Java Benchmarking

as easy as two timestamps

Aleksey Shipilëv
aleksey.shipilev@oracle.com, @shipilev

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Intro

Intro: Warming up...

«How much for instantiating a String?»

```java
long time1 = System.nanoTime();
for (int i = 0; i < 1000; i++) {
  String s = new String("");
}
long time2 = System.nanoTime();
System.out.println("Time:" + (time2 - time1));
```

Theory

Theory: Why would people benchmark?

In the name of...

1. **Holywar**: Node.js – But Java... – Node.js!
2. **Marketing**: Check we are meeting the (release) criteria
3. **Engineering**: Isolate a performance phenomena, make a reference point for improvements
4. **Science**: Understand the performance model, and predict the future behavior

Theory: In the name of Holywar

My favorite example: Computer Language Benchmarks Game:[1]

- Most comparisons are hardly fair: e.g. AOT vs. JIT
- Measures what exactly? E.g. pidigits measures the speed of FFI to GNU GMP
- Lots of disclaimers these results are misrepresentative of the real world (alas, nobody reads them or cares enough)
- People love it, since it gives you **numbers**, which you can then take as your shield and sword in Internet debates

[1] `http://benchmarksgame.alioth.debian.org/`

Theory: In the name of Marketing

My favorite example: SPEC benchmarks

- Reference benchmark suites, agreed upon by the vendors
- Provide the reference points, for which one can set the success criteria, use in adverts, tweet obnoxious competitive data, etc.
- It does not matter how representative they are – it matters they are The Benchmarks Born at the Fiery Summit of Orodruin

Theory: In the name of Engineering

«If you can't measure it, you can't optimize it»

- Need the conditions where the system is running in a predictable state, so we are able to quantify improvements
- These benchmarks usually focus on particular pieces of system, and have more resolution than «marketing» benchmarks

Theory: In the name of Science

«Science Town PD: To Explain and Predict»

- Derive the sound performance model from the results
- Use the performance model to predict the future behavior: keep calm and deploy to production
- The most sweaty, and the most reliable target for benchmarking

Theory: Why would people benchmark?

In the name of...

1. **Holywar**: Node.js – But Java... – Node.js!
2. **Marketing**: check we are meeting the (release) criteria
3. **Engineering**: isolate a performance phenomena, make a reference point for improvements
4. **Science**: understand the performance model, and predict the future behavior

Theory: «Scientific» approach

Ultimate Question

How does a benchmark react on changing the external conditions?

Or, how far the **actual** performance model is from the **mental** one?

1. Fool-proof: do these results even make any sense?
2. Negative control: benchmark reacts on change, but shouldn't?
3. Positive control: benchmark should not react on change, but does?

Theory: «Engineering» approach

Ultimate Question

Why doesn't my benchmark run faster?

Directly observe if our experimental setup is sane:

1. Where are the bottlenecks?
2. Do we expect those things to be bottlenecks?
3. Are these benchmarks running in the same mode?

Theory: JMH

JMH is a Serious Business:
`http://openjdk.java.net/projects/code-tools/jmh/`

- When used properly, helps to mitigate VM quirks
- Aids running lots of benchmarks in different conditions
- Internal profiling to quickly triage the issues
- JVM languages support: Java, Scala, Groovy, Kotlin
- ...or anything else callable from Java (e.g. Nashorn, etc.)

Scientific

Scientific: Story

In this section, we explore some of the methodology implications when doing the benchmarks. People tend to think this story is a deal-breaker when trying to build their own benchmark harnesses.

Complete story and narrative is here:
`http://shipilev.net/blog/2014/nanotrusting-nanotime/`

Models: Model Problem

«Jessie, it's time to cook some benchmarks...»

«What is the cost of `volatile` write?»

It seems like a very easy question...
Let's measure it! Shall we?

Models: Easy...

```
public class VolatileWrite {
  int v; volatile int vv;

  @Benchmark
  int baseline1()    { return 42; }

  @Benchmark
  int incrPlain()    { return v++; }

  @Benchmark
  int incrVolatile() { return vv++; }
}
```

Models: ...does it!

```
public class VolatileWrite {
  int v; volatile int vv;

  @Benchmark
  int baseline1()     { return 42; }   // 2.0 ns

  @Benchmark
  int incrPlain()     { return v++; }  // 3.5 ns

  @Benchmark
  int incrVolatile() { return vv++; } // 15.1 ns
}
```

Models: Fatal Flaw

```
volatile int vv;

@Benchmark
int incrVolatile() { return vv++; }
```

- Measuring in very unfavorable case, when benchmark is choked by volatiles. We are pushing the system to its «edge» condition. This almost never happens in production.
- What do we really need to know is:
  «What is the volatile cost in realistic conditions?»

Models: Backoffs

```
@Param int tokens;

volatile int vv;

@Benchmark
int incrVolatile() {
  Blackhole.consumeCPU(tokens); // burn time
  return vv++;
}
```

- «Burn off» a few cycles before doing heavy-weight op
- Juggle `tokens` ⇒ juggle operation mix

Models: Backoffs

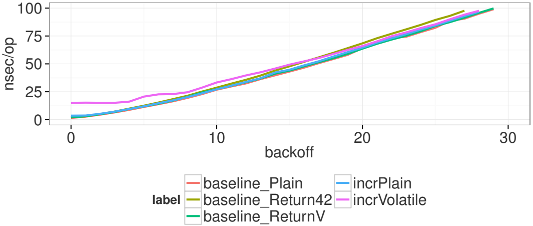
- Take a few baselines while we are at it: which one is correct?

```
@Benchmark
void baseline_Plain()
   { BH.consumeCPU(tokens); }

@Benchmark
int baseline_Return42()
   { BH.consumeCPU(tokens); return 42; }

@Benchmark
int baseline_ReturnPlain()
   { BH.consumeCPU(tokens); return v; }
```

Models: Measuring...

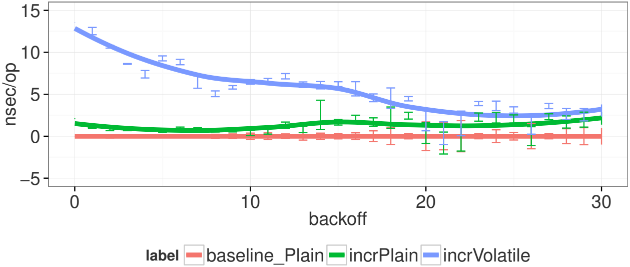
A picture with two robots
in a very hot environment,
one deeply frustrated by
its own actions,
which benefits another
(trickier) robot

«Bender B. Rodriguez regrets using Excel to draw the charts»



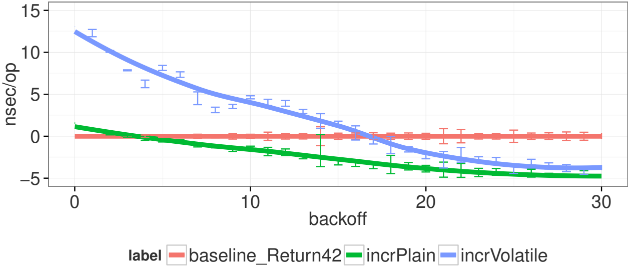
Models: Subtracting `baselinePlain`

- Absolute `volatile` cost gets compensated very well!
- Can we really subtract the baselines?

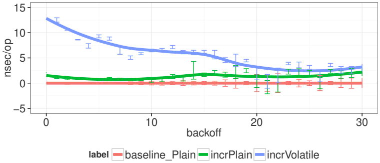


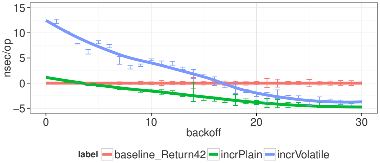
Models: Subtracting `baseline_Return42`

- We added some code in the baseline, and it runs **faster**?
- Nothing surprising: *performance is not usually composable*



Models: WTF is different?

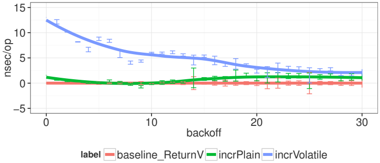


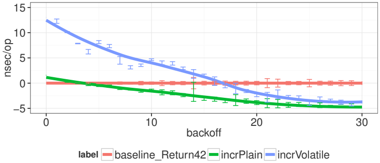


```
@Benchmark
void base_Plain() {
  BH.consumeCPU(tkns);
}
.
```

```
@Benchmark
int base_Ret42() {
  BH.consumeCPU(tkns);
  return 42;
}
```

Models: WTF is different?



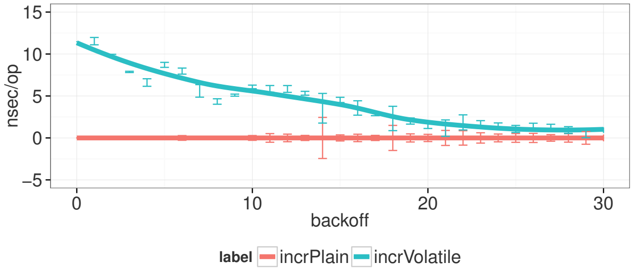


```
@Benchmark
int base_RetV() {
  BH.consumeCPU(tkns);
  return v;
}
```

```
@Benchmark
int base_Ret42() {
  BH.consumeCPU(tkns);
  return 42;
}
```

Models: Bottom Line

- Different baselines act differently: they are **tests** themselves!
- Therefore, we can just compare `plain` and `volatile`:



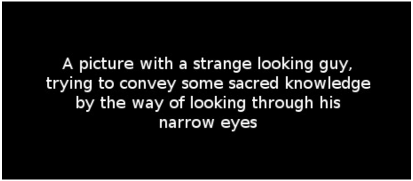
Models: Conclusion

This is what models are for!

- Explore the system behavior outside the (randomly) chosen configuration points
- Allow to predict the system behavior in future conditions
- Catch the experimental setup problems (control!)
- Combinatorial experiments help to create different operation mixes, and derive the individual op costs from their composite performance

Models: You Are Joking, Right?

«Combinatorial experiments help to create different operation mixes, and derive the individual op costs from their composite performance»



A picture with a strange looking guy, trying to convey some sacred knowledge by the way of looking through his narrow eyes

`System.nanoTime`!
Measure each part individually!

Timers: Verifying infrastructure

Why not?

```java
// call continuously
public long measure() {
  long startTime = System.nanoTime();
  work();
  return System.nanoTime() - startTime;
}
```

Timers: Measuring Latency

Latency = time to call `System.nanoTime`

```
@Benchmark
public long latency_nanotime() {
  return System.nanoTime();
}
```

Timers: Measuring Granularity

Granularity = the minimum non-zero difference between two consecutive calls

```
private long lastValue;

@Benchmark
public long granularity_nanotime() {
  long cur;
  do {
    cur = System.nanoTime();
  } while (cur == lastValue);
  lastValue = cur;
  return cur;
}
```

Timers: Typical Case [Linux]

```
Java(TM) SE Runtime Environment, 1.7.0_45-b18
Java HotSpot(TM) 64-Bit Server VM, 24.45-b08
Linux, 3.13.8-1-ARCH, amd64

Running with 1 threads and [-client]:
  granularity_nanotime: 26.300 +- 0.205 ns
      latency_nanotime: 25.542 +- 0.024 ns

Running with 1 threads and [-server]:
  granularity_nanotime: 26.432 +- 0.191 ns
      latency_nanotime: 26.276 +- 0.538 ns
```

Timers: Typical Case [Solaris]

```
Java(TM) SE Runtime Environment, 1.8.0-b132
Java HotSpot(TM) 64-Bit Server VM, 25.0-b70
SunOS, 5.11, amd64

Running with 1 threads and [-client]:
   granularity_nanotime: 29.322 +- 1.293 ns
       latency_nanotime: 29.910 +- 1.626 ns

Running with 1 threads and [-server]:
   granularity_nanotime: 28.990 +- 0.019 ns
       latency_nanotime: 30.862 +- 6.622 ns
```

Timers: Typical Case [Windows]

```
Java(TM) SE Runtime Environment, 1.7.0_51-b13
Java HotSpot(TM) 64-Bit Server VM, 24.51-b03
Windows 7, 6.1, amd64

Running with 1 threads and [-client]:
   granularity_nanotime: 371,419 +- 1,541 ns
       latency_nanotime: 14,415 +- 0,389 ns

Running with 1 threads and [-server]:
   granularity_nanotime: 371,237 +- 1,239 ns
       latency_nanotime: 14,326 +- 0,308 ns
```

Timers: Epic Case [Windows]

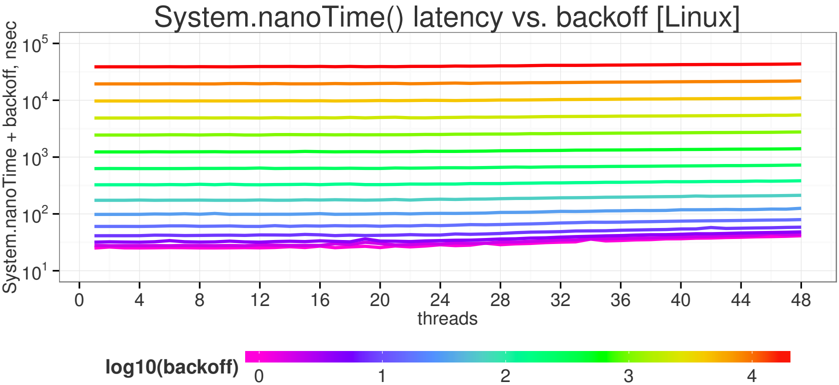
```
Java(TM) SE Runtime Environment, 1.8.0-b132
Java HotSpot(TM) 64-Bit Server VM, 25.0-b70
Windows Server 2008, 6.0, amd64

Running with 32 threads and [-client]:
   granularity_nanotime: 15137.504 +-   97.132 ns
       latency_nanotime: 15190.080 +- 1760.500 ns

Running with 32 threads and [-server]:
   granularity_nanotime: 15118.159 +-  121.671 ns
       latency_nanotime: 15176.690 +- 1504.406 ns
```

Timers: Model Experiment

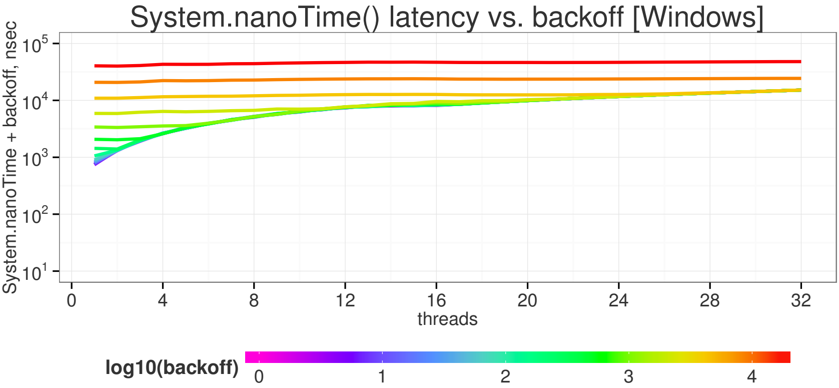
- But if `System.nanoTime()` is heavy and potentially non-scaling, then we run the system into oblivion?
- Let's figure out when it starts to Detroit:

```
@Param
int backoff;

@Benchmark
public long nanotime() {
  Blackhole.consumeCPU(backoff);
  return System.nanoTime();
}
```

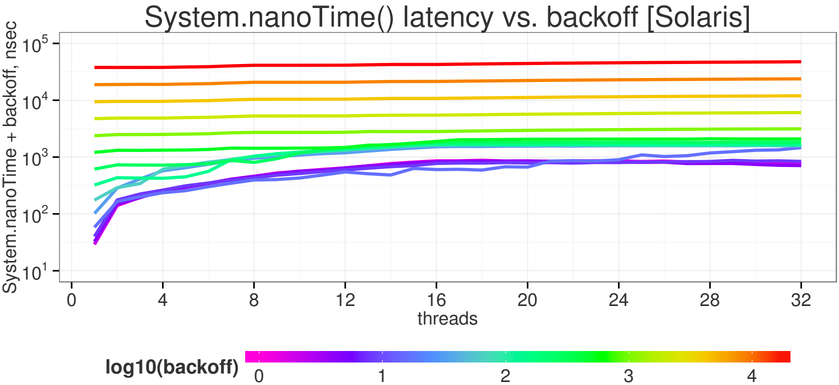
Timers: Seems OK [Linux]



Timers: Double U. Tee. Eff. [Windows]



Timers: Paying for Monotonicity [Solaris]



Timers: Typical Case [Mac OS X]

```
Java(TM) SE Runtime Environment, 1.8.0-b132
Java HotSpot(TM) 64-Bit Server VM, 25.0-b70
Mac OS X, 10.9.2, x86_64

Running with 1 threads and [-server]:
   granularity_nanotime: 1009.623 +-  2.140 ns
       latency_nanotime: 44.145 +-  1.449 ns

Running with 4 threads and [-server]:
   granularity_nanotime: 1044.703 +- 32.103 ns
       latency_nanotime: 56.111 +-  3.397 ns
```

Timers: Summing Up

`System.nanoTime` – is a new `String.intern`!

- Giving users the `nanoTime` is handing over a loaded gun
- `nanoTime` is may and should be used in selected cases, when you can foresee all disadvantages
- Most frequently, the direct measurement is not available, and we have to derive the models from the collateral evidence

Timers: Stop Kidding Already?

A picture of the dog that is derp-high on butterscotch, but still feeling OK

Our code blocks are heavy enough to keep `nanoTime()` granularity and latency at bay!

Omission: Heavy Benchmark is Heavy

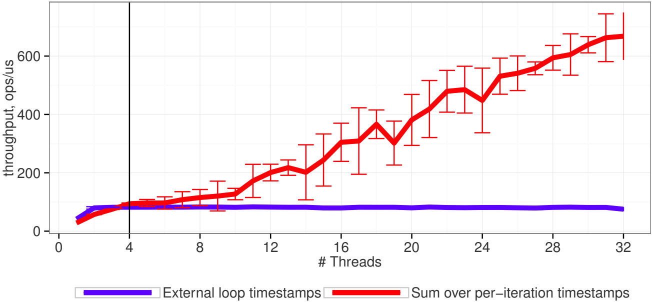
```
public long measure() {
  long ops = 0;
  long startTime = System.nanoTime();
  while(!isDone) {
    setup(); // want to skip this
    work();
    ops++;
  }
  return ops / (System.nanoTime() - startTime);
}
```

Omission: Measuring the Separate Block

```
public long measure() {
  long ops = 0;
  long realTime = 0;
  while(!isDone) {
    setup(); // skip this
    long time = System.nanoTime();
      work();
    realTime += (System.nanoTime() - time);
    ops++;
  }
  return ops / realTime;
}
```

Omission: Checking Empty setup()...

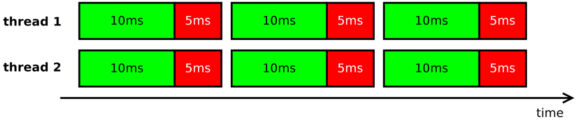
Measuring the throughput... it grows past the CPU count?!



Omission: Hint

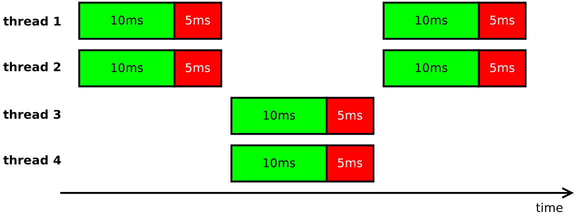
```
public long measure() {
  long ops = 0;
  long realTime = 0;
  while(!isDone) {
    setup(); // skip this
    long time = System.nanoTime();
      work();
    realTime += (System.nanoTime() - time);
    ops++;
    ...WHOOPS, WE DE-SCHEDULE HERE...
  }
  return ops / realTime;
}
```

Omission: Basic Example



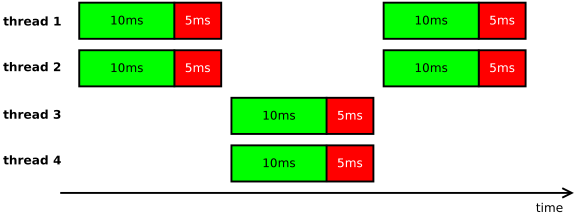
- Measuring the operation time, 10 ms/op on average ⇒ each i-th thread thinks its individual throughput is $\lambda_i =$ 100 ops/sec
- We have two threads, and therefore $\sum_{i=1}^{N} \lambda_i =$ 200 ops/sec

Omission: A Fistful of Threads More



- Each thread still believes $\lambda_i = 100$ ops/sec!
- Now we have four threads $\Rightarrow \sum_{i=1}^{N} \lambda_i = 400$ ops/sec

Omission: A Fistful of Threads More



- Each thread still believes $\lambda_i = 100$ ops/sec!
- Now we have four threads $\Rightarrow \sum_{i=1}^{N} \lambda_i =$ 400 ops/sec

Omission: Conclusion

A picture with a guy in a freezer,
waiting for beer to thaw,
while the alien spaceships
destroy the city around.

"Phillip J. Fry is experiencing the major safepoint event"

Timers skip the beats, and may grossly under/overestimate the durations.

- Every performance metric that includes time is at fault
- Very easy to blow up on overloaded systems
- Very easy to blow up when measurers coordinate with workload

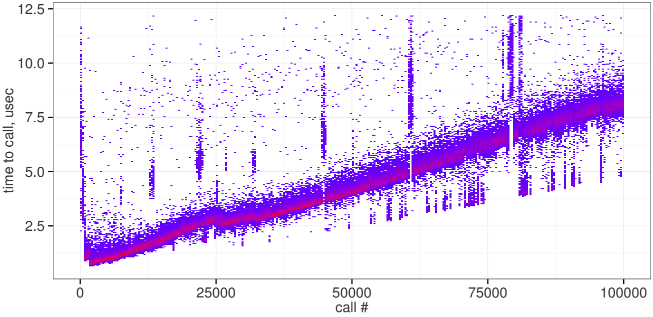
S.S.: (TGIF) Thank God It's Fibonacci

Is there a problem, officer?

```java
public class FibonacciGen {
  BigInteger n1 = ONE; BigInteger n2 = ZERO;

  @Benchmark
  public BigInteger next() {
    BigInteger cur = n1.add(n2);
    n2 = n1; n1 = cur;
    return cur;
  }
}
```

S.S.: Timing Each Call...

Whoops, this benchmark has no steady state, indeed:



S.S.: Pitfalls

No steady state – can **not** use the time-based benchmarks!
The longer we measure, the «slower» the result appears:

duration, sec	throughput, us/op
1	5.013 ± 0.006
2	7.087 ± 0.009
4	10.021 ± 0.017
8	14.159 ± 0.010

S.S.: Pick Your Poison

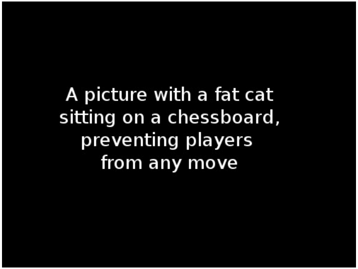
Time-based benchmarks:

- Measuring in God knows what conditions
- How should one compare two implementations?
  (if you are lucky, and your performance model is linear...)

Work-based benchmarks:

- Burning ourselves with timers latency/granularity
- Burning ourselves with omission
- Burning ourselves with transients

S.S.: Conclusion



«The only winning move is not to play at all»

Non-steady state benchmarks force you to choose between all the bad options.

Non-steady state benchmarks are the large P.I.T.A!

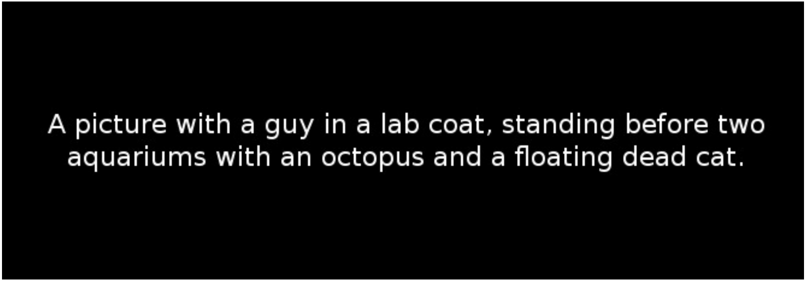
S.S.: Palliative Relief

Measure in large batches!

```
@Setup(Level.Iteration)
public void setup() {
  n1 = BigInteger.ZERO; n2 = BigInteger.ONE;
}

@Benchmark
@Measurement(batchSize = 5000)
public BigInteger next() {
  BigInteger cur = n1.add(n2);
  n2 = n1; n1 = cur;
  return cur;
}
```

Engineering

Engineering: Comparisons



A picture with a guy in a lab coat, standing before two aquariums with an octopus and a floating dead cat.

You want your results to be comparable.

- Every tiny little uncontrolled detail is a free variable
- Libraries are the large complexes of tiny details
- Language runtimes are **galaxies** of tiny details

Engineering: Story

This is a weird story of Java vs. Scala comparison coming from StackOverflow, where people are bound to that believe tail-recursion optimization is the best thing that happened in computer science since the sliced bread.

Complete story and narrative is here:

`http://shipilev.net/blog/2014/java-scala-divided-we-fail/`

Engineering: Scala's @tailrec

```
@tailrec private def
isDivisible(v: Int, d: Int, l: Int): Boolean = {
  if (d > l) true
  else (v % d == 0) && isDivisible(v, d + 1, l)
}

@Benchmark
def test(): Int = {
  var v = 10
  while(!isDivisible(v, 2, l))
    v += 2
  v
}
```

Engineering: Java's absence-of-tailrec

```
private boolean isDivisible(int v, int d, int l)
  if (d > l) return true;
  else
    return (v % d == 0) && isDivisible(v, d+1, l)
}

@Benchmark
public int test() {
  int v = 10;
  while(!isDivisible(v, 2, l))
    v += 2;
  return val;
}
```

Engineering: Measuring

```
 Benchmark      lim         Score  Score error   Units
-----------------------------------------------------
 ScalaBench      1         0.002        0.000   us/op
 ScalaBench      5         0.494        0.005   us/op
 ScalaBench     10        24.228        0.268   us/op
 ScalaBench     15      3457.733       33.070   us/op
 ScalaBench     20   2505634.259    15366.665   us/op
 JavaBench       1         0.002        0.000   us/op
 JavaBench       5         0.252        0.001   us/op
 JavaBench      10        12.782        0.325   us/op
 JavaBench      15      1615.890        7.647   us/op
 JavaBench      20   1053187.731    20502.217   us/op
```

Engineering: Profiling Java

```
Result: 12.719 +-(99.9%) 0.284 us/op [Average]

....[Thread state distributions].......................
 91.3%        RUNNABLE
  8.7%        WAITING

....[Thread state: RUNNABLE]...........................
 58.0%  63.5% n.s.JavaBench.isDivisible
 32.9%  36.1% n.s.JavaBench.test

....[Thread state: WAITING]............................
  8.7% 100.0% <irrelevant>
```

Engineering: Profiling Scala

```
Result: 24.076 +-(99.9%) 0.728 us/op [Average]

....[Thread state distributions]......................
 91.4%        RUNNABLE
  8.6%        WAITING

....[Thread state: RUNNABLE]..........................
 90.6%  99.1% n.s.ScalaBench.test
  0.9%   0.9% n.s.generated.ScalaBench_test.test_avgt_jmhLo

....[Thread state: WAITING]...........................
  8.6% 100.0% <irrelevant>
```

Engineering: Coarse-grained profilers

Coarse-grained (method-level) profilers are useless in diagnosing the problems in nano- and micro-benchmarks.

Additional penalty points if they are sampling at safepoints.

Engineering: JMH perfasm

```
java -jar benchmarks.jar ... -prof perfasm
```

Surprisingly easy to marry these three things:

1. Linux `perf` provides light-weight PMU sampling
2. JVM debug info maps events back to VM methods
3. `-XX:+PrintAssembly` maps events back to Java code

Actually, there are lots of good profilers already, but most of the time you don't need «big guns» to quickly analyze benchmarks.

Engineering: Hottest thing in Scala

One true and solid x86 division:

```
 clocks    insns    code
-------------------------------------------------------
; n.s.g.ScalaBench_test::test_avgt_jmhLoop
...
  0.27%     0.17%   cltd
  2.24%    17.26%   idiv    %ecx
 77.99%    66.44%   test    %edx,%edx
...
```

How can you possibly be 2x faster than this?

Engineering: Hottest thing in Java

```
 clocks    insns   code
-------------------------------------------------------
; n.s.JavaBench::isDivisible
...
  1.68%     2.76%  cltd
  0.06%     0.16%  idiv    %ecx
 27.59%    36.37%  test    %edx,%edx
 ...
  0.04%            cltd
                   idiv    %r10d
 12.24%     1.54%  test    %edx,%edx
 ...
  0.01%            callq   <recursive-call>
```

Engineering: Second hottest thing in Java

```
 clocks    insns    code
------------------------------------------------------
; n.s.g.JavaBench_test::test_avgt_jmhLoop
 ...
  1.34%    0.21%    imul   $0x55555556,%rdx,%rdx
  1.25%    0.20%    sar    $0x20,%rdx
  1.15%    2.36%    mov    %edx,%esi
  0.95%    1.51%    sub    %r10d,%esi            ; irem
 ...
```

[2] http://www.hackersdelight.org/divcMore.pdf

Engineering: Second hottest thing in Java

```
 clocks    insns    code
------------------------------------------------------
; n.s.g.JavaBench_test::test_avgt_jmhLoop
 ...
  1.34%    0.21%    imul    $0x55555556,%rdx,%rdx
  1.25%    0.20%    sar     $0x20,%rdx
  1.15%    2.36%    mov     %edx,%esi
  0.95%    1.51%    sub     %r10d,%esi            ; irem
 ...
```

Beautiful trick of substituting the remainder with constant multiplication and binary shift! [2]

[2] http://www.hackersdelight.org/divcMore.pdf

Engineering: Quick Explanation

```
// inlines twice, specializes for d={2,3}
private boolean isDivisble(int v, int d, int l) {
  ...
  return (v % d == 0) && isDivisble(v, d+1, l);
}

@Benchmark
public int test() {
  int v = 10;
  while(!isDivisble(v, 2, l))
    v += 2;
  return val;
}
```

Engineering: Make «d» unpredictable

```
 Benchmark      lim         Score  Score error  Units
-----------------------------------------------------
 ScalaBench       1         0.002        0.000  us/op
 ScalaBench       5         0.489        0.002  us/op
 ScalaBench      10        23.777        0.116  us/op
 ScalaBench      15      3379.870        5.737  us/op
 ScalaBench      20   2468845.944     2413.573  us/op
 JavaBench        1         0.003        0.000  us/op
 JavaBench        5         0.465        0.001  us/op
 JavaBench       10        22.989        0.095  us/op
 JavaBench       15      3453.116       16.390  us/op
 JavaBench       20   2518726.451     4374.482  us/op
```

Engineering: Conclusion

«Days since the last benchmarking accident: 0» (@gvsmirnov)

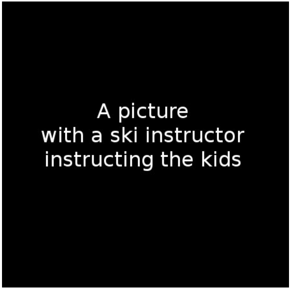
Benchmarks without analysis make me a really sad panda.

You show me nice charts: Language A vs. Language B, Nashorn vs. Rhino, Graal vs. C2, etc, and all I see is

BAYESIAN NOISE

Fin

Fin: Conclusion



«If you don't analyze the benchmarks, you've gonna waste a good time»

The superficial conclusions almost always feed on existing biases, and are almost always wrong.

Benchmarks are for understanding the Reality, not for reinforcing your prejudices about the Universe.