Phaser And StampedLock Concurrency Synchronizers

Dr Heinz M. Kabutz

Last updated 2013-09-12

Javaspecialists.eu
java training

Heinz Kabutz

- **Brief Biography**
  - German from Cape Town, now lives in Chania
  - PhD Computer Science from University of Cape Town
  - The Java Specialists' Newsletter
  - Java Champion since 2005
- **Advanced Java Courses**
  - Concurrency Specialist Course
    - Offered in Crete in 2014 or in-house at your company
  - http://www.javaspecialists.eu

Why Synchronizers?

Why Synchronizers?

- **Synchronizers keep shared mutable state consistent**
  - **Don't need if we can make state immutable or unshared**
- **But many applications need large amounts of state**
  - **Immutable would stress the garbage collector**
  - **Unshared would stress the memory volume**
- **Some applications have hash maps of hundreds of GB!**

Coarse Grained Locking

- **Overly coarse-grained locking means the CPUs are starved for work**
  - **Only one core is busy at a time**
- **Took 51 seconds to complete**

Fine Grained Locking

- **"Synchronized" causes *voluntary context switches***
  - **Thread cannot get the lock, so it is parked**
    - **Gives up its allocated time quantum**
- **Took 745 seconds to complete**

- **It appears that system time is 50% of the total time**
  - **So should this not have taken the same elapsed time as before?**

Independent Tasks With No Locking

- **Instead of shared mutable state, every thread uses only local data and in the end we merge the results**
- **Took 28 seconds to complete with 100% utilization**

Nonblocking Algorithms

- **Lock-based algorithms can cause scalability issues**
  - **If a thread is holding a lock and is swapped out, no one can progress**
- **Definitions of types of algorithms**
  - ***Nonblocking*: failure or suspension of one thread, cannot cause another thread to fail or be suspended**
  - ***Lock-free:* at each step, *some* thread can make progress**

Phaser

New synchronizer compatible with Fork/Join

Synchronizers - Structural Properties

- **Encapsulate state that determines whether arriving threads should be allowed to pass or forced to wait**
- **Provide methods to manipulate that state**
- **Provide methods to wait (efficiently) for the synchronizer to enter a desired state**

Interface: CountDownLatch

```
public class CountDownLatch {
  CountDownLatch(int count)

  void await() throws InterruptedException
  boolean await(long timeout, TimeUnit unit)
                        throws InterruptedException


  void countDown()
}
```

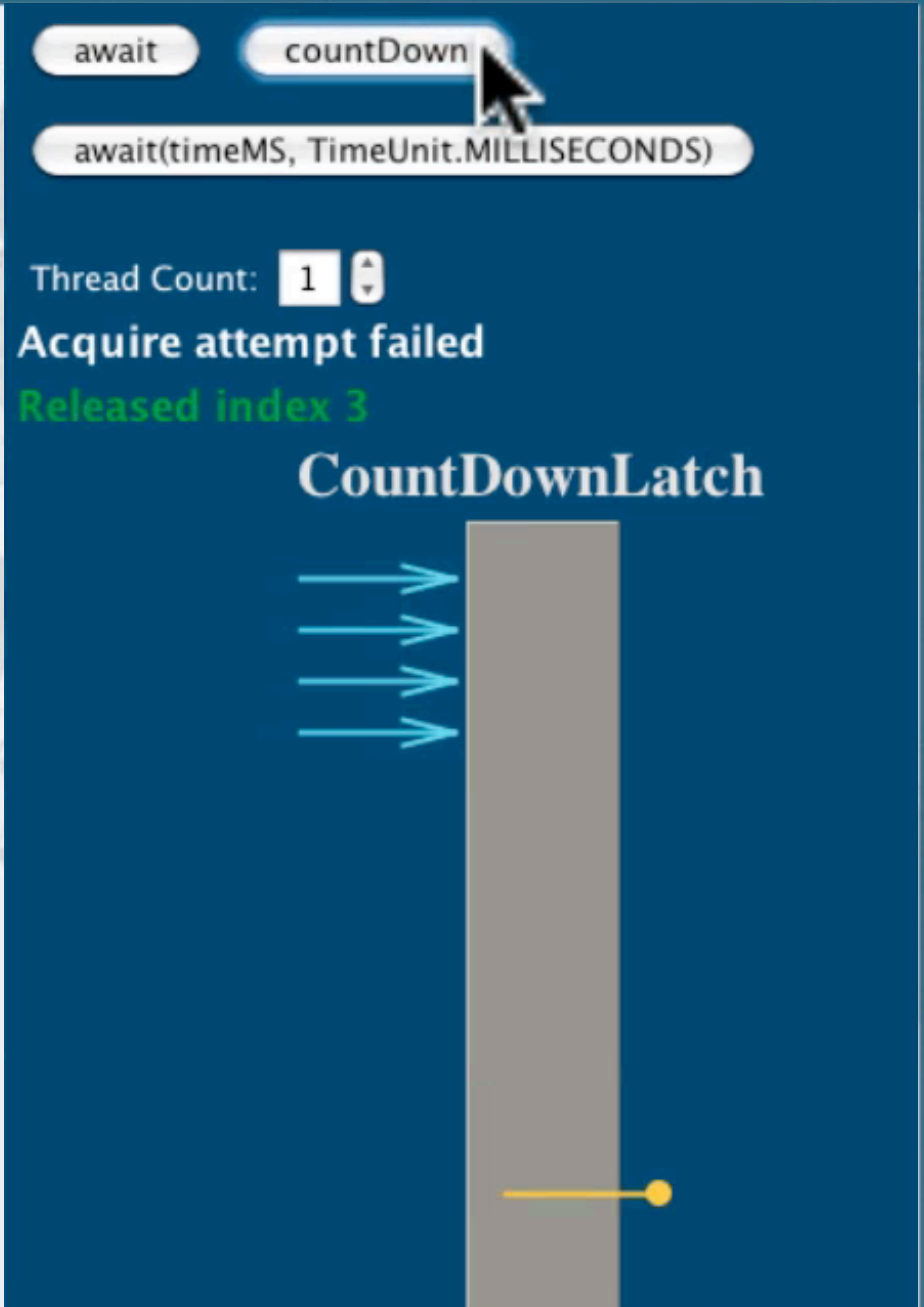
Fixed number of initial permits

A thread can wait for count to reach zero

We can count down, but never up. No reset possible.

CountdownLatch

- **Concurrent Animation Tutorial by Victor Grazi**
  - **www.jconcurrency.com**
- **Threads "await" until the count down latch is zero**
  - **Then they immediately continue**



Thread Count: 1

Acquire attempt failed

Released index 3

CountDownLatch

Code Sample: CountDownLatch

```
Service getService()
    throws InterruptedException {
  serviceCountDown.await();
  return service;
}
```

```
void startDb() {
  db = new Database();
  db.start();
  serviceCountDown.countDown();
}
```

```
void startMailServer() {
  mail = new MailServer();
  mail.start();
  serviceCountDown.countDown();
}
```

Phasers

- Mix of CyclicBarrier and CountDownLatch functionality
  - But with more flexibility
- Registration
  - Number of parties *registered* may vary over time
    - Same as *count* in count down latch and *parties* in cyclic barrier
  - A party can register/deregister itself at any time
  - In contrast, both the other mechanisms have fixed number of parties
- Compatible with Fork/Join framework

Interface: Phaser Registration Methods

```
public class Phaser {
  Phaser(Phaser parent, int parties)

  int register()
  int bulkRegister(int parties)
```

Initial parties - both parameters are optional

Phasers can be arranged in tree to reduce contention

We can change the parties dynamically by calling register()

Phaser Signal And Wait Methods

```
public class Phaser { (continued...)
  int arrive()
  int arriveAndDeregister()


  int awaitAdvance(int phase)
  int awaitAdvanceInterruptibly(int phase,[timeout])
    throws InterruptedException


   int arriveAndAwaitAdvance()
```

Signal only

Wait only - default is to save interrupt

Signal and wait - also saves interrupt

Interface: Phaser Action Method

```
public class Phaser { (continued…)
  protected boolean onAdvance(
    int phase, int registeredParties)

}
```

Override onAdvance() to let phaser finish early

Bunch of lifecycle methods left out

E.g. Coordinated Start Of Threads

- **We want a number of threads to start their work together**
  - **Or as close together as possible, subject to OS scheduling**
- **All threads wait for all others to be ready**
  - **Once-off use**
  - **CountDownLatch or Phaser**

Latch: Waiting For Threads To Start

```
void runTasks(List<Runnable> tasks) throws InterruptedException {
  int size = tasks.size() + 1;
  CountDownLatch latch = new CountDownLatch(size);
  for (Runnable task : tasks) {
    new Thread(() -> {
      try {
        latch.countDown();
        latch.await();
        System.out.println("Running " + task);
        task.run();
      } catch (InterruptedException e) { /* returning */ }
    }).start();
    Thread.sleep(1000);
  }
  latch.countDown();
}
```

Latch: Saving Interruptions

```
public void run() {
  latch.countDown();
  boolean wasInterrupted = false;
  while (true) {
    try {
      latch.await();
      break;
    } catch (InterruptedException e) {
      wasInterrupted = true;
    }
  }
  if (wasInterrupted) Thread.currentThread().interrupt();
  System.out.println("Running: " + task);
  task.run();
}
```

Phaser: Simpler Coding

```
public void runTasks2(List<Runnable> tasks)
    throws InterruptedException {
  Phaser phaser = new Phaser(1); // we register ourselves
  for (Runnable task : tasks) {
    phaser.register(); // and we register all our new threads
    new Thread(() -> {
      phaser.arriveAndAwaitAdvance();
      System.out.println("Running: " + task);
      task.run();
    }).start();
    Thread.sleep(1000);
  }
  phaser.arriveAndDeregister(); // we let the main thread arrive
}
```

phaser.arrive() and phaser.arriveAndAwaitAdvance() also work

Synchronizers Summary

- **CountDownLatch**
  - Makes threads wait until the latch has been counted down to zero
- **CyclicBarrier**
  - A barrier that is reset once it reaches zero
- **Phaser**
  - A flexible synchronizer in Java 7 to do latch and barrier semantics
    - With less code and better interrupt management
    - Is compatible with Fork/Join

StampedLock

Motivation For StampedLock

- **Some constructs need a form of read/write lock**
- **ReentrantReadWriteLock can cause starvation**
  - **Plus it always uses pessimistic locking**
- **StampedLock provides optimistic locking on reads**
  - **Which can be converted easily to a pessimistic lock**
- **Write locks are always pessimistic**
  - **Also called *exclusive* locks**

Read-Write Locks Refresher

- **ReadWriteLock interface**
  - **The writeLock() is *exclusive* - only one thread at a time**
  - **The readLock() is given to lots of threads at the same time**
    - **Much better when mostly reads are happening**
  - **Both locks are pessimistic**

Account With ReentrantReadWriteLock

```
public class BankAccountWithReadWriteLock {
  private final ReadWriteLock lock = new ReentrantReadWriteLock();
  private double balance;
  public void deposit(double amount) {
    lock.writeLock().lock();
    try {
      balance = balance + amount;
    } finally { lock.writeLock().unlock(); }
  }
  public double getBalance() {
    lock.readLock().lock();
    try {
      return balance;
    } finally { lock.readLock().unlock(); }
  }
}
```

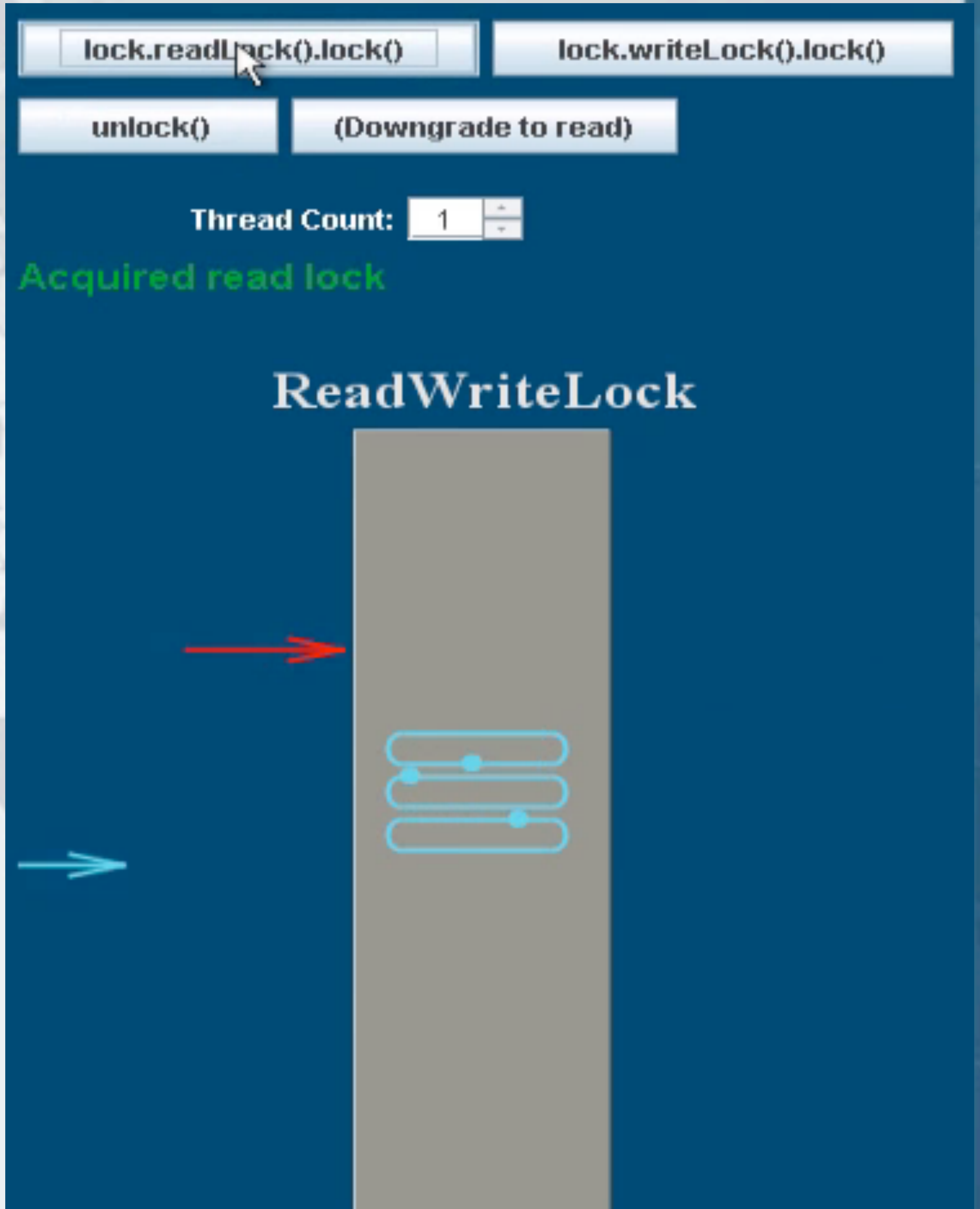
The cost overhead of the RWLock means we need at least 2000 instructions to benefit from the readLock() added throughput

ReentrantReadWriteLock Starvation

- **When readers are given priority, then writers might never be able to complete (Java 5)**
- **But when writers are given priority, readers might be starved (Java 6)**
- **See http://www.javaspecialists.eu/archive/Issue165.html**

Java 5 ReadWriteLock Starvation

- We first acquire some read locks
- We then acquire one write lock
- Despite write lock waiting, read locks are still issued
- If enough read locks are issued, write lock will never get a chance and the thread will be starved!



Thread Count: 1

Acquired read lock

ReadWriteLock In Java 6

- Java 6 changed the policy and now read locks have to wait until the write lock has been issued
- However, now the readers can be starved if we have a lot of writers

Thread Count: 1

Waiting to acquire READ lock

Synchronized vs ReentrantLock

- **ReentrantReadWriteLock, ReentrantLock and synchronized locks have the same memory semantics**
- **However, synchronized is easier to write correctly**

```
synchronized(this)
  // do operation
}
```

```
rwlock.writeLock().lock();
try {
  // do operation
} finally {
  rwlock.writeLock().unlock();
}
```

Bad Try-Finally Blocks

- **Either no try-finally at all**

```
rwlock.writeLock().lock();
// do operation
rwlock.writeLock().unlock();
```

Bad Try-Finally Blocks

- **Or the lock is locked inside the try block**

```
try {
  rwlock.writeLock().lock();
  // do operation
} finally {
  rwlock.writeLock().unlock();
}
```

Bad Try-Finally Blocks

- **Or the unlock() call is forgotten in some places altogether!**

```
rwlock.writeLock().lock();
// do operation
// no unlock()
```

Introducing StampedLock

- **Pros**
  - **Has *much* better performance than ReentrantReadWriteLock**
  - **Latest versions do not suffer from starvation of writers**
- **Cons**
  - **Idioms are more difficult to get right than with ReadWriteLock**
  - **A small difference can make a big difference in performance**
  - **Not nonblocking**

Pessimistic Exclusive Locks (write)

```
public class StampedLock {
  long writeLock()
  long writeLockInterruptibly() throws InterruptedException

  long tryWriteLock()
  long tryWriteLock(long time, TimeUnit unit)
    throws InterruptedException

  void unlockWrite(long stamp)
  boolean tryUnlockWrite()

  Lock asWriteLock()
  long tryConvertToWriteLock(long stamp)
```

Pessimistic Non-Exclusive (read)

```
public class StampedLock {  (continued …)
  long readLock()
  long readLockInterruptibly() throws InterruptedException

  long tryReadLock()

  long tryReadLock(long time, TimeUnit unit)
    throws InterruptedException

  void unlockRead(long stamp)
  boolean tryUnlockRead()

  Lock asReadLock()
  long tryConvertToReadLock(long stamp)
```

Optimistic reads to come ...

Bank Account With StampedLock

```
public class BankAccountWithStampedLock {
  private final StampedLock lock = new StampedLock();
  private double balance;
  public void deposit(double amount) {
    long stamp = lock.writeLock();
    try {
      balance = balance + amount;
    } finally { lock.unlockWrite(stamp); }
  }
  public double getBalance() {
    long stamp = lock.readLock();
    try {
      return balance;
    } finally { lock.unlockRead(stamp); }
  }
}
```

The StampedLock reading is a typically cheaper than ReentrantReadWriteLock

Why Not Use Volatile?

```
public class BankAccountWithVolatile {
  private volatile double balance;

  public synchronized void deposit(double amount) {
    balance = balance + amount;
  }

  public double getBalance() {
    return balance;
  }
}
```

Much easier!
Works because there
are no invariants
across the fields.

Example With Invariants Across Fields

- **Our Point class has x,y coordinates, "belong together"**

```
public class MyPoint {
  private double x, y;
  private final StampedLock sl = new StampedLock();

  // method is modifying x and y, needs exclusive lock
  public void move(double deltaX, double deltaY) {
    long stamp = sl.writeLock();
    try {
      x += deltaX;
      y += deltaY;
    } finally { sl.unlockWrite(stamp); }
  }
```

Optimistic Non-Exclusive "Locks"

```
public class StampedLock {
  long tryOptimisticRead()
```

Try to get an optimistic read lock - might return zero if an exclusive lock is active

```
  boolean validate(long stamp)
```

checks whether a write lock was issued after the tryOptimisticRead() was called

Note: sequence validation requires stricter ordering than apply to normal volatile reads - a new explicit loadFence() was added

```
  long tryConvertToOptimisticRead(long stamp)
```

Code Idiom For Optimistic Read

```
public double optimisticRead() {
  long stamp = sl.tryOptimisticRead();
  double currentState1 = state1,
         currentState2 = state2, ... etc.;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentState1 = state1;
      currentState2 = state2, ... etc.;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return calculateSomething(state1, state2);
}
```

Code Idiom For Optimistic Read

```
public double optimisticRead() {
  long stamp = sl.tryOptimisticRead();
  double currentState1 = state1,
         currentState2 = state2, ... etc.;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentState1 = state1;
      currentState2 = state2, ... etc.;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return calculateSomething(state1, state2);
}
```

We get a stamp to use for the optimistic read

Code Idiom For Optimistic Read

```
public double optimisticRead() {
  long stamp = sl.tryOptimisticRead();
  double currentState1 = state1,
         currentState2 = state2, ... etc.;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentState1 = state1;
      currentState2 = state2, ... etc.;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return calculateSomething(state1, state2);
}
```

We read field values into local fields

Code Idiom For Optimistic Read

```
public double optimisticRead() {
  long stamp = sl.tryOptimisticRead();
  double currentState1 = state1,
         currentState2 = state2, ... etc.;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentState1 = state1;
      currentState2 = state2, ... etc.;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return calculateSomething(state1, state2);
}
```

Next we validate that no write locks have been issued in the meanwhile

Code Idiom For Optimistic Read

```
public double optimisticRead() {
  long stamp = sl.tryOptimisticRead();
  double currentState1 = state1,
         currentState2 = state2, ... etc.;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentState1 = state1;
      currentState2 = state2, ... etc.;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return calculateSomething(state1, state2);
}
```

If they have, then we don't know if our state is clean

Thus we acquire a pessimistic read lock and read the state into local fields

Code Idiom For Optimistic Read

```
public double optimisticRead() {
  long stamp = sl.tryOptimisticRead();
  double currentState1 = state1,
         currentState2 = state2, ... etc.;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentState1 = state1;
      currentState2 = state2, ... etc.;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return calculateSomething(state1, state2);
}
```

Optimistic Read In Point Class

```
public double distanceFromOrigin() {
  long stamp = sl.tryOptimisticRead();
  double currentX = x, currentY = y;
  if (!sl.validate(stamp)) {
    stamp = sl.readLock();
    try {
      currentX = x;
      currentY = y;
    } finally {
      sl.unlockRead(stamp);
    }
  }
  return Math.hypot(currentX, currentY);
}
```

Shorter code path in optimistic read leads to better read performance than with original examples in JavaDoc

Code Idiom For Conditional Change

```
public void changeStateIfEquals(oldState1, oldState2, ...
                                newState1, newState2, ...) {
  long stamp = sl.readLock();
  try {
    while (state1 == oldState1 && state2 == oldState2 ...) {
      long writeStamp = sl.tryConvertToWriteLock(stamp);
      if (writeStamp != 0L) {
        stamp = writeStamp;
        state1 = newState1; state2 = newState2; ...
        break;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
  } finally { sl.unlock(stamp); }
}
```

Code Idiom For Conditional Change

```
public void changeStateIfEquals(oldState1, oldState2, ...
                                newState1, newState2, ...) {
  long stamp = sl.readLock();
  try {
    while (state1 == oldState1 && state2 == oldState2 ...) {
      long writeStamp = sl.tryConvertToWriteLock(stamp);
      if (writeStamp != 0L) {
        stamp = writeStamp;
        state1 = newState1; state2 = newState2; ...
        break;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
  } finally { sl.unlock(stamp); }
}
```

We get a pessimistic read lock

Code Idiom For Conditional Change

```
public void changeStateIfEquals(oldState1, oldState2, ...
                           newState1, newState2, ...) {
  long stamp = sl.readLock();
  try {
    while (state1 == oldState1 && state2 == oldState2 ...) {
      long writeStamp = sl.tryConvertToWriteLock(stamp);
      if (writeStamp != 0L) {
        stamp = writeStamp;
        state1 = newState1; state2 = newState
        break;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
  } finally { sl.unlock(stamp); }
}
```

If the state is not the expected state, we unlock and exit method

Note: the general unlock() method can unlock both read and write locks

Code Idiom For Conditional Change

```
public void changeStateIfEquals(oldState1, oldState2, ...
                           newState1, newStat
  long stamp = sl.readLock();
  try {
    while (state1 == oldState1 && state2 == oldState2 ...) {
      long writeStamp = sl.tryConvertToWriteLock(stamp);
      if (writeStamp != 0L) {
        stamp = writeStamp;
        state1 = newState1; state2 = newState2; ...
        break;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
  } finally { sl.unlock(stamp); }
}
```

We try convert our read lock to a write lock

Code Idiom For Conditional Change

```
public void changeStateIfEquals(oldState1, oldState2, ...
                                newState1, newState2, ...) {
  long stamp = sl.readLock();
  try {
    while (state1 == oldState1 && state2 == oldState2 ...) {
      long writeStamp = sl.tryConvertToWriteLock(stamp);
      if (writeStamp != 0L) {
        stamp = writeStamp;
        state1 = newState1; state2 = newState2; ...
        break;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
  } finally { sl.unlock(stamp); }
}
```

If we are able to upgrade to a write lock (ws != 0L), we change the state and exit

Code Idiom For Conditional Change

```
public void changeStateIfEquals(oldState1, oldState2, ...
                           newState1, newState2, ...) {
  long stamp = sl.readLock();
  try {
    while (state1 == oldState1 && state2 == oldState2 ...) {
      long writeStamp = sl.tryConvertToWriteLock(stamp);
      if (writeStamp != 0L) {
        stamp = writeStamp;
        state1 = newState1; state2 = newState2; ...
        break;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
  } finally { sl.unlock(stamp); }
}
```

Else, we explicitly unlock the read lock and lock the write lock

And we try again

Code Idiom For Conditional Change

```
public void changeStateIfEquals(oldState1, oldState2, ...
                           newState1, newState2, ...) {
  long stamp = sl.readLock();
  try {
    while (state1 == oldState1 && state2 == oldState2 ...) {
      long writeStamp = sl.tryConvertToWriteLock(stamp);
      if (writeStamp != 0L) {
        stamp = writeStamp;
        state1 = newState1; state2 = newStat
        break;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock(
      }
    }
  } finally { sl.unlock(stamp
}
```

If the state is not the expected state, we unlock and exit method

This could happen if between the unlockRead() and the writeLock() another thread changed the values

Code Idiom For Conditional Change

```
public void changeStateIfEquals(oldS
                                newState1
  long stamp = sl.readLock();
  try {
    while (state1 == oldState1 && state2 == oldState2 ...) {
      long writeStamp = sl.tryConvertToWriteLock(stamp);
      if (writeStamp != 0L) {
        stamp = writeStamp;
        state1 = newState1; state2 = newState2; ...
        break;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
  } finally { sl.unlock(stamp); }
}
```

Because we hold the write lock, the tryConvertToWriteLock() method **will** succeed

We update the state and exit

Code Idiom For Conditional Change

```
public void changeStateIfEquals(oldState1, oldState2, ...
                           newState1, newState2, ...) {
  long stamp = sl.readLock();
  try {
    while (state1 == oldState1 && state2 == oldState2 ...) {
      long writeStamp = sl.tryConvertToWriteLock(stamp);
      if (writeStamp != 0L) {
        stamp = writeStamp;
        state1 = newState1; state2 = newState2; ...
        break;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
  } finally { sl.unlock(stamp); }
}
```

Applying To Our Point Class

```
public void moveIfAt(double oldX, double oldY,
                     double newX, double newY) {
  long stamp = sl.readLock();
  try {
    while (x == oldX && y == oldY) {
      long writeStamp = sl.tryConvertToWriteLock(stamp);
      if (writeStamp != 0L) {
        stamp = writeStamp;
        x = newX; y = newY;
        break;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
  } finally { sl.unlock(stamp); }
}
```

Performance StampedLock & RWLock

- **We researched ReentrantReadWriteLock in 2008**
  - Discovered serious starvation of *writers* (exclusive lock) in Java 5
  - And also some starvation of *readers* in Java 6
  - http://www.javaspecialists.eu/archive/Issue165.html
- **StampedLock released to concurrency-interest list Oct 12**
  - Worse *writer* starvation than in the ReentrantReadWriteLock
  - Missed signals could cause StampedLock to deadlock
- **Revision 1.35 released 28th Jan 2013**
  - Changed to use an explicit call to loadFence()
  - Writers do not get starved anymore
  - Works correctly

Performance StampedLock & RWLock

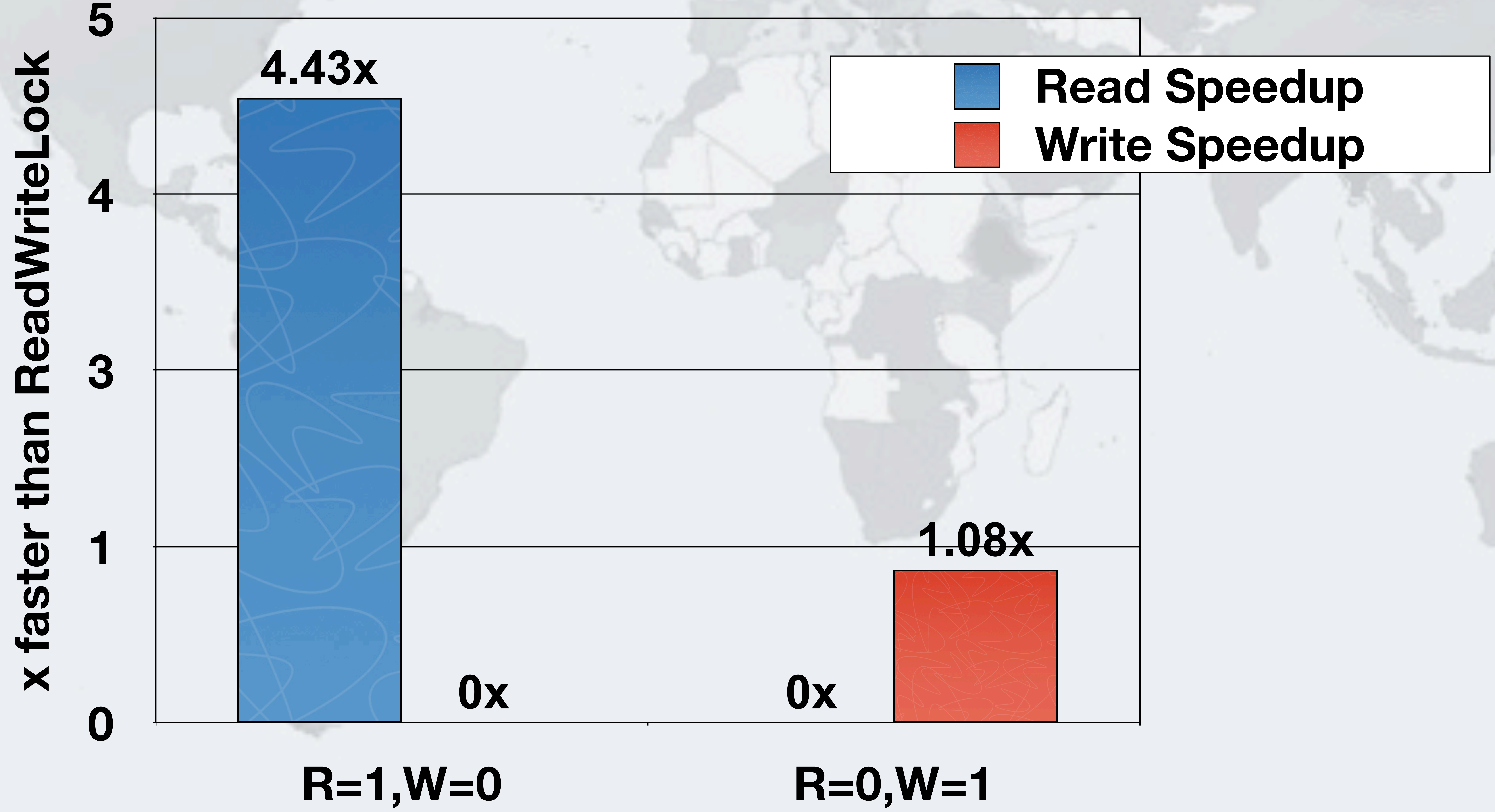
- In our test, we used
  - lambda-8-b75-linux-x64-28_jan_2013.tar.gz
  - Two CPUs, 4 Cores each, no hyperthreading
    - 2x4x1
  - Ubuntu 9.10
  - 64-bit
  - Intel(R) Core(TM) i7 CPU 920 @ 2.67GHz
    - L1-Cache: 256KiB, internal write-through instruction
    - L2-Cache: 1MiB, internal write-through unified
    - L3-Cache: 8MiB, internal write-back unified
  - JavaSpecialists.eu server
    - Never breaks a sweat delivering newsletters

Conversions To Pessimistic Reads

- **In our experiment, reads had to be converted to pessimistic reads less than 10% of the time**
  - **And in most cases, less than 1%**
- **This means the optimistic read worked most of the time**

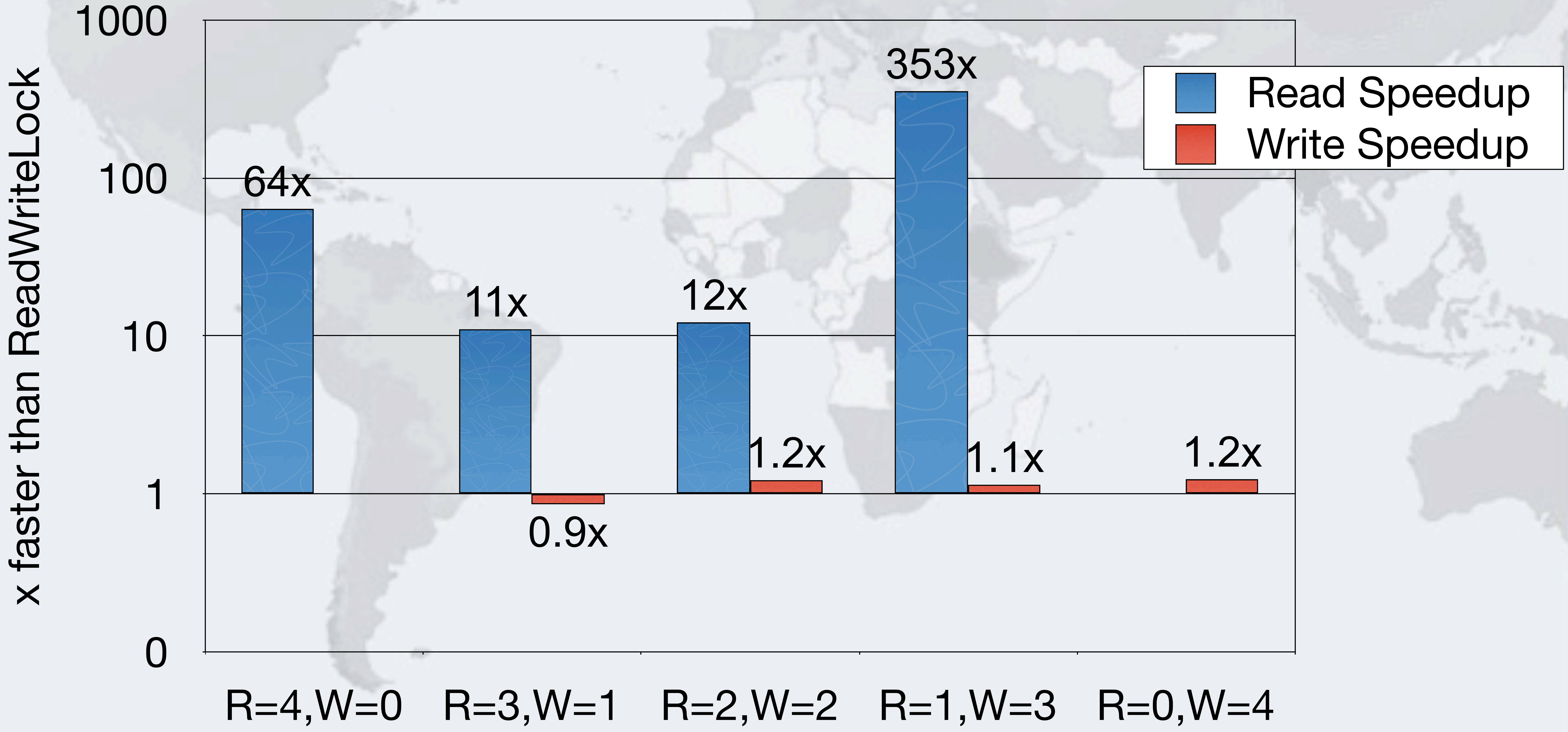
How Much Faster Is StampedLock Than ReentrantReadWriteLock?

- **With a single thread**



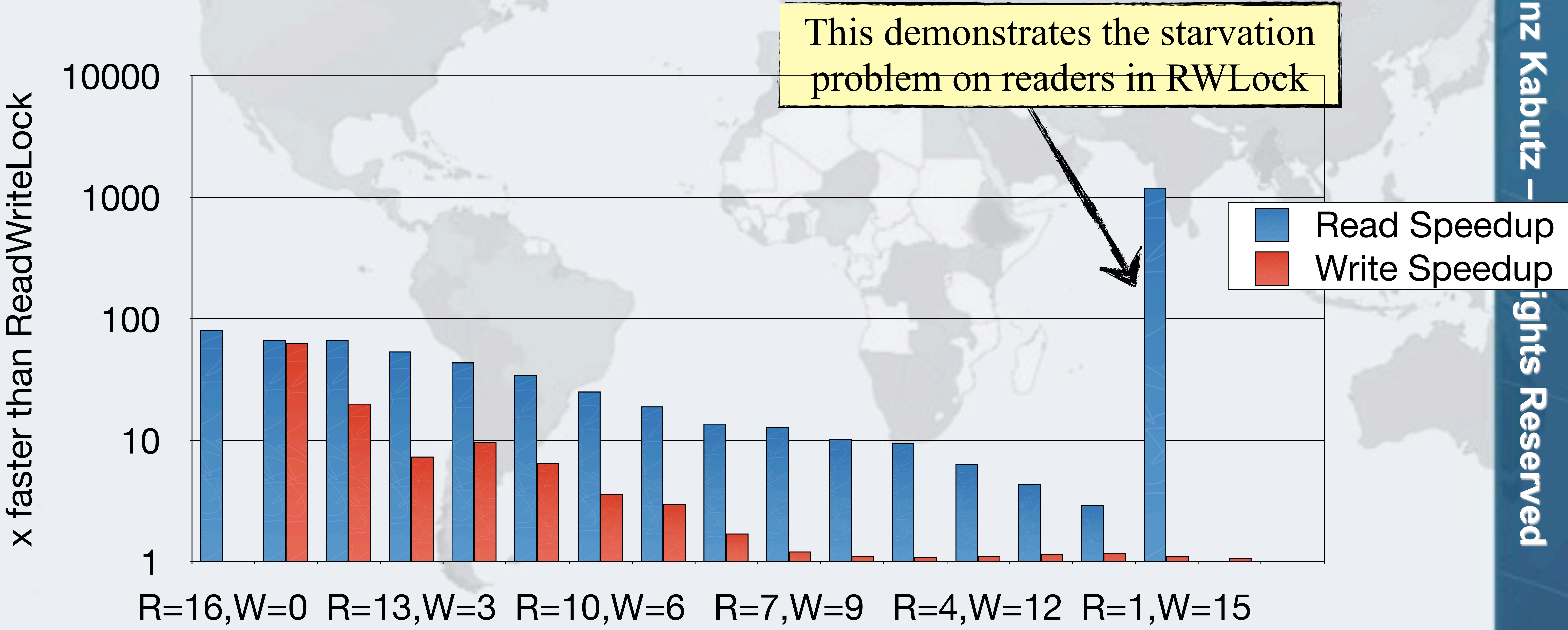
How Much Faster Is StampedLock Than ReentrantReadWriteLock?

- **With four threads**



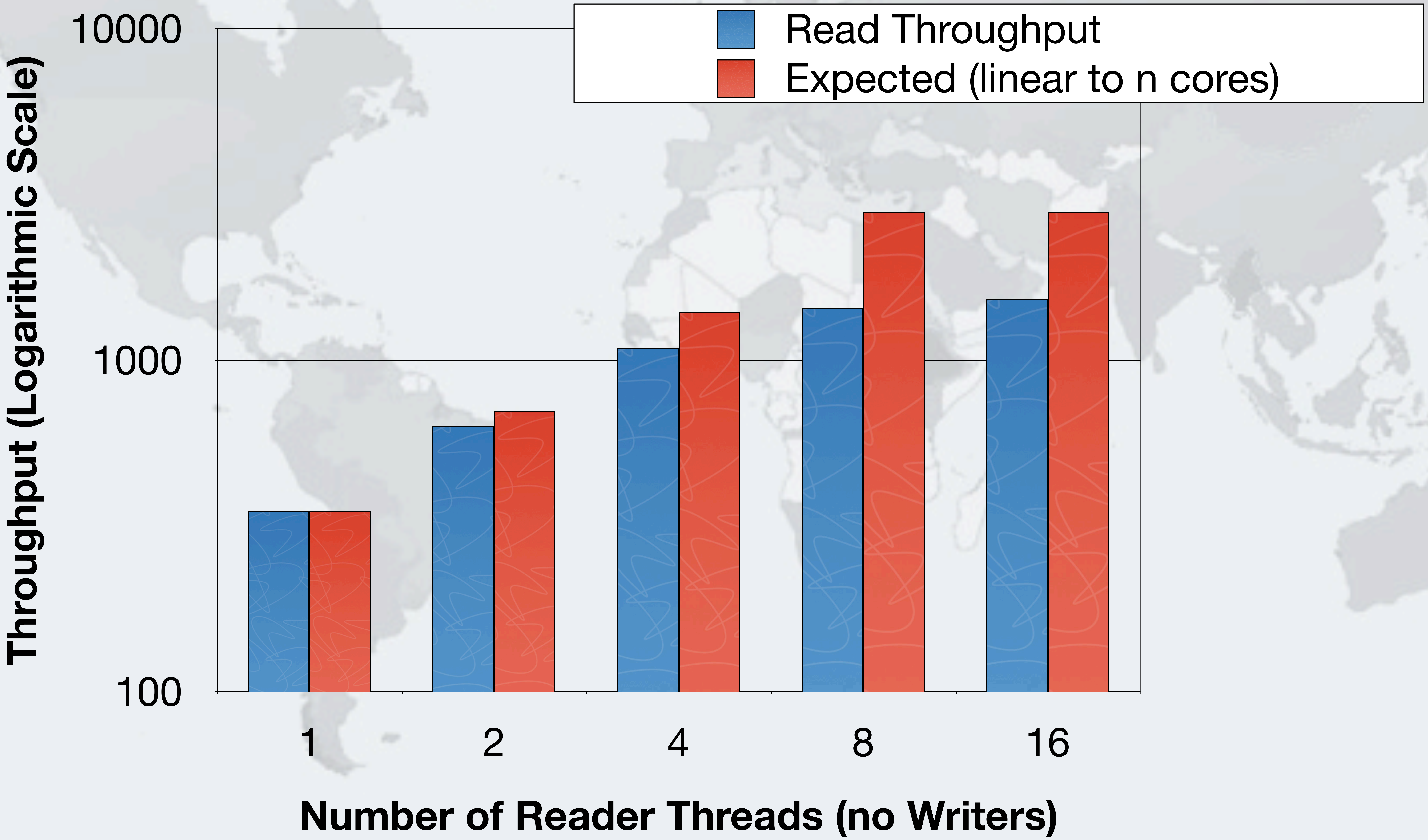
How Much Faster Is StampedLock Than ReentrantReadWriteLock?

- **With sixteen threads**

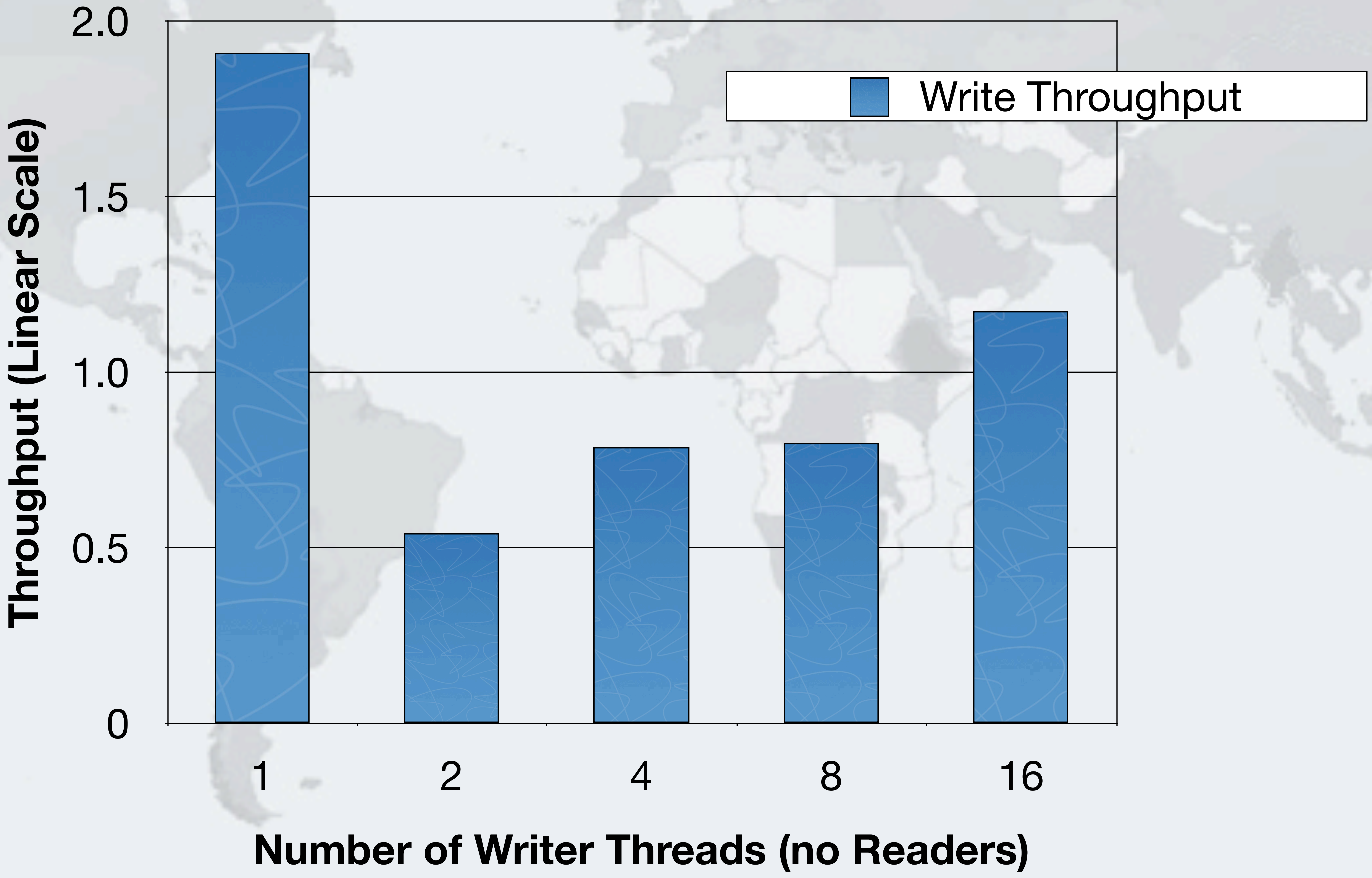


This demonstrates the starvation problem on readers in RWLock

Reader Throughput With StampedLock

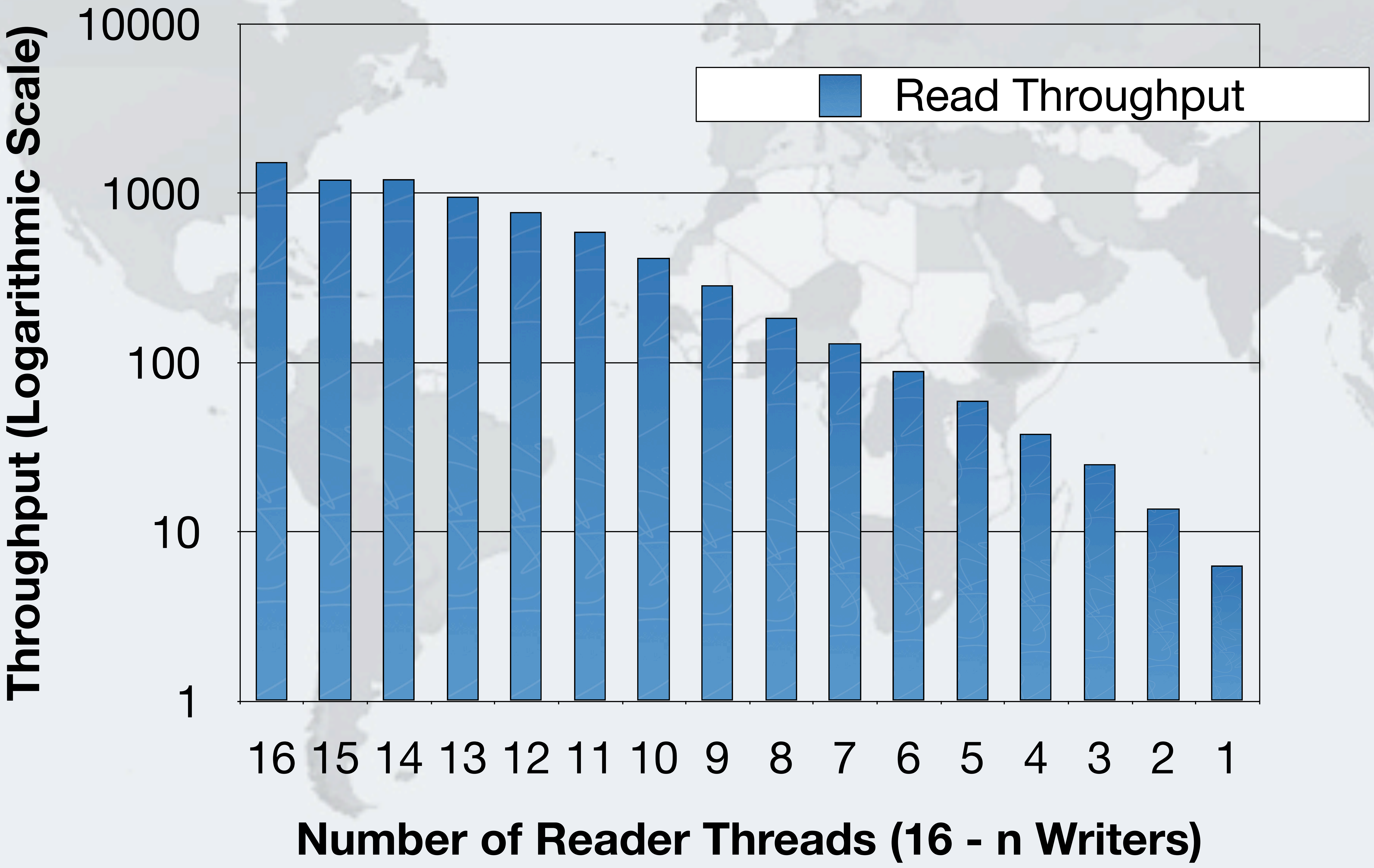


Writer Throughput With StampedLock

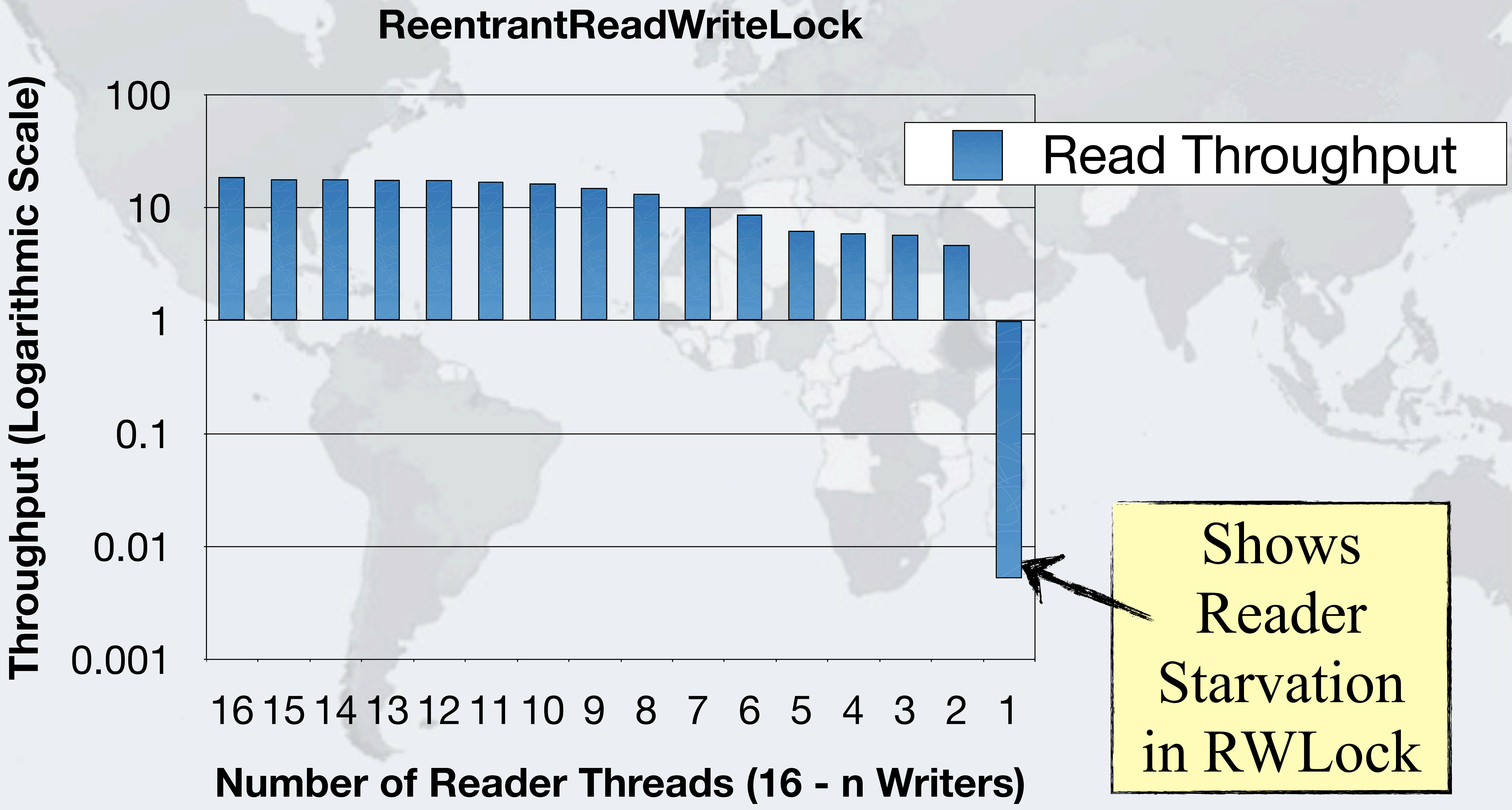


Note: Linear Scale throughput

Mixed Reader Throughput StampedLock



Mixed Reader Throughput RWLock



Shows Reader Starvation in RWLock

Conclusion Of Performance Analysis

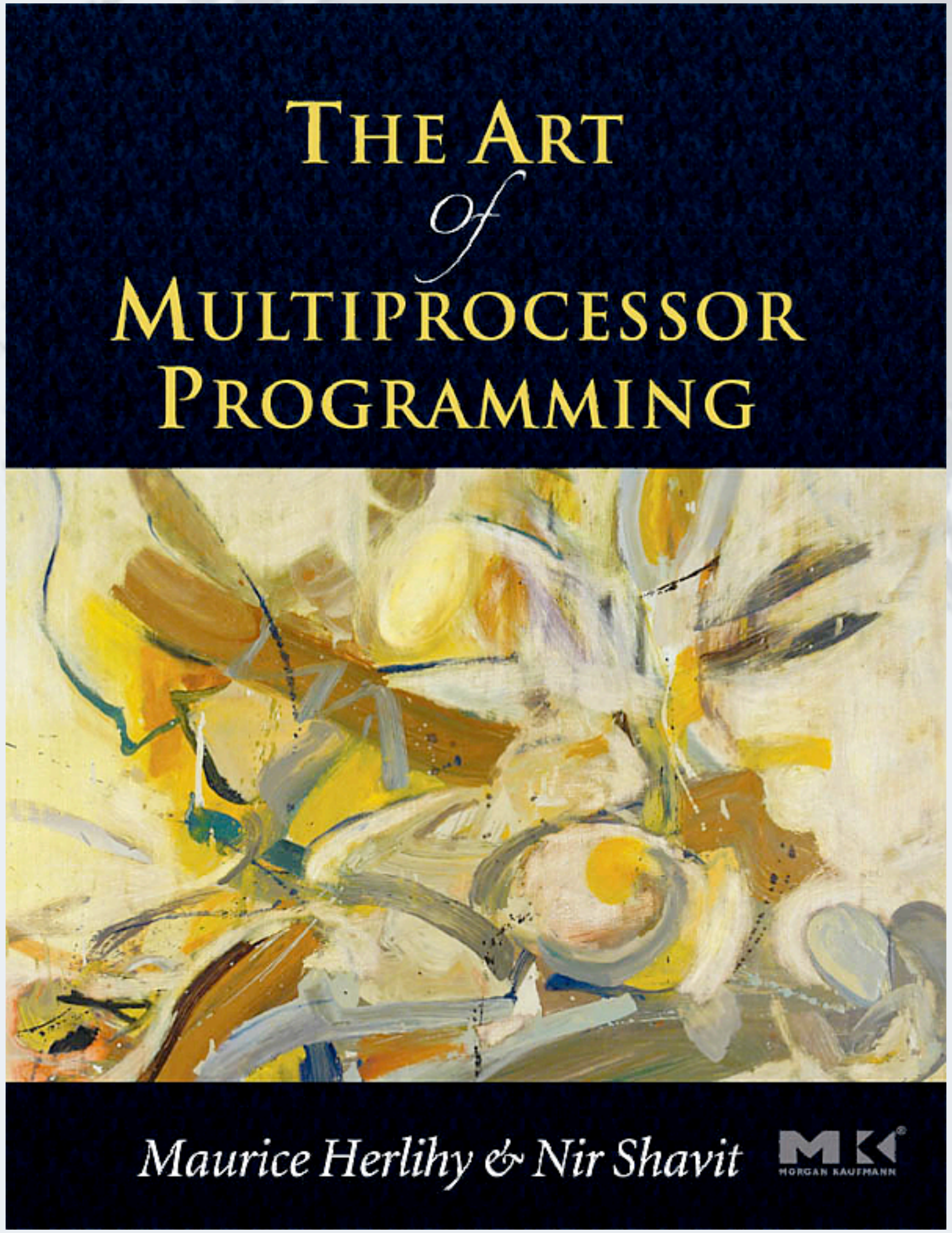
- **StampedLock performed very well in all our tests**
  - **Much faster than ReentrantReadWriteLock**
- **Offers a way to do optimistic locking in Java**
- **Good idioms have a big impact on the performance**

Conclusion

Where to next?

The Art Of Multiprocessor Programming

- **Herlihy & Shavit**
  - Theoretical book on how things work "under the hood"
  - Good as background reading



THE ART
Of
MULTIPROCESSOR
PROGRAMMING

Maurice Herlihy & Nir Shavit

JSR 166

- **http://gee.cs.oswego.edu/**
- **Concurrency-Interest mailing list**
  - **Usage patterns and bug reports on Phaser and StampedLock are always welcome on the list**

Heinz Kabutz (heinz@kabutz.net)

- **The Java Specialists' Newsletter**
  - **Subscribe today:**
    - **http://www.javaspecialists.eu**
- **Concurrency Specialist Course**
  - **Offered in Crete in 2014 or in-house at your company**
  - **http://www.javaspecialists.eu/courses/concurrency.jsp**
- **Questions?**

Phaser And StampedLock Concurrency Synchronizers

heinz@javaspecialists.eu

Questions?