# Multi-threaded Performance And Scalability

Dr Heinz M. Kabutz

http://www.javaspecialists.eu/talks/wjax12/kabutz.pdf

Javaspecialists.eu
java training

# Dr Heinz Kabutz

- Brief Biography
  - German from Cape Town, now lives on an island in Greece
  - PhD Computer Science from University of Cape Town
  - The Java Specialists' Newsletter
  - Java programmer
  - Java Champion since 2005
- Advanced Java Courses
  - Concurrency Specialist Course
  - Java Specialist Master Course
  - Design Patterns Course
  - http://www.javaspecialists.eu

# Dr Heinz Kabutz

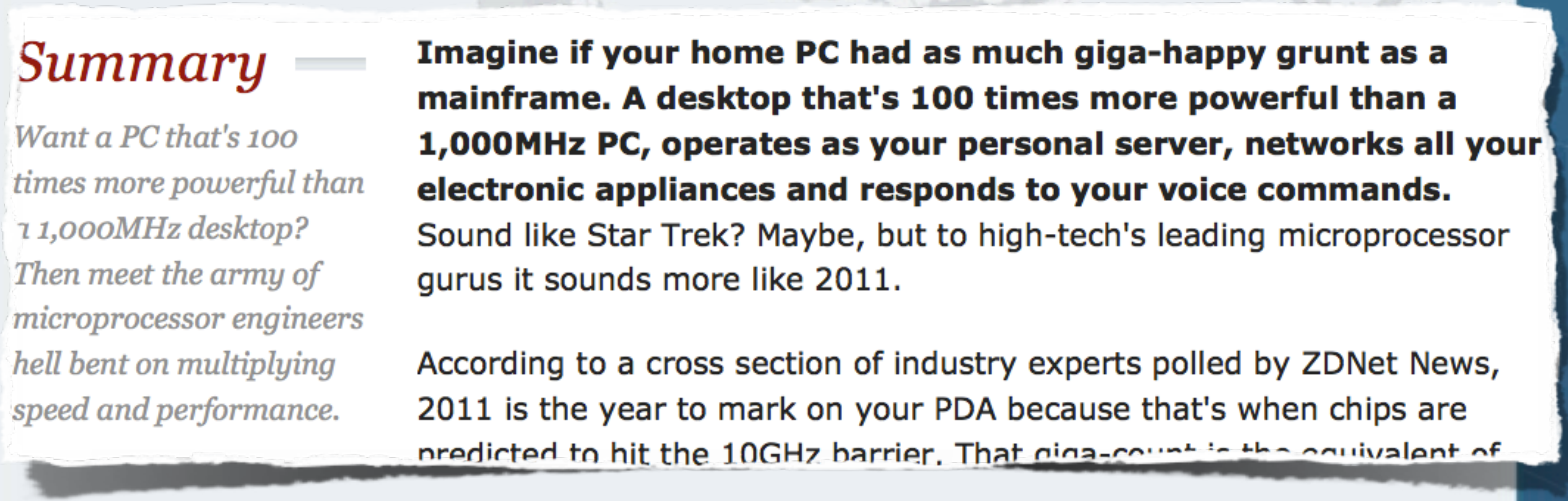
- **Brief Biography**
  - German from Cape Town, now lives on an island in Greece
  - PhD Computer Science from University of Cape Town
  - The Java Specialists' Newsletter
  - Java programmer
  - Java Champion since 2005
- **Advanced Java Courses**
  - Concurrency Specialist Course
  - Java Specialist Master Course
  - Design Patterns Course
  - http://www.javaspecialists.eu

# Threads Help Utilize Our Hardware

- Do something else whilst waiting for IO
  - e.g. blocking IO, progress bars, etc.
- Split a problem into smaller chunks and solve together
  - e.g. fork/join

# Let's Go Fast Fast Fast

- **In 2000, Intel predicted 10GHz chips on desktop by 2011**
  - **http://www.zdnet.com/news/taking-chips-to-10ghz-and-beyond/96055**



## Summary

*Want a PC that's 100 times more powerful than a 1,000MHz desktop? Then meet the army of microprocessor engineers hell bent on multiplying speed and performance.*

**Imagine if your home PC had as much giga-happy grunt as a mainframe. A desktop that's 100 times more powerful than a 1,000MHz PC, operates as your personal server, networks all your electronic appliances and responds to your voice commands.** Sound like Star Trek? Maybe, but to high-tech's leading microprocessor gurus it sounds more like 2011.

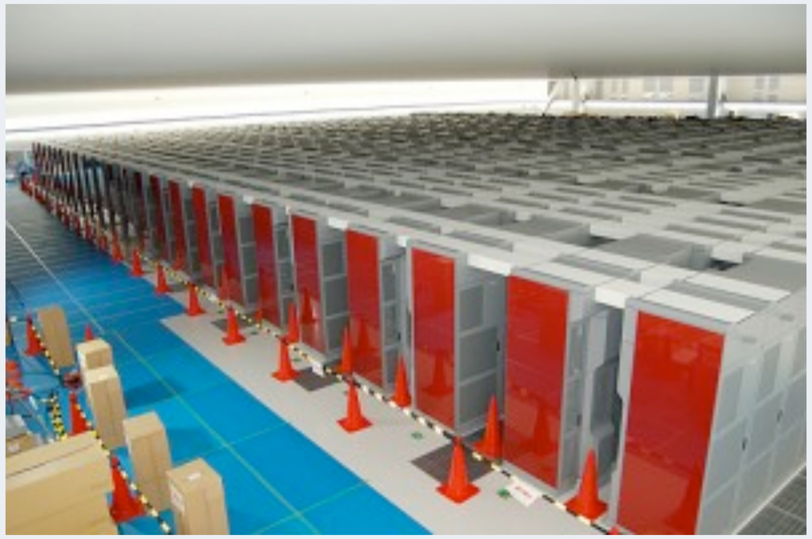
According to a cross section of industry experts polled by ZDNet News, 2011 is the year to mark on your PDA because that's when chips are predicted to hit the 10GHz barrier. That giga-count is the equivalent of

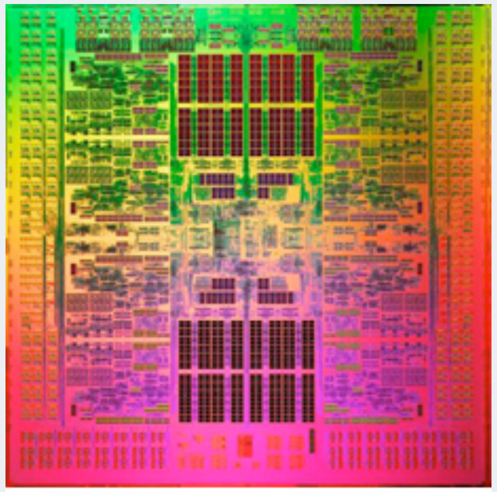
# Let's Go Fast Fast Fast

- **Core i7 990x hit the market early 2011**
  - **3.46GHz clock stretching up to 3.73 GHz in turbo mode**
  - **6 processing cores**
  - **Running in parallel, we get 22GHz of processing power!**

# Let's Go Fast Fast Fast

- Japanese 'K' Computer June 2011
  - 8.2 petaFLOPS
    - 8 200 000 000 000 000 floating point operations per second
    - Intel 8087 was 30 000 FLOPS, 273 billion times slower
  - 548,352 cores from 68,544 2GHz 8-Core SPARC64 VIIIfx processors





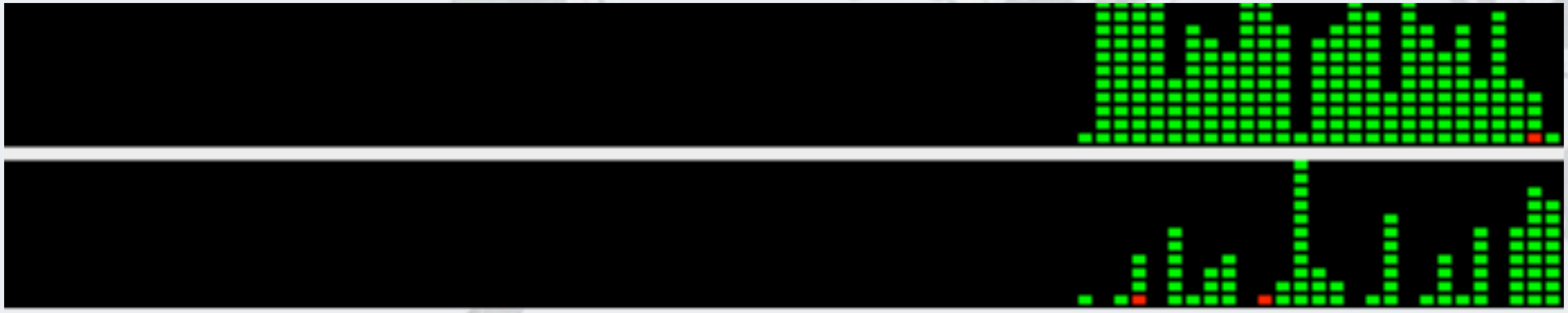
SPARC64™ VIIIfx

# Which Is Faster, "K" Or One Simple Intel "i7"

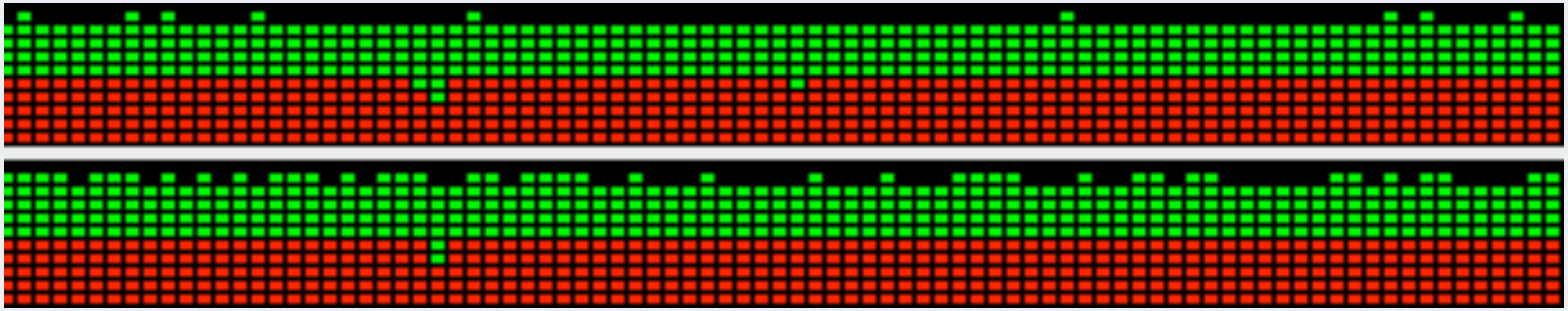
- **Intel i7 has a total speed of 3.73GHz x 6 = 22 GHz**
- **K has a total speed of 2GHz x 548352 = 1096704 GHz**
- **Which is faster?**
  - **If we can parallelize our algorithm, then K is 50,000 times faster**
  - **But if it has to run in serial, then one Intel i7 is almost twice as fast**

# Thinking About Performance

- **We want to utilize all our CPUs with application code**
  - – **Overly coarse-grained locking means the CPUs are starved for work**
    - **Took 51 seconds to complete**

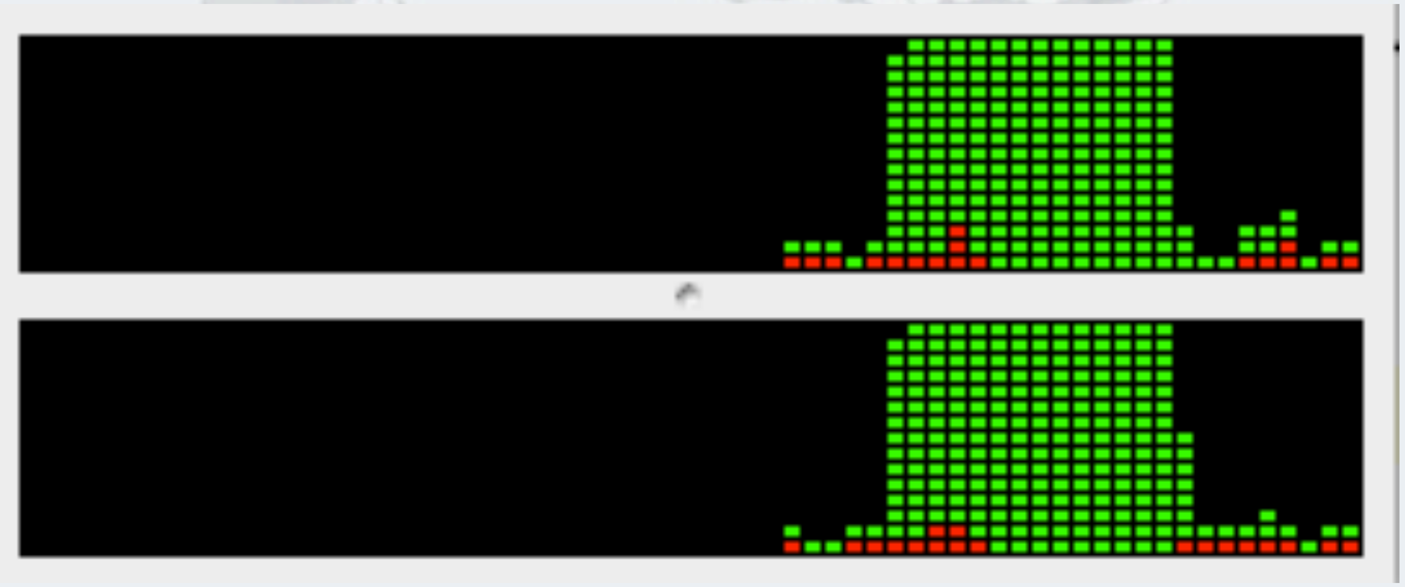


  - – **Too fine-grained locking means we are busy with system code**
    - **Took 745 seconds to complete**



# Thinking About Performance

- **Busy CPUs by using local data and merging results**
  - **– Took 28 seconds to complete**



# Performance Vs Scalability

- **We can measure application performance in many ways**
  - **Latency: How fast *one* unit of work runs**
  - **Throughput: How many units of work can be done per unit of time**
- **A system is *scalable* when the throughput increases with more computing resources such as CPUs or faster IO**

# How Fast Vs How Much

- **In traditional performance optimizations, we try to make our code run *faster***
  - **e.g. cache old results, improve complexity of algorithm**

# How Fast Vs How Much

- **When we tune for scalability, we want to parallelize work**
  - **Thus by adding more CPUs, we can complete more work**

# How Fast Vs How Much

- **Many code tricks that make code run *faster* also make it less scalable**
  - **For example, maintaining previous results adds synchronization**

# 2-Tier Vs Multi-tier System

- **Old 2-tier systems**
  - **Rich client connected to database**
  - **Typically low latency**
  - **Only two layers**
  - **Not scalable to millions of users**

# 2-Tier Vs Multi-tier System

- **Multi-tier systems can scale**
  - **Overall latency might be worse than with 2-tier**
  - **But throughput is much better**
  - **Can scale to millions of users**

# 2-Tier Vs Multi-tier System

- Every system you build should also work in a cluster
  - Don't leave "clustering" as an optional extra for the end

# Evaluating Performance Tradeoffs

- **Always find out the performance requirements**
  - **Are the requirements low latency?**
    - **What is the maximum wait time for your clients?**
  - **Or high throughput?**
    - **How many clients do you want to support at the same time?**

# Amdahl's And Little's Laws

## Performance and Scalability

# Amdahl's Law

- **Some problems can be solved faster by parallelizing portions of it**
  - **N = number of cores**
  - **F = serial portion**

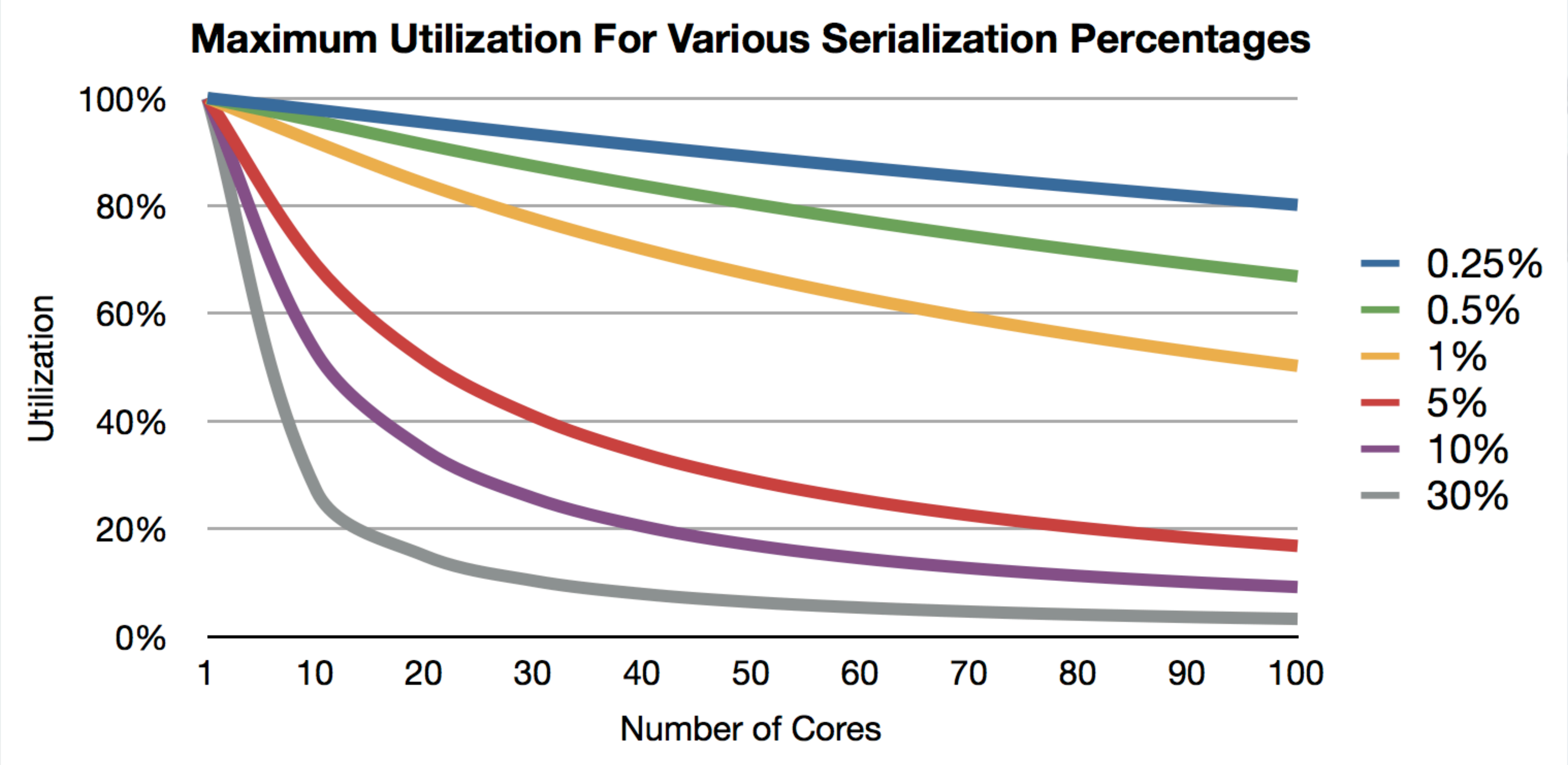
$$\text{Speedup} \leq \frac{1}{F + \frac{(1 - F)}{N}}$$

# Watermelons

- **More workers (N) can *plant* watermelons faster and *harvest* them faster**
- **But no amount of additional workers can make them *grow* any faster (F)**
  - **The *growing* is our serial section**

# Utilization According To Amdahl

- **Even with a small section needing to run in serial, we are limited to how we can speed up our program**



# Problems With Using Amdahl's Law In Practice

- **You cannot accurately predict serial portion (F)**
  - **At best we can explain why the system is slow**
- **Amdahl's law does not set an upper limit on processors**
  - **The most powerful supercomputer "K" has 500,000 cores**
- **It assumes the amount of data remains the same**
  - **Usually data grows as utilization does**

# Little's Law

- **A better law for modeling real systems is Little's Law**
  - **The long-term average number of customers in a *stable system* L is equal to the long-term average effective arrival rate, λ, multiplied by the average time a customer spends in the system, W**
    - **$L = \lambda W$**
  - **Throughput is the inverse of service time**

# Little's Law

- **If your service time is 1ms and you have one server, then the maximum throughput is 1000 transactions per second**
  - **To increase throughput, add more servers or decrease service time**
- **Good paper showing how the law can be used in practice**
  - **http://ie.technion.ac.il/serveng/Lectures/Little.pdf**

# Practical Examples Of Little's Law

- **In a store, the limiting factor is usually the cashiers**
  - **Years ago, Aldi in Germany increased the speed of the cashiers by making them memorize *all* the article codes**
  - **They increased throughput by speeding up their cashiers**
  - **They also limited the number of different types of articles**

# Threading And Little's Law

- A synchronized section only lets one thread in at a time
  - – $\lambda = L/W$
  - – L is 1, since the code is synchronized
  - – W is however long it takes to acquire the lock, call the critical section, release the lock again
  - – If W is 20ms, our maximum throughput is 1/0.02 = 50 per second
- It does not matter how many CPUs are in the system, we are restricted by Little!

# Costs Introduced By Threads

Performance and Scalability

# Costs Introduced By Threads

- **Single-threaded programs do not have to synchronize, context switch or use locks to protect data**
- **Threads can offer performance improvements, but there is a cost**

# Context Switching

- **When we have more runnable threads than CPUs, the operating system will need to do a *context switch***
  - **The thread is swapped out, together with call stack and related data**
  - **Another thread is swapped in**

# Context Switching Costs Thousands Of Cycles

- **On Mac OS X it took on average of 3400 clock cycles**
  - **Thousands sounds like a lot, but it was only 0.001% of performance!**
  - **We should not let the context switch happen unnaturally often**
- **Cost does not only come from the actual context switch, but also the related events**
  - **The cache might need to be filled with new data**
  - **Locking and unlocking might be causing the context switch**

# Memory Synchronization

- **Java uses *memory barriers* to ensure that fields are flushed and caches invalidated**
  - **We use *volatile* and *synchronized* to place memory barriers**
  - **Memory barriers slow us down**
  - **They also limit how our code can be optimized**

# Deaf Piano Tuning Association

- **Tuning involves measurement**
  - **There is a blind piano tuning association**
    - **But no deaf piano tuning association**

# Uncontended Locks Optimized

- **Uncontended locks can be optimized away by HotSpot**
  - **Escape Analysis sees that object never escapes from code block**
    - **The object can then be constructed on the stack or in the registers**
    - **Locking can be removed automatically**

# Lesson:
# Don't Worry About Uncontended Locks

# Spinning Before Actual Blocking

- CPU spinning for a bit before actual locking
  - -XX:+UseSpinning turns on spinning (default off)
  - -XX:PreBlockSpin=20 spin count for maximum spin iterations before entering operating system thread synchronization code (default 10)
- Remember to measure and check that this is helping

# Reducing Lock Contention

## Performance and Scalability

# Reducing Lock Contention

- **The biggest threat to scalability is the exclusive lock**
  - **Amdahl's Law shows that even a small section of serial code will limit the amount of speedup we can achieve**
  - **And with Little's Law L=λW, the serial section always has L=1**
    - **Thus λ=1/W**
- **Our aim would need to be to reduce contended locks**
  - **But of course ensuring that the code is still *safe***

# Reducing Lock Contention

- The biggest threat to scalability is the exclusive lock
  - Amdahl's Law shows that even a small section of serial code will limit the amount of speedup we can achieve
  - And with Little's Law $L=\lambda W$, the serial section always has $L=1$
    - Thus $\lambda=1/W$
- Our aim would need to be to reduce contended locks
  - But of course ensuring that the code is still *safe*

***Safety First***

# Reducing Lock Contention

- **The biggest threat to scalability is the exclusive lock**
  - **Amdahl's Law shows that even a small section of serial code will limit the amount of speedup we can achieve**
  - **And with Little's Law $L=\lambda W$, the serial section always has $L=1$**
    - **Thus $\lambda=1/W$**
- **Our aim would need to be to reduce contended locks**
  - **But of course ensuring that the code is still *safe***

***Safety First***

***Safety First***

# Reducing Lock Contention

- **The biggest threat to scalability is the exclusive lock**
  - **Amdahl's Law shows that even a small section of serial code will limit the amount of speedup we can achieve**
  - **And with Little's Law L=λW, the serial section always has L=1**
    - **Thus λ=1/W**
- **Our aim would need to be to reduce contended locks**
  - **But of course ensuring that the code is still *safe***

***Safety First***

***And measure!***

***Safety First***

# How To Reduce Lock Contention

- **We have three main ways to reduce contention**
  - **Reduce the duration that locks are held**
  - **Reduce frequency with which locks are requested**
  - **Replace exclusive locks with non-locking thread-safe mechanisms**

# Narrowing Lock Scope ("Get In, Get Out")

- **We should always hold locks for as short as possible**
  - **Our performance is limited by how long we hold the locks**
    - **If the lock is held for 2 ms, throughput is maximum of 500 tx/s**
    - **If it is held for only 1ms, throughput can increase to 1000 tx/s**

# AttributeStore With A Long Critical Section

- **We are locking the entire matching method, even the regular expression pattern matching**

```
@ThreadSafe
public class AttributeStore {
  @GuardedBy("this")
  private final Map<String, String> attributes =
    new HashMap<>();
  public synchronized boolean userLocationMatches(
      String name, String regexp) {
    String key = "users." + name + ".location";
    String location = attributes.get(key);
    if (location == null)
      return false;
    else
      return Pattern.matches(regexp, location);
  }
}
```

# A Better Way To Write "userLocationMatches"

- **Faster is to lock only the portion that is necessary**
  - **In addition, we are encapsulating the lock by using a private field**

```
public boolean userLocationMatches(
    String name, String regexp) {
  String key = "users." + name + ".location";
  String location;
  synchronized (attributes) {
    location = attributes.get(key);
  }
  if (location == null)
    return false;
  else
    return Pattern.matches(regexp, location);
}
```

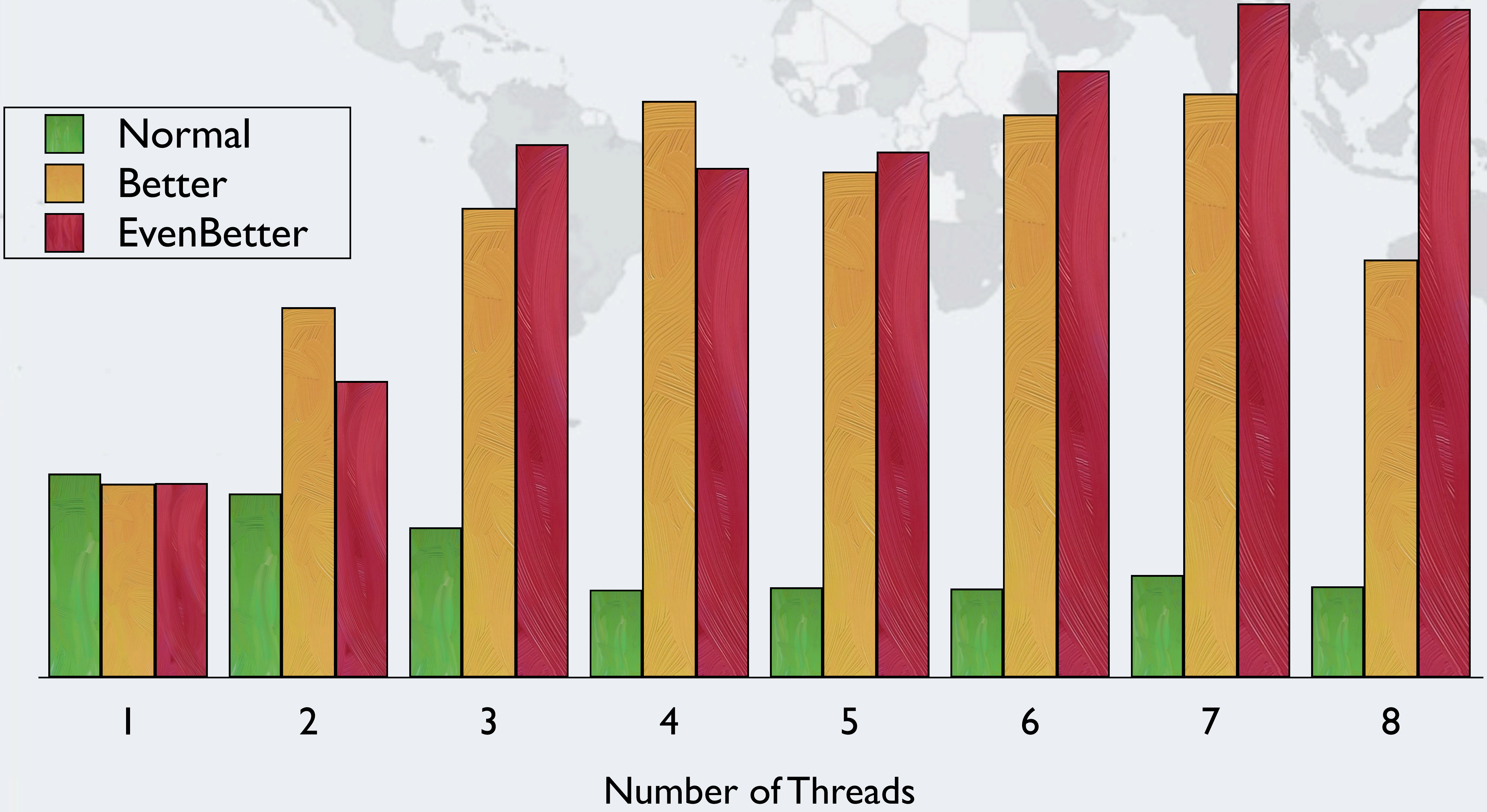
# Or Use A ConcurrentHashMap

- **The ConcurrentHashMap is non-blocking on reads**
  - **The serial section is reduced to just a memory barrier via volatile**

```
@ThreadSafe
public class EvenBetterAttributeStore {
  @GuardedBy("this")
  private final Map<String, String> attributes =
      new ConcurrentHashMap<>();

  public boolean userLocationMatches(
      String name, String regexp) {
    String key = "users." + name + ".location";
    String location = attributes.get(key);
    if (location == null)
      return false;
    else
      return Pattern.matches(regexp, location);
  }
}
```

# AttributeStore Performance Comparisons

- Throughput for a million lookups on an 8-core machine



# Reducing Lock Granularity

- **We can use *lock splitting* or *lock striping* to reduce contention**
- **Imagine if there was one lock for the entire application**
  - **Completely unrelated parts of the program would be run in serial**
- **If a class has unrelated fields, we can use separate locks**
- **In ServerStatus (next slide) we could use two locks to allow updating of users and queries at the same time**

# ServerStatus Uses A Single Lock

```
@ThreadSafe
public class ServerStatus {
  @GuardedBy("this")
  private final Set<String> users = new TreeSet<>();
  @GuardedBy("this")
  private final Set<String> queries = new TreeSet<>();

  public synchronized void addUser(String user) {
    users.add(user);
  }
  public synchronized void addQuery(String query) {
    queries.add(query);
  }
  public synchronized void removeUser(String user) {
    users.remove(user);
  }
  public synchronized void removeQuery(String query) {
    queries.remove(query);
  }
}
```

# ServerStatus Using Two Locks To Split Locking

```
@ThreadSafe
public class ServerStatus {
  @GuardedBy("users")
  private final Set<String> users = new TreeSet<>();
  @GuardedBy("queries")
  private final Set<String> queries = new TreeSet<>();

  public void addUser(String user) {
    synchronized(users) { users.add(user); }
  }
  public void addQuery(String query) {
    synchronized(queries) { queries.add(query); }
  }
  public void removeUser(String user) {
    synchronized(users) { users.remove(user); }
  }
  public void removeQuery(String query) {
    synchronized(queries) { queries.remove(query); }
  }
}
```

# CopyOnWriteArraySet Can Help To Avoid Locking

- We might also be able to use a thread-safe collection like CopyOnWrite if the queries exceed the modifications

```
@ThreadSafe
public class ServerStatus {
  private final Set<String> users =
    new CopyOnWriteArraySet<>();
  private final Set<String> queries =
    new CopyOnWriteArraySet<>();

  public void addUser(String user) {
    users.add(user);
  }
  public void addQuery(String query) {
    queries.add(query);
  }
  public void removeUser(String user) {
    users.remove(user);
  }
  public void removeQuery(String query) {
    queries.remove(query);
  }
}
```

# Lock Striping

- **We can decrease the probability of contention by splitting our data structures into many pieces**
- **ConcurrentHashMap contains an array of sub-maps**
  - **The concurrency level constructor parameter specifies how many segments we want to have inside the map**
    - **Should be the number of threads that need concurrent access**
    - **Concurrency level increases memory usage.  For an empty map:**

| **Concurrency Level** | **Bytes** |
|---|---|
| 2 | 480 |
| 16 (default) | 2272 |
| 256 | 34912 |

  - **Note that ConcurrentHashMap in Java 8 will probably work with a tree structure of segments**

# Avoiding Hot Fields

- **Even a small portion of serial code will stop scalability**
- **For example, ConcurrentLinkedQueue does not maintain the number of elements inside**
  - **Doing so would introduce a "hot" field**
  - **We would not be able to add and remove elements at the same time**
  - **Instead, every time we ask for size() it counts the elements**
  - **It is optimized for the most common cases: add() and remove()**

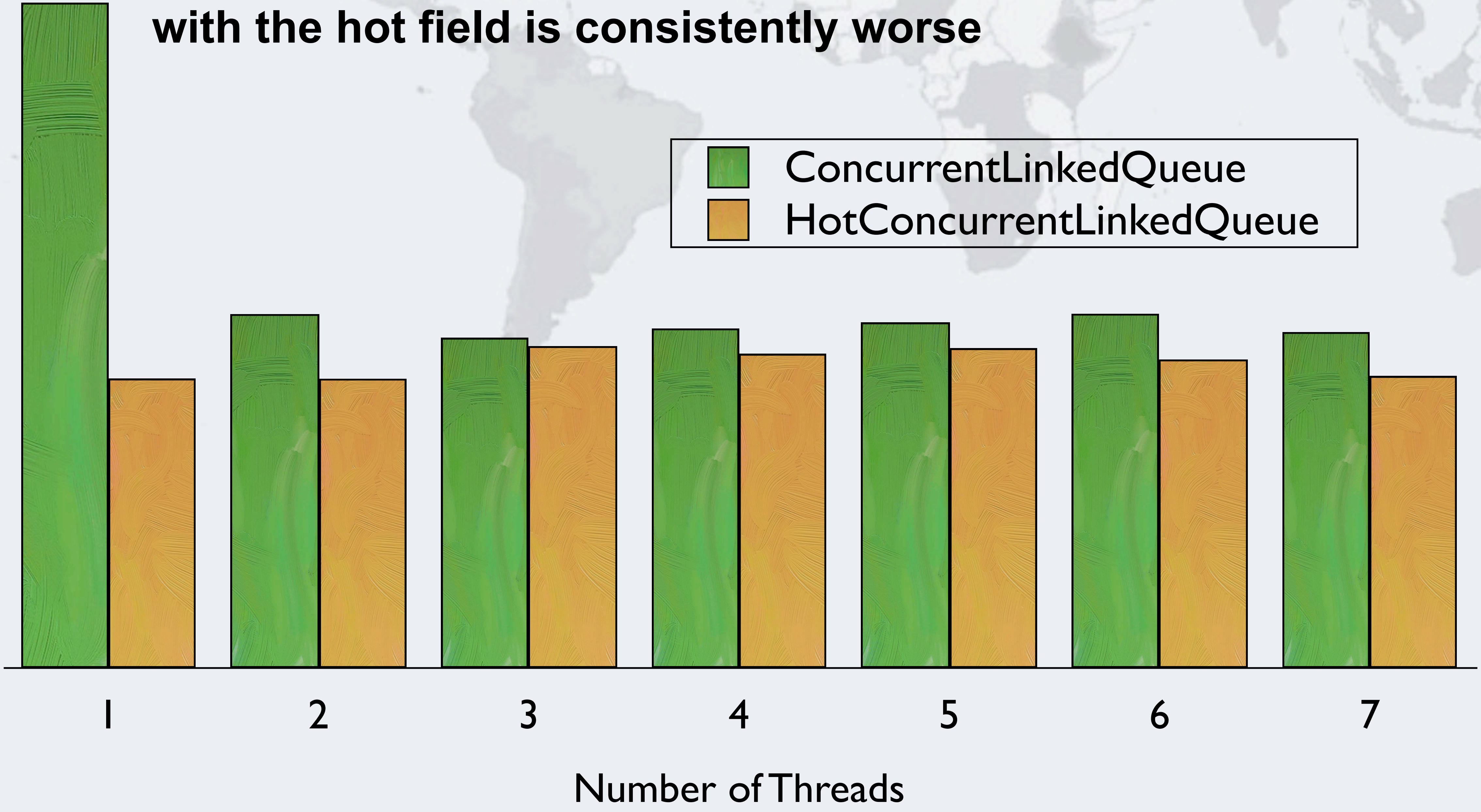
# ConcurrentLinkedQueue With *Hot Field*

- **Our HotConcurrentLinkedQueue introduces a *hot field* that caches the number of elements**

```
public class HotConcurrentLinkedQueue<E>
    extends ConcurrentLinkedQueue<E> {
  private final AtomicInteger elements = new AtomicInteger();
  public boolean offer(E e) {
    boolean success = super.offer(e);
    if (success) elements.incrementAndGet();
    return success;
  }
  public E poll() {
    E e = super.poll();
    if (e != null) elements.decrementAndGet();
    return e;
  }
  public int size() {
    return elements.get();
  }
}
```

# Performance Of ConcurrentLinkedQueues

- **Throughput of the two queues on an 8-core system**
  - **Note that throughput is terrible for multi-core access, but the queue with the hot field is consistently worse**



# Alternatives To Exclusive Locks

- **We can use more scalable alternatives to exclusive locks**
  - **ReadWriteLock allows several threads to read at the same time but only for one to write**
  - **Some of the concurrent collections allow better scalability**
    - **They typically use a combination of volatile and compare-and-set**
  - **Immutable objects reduce the need for locking**
  - **Atomic fields provide volatile access and compare-and-set**
- **Contended fields based on compare-and-set can have worse performance due to too many retries**

# Unix "vmstat"

- **Busy system**

```
procs  -----------memory---------- -swap- --io-- --system-- ----cpu----
 r  b  swpd    free   buff   cache  si so bi  bo   in    cs us sy id wa
 3  0     0 2666092 223300 4388744   0  0  0  68 1506 56459 24   2 74   0
 3  0     0 2644168 223300 4388744   0  0  0   8 1298 61687 31   2 68   0
 3  0     0 2643668 223300 4388744   0  0  0   0 1296 60977 25   1 73   0
 4  0     0 2644064 223300 4388744   0  0  0  12 1311 59997 27   2 71   0
 2  0     0 2643660 223300 4388748   0  0  0   8 1423 68424 25   2 73   0
 4  0     0 2643876 223300 4388748   0  0  0   0 1555 65415 26   2 72   0
 3  0     0 2620896 223308 4388748   0  0  0 132 1349 56320 31   2 67   0
```

- **Quiet system**

```
procs -----------memory---------- -swap- --io-- --system-- ----cpu-----
 r  b swpd    free   buff  cache   si so bi  bo   in    cs us sy  id wa
 0  0    0 2661188 223524 4388964    0  0  0   0  228   212  0  0 100  0
 0  0    0 2660800 223524 4388968    0  0  0   0  135   141  0  0 100  0
 0  0    0 2660676 223524 4388968    0  0  0   0   83    83  0  0 100  0
 0  0    0 2660676 223524 4388968    0  0  0   0  103    91  0  0 100  0
 0  0    0 2660676 223524 4388968    0  0  0   0  170   157  0  0 100  0
 0  0    0 2660676 223524 4388968    0  0  0   0  111   112  0  0 100  0
```

# Unix "vmstat"

- **The context switching (system/cs) is very large, telling us that threads are not using their time quantum**

```
procs  -----------memory---------- -swap- --io-- --system-- ----cpu----
 r  b  swpd    free   buff   cache  si  so bi   bo    in     cs us  sy id  wa
 3  0     0 2666092 223300 4388744   0   0  0   68  1506  56459 24   2 74   0
 3  0     0 2644168 223300 4388744   0   0  0    8  1298  61687 31   2 68   0
 3  0     0 2643668 223300 4388744   0   0  0    0  1296  60977 25   1 73   0
 4  0     0 2644064 223300 4388744   0   0  0   12  1311  59997 27   2 71   0
 2  0     0 2643660 223300 4388748   0   0  0    8  1423  68424 25   2 73   0
 4  0     0 2643876 223300 4388748   0   0  0    0  1555  65415 26   2 72   0
 3  0     0 2620896 223308 4388748   0   0  0  132  1349  56320 31   2 67   0
```

# Why Might The CPUs Not Be Fully Loaded?

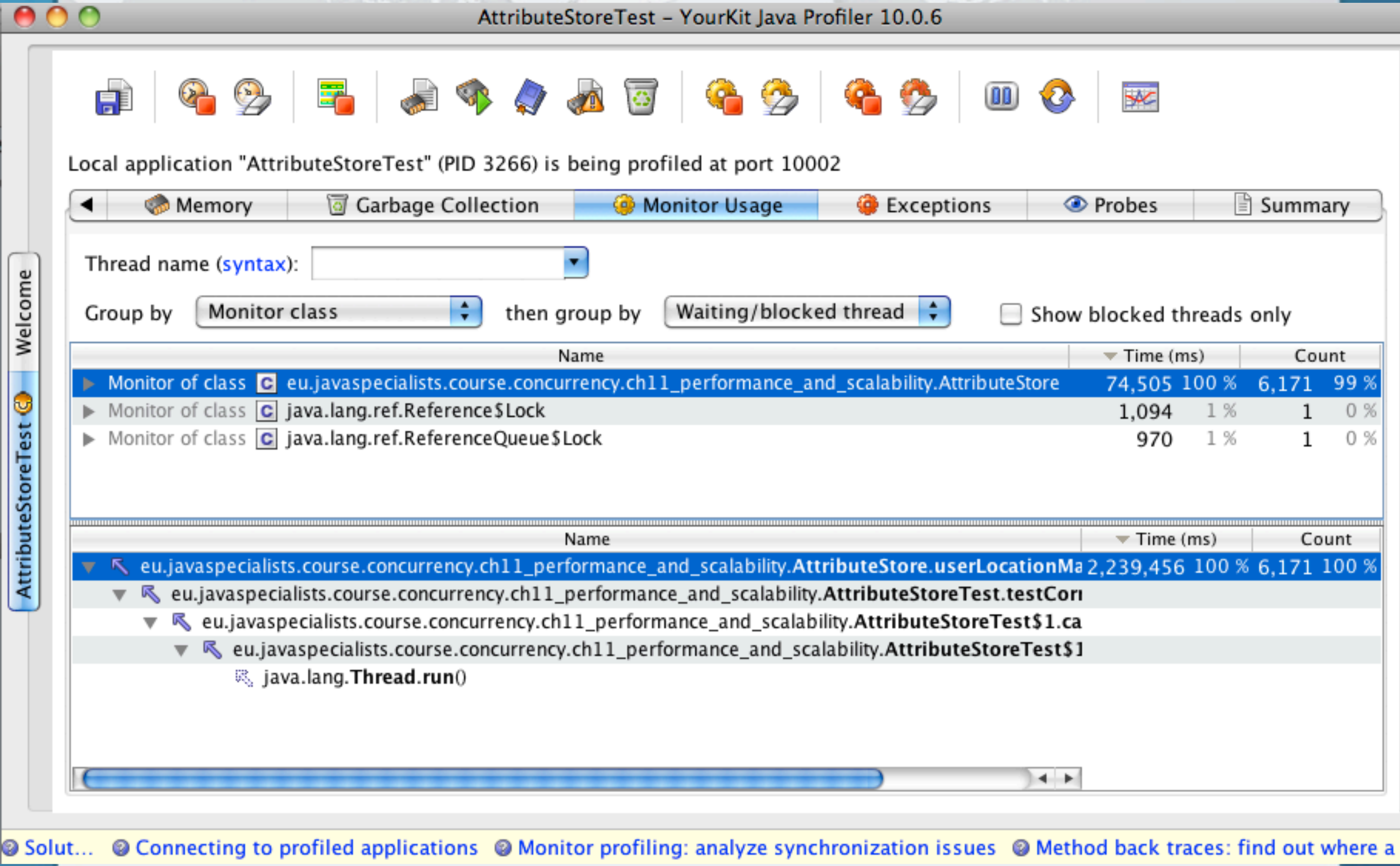
- There are several reasons the CPUs might not get hot
  - Insufficient load
    - The test data set might be too small
    - Our test script might not be adequately loading the system
    - Our test environment might not be powerful enough
  - I/O bound
    - If the application is disk-bound you will see a lot of disk io
    - Windows: perfmon or taskmgr (with the correct columns selected)
    - Unix: iostat or vmstat
  - Externally bound
    - We might be waiting for the database or a web service
    - Use a sampling profiler to see what our threads are waiting for
  - Lock contention - more next slide

# How To Find "Hot Locks"

- **Profiling**
  - **A profiling tool like YourKit shows the most contended locks**
- **Thread dumps**
  - **A cheap way of finding "hot locks" is to take several thread dumps**
  - **A heavily contended lock will usually show up several times**

```
"pool-9-thread-2" prio=10 runnable
   java.lang.Thread.State: RUNNABLE
        at SynchronizedOuter.someMethod
        - locked <0x00000007555c3de8>
        at SynchronizedInnerOuterTest$2.callMethod
        at SynchronizedInnerOuterTest$2.run

"pool-9-thread-1" prio=10 waiting for monitor entry
   java.lang.Thread.State: BLOCKED
        at SynchronizedOuter.someMethod
        - locked <0x00000007555c3de8>
        at SynchronizedInnerOuterTest$2.callMethod
        at SynchronizedInnerOuterTest$2.run
```

# Multi-threaded Performance and Scalability



AttributeStoreTest – YourKit Java Profiler 10.0.6

Local application "AttributeStoreTest" (PID 3266) is being profiled at port 10002

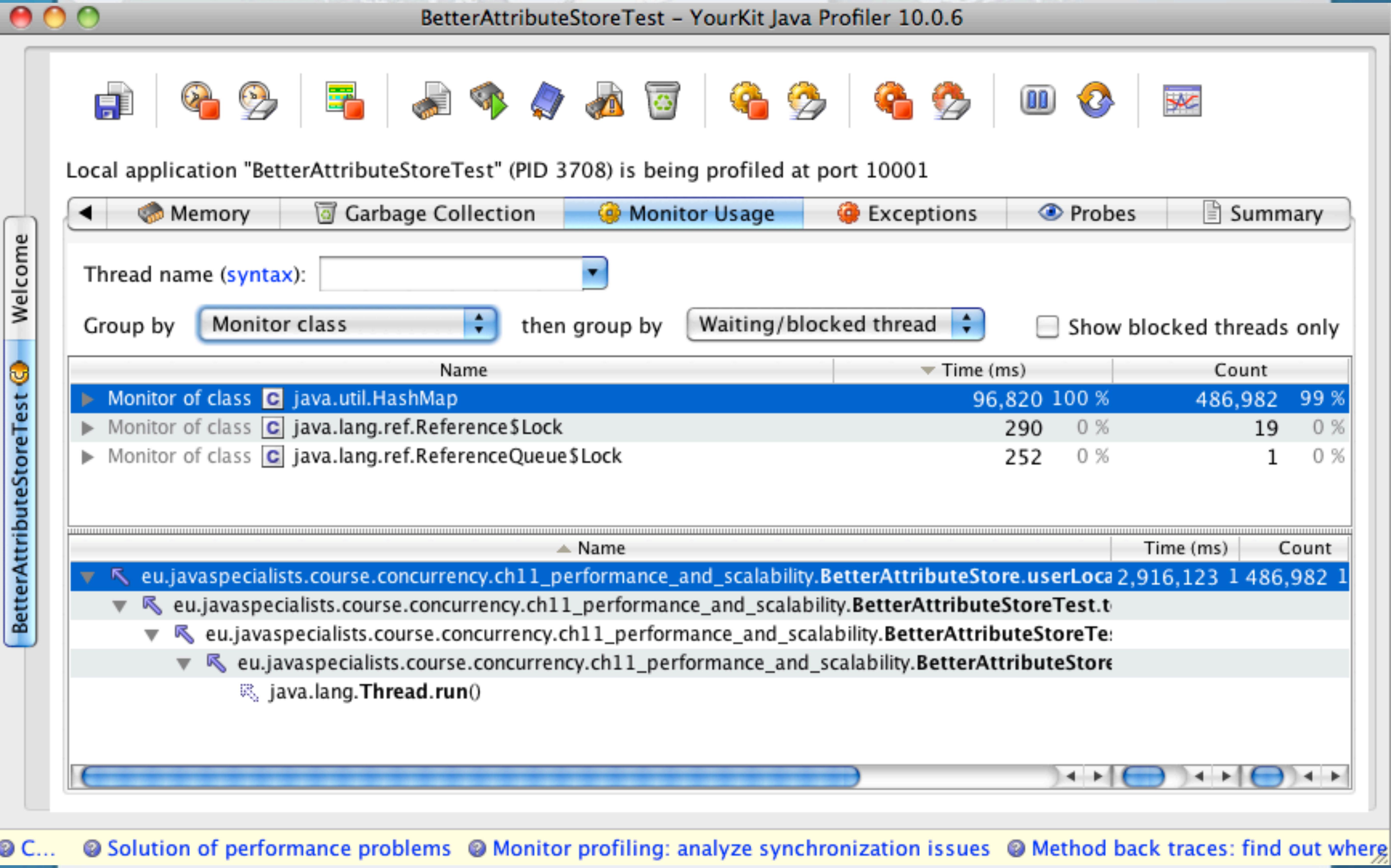
Thread name (syntax):

Group by: Monitor class; then group by: Waiting/blocked thread; □ Show blocked threads only

| Name | Time (ms) | | Count | |
|---|---|---|---|---|
| Monitor of class eu.javaspecialists.course.concurrency.ch11_performance_and_scalability.AttributeStore | 74,505 | 100 % | 6,171 | 99 % |
| Monitor of class java.lang.ref.Reference$Lock | 1,094 | 1 % | 1 | 0 % |
| Monitor of class java.lang.ref.ReferenceQueue$Lock | 970 | 1 % | 1 | 0 % |

| Name | Time (ms) | | Count | |
|---|---|---|---|---|
| eu.javaspecialists.course.concurrency.ch11_performance_and_scalability.AttributeStore.userLocationMa | 2,239,456 | 100 % | 6,171 | 100 % |
| eu.javaspecialists.course.concurrency.ch11_performance_and_scalability.AttributeStoreTest.testCon | | | | |
| eu.javaspecialists.course.concurrency.ch11_performance_and_scalability.AttributeStoreTest$1.ca | | | | |
| eu.javaspecialists.course.concurrency.ch11_performance_and_scalability.AttributeStoreTest$1 | | | | |
| java.lang.Thread.run() | | | | |

Solut… Connecting to profiled applications Monitor profiling: analyze synchronization issues Method back traces: find out where a



BetterAttributeStoreTest – YourKit Java Profiler 10.0.6

Local application "BetterAttributeStoreTest" (PID 3708) is being profiled at port 10001

Thread name (syntax):

Group by: Monitor class then group by: Waiting/blocked thread ☐ Show blocked threads only

| Name | Time (ms) | | Count | |
|---|---|---|---|---|
| ▶ Monitor of class C java.util.HashMap | 96,820 | 100 % | 486,982 | 99 % |
| ▶ Monitor of class C java.lang.ref.Reference$Lock | 290 | 0 % | 19 | 0 % |
| ▶ Monitor of class C java.lang.ref.ReferenceQueue$Lock | 252 | 0 % | 1 | 0 % |

| Name | Time (ms) | Count |
|---|---|---|
| ▼ eu.javaspecialists.course.concurrency.ch11_performance_and_scalability.BetterAttributeStore.userLoca | 2,916,123 1 | 486,982 1 |
| ▼ eu.javaspecialists.course.concurrency.ch11_performance_and_scalability.BetterAttributeStoreTest.t | | |
| ▼ eu.javaspecialists.course.concurrency.ch11_performance_and_scalability.BetterAttributeStoreTe: | | |
| ▼ eu.javaspecialists.course.concurrency.ch11_performance_and_scalability.BetterAttributeStore | | |
| java.lang.Thread.run() | | |

C… Solution of performance problems Monitor profiling: analyze synchronization issues Method back traces: find out where

# HotSpot Options For Lock Performance

- **We can control how HotSpot does locking**
  - **-XX:+DoEscapeAnalysis**
    - **Elides locks on objects that cannot escape**
  - **-XX:+EliminateLocks**
    - **Does lock coarsening using roach motel semantics**
  - **-XX:+UseBiasedLocking**
    - **Locks are assumed to be given to a single thread**
      - **This might have to be undone if another thread needs the lock**
    - **Additional flags control how quickly biased locking is applied**
      - **-XX:BiasedLockingStartupDelay= 4000**
      - **-XX:BiasedLockingBulkRebiasThreshold=20**
      - **-XX:BiasedLockingBulkRevokeThreshold=40**
      - **-XX:BiasedLockingDecayTime=25000**

# Hardware Support For Concurrency

## Atomic Variables and Nonblocking Synchronization

Javaspecialists.eu
java training

# 15.2: Hardware Support For Concurrency

- **Exclusive locking is a *pessimistic* technique**
  - **Imagine every time you wanted to go outside to hang up your washing, you put on your alarm, bolt your door and close the security gate**
- **For fine-grained work, it is better to use *optimistic locking***
  - **"Everything is going to be alright"**
  - **We try an operation and if it does not succeed, we try again**
  - **This depends on *collision detection* instead of explicit locking**
- **Modern processors have multi-processing instructions**
  - **Compare And Swap (CAS), Load Linked / Store Conditional**
  - **These are used by Java concurrent structures to improve throughput**

# Compare And Swap (CAS)

- **Most processor architectures use Compare-And-Swap**
  - **For example, Intel, AMD and Sparc**
  - **Others such as Alpha AXP, IBM PowerPC, MIPS and ARM, implement it with a pair of instructions**
    - **load-linked and store-conditional**
- **Compare-and-swap does the following atomically**
  - **We pass in three operands**
    - **Memory location** V, **expected old value** A **and new value** B
  - **CAS updates** V **to the new value** B, **only if** V **contains expected** A
  - **The old value at** V **is returned always**
- **Compare-and-set is similar, but returns true or false**
  - **Unfortunately only compare-and-set is available in Java**

# Managing Conflicts With Compare-and-Swap

- **If two threads try to write at the same time, one will "win" and the other will "lose"**
  - **The winner will see the same result returned as his "new value"**
  - **The loser will see the "new value" of the winner thread**
- **Because there is no locking, there can be no deadlock**
  - **However, in unusual cases, we could get a *livelock***

# CAS Support In The JVM

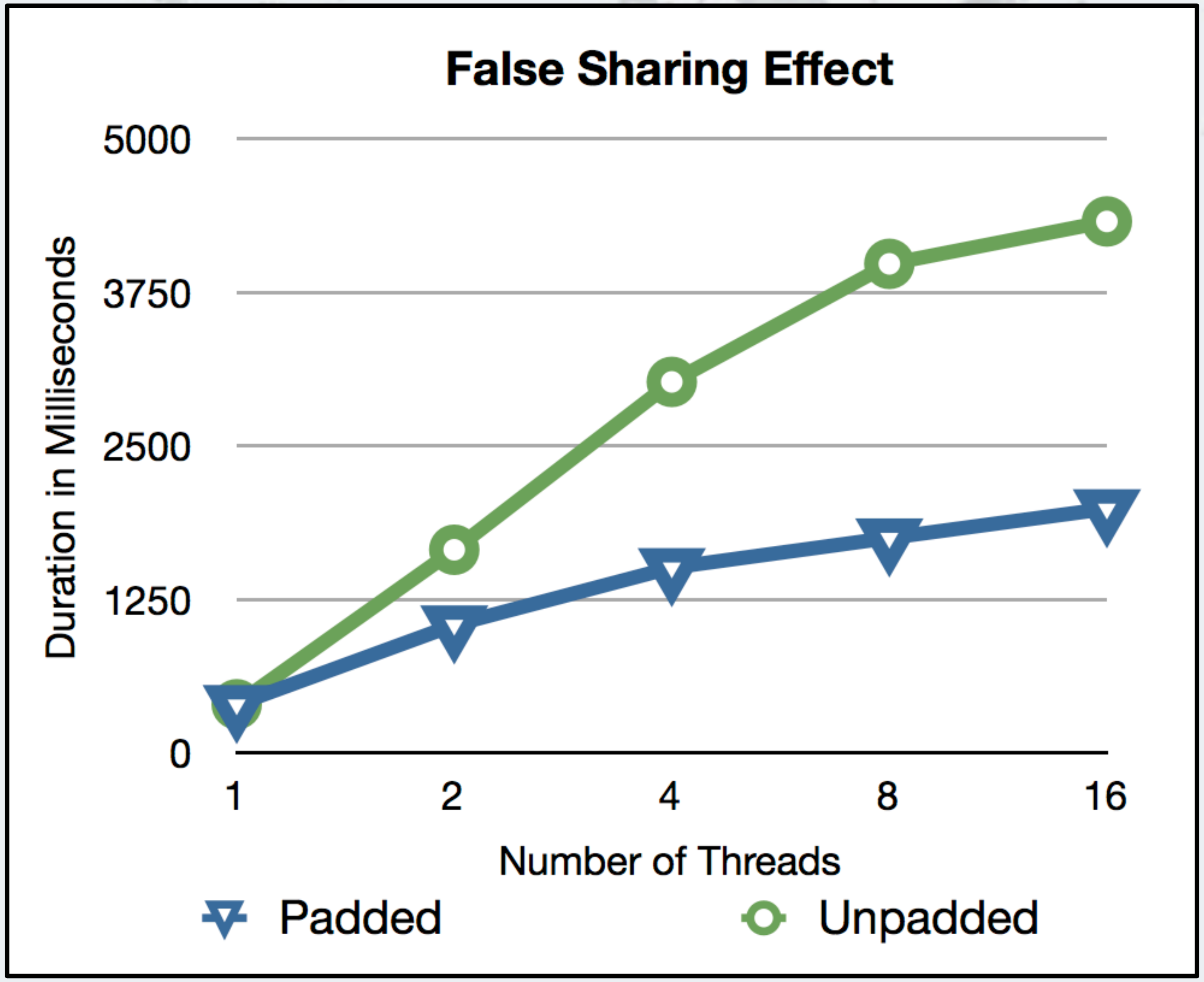
- **Prior to Java 5, we needed to use native code to do CAS**
- **Nowadays, it is available via the atomic classes and via the sun.misc.Unsafe class**
  - **Although sun.misc.Unsafe should be avoided where possible**
- **The CAS methods are compiled as efficiently as possible**
  - **If the hardware supports it, it becomes a single CPU instruction**
    - **Even though it is declared as a native method call, we do not have the call cost overhead of JNI**
  - **If it does not support it, it is typically implemented as a spin lock**
- **The JDK uses this in a number of classes in the java.util.concurrent package**

# Shared Cache Lines

- Our modern architectures consists of several layers of memory caches
- Memory is stored in units called "cache lines"
  - These can be anything from 32 to 256 bytes large
- When a Java object is too small, it might have to share the same cache line with another object
  - This can make volatile field access slower
- Trick by Martin Thompson is to pad the Java object with dummy data, thus forcing it to be in its own cache line
  - http://mechanical-sympathy.blogspot.com/2011/07/false-sharing.html

# Performance Advantage Of Padding

- **Graph shows how long it takes for all the threads to complete writing to volatile fields**



# "Padded Atomic Long"

- **With "padding" field, the object might own the cache line**
  - **This is only important for highly contended fields**
  - **Future compilers might do this automatically**
- **One way of padding is to use a "long array" and then set just an element in the middle of the array with Unsafe**

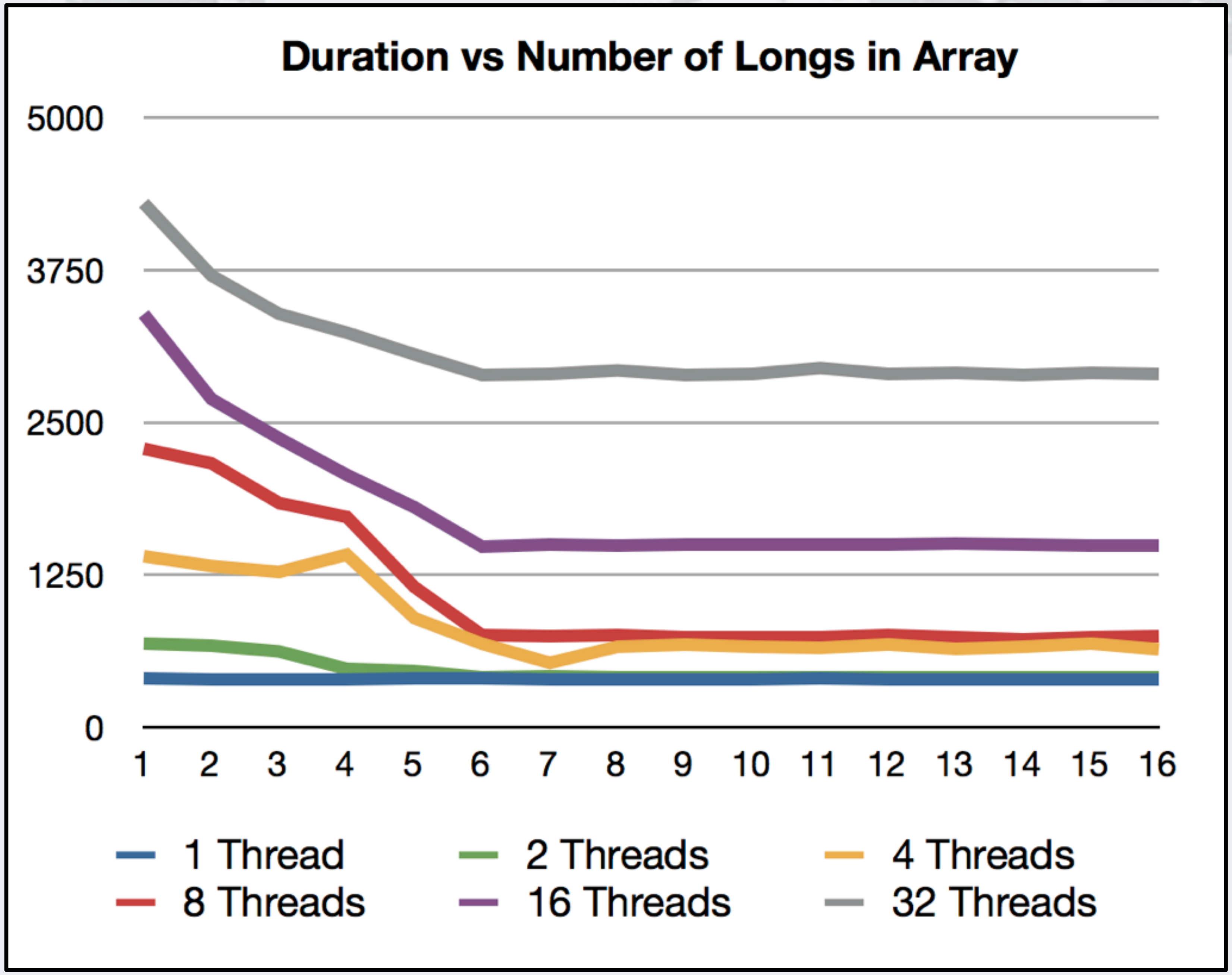
```
private static final Unsafe UNSAFE = Unsafe.getUnsafe();
private static long longArrayIndex =
  UNSAFE.arrayBaseOffset(long[].class);
private static long longArrayIndexScale =
  UNSAFE.arrayIndexScale(long[].class);

private static void set(long[] data, int idx, long value) {
  UNSAFE.putLongVolatile(data,
      longArrayIndex + longArrayIndexScale * idx, value);
}
```

Javaspecialists.eu

15.2: Hardware Support For Concurrency

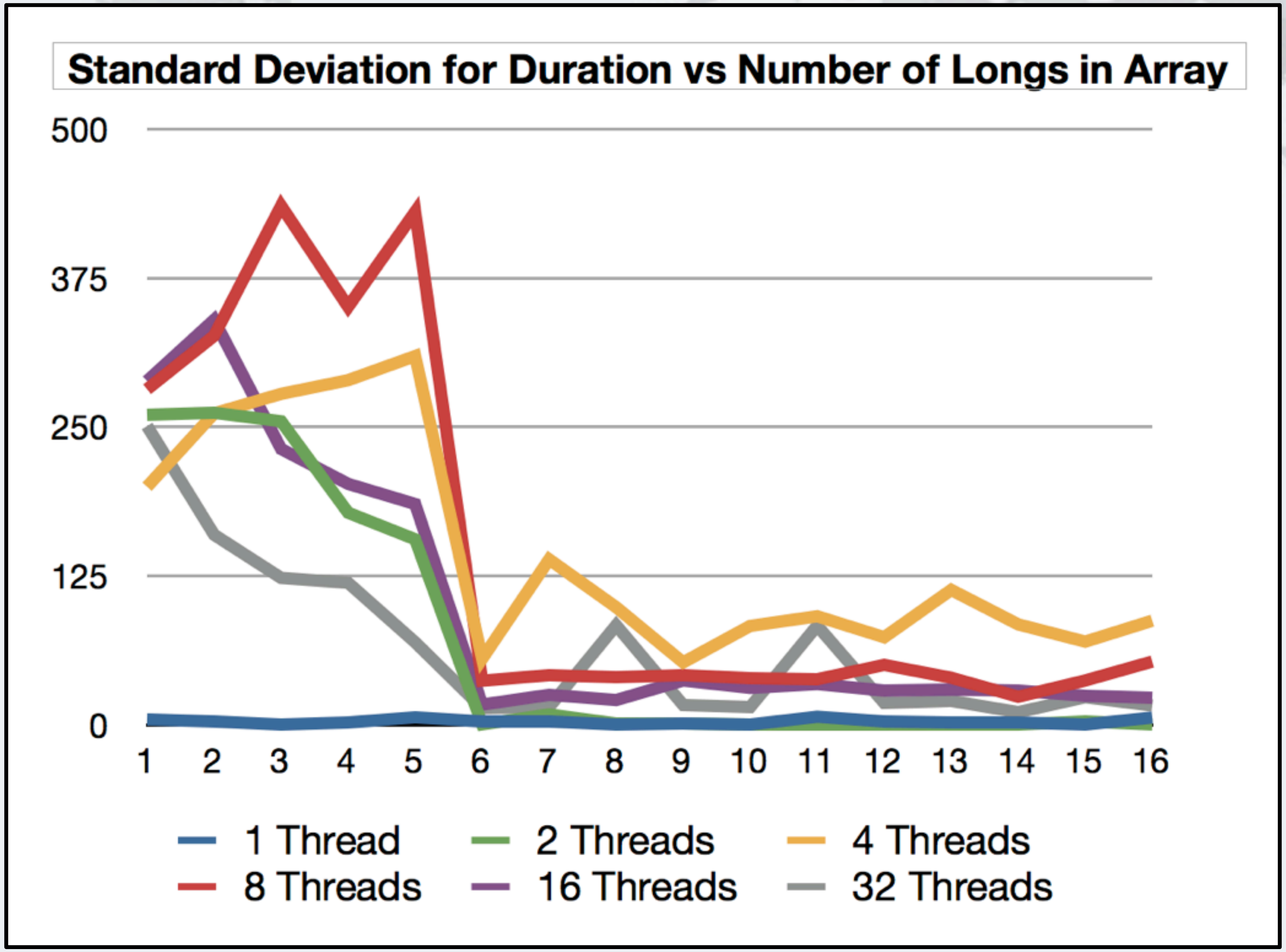
# Duration Of Writing Vs Amount Of Padding

- **This is on an 8-core machine**



# Standard Deviation

- **With more padding, the standard deviation was less**



# Atomic Variable Classes

## Atomic Variables and Nonblocking Synchronization

# 15.3: Atomic Variable Classes

- Atomics are optimistic about clashes
  - They assume there will be no clash (optimism)
  - Compare-and-swap is used to set the field, if possible
  - If there was a clash, we simply try again
- With not too much contention, atomics are blazingly fast
  - But with heavy contention, atomics can sometimes be slower
    - They might clash so much that the "try again" is more than locks
- Useful for writing high-performant concurrent code
  - Especially for fine-grained fields like counters
- Atomic compareAndSet is compiled to a CPU instruction

# Atomics As "Better Volatiles"

- **Atomics store the value internally as a *volatile* field**
  - **We thus have the same visibility semantics**
- **There is little reason to use volatile directly**
  - **Atomic classes also have other advantages in that they solve some read-modify-write race conditions**
    - **Though check-then-act would usually require a loop**

# Types Of Atomic Classes

- The following types have atomics built in
  - AtomicBoolean
  - AtomicInteger
    - Use for int, short, byte and float (use Float.floatToIntBits(float))
  - AtomicLong
    - Use for long and double
  - AtomicReference
- There are also atomic array classes
    - Necessary as you can never make values of an array volatile!
  - AtomicIntegerArray
  - AtomicLongArray
  - AtomicReferenceArray

# How Do Atomics Work?

- **Thread safe without explicit locking**
  - **Tries to update the value repeatedly until success**
  - **Only works if there are no invariants across fields**
- **Note: AtomicInteger.equals() is not overridden**
- **Here is how addAndGet() works**

```
public final int addAndGet(int delta) {
  for (;;) {
    int current = get();
    int next = current + delta;
    if (compareAndSet(current, next))
      return next;
  }
}
```

# Atomic Bank Account

```
public class BankAccount {
  private final AtomicInteger balance =
    new AtomicInteger();

  public BankAccount(int balance) {
    this.balance.set(balance);
  }
  public void deposit(int amount) {
    balance.addAndGet(amount);
  }
  public void withdraw(int amount) {
    deposit(-amount);
  }
  public int getBalance() {
    return balance.intValue();
  }
}
```

# Performance Comparison: Locks Vs Atomics

- In our comparison, we will compare various locking algorithms for generating pseudo random numbers
- The basis is PseudoRandom

```
public class PseudoRandom {
  protected int calculateNext(int prev) {
    prev ^= prev << 6;
    prev ^= prev >>> 21;
    prev ^= (prev << 7);
    return prev;
  }
}
```

# SynchronizedPseudoRandom

- **A random number generator protected with synchronized**

```
@ThreadSafe
public class SynchronizedPseudoRandom extends PseudoRandom {
  private final Object lock = new Object();
  @GuardedBy("lock") private int seed;

  public SynchronizedPseudoRandom(int seed) {
    this.seed = seed;
  }

  public int nextInt(int n) {
    synchronized (lock) {
      int s = seed;
      seed = calculateNext(s);
      int remainder = s % n;
      return remainder > 0 ? remainder : remainder + n;
    }
  }
}
```

15.3: Atomic Variable Classes

# ReentrantLockPseudoRandom

```
@ThreadSafe
public class ReentrantLockPseudoRandom extends PseudoRandom {
  private final Lock lock = new ReentrantLock(false);
  @GuardedBy("lock") private int seed;

  public ReentrantLockPseudoRandom(int seed) {
    this.seed = seed;
  }

  public int nextInt(int n) {
    lock.lock();
    try {
      int s = seed;
      seed = calculateNext(s);
      int remainder = s % n;
      return remainder > 0 ? remainder : remainder + n;
    } finally {
      lock.unlock();
    }
  }
}
```

# AtomicPseudoRandom

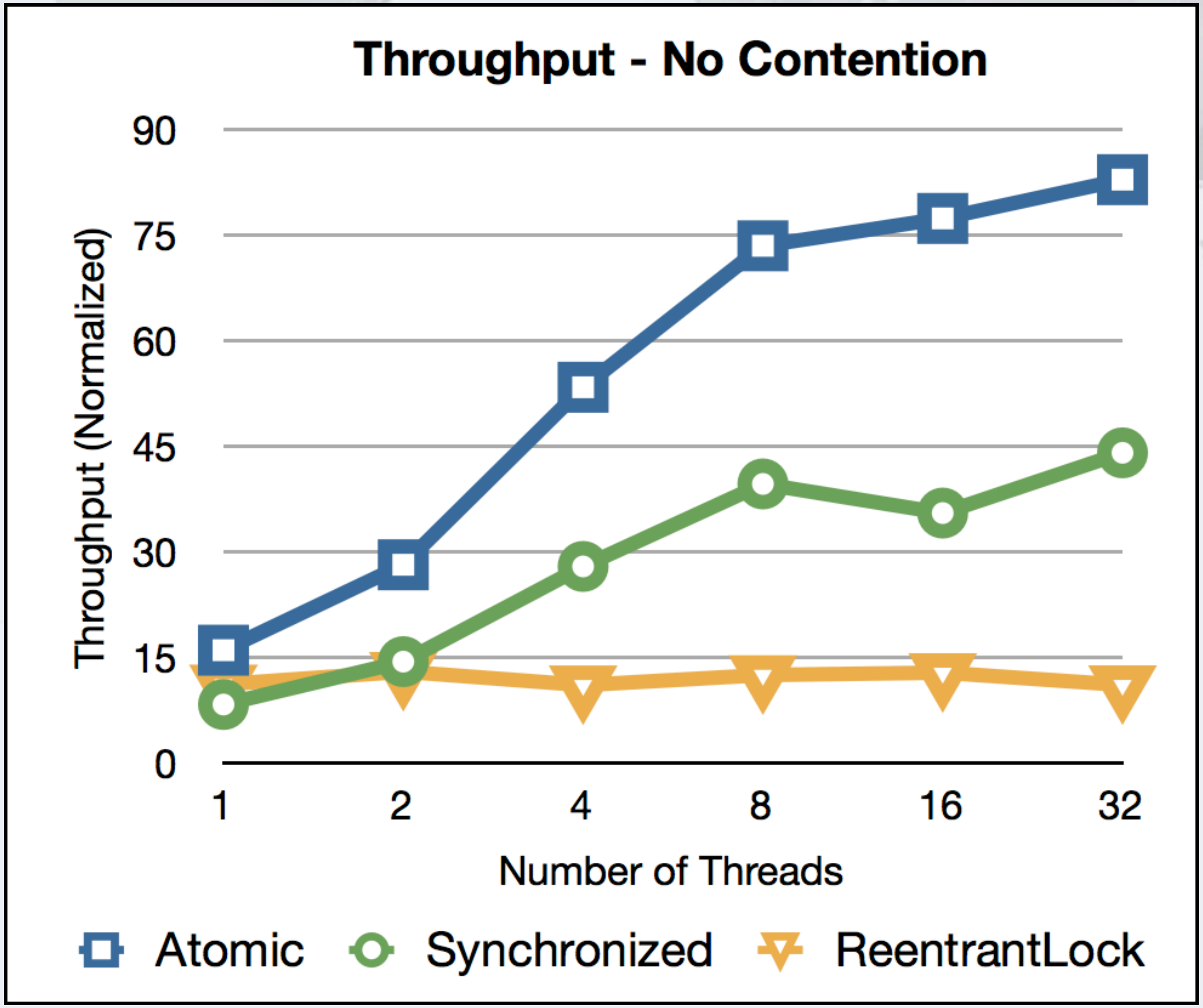
```
@ThreadSafe
public class AtomicPseudoRandom extends PseudoRandom {
  private final AtomicInteger seed;

  public AtomicPseudoRandom(int seed) {
    this.seed = new AtomicInteger(seed);
  }

  public int nextInt(int n) {
    while (true) {
      int currentSeed = seed.get();
      int nextSeed = calculateNext(currentSeed);
      if (seed.compareAndSet(currentSeed, nextSeed)) {
        int remainder = currentSeed % n;
        return (remainder > 0) ? remainder : remainder + n;
      }
    }
  }
}
```

15.3: Atomic Variable Classes

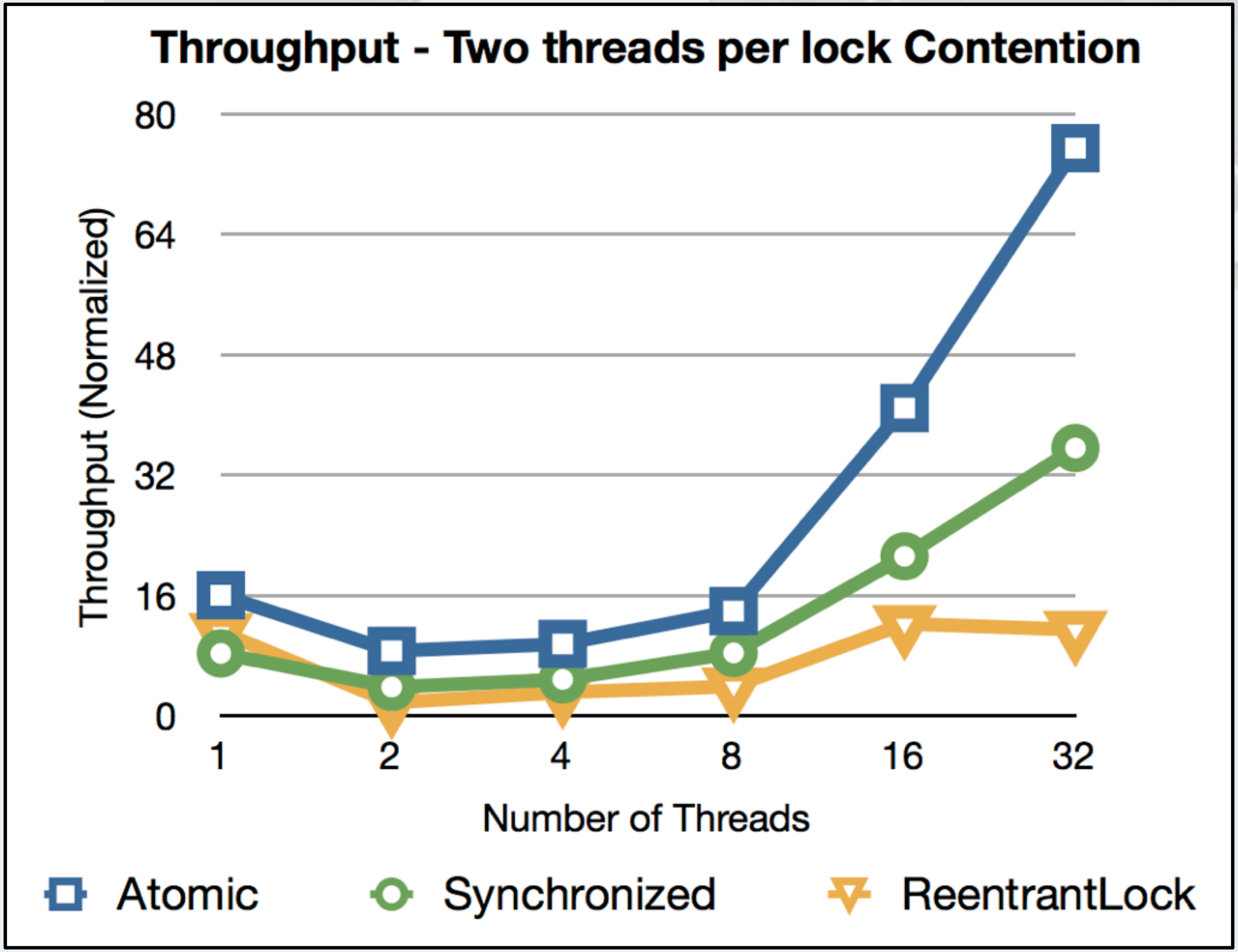
# Explanation Of Experiment

- **Running on 8-core machine with Linux and JDK 1.6.0_29**
- **60 data points for warmup, after that 30 data points**
- **Throughput values were normalized across all the tests**
  - **Thus a value of 80 in one test and 1 in another means that throughput is 80 times faster**
- **We tested 1,2,4,8,16 and 32 threads**
  - **For "no contention", each thread has its own lock**
  - **For "full contention", one lock is shared by all**
  - **We also have tests with some contention**

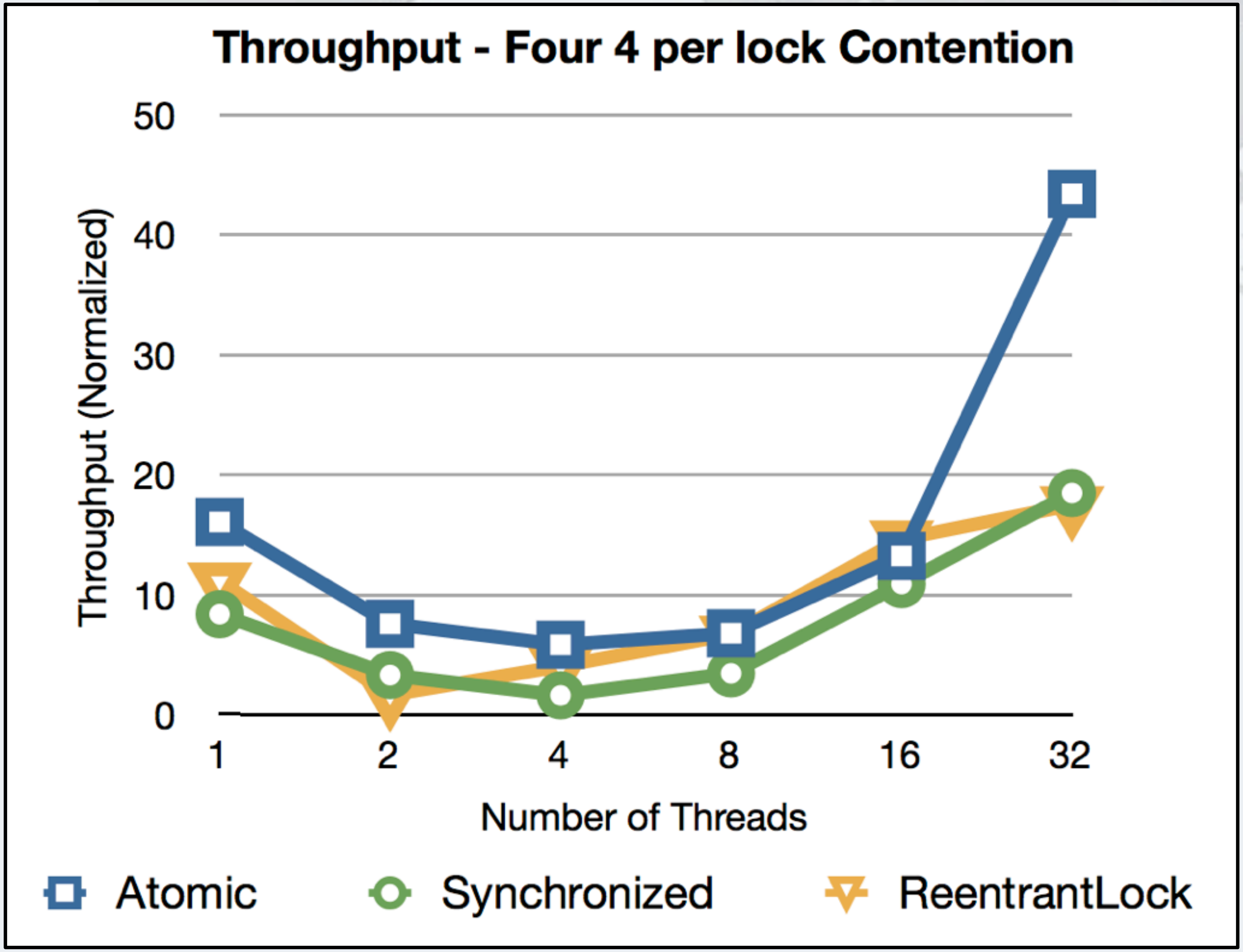
# Throughput With No Contention



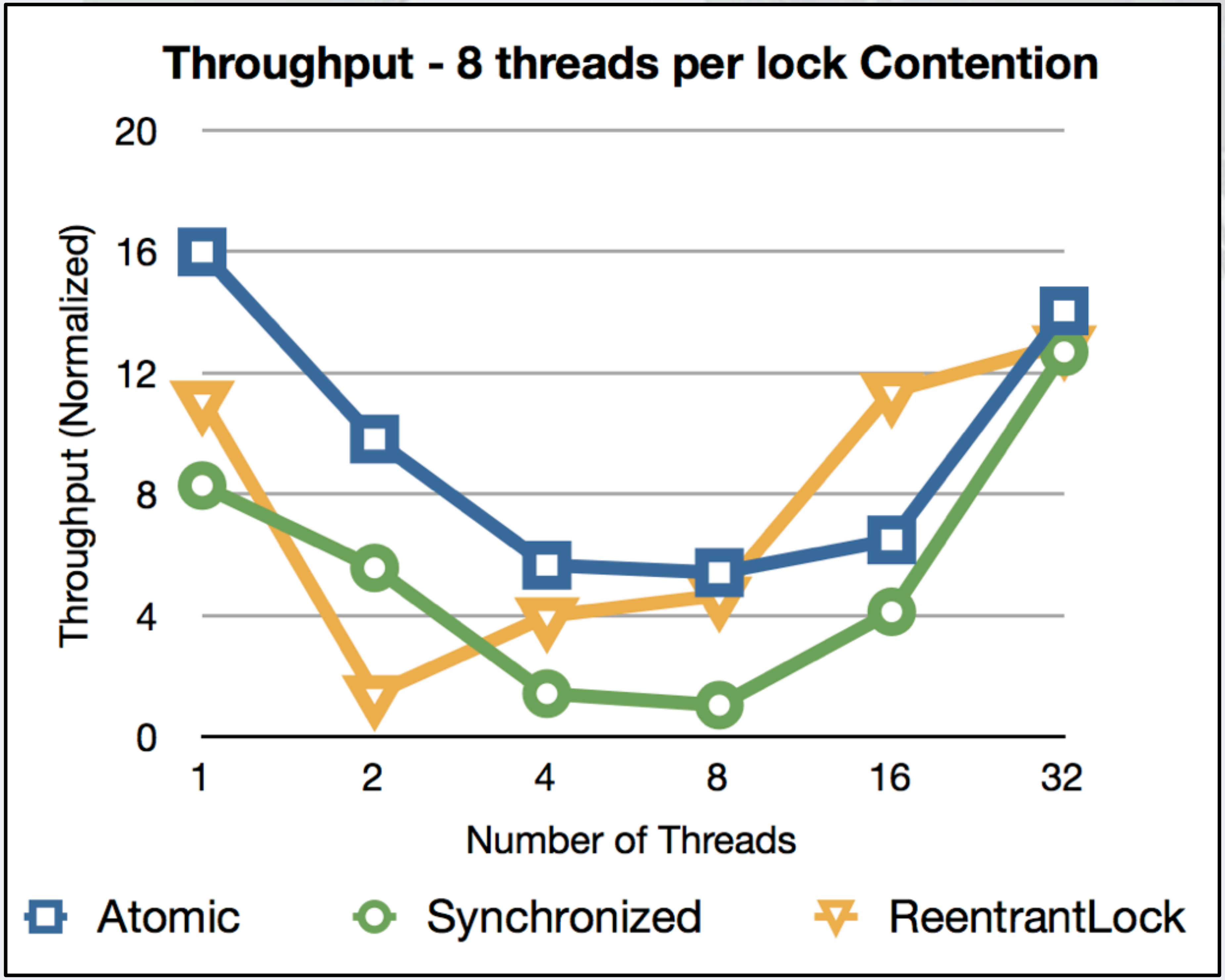
# Throughput With Minor Contention



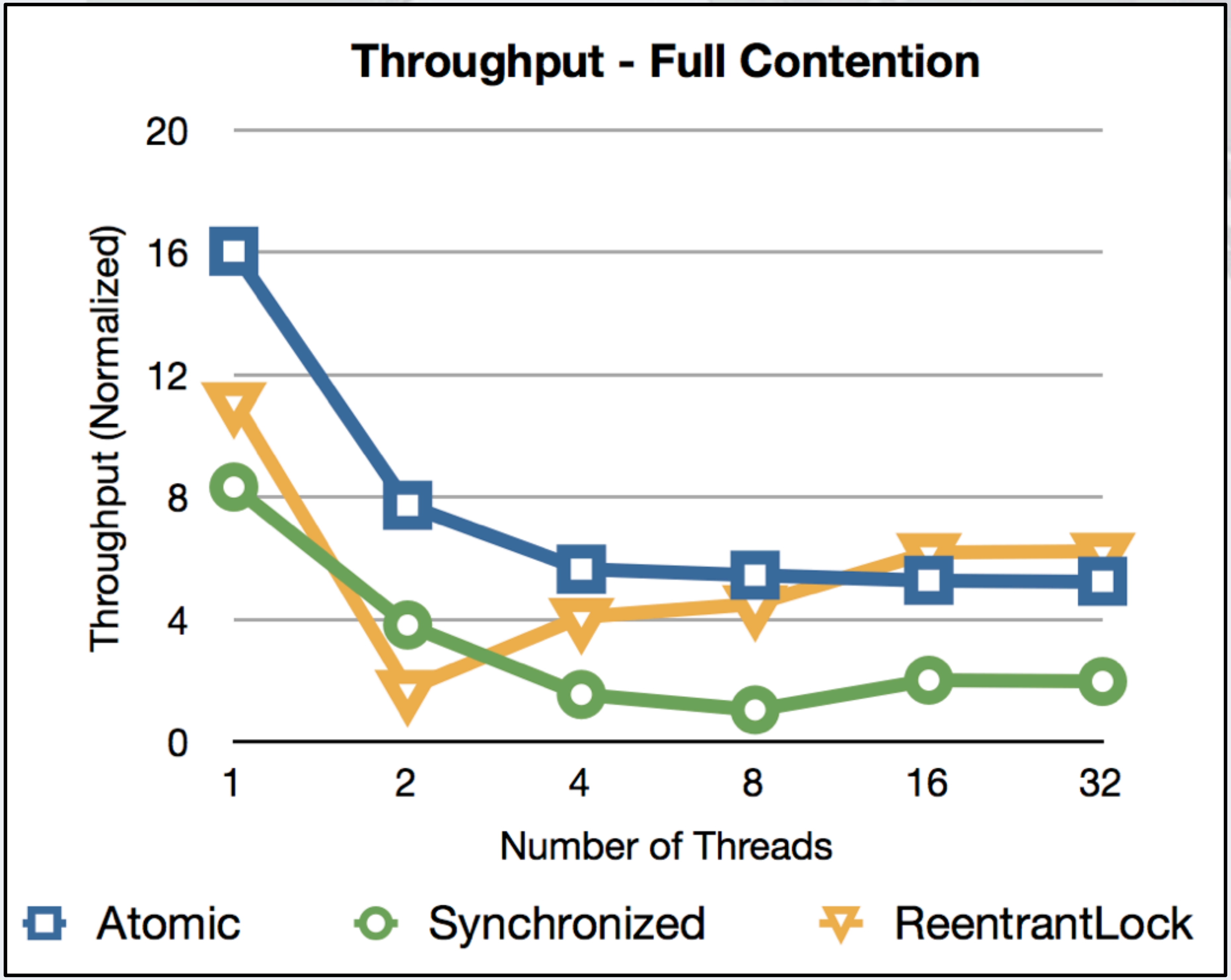
# Throughput With 50% Contention



# Throughput With High Contention

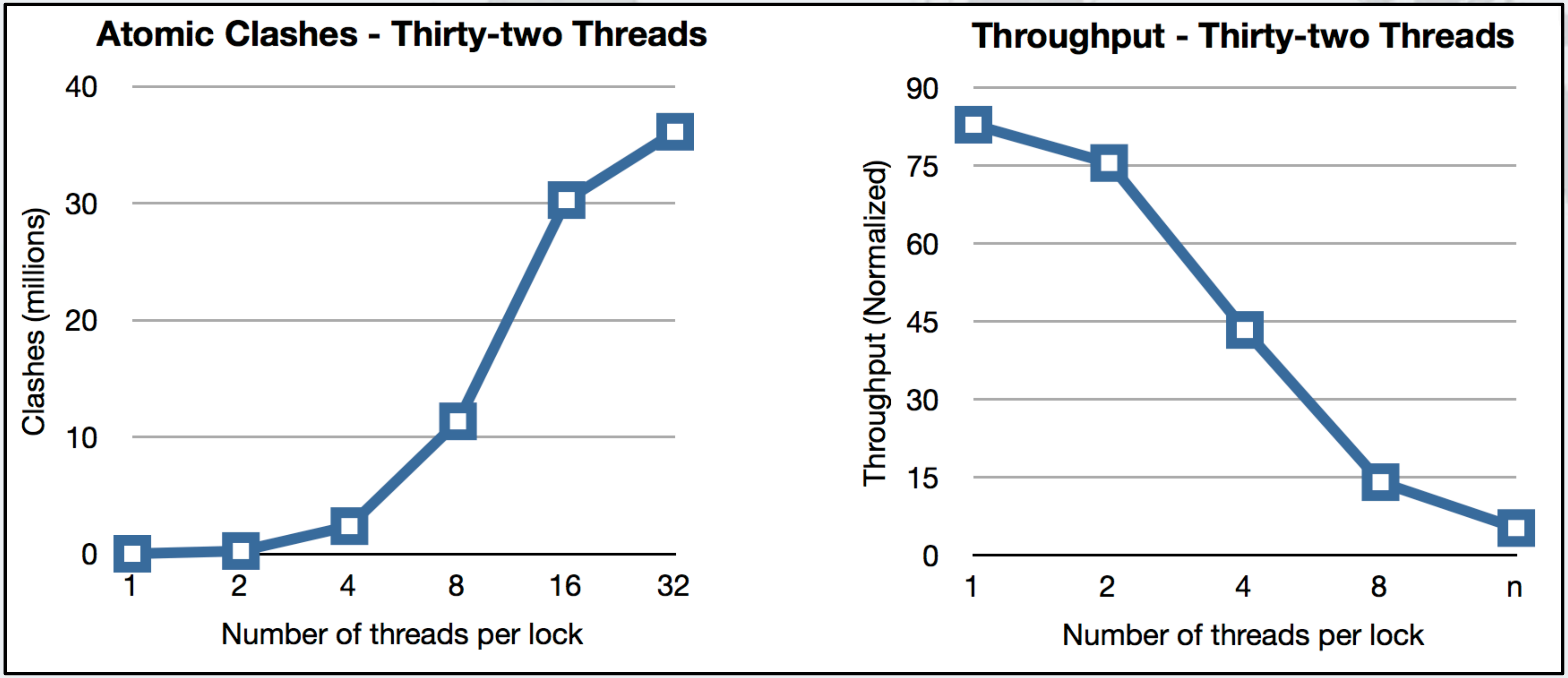


# Throughput With All Threads On One Lock



# Atomic Spin Loops

- **Not surprisingly, as contention increases, number of clashes increase with atomics**
  - **Active threads sharing an atomic should be less than physical cores**



# Nonblocking Algorithms

## Atomic Variables and Nonblocking Synchronization

# 15.4: 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**
- **We use Compare-And-Swap (CAS) to write nonblocking and lock-free algorithms**
- **Nonblocking algorithms cannot deadlock or cause priority inversion**

# A Nonblocking Stack

- **Stack is one of the easiest data structures to write**
- **In this case, we use the AtomicReference to point to top**
  - **We don't have method for telling us the size, just push() and pop()**

```
@ThreadSafe
public class ConcurrentStack<E> {
  private final AtomicReference<Node<E>> top
      = new AtomicReference<>();

  @Immutable
  private static class Node<E> {
    public final E item;
    public final Node<E> next;
    public Node(E item, Node<E> next) {
      this.item = item;
      this.next = next;
    }
  }
```

# Stack "push()" And "pop()" Methods

```
  public void push(E item) {
    Node<E> oldHead, newHead;
    do {
      oldHead = top.get();
      newHead = new Node<>(item, oldHead);
    } while (!top.compareAndSet(oldHead, newHead));
  }

  public E pop() {
    Node<E> oldHead, newHead;
    do {
      oldHead = top.get();
      if (oldHead == null)
        return null;
      newHead = oldHead.next;
    } while (!top.compareAndSet(oldHead, newHead));
    return oldHead.item;
  }
}
```

# Speculative Work

- In algorithms using CAS, we usually need to be optimistic and do work that might need to be redone
  - In our ConcurrentStack, we construct a new Node in the push() method, hoping that no one else beat us to it

```
public void push(E item) {
  Node<E> oldHead, newHead;
  do {
    oldHead = top.get();
    newHead = new Node<>(item, oldHead);
  } while (!top.compareAndSet(oldHead, newHead));
}
```

  - This is a characteristic of non-blocking algorithms

# Highly Scalable HashTable By Cliff Click

- For more than 50 cores, use the non-blocking collections
  - http://sourceforge.net/projects/high-scale-lib/
- Replacement for java.util and java.util.concurrent classes
- State machine based solution for solving concurrency
  - More complicated than simply locking
  - But scales to a thousand cores

# Atomic Field Updaters

- **A field can be updated atomically without using an Atomic class to hold the value**
  - **This saves having an extra object to hold the balance field**
- **We can get an AtomicFieldUpdater for the class' field**
- **Note:**
  - **Up to Java 6, ConcurrentLinkedQueue used AtomicReferenceFieldUpdater to change the fields**
  - **Since Java 7, they use sun.misc.Unsafe and directly access fields using CAS**
    - **Reason is that the atomic field updaters do a lot of checking on every call**

# Bank Account Using Field Updater

```
public class BankAccount {
  private volatile int balance;

  private static final AtomicIntegerFieldUpdater<BankAccount>
      balanceUpdater = AtomicIntegerFieldUpdater.newUpdater
          (BankAccount.class, "balance");

  public BankAccountWithFieldUpdater(int balance) {
    this.balance = balance;
  }
  public void deposit(int amount) {
    balanceUpdater.addAndGet(this, amount);
  }
  public void withdraw(int amount) {
    deposit(-amount);
  }
  public int getBalance() {
    return balance;
  }
}
```

# Class "sun.misc.Unsafe"

- In the OpenJDK, the sun.misc.Unsafe class is used to access memory directly
  - Calls to some of the methods such as compareAndSwap(), putIntVolatile() and others, can be compiled to a single CPU instruction
- Incorrect use can cause segment violations
  - JVM crashes spectacularly
- "sun.misc.Unsafe" is dangerous - use with caution!
  - Portability is not such a big issue though, sun.misc.Unsafe is supported correctly by all production JVM implementors
- In Java 8 it might be disabled for non-system classes

# Bank Account Using "sun.misc.Unsafe"

- **We start by finding the offset of the "balance" field**
  - **"UnsafeProvider" is left as an *exercise to the reader***

```
public class BankAccount {
  private static final sun.misc.Unsafe UNSAFE;
  private static final long balanceOffset;

  static {
    try {
      UNSAFE = Unsafe.getUnsafe();
      balanceOffset = UNSAFE.objectFieldOffset(
        BankAccount.class.getDeclaredField("balance"));
    } catch (Exception e) {
      throw new Error(e);
    }
  }

  private volatile int balance;
```

# Rest Of BankAccount With Unsafe

```
  public BankAccount(int balance) {
    this.balance = balance;
  }
  public void deposit(int amount) {
    while (true) {
      int current = UNSAFE.getInt(this, balanceOffset);
      int next = current + amount;
      if (UNSAFE.compareAndSwapInt(
        this, balanceOffset, current, next))
          return;
    }
  }
  public void withdraw(int amount) {
    deposit(-amount);
  }
  public int getBalance() {
    return balance;
  }
}
```

# Conclusion

## Performance and Scalability

Javaspecialists.eu
java training

# Conclusion

- Traditional optimizations try to speed up a single method
  - Change complexity or cache previous results
  - – In multi-threading, this can introduce bottlenecks and "hot fields"
  - – Algorithms might also be more difficult to parallelize
- Measure your performance
  - – Only optimize contended locks
  - – Use good tooling to discover the hottest locks
- Narrow your lock scope ("Get in, Get out")
  - – Do not write 2000 line long synchronized methods
  - – Little and Amdahl will love you for it
- Learn how concurrency works in Java

# Multi-threaded Performance And Scalability

Questions?

heinz@javaspecialists.eu

http://www.javaspecialists.eu/talks/wjax12/kabutz.pdf

Javaspecialists.eu
java training