MOVING JAVA FORWARD

(The Art of) (Java) Benchmarking

Aleksey Shipilev
Java Platform Performance, Oracle

Agenda

Introduction

Basic Theory

Java Benchmarking

Tools

Introduction

- Computer Science → Software Engineering
  - Way to construct software to meet functional requirements
  - Usually don't care about HW and data specifics
  - Abstract and composable, “formal science”

- Software Performance Engineering
  - “Real world strikes back!”
  - Researching complex interactions between hardware, software, and data
    - Unable to *predict* performance, can only *measure* it
  - Based on empirical evidence, i.e. “natural science”
    - Characterization → Hypotheses → Predictions → Experiments
    - Benchmarks are ultimate tools for conducting experiments

Benchmarks

- What's the benchmark anyway?
  - Def.: “Benchmark is the application used to measure performance”
    - Take an application, get two timestamps, now you have a benchmark
  - Every benchmark run is computational experiment

- Correct benchmark is tricky to get right
  - It has a metric: each run yields metric value (ops/sec, sec, ops/W)
  - Reliable: reproducible results, reacts correctly to changing environment
  - Objective: Tests the Right Thing™
  - Easy to run
  - Self-checking

Benchmark Taxonomy

- Real-World Applications
  - Launch manually
  - Measure with stopwatch, voltmeter, oscillograph
- Automated Scenarios of Real-World Applications
  - Record the trace through the application, feed the trace to the robot
  - Let other robots to measure time, power, traffic, etc.
- Synthetic (Macro-)Benchmarks
  - Develop golden application along with all the testing infrastructure
  - Hard to develop, lots of code, huge investment
- Microbenchmarks
  - Focus on one specific part of the code, throwing everything else out
  - Easy to develop, (everyone tends to think of it as) small investment

How Could This Go Wrong?

- Good design of experiment requires one to know the field
  - Anyone to perform heart surgery without 10+ years surgeon's experience?
- Running the benchmark is NOT the final step
  - Asking the right question is the first step
  - Interpreting the results is hard
  - Detecting Type III Errors is hard
  - Convincing yourself to fix something and run again is the hardest part
- Being effective at benchmarking means having less retries
  - Translation: don't do silly things → learn what NOT to do
  - Catch mistakes early and eagerly
  - Too many retries? Learn to give up.

You're Not Alone

- Transaction Processing Performance Council
  - Design documents for standard benchmarks
  - Resolve a lot of benchmark design problems
- Standard Performance Evaluation Corporation
  - Industry-standard performance benchmarks
    - SPECcpu, SPECvirt_sc, SPECjbb, SPECjvm, SPECjEnterprise
    - Ready-to-run implementations:
      - Download/Buy → Deploy → Run → ??? → PROFIT!
  - Updated to match newer trends in hardware and software
- Performance teams
- Peer reviews

Agenda

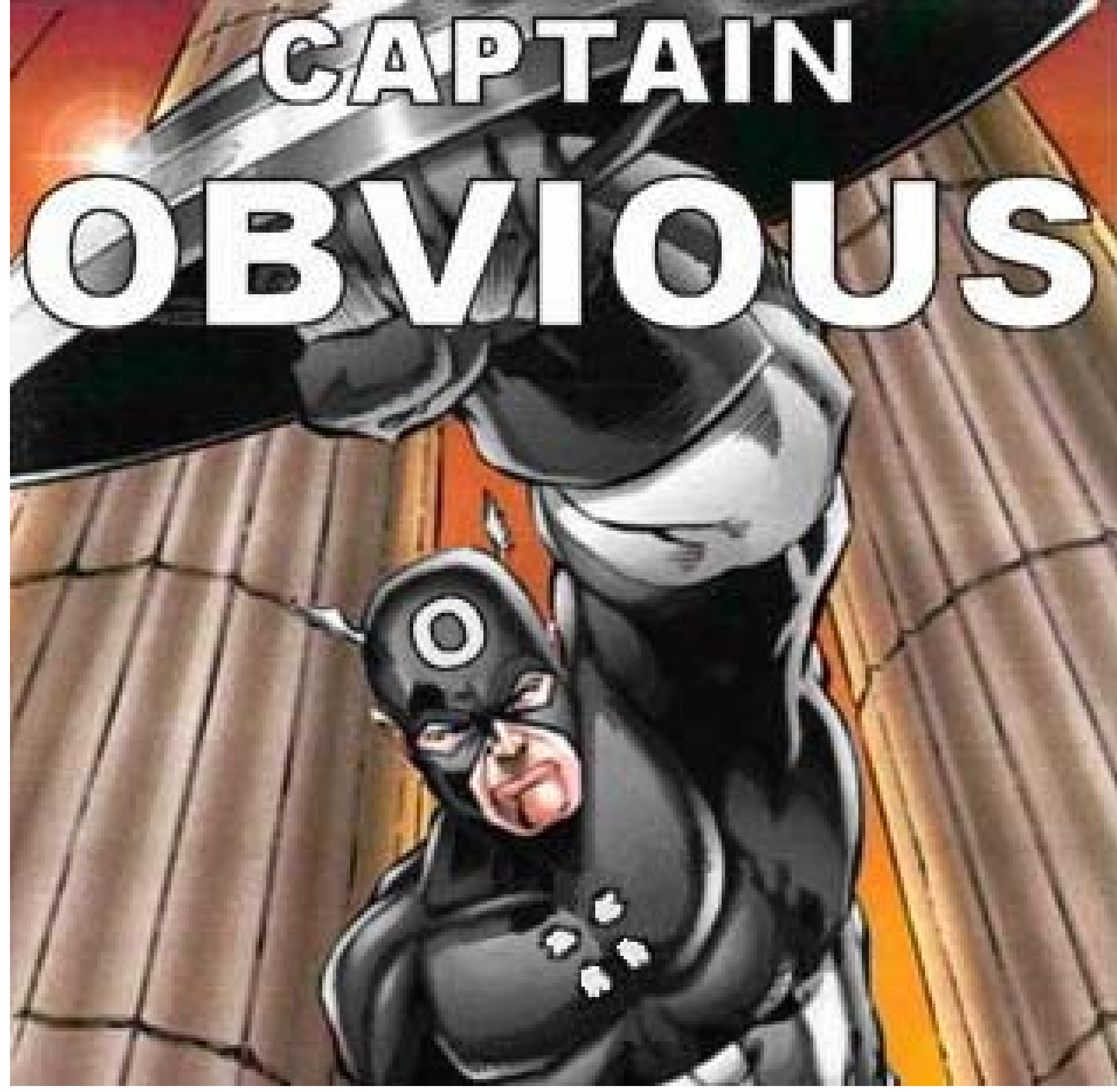
Introduction

Basic Theory

Java Benchmarking

Tools

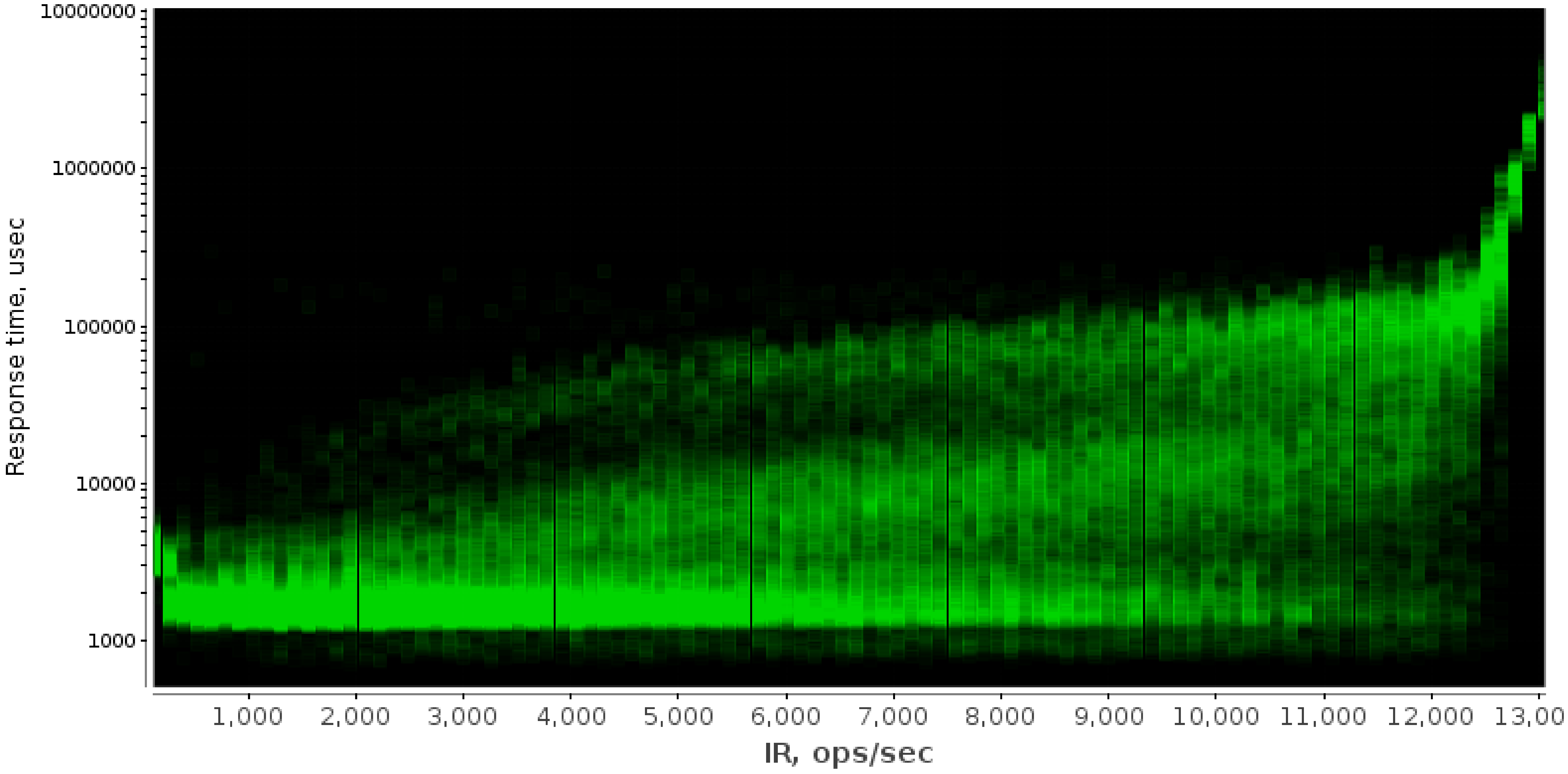
Metrics



- Performance metrics have two major manifestations
  - How many operations per unit of time? (λ, throughput, bandwidth)
  - How long one operation takes? (τ, response time, latency)
  - “Bandwidth only tells parts of the story” © pingtest.net
- It turns out, λ is much easier to improve
  - DDR2 PC2-3200: 3200 Mb/sec, CL 4
  - DDR3 PC3-17000: 17000 Mb/sec, CL 15
- In most queued systems, τ is at the mercy of λ
  - “Nine women can't make a baby in one month”
    - If you pipeline them, you might have a baby each month, each day, each second
    - Now what if you have the bounded queue of available midwives?

Metrics

- Wider you go, slower you get: “hockey stick”



Composability

- Assume you have two code blocks, A and B
  - Any difference with A and B running exclusively (…) or concurrently (||)?
- Functionality:
  - *Functionality(A ... B) = Functionality(A || B)*
  - “Black box abstraction”: behavior is the same, unless you want otherwise
- Performance:
  - *Performance(A … B)* ***?*** *Performance(A || B)*
  - Who can bet?
    - A and B are competing for processing resources
    - It could be >, e.g. blocks are fighting for the cache
    - It could be <, e.g. single-threaded blocks on multicore machine
    - It could be even =, e.g. for perfectly multi-threaded CPU-bound tasks

Composability

- Lesson: “Black Box Abstraction *does not* work for performance”
  - In the real world, processes and threads are competing for resources
  - How to estimate performance then?
  - True even for uniprocessor machines: thread preemption
- Very important for manycore world
  - Virtually all the code is executed concurrently with something else
  - Single-threaded benchmarks are then *useless* to predict real performance
    - Unless you can expect your code is the One Mega Application
- Should I test all the juxtapositions with every other program?
  - Exponential explosion of test configuration space
  - There's good approximation: run the same code in multiple threads

God Does Play Dice

- Empirical data is subject to variance
  - Replication is your friend
    - Many runs, many iterations, statistics
    - Runs on different machines under different conditions
- Scientific Control
  - Positive: expect the phenomenon to appear
  - Negative: expect the phenomenon to *not* appear
- Statistical Inference
  - System A runs **50** ops/sec, System B runs **40** ops/sec, is System A faster?

"Piled Higher and Deeper" by Jorge Cham
www.phdcomics.com

God Does Play Dice

- Empirical data is subject to variance
  - Replication is your friend
    - Many runs, many iterations, statistics
    - Runs on different machines under different conditions
- Scientific Control
  - Positive: expect the phenomenon to appear
  - Negative: expect the phenomenon to *not* appear
- Statistical Inference
  - System A runs **50** ops/sec, System B runs **40** ops/sec, is System A faster?
    - Case 1. A = **50** ± 12 ops/sec, B = **40** ± 24 ops/sec (not really)
    - Case 2. A = **50** ± 3 ops/sec, B = **40** ± 2 ops/sec (*probably*, yes)

"Piled Higher and Deeper" by Jorge Cham
www.phdcomics.com

David Brent, “The Office”

“Those of you who think you know everything are annoying to those of us who do.”

“Imperial Examination” in performance

- HW specifics
  - cpu/memory layout, cache sizes and associativity, power states, etc.
- OS specifics
  - threading model, thread scheduling and affinity, system calls performance, etc.
- Libraries specifics
  - algorithms, tips and tricks for better performance, etc.
- Compilers specifics
  - high-level and low-level optimizations, tips and tricks, etc.
- Algos specifics
  - algorithmic complexity, data access patterns, etc.
- Data specifics
  - representative sizes and values, operation mix, etc.

Why do I need to know that?

- The Ultimate Question:

How does my benchmark perform?

Why do I need to know that?

- The Ultimate Question:

~~How does my benchmark perform?~~

Why doesn't my benchmark perform *better*?

Why do I need to know that?

- The Ultimate Question:

~~**How does my benchmark perform?**~~

Why doesn't my benchmark perform *better*?

- The answer tells many things about quality of your experiment
  - What are we really stressing?
  - Are we stressing the part we want to stress?
  - What needs to be changed in benchmark to make it right?

Agenda

Introduction

Basic Theory

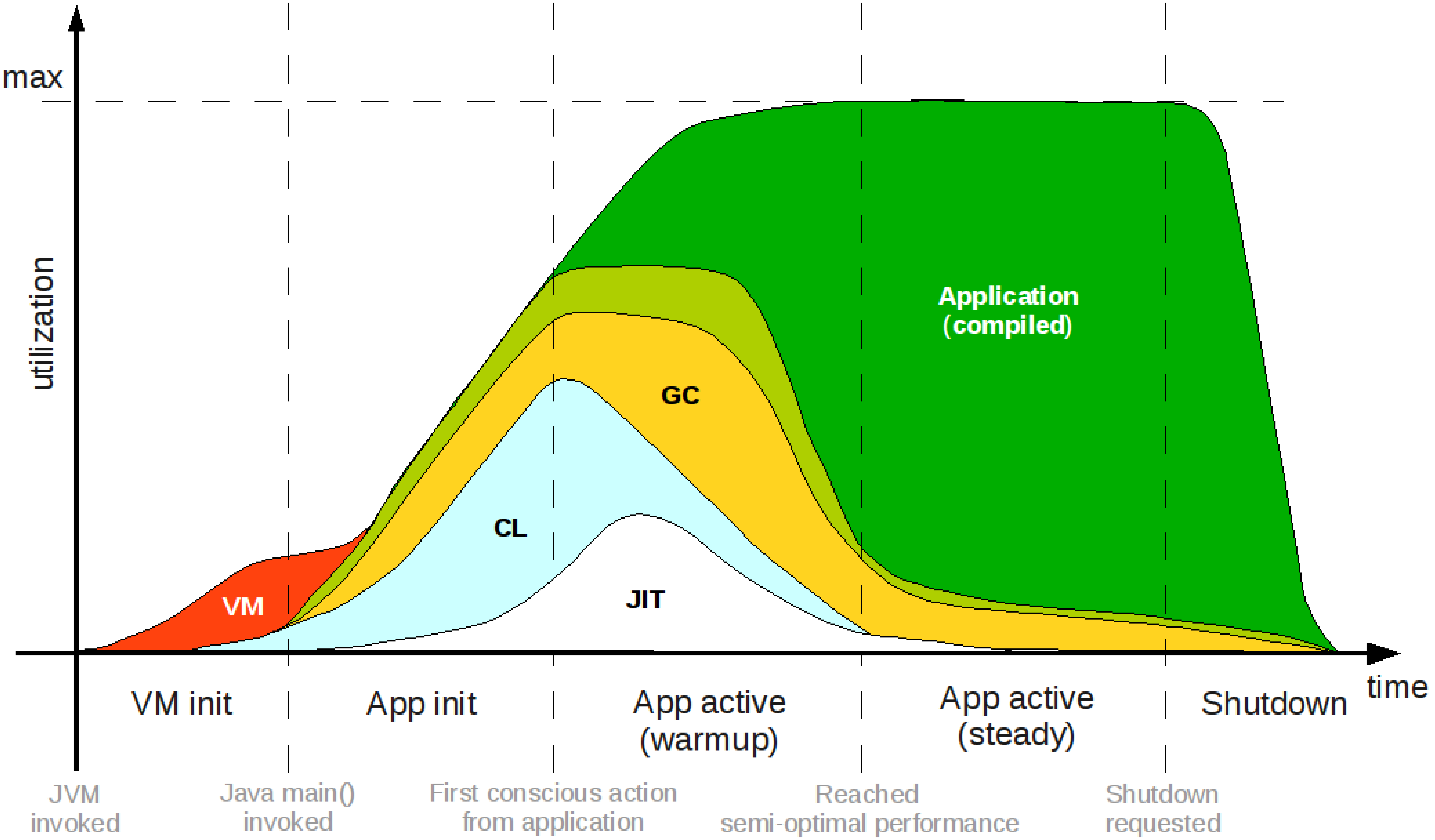
Java Benchmarking

Tools

Dynamic Runtimes? Bring It On!

- Also need to know about:
  - Virtual Machines
    - Class loading, class verification, class lifecycle
  - JIT
    - Code lifecycle
    - Profiling, compile plans, OSR
    - Aggressive optimizations
    - Usual and pathological work modes
  - GC
    - Types, algos used, throughput vs. response time
    - Usual and pathological work modes

Typical Java Runtime Lifecycle (Not to scale!)



Dr. Cliff Click

“Micro-benchmarks are like a microscope.
Magnification is high, but what the heck are you looking at?”

Pitfall #0. Type III Errors

- PiDigits subtest from Computer Language Benchmarks Game
  - Computes first N digits in decimal form of π
    - Requires arbitrary precision for series coefficients
  - So, this is what you call Java benchmark?
    - Uses native GNU MP for computation
    - Java is used to arrange native calls
      - ...so up to 90% of time is spent in GNU MP
  - This code will never pass sensible code review
    - Do you really want to measure it?

```
public void _run() {
    // ...

    acquire(sema[op1], 1);
    sema[op1].release();
    acquire(sema[op2], 1);
    sema[op2].release();

    if (instr == MUL) {
          GmpUtil.mpz_mul_si(...);
    } else if (instr == ADD) {
          GmpUtil.mpz_add(...);
    } else if (instr == DIV_Q_R) {
          GmpUtil.mpz_tdiv_qr(...);
          sema[op3].release();
    }
    sema[dest].release();
 }
```

Pitfall #0. Type III Errors

Lessons Learned:

Choose what you want to measure.

Assess what you had really measured.

Fix it.

Repeat.

Pitfall #1. Improper Warmup

- CLBG has two major metrics
  - Execution time
    - Measured in scripting equivalent of “time java $*”
  - Memory footprint
- Any dynamic runtime is penalized
  - Includes time to initialize, aggressive optimizations, adaptive algos
    - Shorter the test, more the overheads
  - So then, if you're competing with static compilers
    - Gotta watch for init time!
      - Don't use complex APIs
      - Move everything in native code?

Pitfall #1. Improper Warmup

- There *are* some “steady state” tests:

```
pidigits.javasteady:
     public static void main(String[] args){
          pidigits m = new pidigits(Integer.parseInt(args[0]));
          for (int i=0; i<65; ++i) m.pidigits(false);
          m.pidigits(true);
     }
```

- Now compare to the “standard” test:

```
pidigits.java:
     public static void main(String[] args){
          pidigits m = new pidigits(Integer.parseInt(args[0]));
          // for (int i=0; i<19; ++i) m.pidigits(false);
          m.pidigits(true);
     }
```

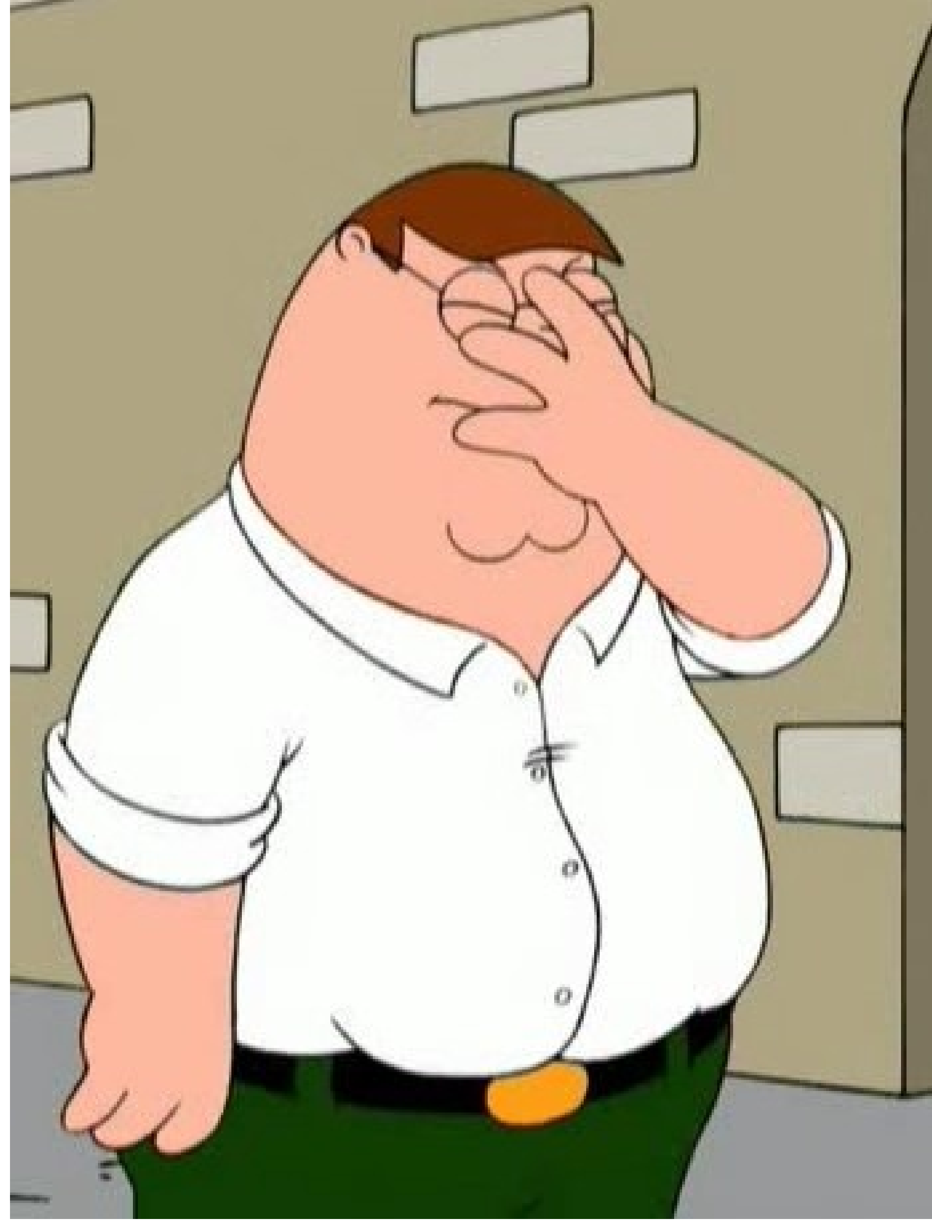
Pitfall #1. Improper Warmup

- There *are* some “steady state” tests:

```
pidigits.javasteady:
    public static void main(String[] args){
        pidigits m = new pidigits(Integer.parseInt(args[0]));
        for (int i=0; i<65; ++i) m.pidigits(false);
        m.pidigits(true);
    }
```

- CLBG maintainer furiously defended:
  - “... *The JavaOne Moscow presenter's slides fail to show that combined time measurement was* ***correctly divided by 65 to give the average.***”

Pitfall #1. Improper Warmup



- There *are* some “steady state” tests:

```
pidigits.javasteady:
    public static void main(String[] args){
        pidigits m = new pidigits(Integer.parseInt(args[0]));
        for (int i=0; i<65; ++i) m.pidigits(false);
        m.pidigits(true);
    }
```

- CLBG maintainer furiously defended:
  - “... *The JavaOne Moscow presenter's slides fail to show that combined time measurement was* ***correctly divided by 65 to give the average.***”
    - Yes, right, however:
      - Warmup iterations skip I/O, hitting deopt on “real” iteration
      - If, say, first iteration takes 20x longer, then average is skewed >13%
    - Averaging scores from different modes makes *little* sense

Pitfall #1. Improper Warmup

Lessons Learned:

Warmup the code.

Embrace warmup as distinguished lifecycle phase.

Warmup on the same code and data paths.

Pitfall #2. Observing Unrelated Effect

- Found in EclipseLink tests
  - One of the tests which measures synchronized method performance in uncontended scenario
- Published data
  - Negative scientific control **FAIL**:
    - block: 8244087 usec
    - method: 13383707 usec
  - Published conclusion:
    - “synchronized(this) { } is better”
  - JVM engineers:
    - “WTF, those two are equivalent”

```
public class SynchTest {

    int i;

    @GenerateMicroBenchmark
    void testSynchInner() {
        synchronized (this) {
            i++;
        }
    }

    @GenerateMicroBenchmark
    synchronized void testSynchOuter() {
        i++;
    }
}
```

This code is the mock. **Do** Try This At Home.

Pitfall #2. Observing Unrelated Effect

- Trying to reproduce
  - Calling each method many times
    - First, call synchInner() for 10 secs
    - Then, call synchOuter() for 10 secs
    - 1 thread, monitor is always uncontended
  - Results:
    - synchInner:

      40988 ± 218 ops/sec
    - synchOuter:

      261602 ± 11511 ops/sec

```
public class SynchTest {

    int i;

    @GenerateMicroBenchmark
    void testSynchInner() {
        synchronized (this) {
            i++;
        }
    }

    @GenerateMicroBenchmark
    synchronized void testSynchOuter() {
        i++;
    }
}
```

This code is the mock. **Do** Try This At Home.

Pitfall #2. Observing Unrelated Effect

- Solution
  - BiasedLocking is not enabled at once
    - By default, late 5000 msec after start
    - First test uses *thin* syncs
    - Second test uses *biased* syncs
  - Re-measuring with -XX:BiasedLockingStartupDelay=0
    - synchInner:
      287905 ± 12298 ops/sec
    - synchOuter:
      286962 ± 10114 ops/sec

```
public class SynchTest {

    int i;

    @GenerateMicroBenchmark
    void testSynchInner() {
        synchronized (this) {
            i++;
        }
    }

    @GenerateMicroBenchmark
    synchronized void testSynchOuter() {
        i++;
    }
}
```

This code is the mock. **Do** Try This At Home.

Pitfall #2. Observing Unrelated Effect

Lessons Learned:

Every result needs an explanation.

Surprising results can warn you about bad design.

Let more eyeballs to look over your data.

Pitfall #3. Weird Modes

- Not so long ago in some blog:

> “
>
> Today I want to show how you could compare e.g. different algorithms.
> You could simply do:
>
> ```
> int COUNT = 1000000;
> long firstMillis = System.currentTimeMillis();
> for(int i = 0; i < COUNT; i++) {
>   runAlgorithm();
> }
> System.out.println(System.currentTimeMillis()-firstMillis) / 1000.0 / COUNT);
> ```
>
> There are several problems with this approach, which results in very unpredictable results: [...]
>
> **You should turn off the JIT-compiler with specifying -Xint as a JVM option, otherwise your outcome could depend on the (unpredictable) JIT and not on your algorithm.** You should start with enough memory, so specify e.g.-Xms64m -Xmx64m, because JVM memory allocation could get time consuming. Avoid that the garbage collector runs while your measurement: -Xnoclassgc
>
> ”

Pitfall #3. Weird Modes

- Not so long ago in some blog:

> “
>
> Today I want to show how you could compare e.g. different algorithms.
> You could simply do:
>
> ```
> int COUNT = 1000000;
> long firstMillis = System.currentTimeMillis();
> for(int i = 0; i < COUNT; i++) {
>   runAlgorithm();
> }
> System.out.println(System.currentTimeMillis()-firstMillis) / 1000.0 / COUNT);
> ```
>
> There are several problems with this approach, which results in very unpredictable results: [...]
>
> **You should turn off the JIT-compiler with specifying -Xint as a JVM option, otherwise your outcome could depend on the (unpredictable) JIT and not on your algorithm.** You should start with enough memory, so specify e.g.-Xms64m -Xmx64m, because JVM memory allocation could get time consuming. Avoid that the garbage collector runs while your measurement: -Xnoclassgc
>
> ”

Pitfall #3. Weird Modes

- Run multiple doIteration()-s
  - On each iteration **bitset** is expanding
  - Throughput goes down, and down, and down
  - Suspected “JIT variability”
- Let's turn off JIT!
  - Overall throughput plummeted
  - Relative **bitset**-induced degradation is lower, given overall decrease
  - “Who-hoo, interpreter is more predictable!”
- Time to spread the word
  - But finally, took a look in the profiler, and realized **bitset** is the culprit

```
public class Test {
    private static BitSet bitset;

    private void doWork() {
        // do work
        bitset.set(n, r);
        // do more work
    }

    public void doIteration() {
        doWork();
    }
}
```

This code is the mock. **Do** Try This At Home.

Pitfall #3. Weird Modes

Lessons Learned:

Run with the same options you would expect in production.

Have a strong evidence additional options are needed for test.

Understand what different options actually affect.

Pitfall #4. Dead-code elimination

- Modern JIT compilers are so modern...
  - ...they do automatic constant propagation and dead-code elimination
  - That means smart compiler can detect “useless” blocks of code, and optimize out
    - The result of computation is not used? No other side effects? Eliminate!
    - What if that was our benchmarked code?
- Rather easy to fight
  - Consume the result or provide another side-effect
    - Print the result out
    - Store in field of always reachable object
    - Compare with non-trivial, but impossible value, and throw exception otherwise
    - (I would really like to see some compiler hint, i.e. “Unsafe.sideEffect(Object o)”)

Pitfall #4. Dead-code elimination

- Typical example
  - Found in Apache Commons Math tests
- Typical result
  - StrictMath: 71 msecs
  - FastMath: 39 msecs
  - Math: 0 msecs
- That's one super-fast java.util.Math!

```
@Test
public void testLog() {
    double x = 0;
    long time = System.nanoTime();
    for (int i = 0; i < RUNS; i++)
        x += StrictMath.log(Math.PI + i);
    long strictMath = System.nanoTime() - time;

    x = 0;
    time = System.nanoTime();
    for (int i = 0; i < RUNS; i++)
        x += FastMath.log(Math.PI + i);
    long fastTime = System.nanoTime() - time;

    x = 0;
    time = System.nanoTime();
    for (int i = 0; i < RUNS; i++)
        x += Math.log(Math.PI + i);
    long mathTime = System.nanoTime() - time;

    report("log",strictMath,fastTime,mathTime);
}
```

Pitfall #4. Dead-code elimination

- Typical example
  - Found in Apache Commons Math tests
- Typical result
  - StrictMath: 71 msecs
  - FastMath: 39 msecs
  - Math: 0 msecs
- What's going on?
  - “x” is not used, smart JIT deduced there are no side effects in Math.log(...) call → BOOM!
  - Printing out “x” resolves the issue
  - Math: 23 msecs

```
@Test
public void testLog() {
    double x = 0;
    long time = System.nanoTime();
    for (int i = 0; i < RUNS; i++)
        x += StrictMath.log(Math.PI + i);
    long strictMath = System.nanoTime() - time;

    x = 0;
    time = System.nanoTime();
    for (int i = 0; i < RUNS; i++)
        x += FastMath.log(Math.PI + i);
    long fastTime = System.nanoTime() - time;

    x = 0;
    time = System.nanoTime();
    for (int i = 0; i < RUNS; i++)
        x += Math.log(Math.PI + i);
    long mathTime = System.nanoTime() - time;

    report("log",strictMath,fastTime,mathTime);
}
```

Pitfall #4. Dead-code elimination

Lessons Learned:

Computers are lazy.

Runtime may not bother computing, if result is not claimed.

Pitfall #5. Incomparable Results

- Bob (pun intended) had developed a new fancy Java collection
  - Bob claims it's uber-fast and thread-safe
- Alice is trying to get performance data with two tests
  - Test scalability for shared accesses
    - One collection instance for all the threads, #Threads = #CPUs
  - Test scalability for exclusive accesses
    - One collection per each thread, #Threads = #CPU
- Why not one collection instance for single thread?
  - Obviously not fair – system load will differ drastically
  - Note both two tests should saturate the system

Pitfall #5. Incomparable Results

- Alice's first measurement:
  - 4 threads, 4 logical cores
  - Exclusive access

    615 ± 12 ops/sec
  - Shared access

    828 ± 21 ops/sec

- Bob: “Say what?”
  - Shared access is faster?
  - This can't be happening!

```
public class AliceBobMakerTest {
    List<String> keys = new ArrayList<>();
    Map<String, String> map = BobMaker.newMap();

    @GenerateMicroBenchmark
    public void test() {
        for(String key : keys) {
            String value = map.get(key);
            if (!value.equals(key)) {
                // make side effect
                throw new ISE("Violated");
            }
        }
    }
}
```

This code is the mock. **Do** Try This At Home.

Pitfall #5. Incomparable Results

- Memory footprint
  - It happens, one collection takes ~250 Kb
  - L2 cache = 256 Kb
  - L3 cache = 3072 Kb
- These two tests are incomparable:
  - Exclusive mode:
    - 4 instances x 250 Kb = 1000 Kb
  - Shared mode:
    - 1 instances x 250 Kb = 250 Kb
- Might be other non-obvious differences

	Exclusive	**Shared**
1 thread	314 ± 14	296 ± 11
2 threads	561 ± 21	554 ± 12
4 threads	615 ± 12	828 ± 21
8 threads	598 ± 15	815 ± 15
16 threads	595 ± 12	829 ± 16
32 threads	644 ± 43	915 ± 34

Pitfall #5. Incomparable Results

Lessons Learned:

Carefully assess what results you *can* compare.

(Turns out comparable results are rare exceptions)

Pitfall #6. Cross-Language Benchmarks

- Larger the platforms you're comparing, more the subtle differences
  - Used concepts, algorithms, libraries, default settings, hardware support
  - Comparing several large platforms is Mammoth task
    - It's nearly impossible to create *useful* cross-language benchmark
- Comparing $lang1 vs. $lang2
  - Probably the easiest kickoff for never-ending holy war
    - All about wrong question: “Is $lang1 faster/slower than $lang2?”
    - Tons of troll food expected
  - Smart people see the educational opportunity here:
    - Valid question: “***WHY*** $program1 running with $impl1 of $lang1 performs different from $program2 running with $impl2 of $lang2 on $hardware?”
    - Most of the time requires deep knowledge of both $impl1 and $impl2

Pitfall #6. Cross-Language Benchmarks

Lessons Learned:

Don't do it.

Do that *only* to answer “Why performance is different?”

Pitfall #7. Infrastructure Overheads

- Service code is executed within the same benchmark
  - In case of micros, the impact of service code is comparable with tested code itself
  - Very, very, **VERY** easy to shoot oneself in the foot
- Usual overheads
  - Counters
    - Useless contentions, redundant dereferences, etc.
    - (example follows)
  - Timers
    - It's OK to take the timestamp every now and then
    - “Mad Hatter” syndrome: some benchmarks do that *millions* times each second
  - I/O
    - Are you synchronously writing the logs to disk?

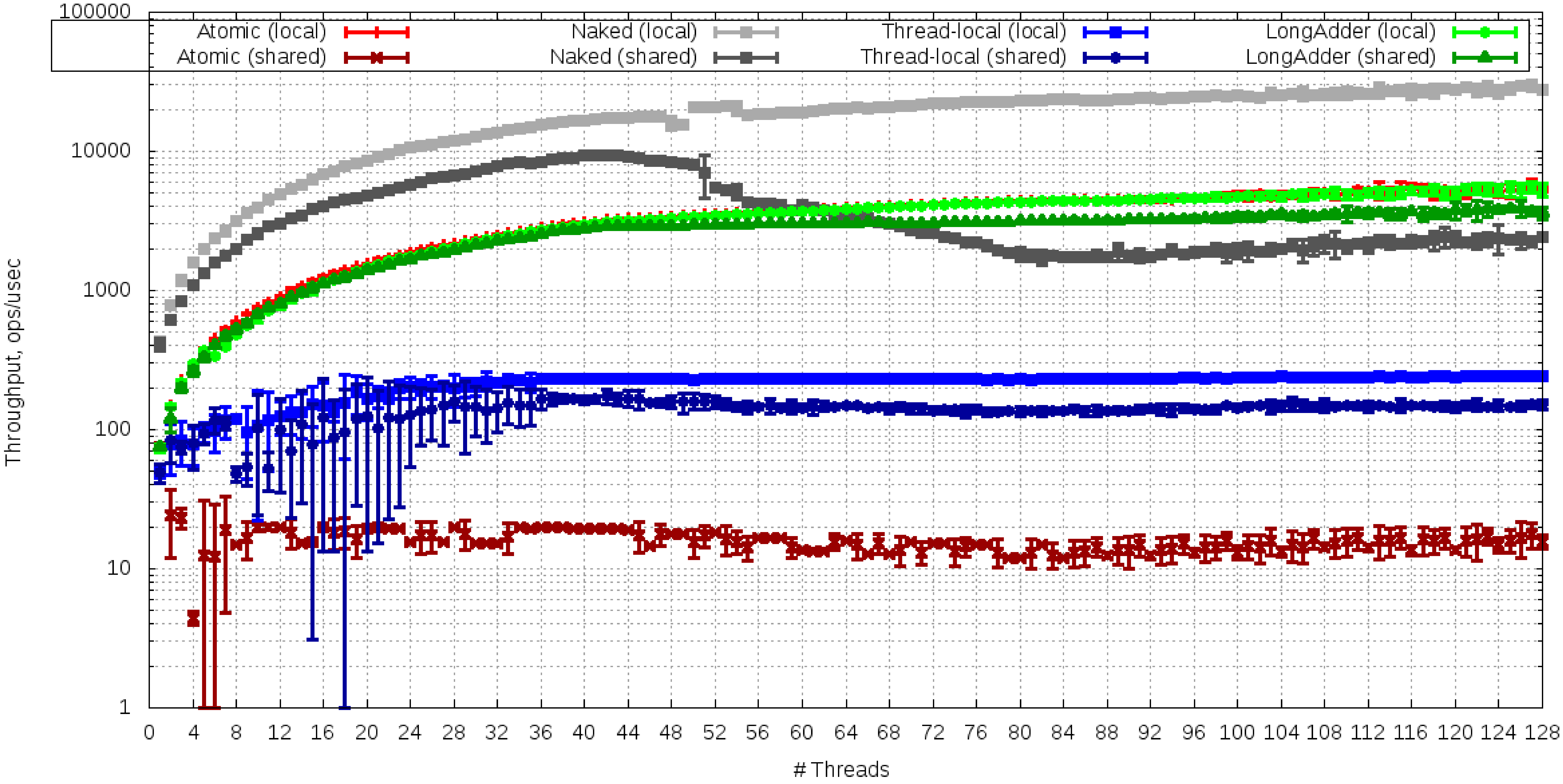
Pitfall #7. Infrastructure Overheads

- Mind the simple test
  - Measuring *something*
  - Metric: computations/sec
- How to count successful operations?
  - Sometimes you don't know the thread context
  - ...so the counter should be thread-safe anyway
- Our options are:
  1. Naked counter
  2. Atomic counter
  3. jsr166e-ish LongAdder
  4. ThreadLocal “counter”

```
public class MyBrokenBenchmark {

    private long counter;

    private AtomicLong atomicCounter;

    private LongAdder longAdderCounter;

    private ThreadLocal<Long> tlCounter =
        new ThreadLocal<Long>();

    public void test() {
        // { increment counter here }
        // do work here
    }
}
```

This code is the mock. **Do** Try This At Home.

Pitfall #7. Infrastructure Overheads



Intel Westmere-EX (E7-4860) 2.27Ghz, 4x10x2 = 80 HW threads, RHEL 5.5, JDK 7b148. High/low whiskers are (-2σ, +2σ) for 5 samples

Pitfall #7. Infrastructure Overheads

Lessons Learned:

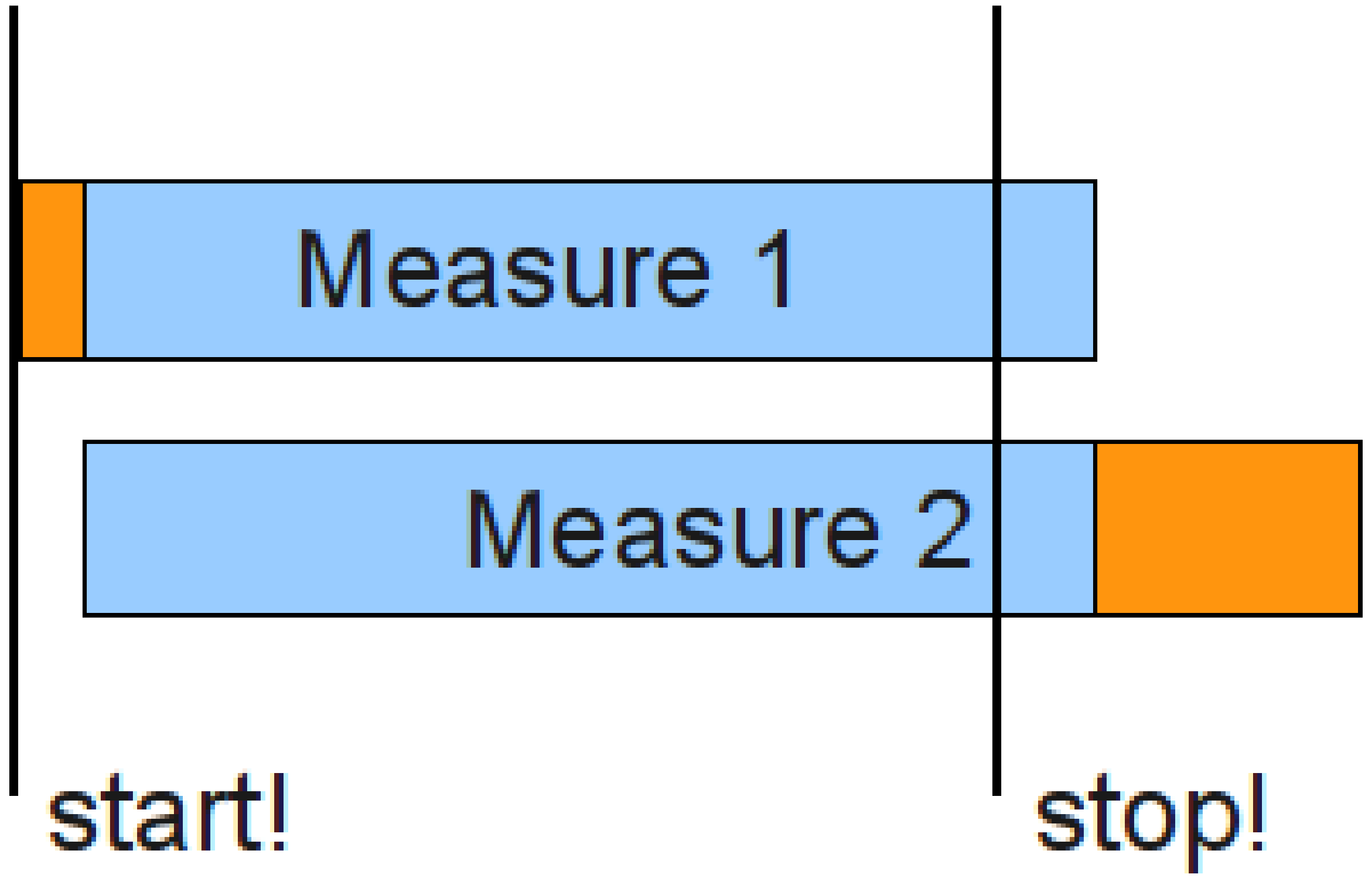
Your test infrastructure is your best friend.

Your test infrastructure can turn out to be your worst enemy.

Always suspect it's faulty, unless proven otherwise.

Pitfall #8. Thread Scheduling

- Thread scheduling is not always deterministic
  - (Unless you are running realtime OS)
  - Thread start/stop times are then non-deterministic
  - Even if you park threads on locks, and let go at once
- Mind N threads doing the same work
  - Do they finish at the same time? Nope.
    - Some thread can finish prematurely
    - Then, other threads are free to proceed in better conditions
      - Distorts the measurement
      - Gives unreasonably high throughput
    - One of the most frequent jitter sources

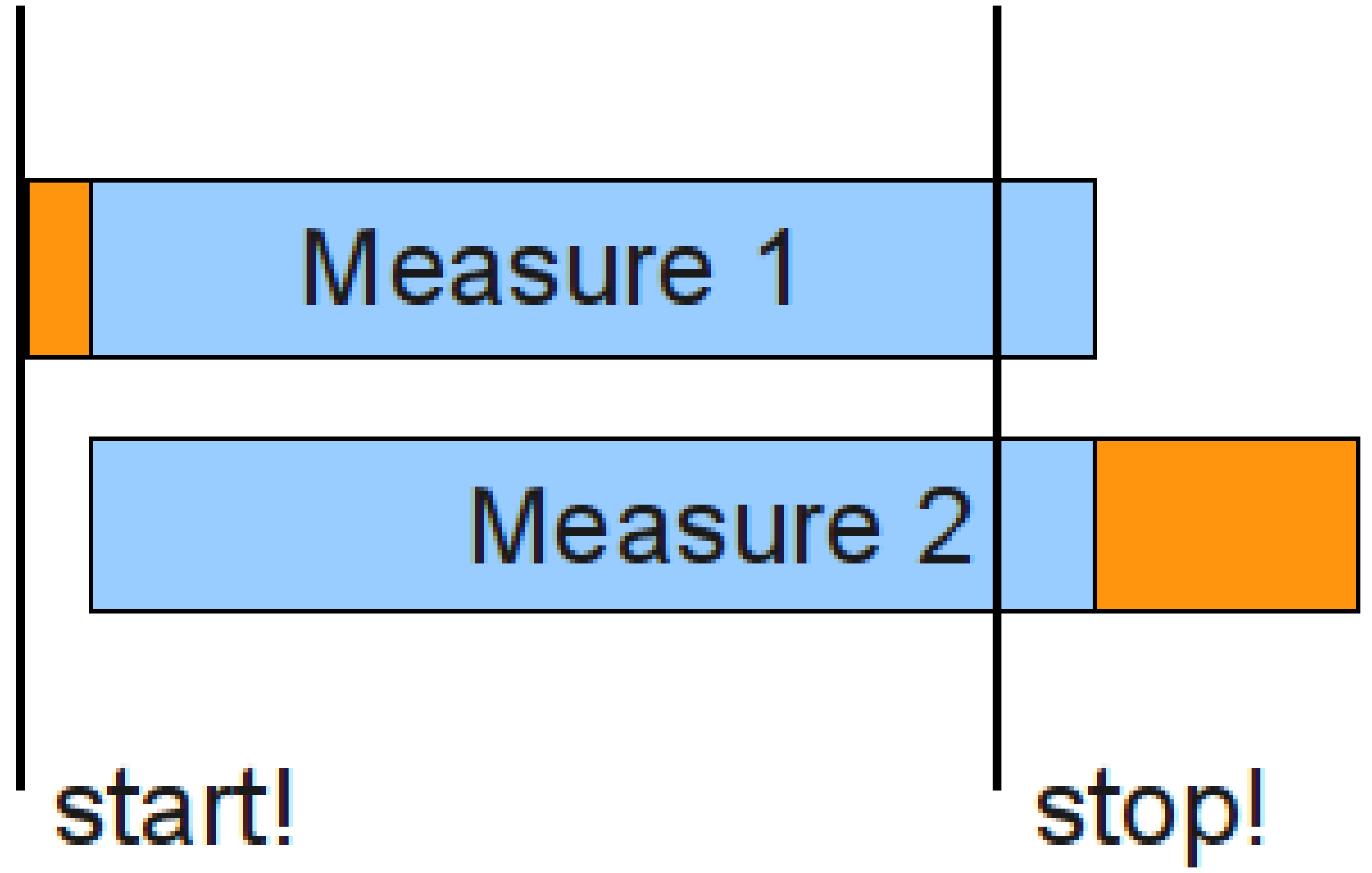


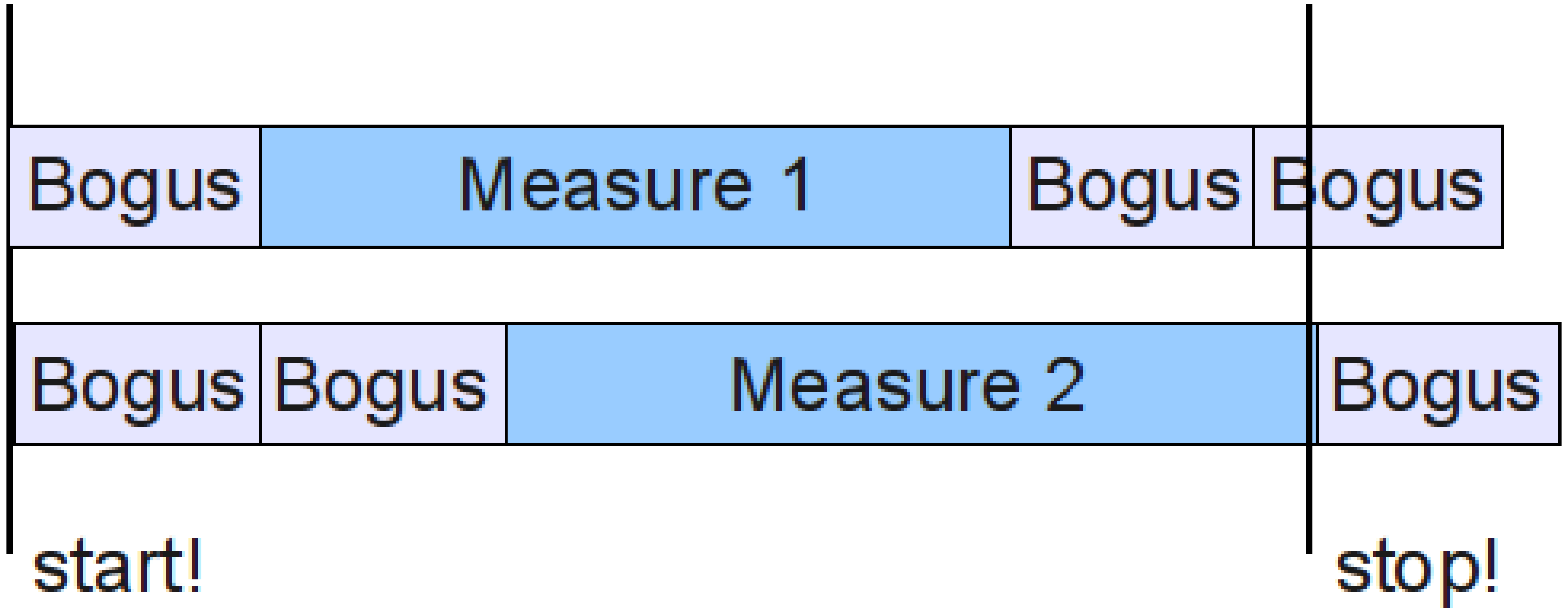
Pitfall #8. Thread Scheduling

- Need to insert bogus iterations
  - Measured parts are always executing as if all the threads are busy
- Simple test
  - Running 24 threads on 24-way host
  - Doing something quite heavy
  - Without bogus iterations

    957 ± 32 ops/min
  - With bogus iterations

    821 ± 5 ops/min





Pitfall #8. Thread Scheduling

Lessons Learned:

High benchmark variance is telling you something.

Usually it means someone is making critical decisions behind your back.

Pitfall #9. Compile Plans

- Runtime is able to adapt to profile
  - Benefits (lots of) real applications
  - Subtle trouble for benchmarking
- Profile might be conservative
  - Running test T1: runtime gathers profile P(T1), compiles for it
  - Running another test T2, runtime gathers additional profile P(T2), so new profile is P(T1+T2)
  - Runtime has no hints to indicate “now everything had changed, scrap all you have”

```
public class CompilePlanTest {
    Counter counter1 = new CounterImpl1();
    Counter counter2 = new CounterImpl2();

    @GenerateMicroBenchmark
    public void testM1() {
        test(counter1);
    }

    @GenerateMicroBenchmark
    public void testM2() {
        test(counter2);
    }

    public void test(Counter counter) {
        for(int c = 0; c < LIMIT; c++) {
            counter.inc();
        }
    }
}
```

This code is the mock. **Do** Try This At Home.

Pitfall #9. Compile Plans

- Executing one-by-one in single JVM
  - testM1: 394 ± 7 ops/msec
  - testM2: 11 ± 1 ops/msec
- Executing each one in fresh JVM
  - testM1: 396 ± 3 ops/msec
  - testM2: 381 ± 16 ops/msec
- Single test is aggressively optimized
  - JVM is able to inline inc() { i++; }, unroll the loop, figure out bunch of increments is actually { i += $unroll_limit; }, furiously faster!

```
public class CompilePlanTest {
    Counter counter1 = new CounterImpl1();
    Counter counter2 = new CounterImpl2();

    @GenerateMicroBenchmark
    public void testM1() {
        test(counter1);
    }

    @GenerateMicroBenchmark
    public void testM2() {
        test(counter2);
    }

    public void test(Counter counter) {
        for(int c = 0; c < LIMIT; c++) {
            counter.inc();
        }
    }
}
```

This code is the mock. **Do** Try This At Home.

Pitfall #9. Compile Plans

Lessons Learned:

Profiles are usually to be “shaken, not stirred”

When measuring multi-method performance, warm up everything.

When measuring single-method performance, better start fresh VM.

Agenda

Introduction

Basic Theory

Java Benchmarking

Tools

Essential Tools

- Your Brain, equipped with some plug-ins*
  - “WTFISGOINGON” for doubting and rechecking facts
  - “LETMETHINK” for building hypotheses, and appropriate tests
  - “THISCANTBEHAPPENNING” for checking data consistency
  - “THATWASSTUPID” for painless rejection of results
- Your Hands
  - Accurate, for making neat experiments
  - Strong and powerful, for processing tons of experimental data
- Your Peripheral I/O Devices
  - Exchanging the results with fellow researchers, and conducting peer review
  - Accessing experiment history

* these plug-ins do not come with basic package

Collateral Tools

- Application Profilers
  - VisualVM, JRockit Mission Control, Oracle Solaris Studio Performance Analyzer
- Whole-System Profilers
  - top, vmstat, mpstat, iostat, dtrace, strace, etc.
- JRE profiling and debug information
  - -XX:+PrintCompilation, -verbose:gc, -verbose:class, -Xprintflags
- Disassemblers
  - http://wikis.sun.com/display/HotSpotInternals/PrintAssembly
- Hardware Counters
  - Sun Studio Performance Analyzer, oprofile, VTune, etc.

Q & A

Q & A Ultimate Answer:
“It Depends”

JavaOne™