DataFlowVariable[Int]

  val main = thread {
    println("Thread 'main'")
    x << 1
    println("'x' set to: " + x())
    println("Waiting for 'y' to be set...")
    if (x() > y()) {
      z << x
      println("'z' set to 'x': " + z())
    } else {
      z << y
      println("'z' set to 'y': " + z())
    }

    x.shutdown
    y.shutdown
    z.shutdown
    v.shutdown
  }
```

Example: Variables 2

```
object Test5 extends Application {

  val setY = thread {
    println("Thread 'setY', sleeping...")
    Thread.sleep(5000)
    y << 2
    println("'y' set to: " + y())
  }

  // shut down the threads
  main ! 'exit
  setY ! 'exit

  System.exit(0)
}
```

Example: Streams

```
object Test4 extends Application {
  import DataFlow._

  def ints(n: Int, max: Int, stream: DataFlowStream[Int]): Unit = if (n != max) {
    println("Generating int: " + n)
    stream <<< n
    ints(n + 1, max, stream)
  }
  def sum(s: Int, in: DataFlowStream[Int], out: DataFlowStream[Int]): Unit = {
    println("Calculating: " + s)
    out <<< s
    sum(in() + s, in, out)
  }
  def printSum(stream: DataFlowStream[Int]): Unit = {
    println("Result: " + stream())
    printSum(stream)
  }

  val producer = new DataFlowStream[Int]
  val consumer = new DataFlowStream[Int]

  thread { ints(0, 1000, producer) }
  thread { sum(0, producer, consumer) }
  thread { printSum(consumer) }
}
```

Example: Streams (Oz)

```
fun {Ints N Max}
  if N == Max then nil
  else
    {Delay 1000}
    N|{Ints N+1 Max}
  end
end

fun {Sum S Stream}
  case Stream of nil then S
  [] H|T then S|{Sum H+S T} end
end

local X Y in
  thread X = {Ints 0 1000} end
  thread Y = {Sum 0 X} end
  {Browse Y}
end
```

Implementations

- Mozart Oz
  - http://www.mozart-oz.org/
- Jonas Boner's Scala library (now part of Akka)
  - http://github.com/jboner/scala-dataflow
  - dataflow variables and streams
- Ruby library
  - http://github.com/larrytheliquid/dataflow
  - dataflow variables and streams
- Groovy
  - http://code.google.com/p/gparallelizer/

Variations

- Futures
  - Originated in Multilisp
  - Eager/speculative evaluation
  - Implementation quality matters
- I-Structures
  - Id, pH (Parallel Haskell)
  - Single assignment arrays
  - cannot be rebound => no streams

Problems

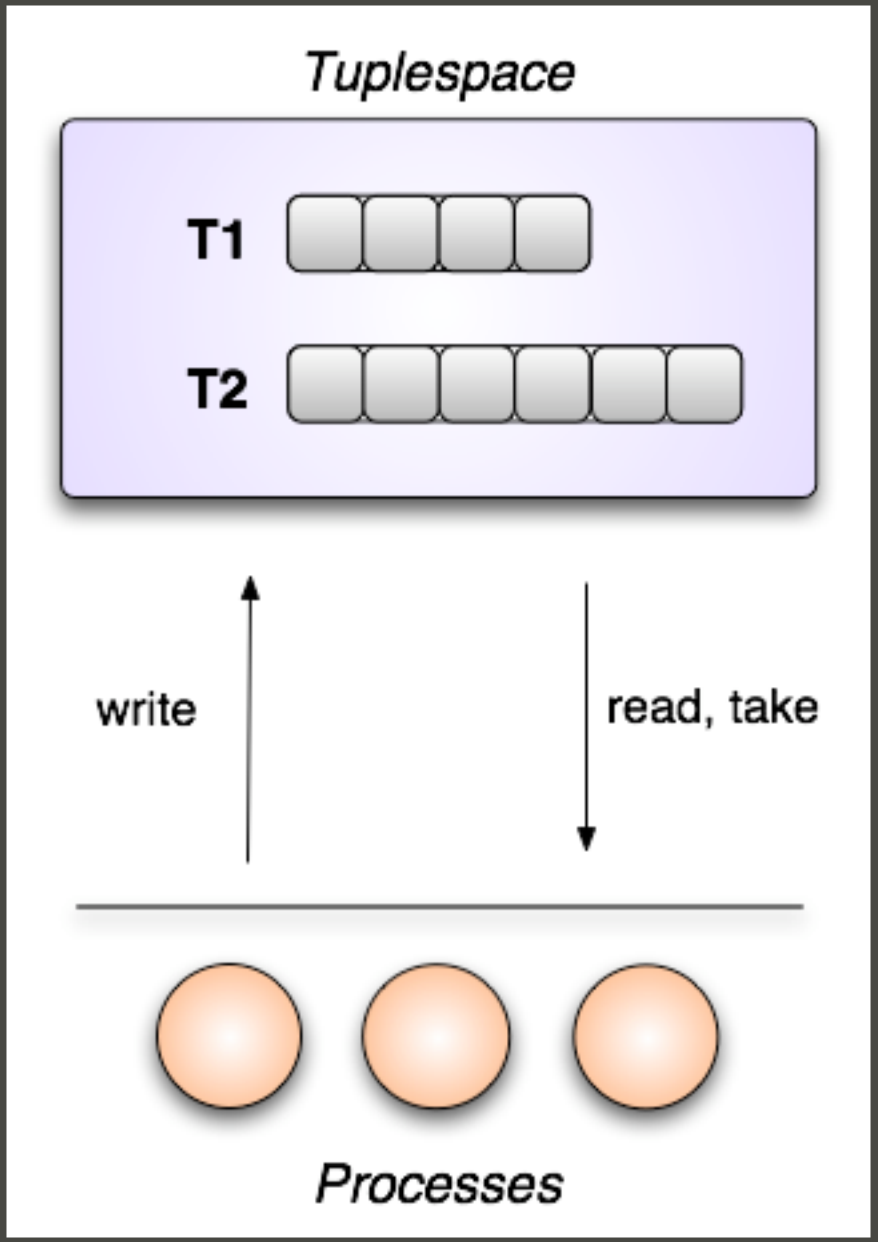
- Can't handle non-determinism
  - like a server
  - Need ports
    - this leads to actor like things

Tuple Spaces

- Originated in Linda (1984)
- Popularized by Jini

The Model

- Three operations
  - write() (out)
  - take()   (in)
  - read()



O'REILLY OSCON Open Source Convention

The Model

- Space uncoupling
- Time uncoupling
- Readers are decoupled from Writers
- Content addressable by pattern matching
- Can emulate
  - Actor like continuations
  - CSP
  - Message Passing
  - Semaphores

O'REILLY OSCON Open Source Convention

Example

```
public class Account implements Entry {
  public Integer accountNo;
  public Integer value;
  public Account() { ... }
  public Account(int accountNo, int value) {
    this.accountNo = newInteger(accountNo);
    this.value = newInteger(value);
  }
}

try {
  Account newAccount = new Account(accountNo, value);
  space.write(newAccount, null, Lease.FOREVER);
}

space.read(accountNo);
```

Implementations

- Jini/JavaSpaces
  - http://incubator.apache.org/river/RIVER/index.html
- BlitzSpaces
  - http://www.dancres.org/blitz/blitz_js.html
- PyLinda
  - http://code.google.com/p/pylinda/
- Rinda
  - built in to Ruby

Problems

- Low level
- High latency to the space - the space is contention point / hot spot
- Scalability
- More for distribution than concurrency

Projects

- Scala
- Erlang
- Clojure
- Kamaelia
- Haskell
- Axum/F#
- Mozart/Oz
- Akka

Work to be done

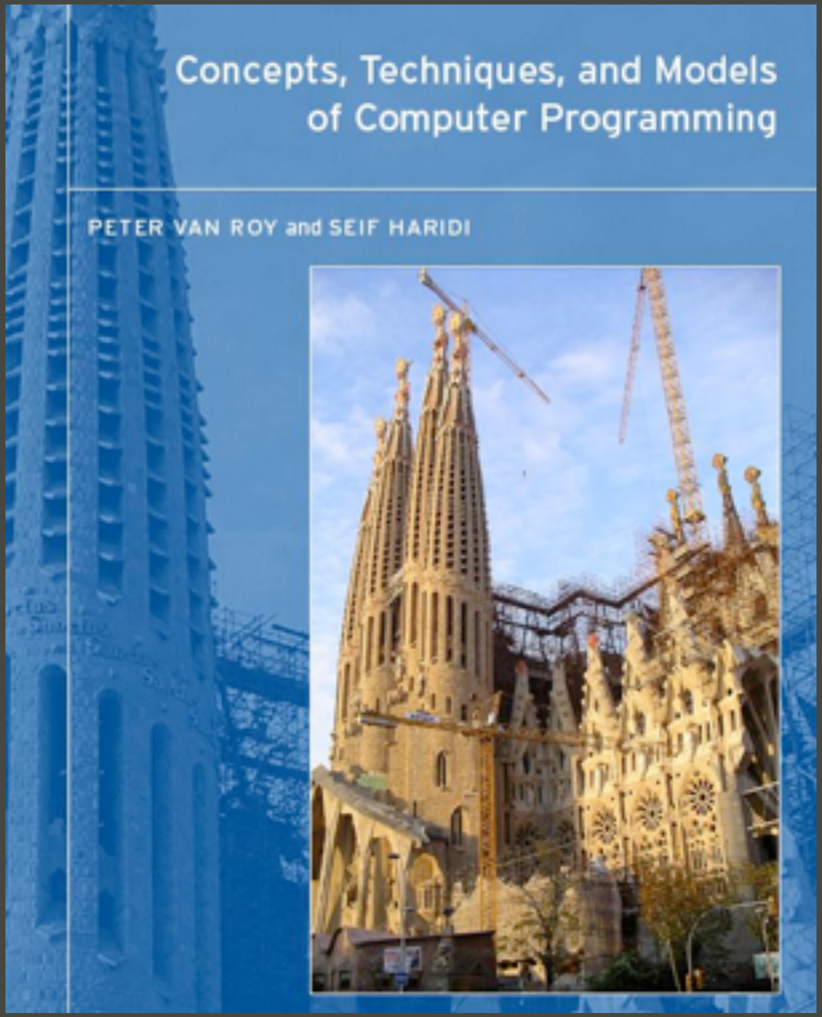
- More in depth comparisons on 4+ core platforms
- Higher level frameworks
- Application architectures/patterns
  - Web
  - Middleware

Final thoughts

- Shared State is troublesome
  - immutability or
  - no sharing
- It's too early

References

- Actors: A Model of Concurrent Computation in Distributed Systems - Gul Agha - MIT Press 1986
- Concepts, Techniques, and Models of Computer Programming - Peter Van Roy and Seif Haridi - MIT Press 2004



Thanks!

- Q&A