From Smile To Tears: Emotional StampedLock

Dr Heinz M. Kabutz

Last updated 2013-11-05

Javaspecialists.eu
Java Training

Heinz Kabutz

- **Author of The Java Specialists' Newsletter**
  - **Articles about advanced core Java programming**
- **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**
- **Some applications need large amounts of state**
  - **Immutable could stress the garbage collector**
  - **Unshared could stress the memory volume**

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**

"Good" And "Bad" Context Switches

- **"Good" Context Switch**
  - **Thread has used up its time quantum and can be swapped out by the OS in a single clock cycle**
  - **Also called "Involuntary" context switch**
- **"Bad" Context Switch**
  - **Executing thread needs to stop because it cannot acquire a resource held by another suspended thread**
  - **Also called "Voluntary" context switch**
  - **Can cost tens of thousands of clock cycles**

Fine Grained Locking

- **"Synchronized" causes *"bad" 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 Lock-free Algorithms

- **Lock-based algorithms can cause scalability issues**
  - **If a thread is holding a lock and is swapped out, no one can progress**
  - **Amdahl's and Little's laws explain why we can't scale**
- **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**

StampedLock

Javaspecialists.eu
Java Training

Motivation For StampedLock

- Some constructs need a form of read/write lock
- ReentrantReadWriteLock can cause starvation
  - Plus it always uses pessimistic locking

Motivation For StampedLock

- **StampedLock provides optimistic locking on reads**
  - **Which can be converted easily to a pessimistic read**
- **Write locks are always pessimistic**
  - **Also called *exclusive* locks**
- **StampedLock is not reentrant**

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(); }
  }
}
```

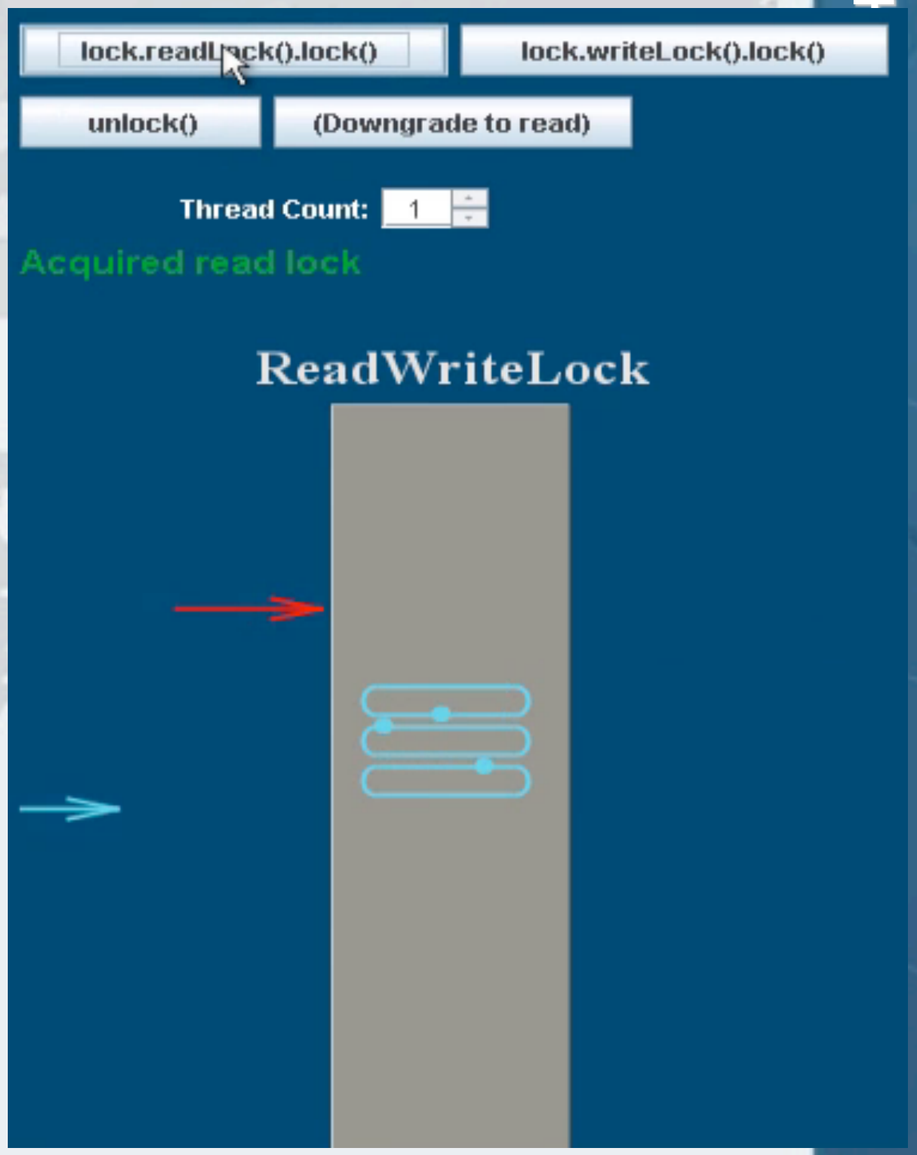
The cost overhead of the RWLock means we need at least 2000 instructions to benefit from the readLock() added throughput

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)
- 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!



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**

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 better performance than ReentrantReadWriteLock**
  - **Latest versions do not suffer from starvation of writers**
- **Cons**
  - **Idioms are more difficult than with ReadWriteLock**
    - **A small change in idiom code can make a big difference in performance**
  - **Not nonblocking**
  - **Non-reentrant**

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

- **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, st
}
```

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 boolean 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; ...
        return true;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
    return false;
  } finally { sl.unlock(stamp); }
}
```

Code Idiom For Conditional Change

```
public boolean 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; ...
        return true;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
    return false;
  } finally { sl.unlock(stamp); }
}
```

We get a pessimistic read lock

Code Idiom For Conditional Change

```
public boolean 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 = newSt
        return true;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
    return false;
  } 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 boolean changeStateIfEquals(oldState1, oldState2, ...
                                   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; ...
        return true;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
    return false;
  } finally { sl.unlock(stamp); }
}
```

We try convert our read lock to a write lock

Code Idiom For Conditional Change

```
public boolean 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; ...
        return true;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
    return false;
  } 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 boolean 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; ...
        return true;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
    return false;
  } 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 boolean 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 = newS
        return true;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
    return false;
  } finally { sl.unlock(sta
}
```

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 boolean changeStateIfEquals(

  long stamp = sl.readLock();
  try {
    while (state1 == oldState1 && state2 == oldState2 ...) {
      long writeStamp = sl.tryConvertToWriteLock(stamp);
      if (writeStamp != 0L) {
        stamp = writeStamp;
        state1 = newState1; state2 = newState2; ...
        return true;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
    return false;
  } 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 boolean 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; ...
        return true;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
    return false;
  } finally { sl.unlock(stamp); }
}
```

Applying To Our Point Class

```
public boolean 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;
        return true;
      } else {
        sl.unlockRead(stamp);
        stamp = sl.writeLock();
      }
    }
    return false;
  } 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 12th Oct 2012**
  - **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

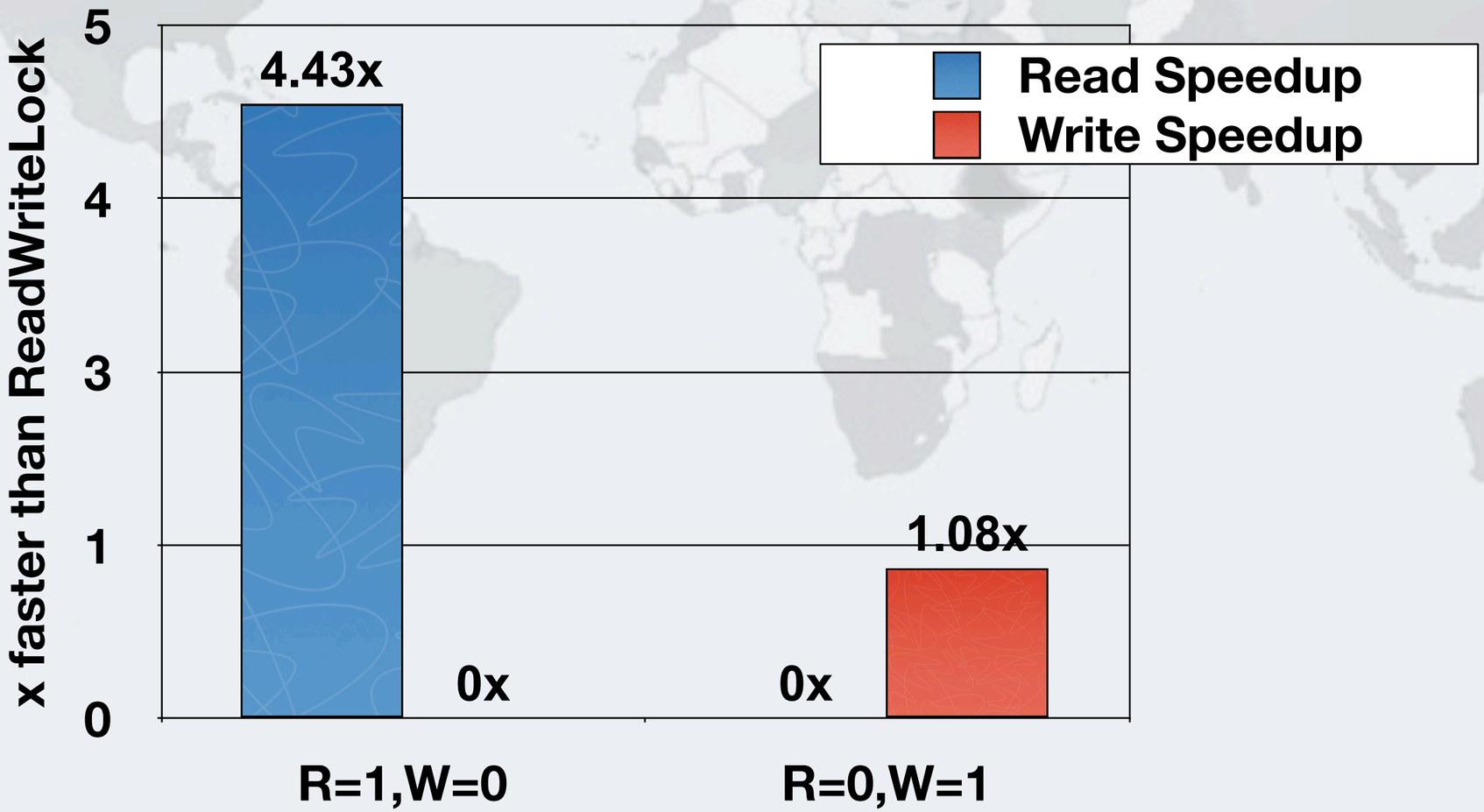
- In our test, we used
  - lambda-8-b75-linux-x64-28_jan_2013.tar.gz
  - Two CPUs, 4 Cores each, no hyperthreading
    - 2x4x1
  - Ubuntu 9.10
  - 64-bit
  - Intel(R) Core(TM) i7 CPU 920 @ 2.67GHz
    - L1-Cache: 256KiB, internal write-through instruction
    - L2-Cache: 1MiB, internal write-through 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**

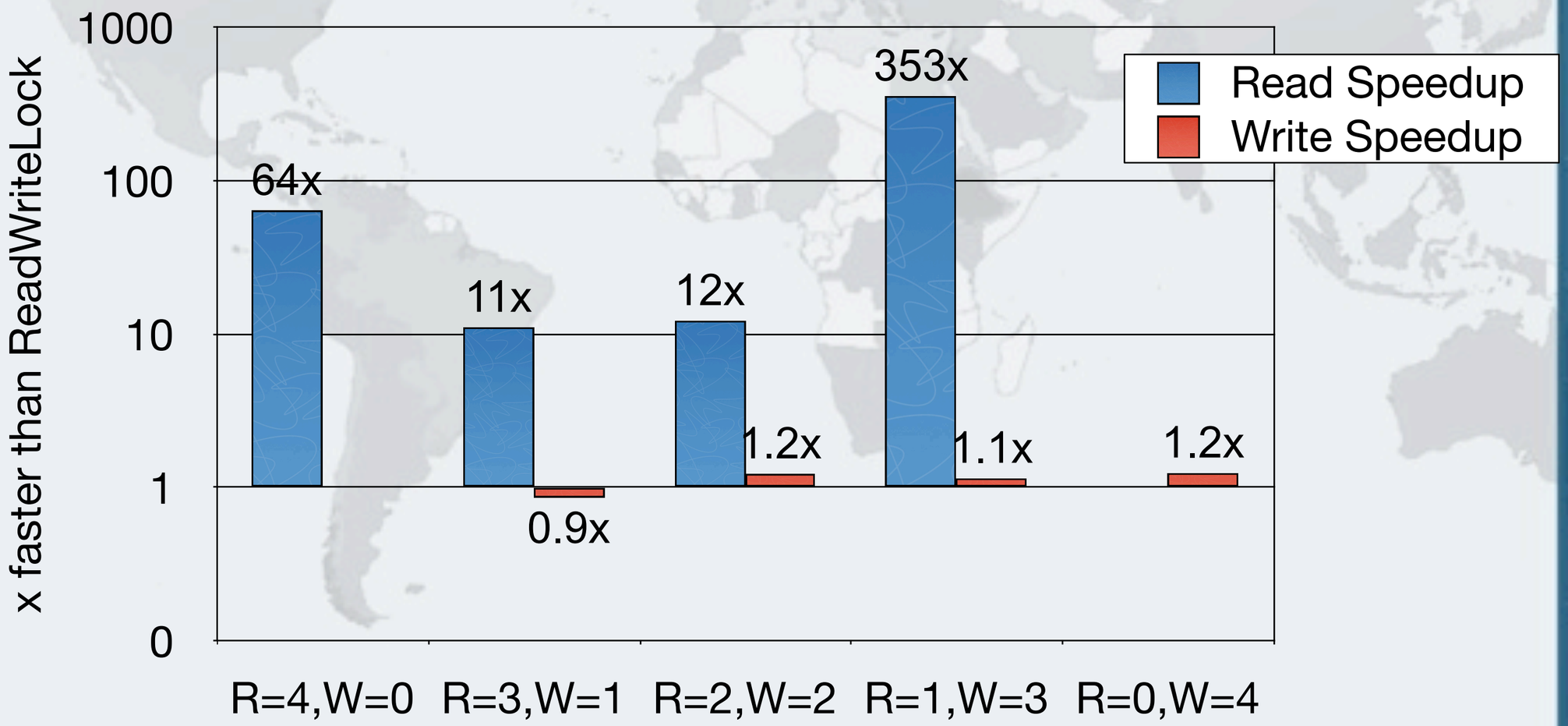
How Much Faster Is StampedLock Than ReentrantReadWriteLock?

- **With a single thread**



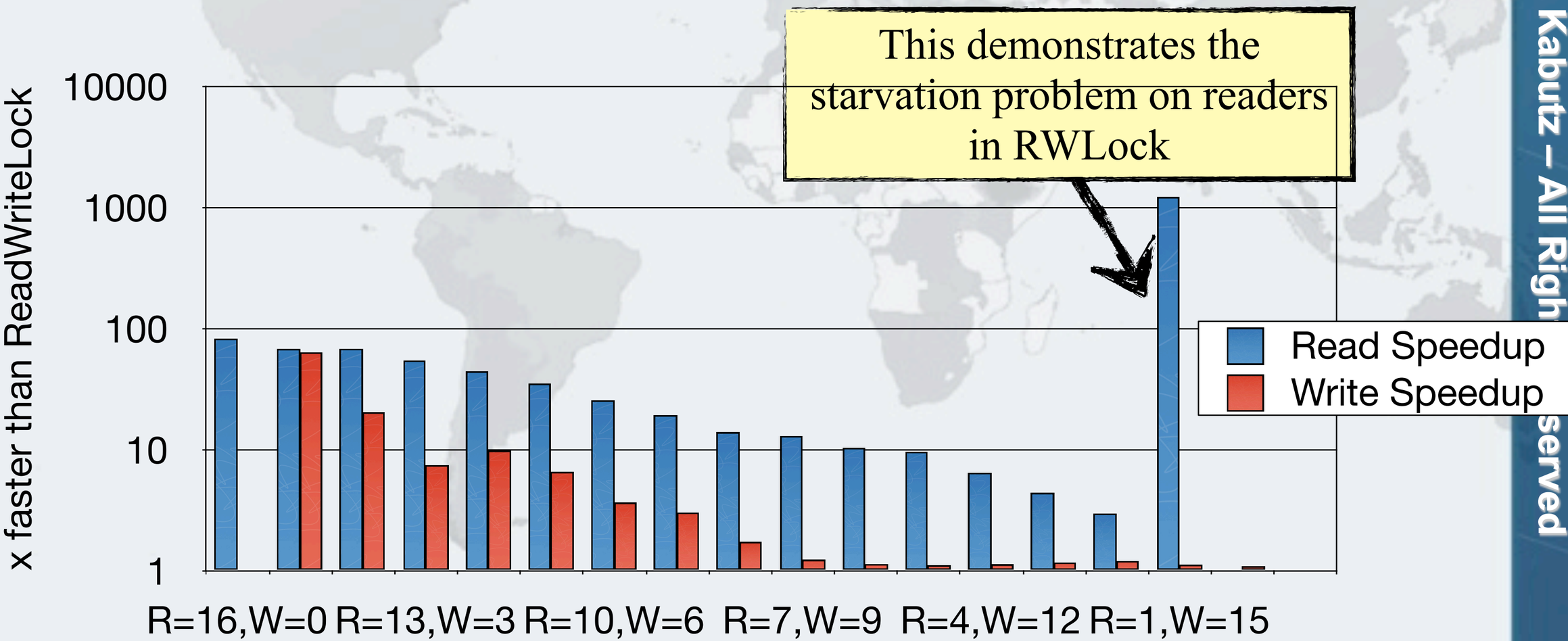
How Much Faster Is StampedLock Than ReentrantReadWriteLock?

- **With four threads**

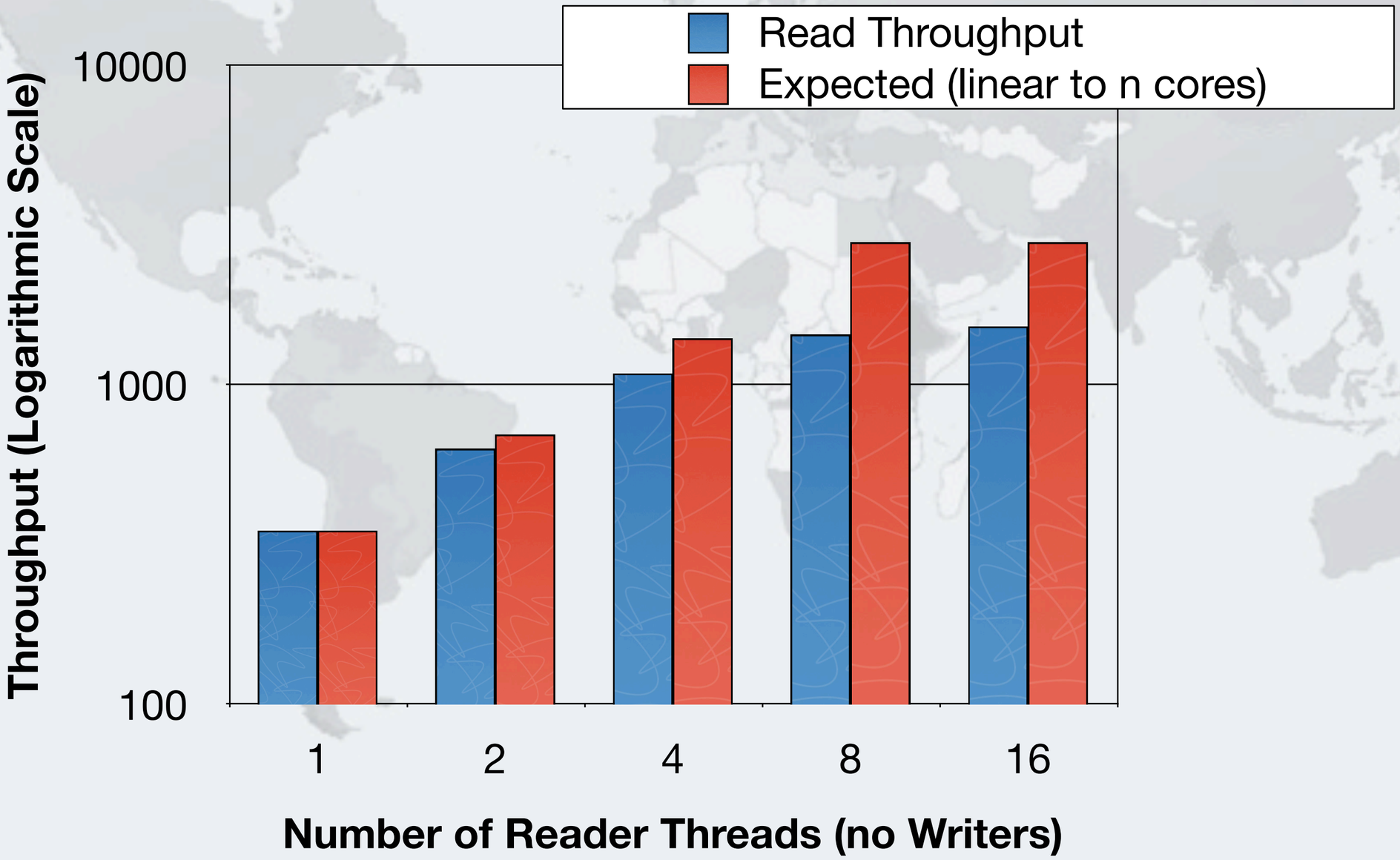


How Much Faster Is StampedLock Than ReentrantReadWriteLock?

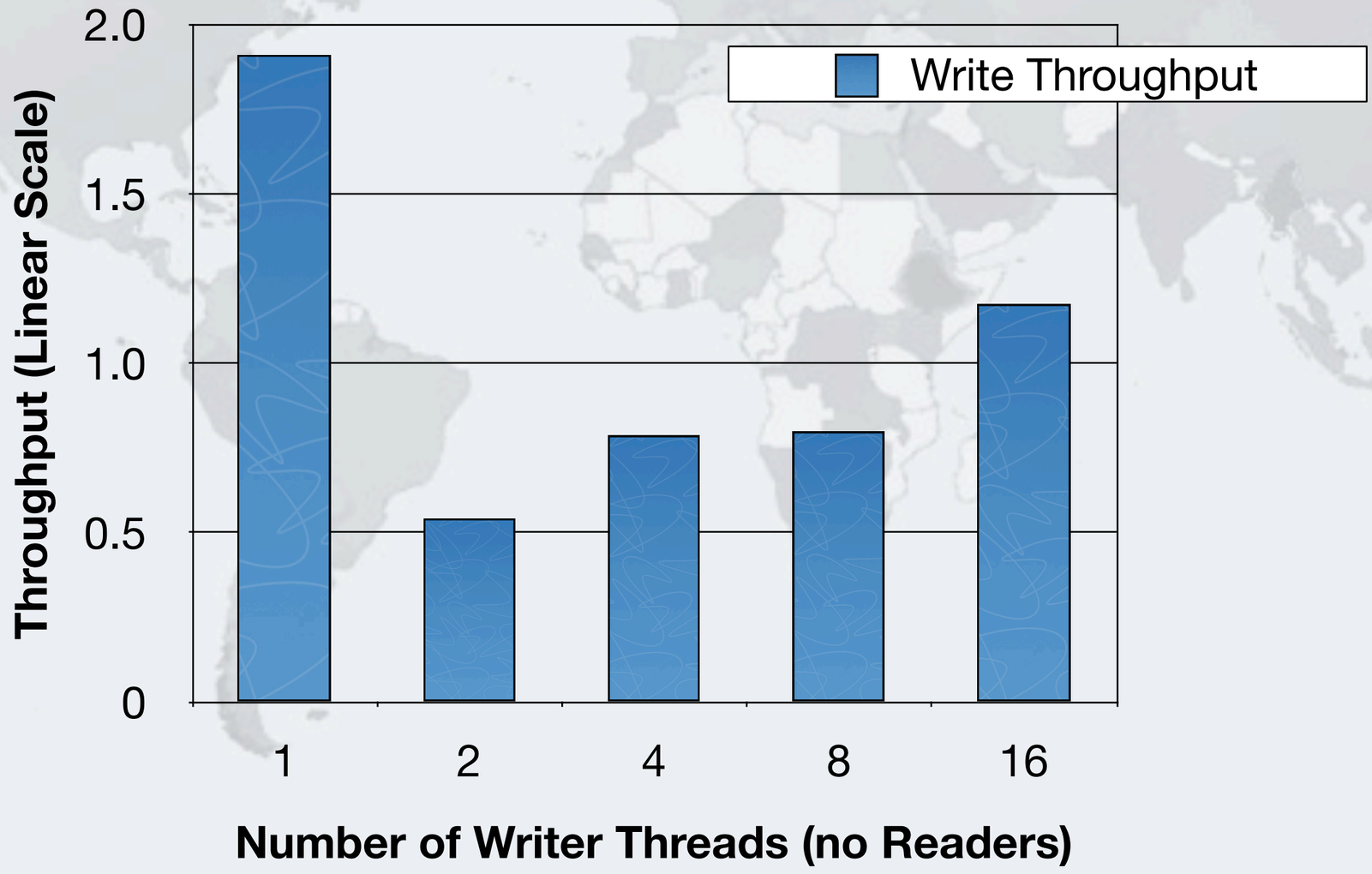
- **With sixteen threads**



Reader Throughput With StampedLock

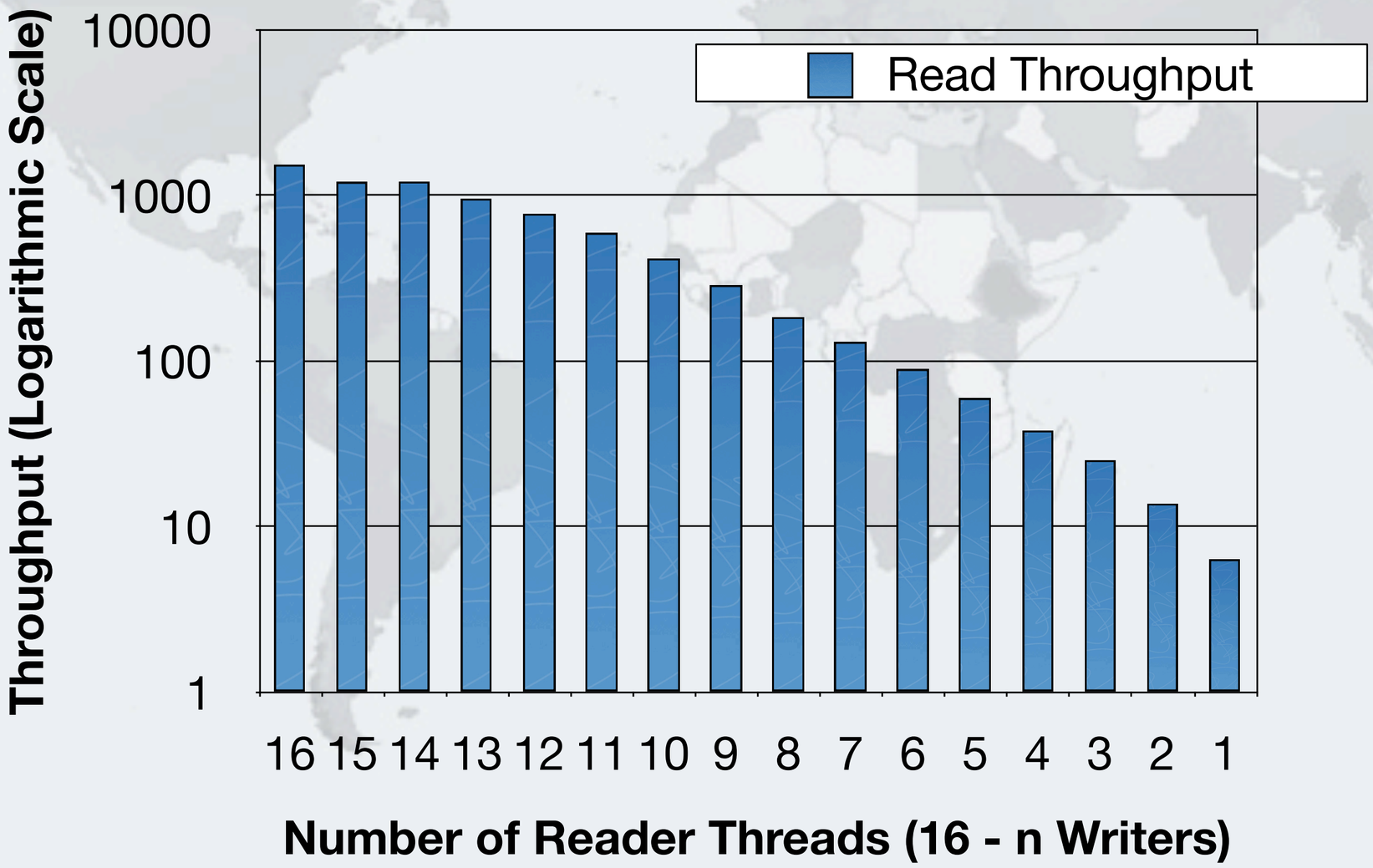


Writer Throughput With StampedLock

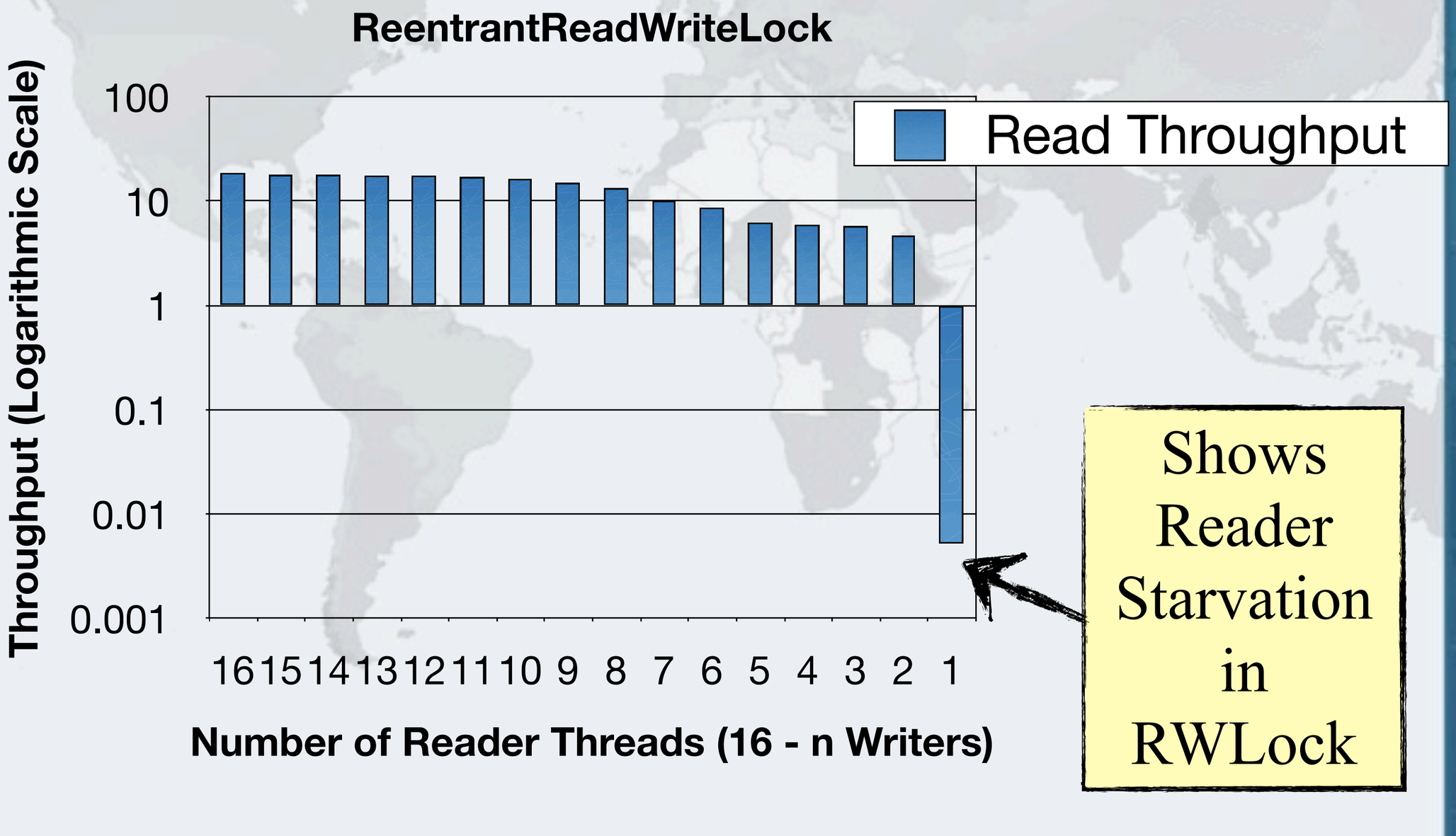


Note: Linear Scale throughput

Mixed Reader Throughput StampedLock



Mixed Reader Throughput 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**

Idioms With Lambdas

Idioms With Lambdas

- **Java 8 lambdas allow us to define a structure of a method, leaving the details of what to call over to users**
  - **A bit like the "Template Method" Design Pattern**

```
List<String> students = new ArrayList<>();
Collections.addAll(students, "Anton", "Heinz", "John");
students.forEach((s) -> System.out.println(s.toUpperCase()));
```

```
ANTON
HEINZ
JOHN
```

LambdaFAQ.org

- **Edited by Maurice Naftalin**
  - **Are lambda expressions objects?**
  - **Why are lambda expressions so-called?**
  - **Why are lambda expressions being added to Java?**
  - **Where is the Java Collections Framework going?**
  - **Why are Stream operations not defined directly on Collection?**
  - **etc.**

Idioms For Using StampedLock

```
import java.util.concurrent.locks.*;
import java.util.function.*;

public class LambdaStampedLock extends StampedLock {
  public void writeLock(Runnable writeJob) {
    long stamp = writeLock();
    try {
      writeJob.run();
    } finally {
      sl.unlockWrite(stamp);
    }
}
```

```
lsl.writeLock(
    () -> {
      x += deltaX;
      y += deltaY;
    }
);
```

Idioms For Using StampedLock

```
public Object optimisticRead(Supplier<?> supplier) {
  long stamp = tryOptimisticRead();
  Object result = supplier.get();
  if (!validate(stamp)) {
    stamp = readLock();
    try {
      result = supplier.get();
    } finally {
      unlockRead(stamp);
    }
  }
  return result;
}
```

```
double[] xy = (double[])lsl.optimisticRead(
  () -> new double[]{x, y}
);
return Math.hypot(xy[0], xy[1]);
```

Idioms For Using StampedLock

```
public static boolean conditionalWrite(
    BooleanSupplier condition, Runnable action) {
  long stamp = readLock();
  try {
    while (condition.getAsBoolean()) {
      long writeStamp = tryConvertToWriteLock(stamp);
      if (writeStamp != 0) {
        action.run();
        stamp = writeStamp;
        return true;
      } else {
        unlockRead(stamp);
        stamp = writeLock();
      }
    }
    return false;
  } finally {
    unlock(stamp);
  }
}
```

```
return lsl.conditionalWrite(
    () -> x == oldX && y == oldY,
    () -> { x = newX; y = newY; }
);
```

Nonblocking Point

Javaspecialists.eu
Java Training

Nonblocking Point

- **Instead of relying on synchronizers, use non-blocking algorithm**
  - **Might create additional objects**
    - **But a contended StampedLock will also create objects**
  - –

Store State Inside AtomicReference

```
public class PointNonblocking {
  public static final double[] INITIAL = new double[]{0, 0};
  private final AtomicReference<double[]> xy =
      new AtomicReference<>(INITIAL);

  public void move(double deltaX, double deltaY) {
    double[] current, next;
    do {
      current = xy.get();
      double x = current[0];
      double y = current[1];
      next = new double[]{x + deltaX, y + deltaY};
    } while (!xy.compareAndSet(current, next));
  }
```

Reading Does Not Create Objects

```
public double distanceFromOrigin() {
  double[] current = xy.get();
  double x = current[0];
  double y = current[1];
  return Math.hypot(x, y);
}
```

Conditional Write Can Make Objects

```
public boolean moveIfAt(double oldX, double oldY,
                        double newX, double newY) {
  double[] current, next;
  do {
    current = xy.get();
    double x = current[0];
    double y = current[1];
    if (x != oldX || y != oldY) {
      return false;
    }
    next = new double[]{newX, newY};
  } while (!xy.compareAndSet(current, next));
  return true;
 }
}
```

Which Is Fastest?

- **StampedLock, synchronized or non-blocking?**
  - **Depends on how you measure**
  - **For multiple readers, lock-free is probably faster**
    - **http://mechanical-sympathy.blogspot.de/2013/08/lock-based-vs-lock-free-concurrent.html**
  - **But synchronized might be faster than both in some cases**
  - **Depends on how you use it**
    - **(Great consultant answer :-))**

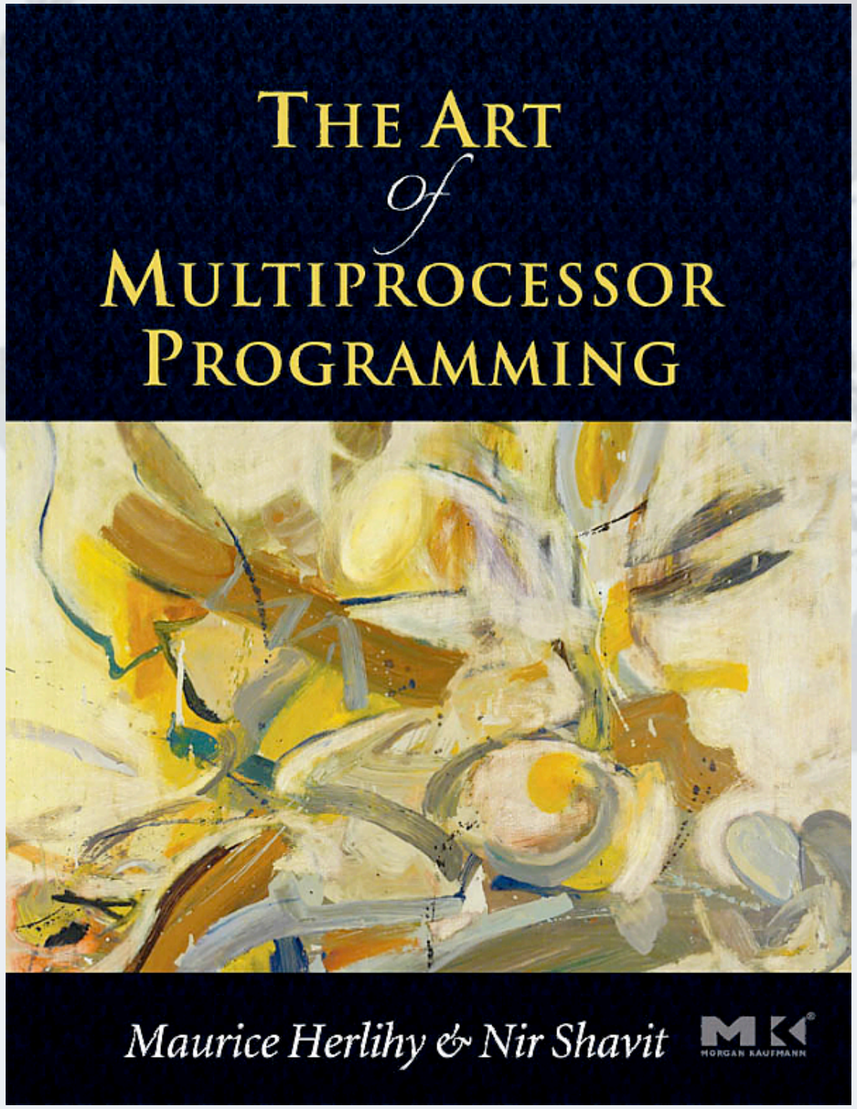
Conclusion

Where to next?

Javaspecialists.eu
Java Training

The Art Of Multiprocessor Programming

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



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

Mechanical Sympathy - Martin Thompson

- **Mailing list**
  - **mechanical-sympathy@googlegroups.com**
- **Blog**
  - **http://mechanical-sympathy.blogspot.com**

Heinz Kabutz (heinz@kabutz.net)

- **The Java Specialists' Newsletter**
  - **Subscribe today:**
    - **http://www.javaspecialists.eu**
- **Questions?**

From Smile To Tears: Emotional StampedLock

heinz@javaspecialists.eu

Questions?

Javaspecialists.eu
java training

The Java Specialists' Newsletter

Javaspecialists.eu
java training