Measuring Performance with JMH

the Java Microbenchmark Harness

HJUG - July 2016

William Price
Software Architect
PROS, Inc.

$ whoami

- **Life-long code monkey**
  BASIC C C# Java JS PHP VB (and others)
- **Java**
  @since J2SE 1.3
- **PROS, Inc.**
  2005 - present
- **LPTV broadcasting / videography**
  Uncompressed, full 1080p60 HD-SDI is ~3Gbps
- **DSLR amateur photography**
  Canon 7D - final launch of space shuttle Discovery STS-133
- **How things work**

Online resources

Slides and source code available on GitHub:

wrprice / hjug-jmh-demo

http://openjdk.java.net/projects/code-tools/jmh/

https://github.com/melix/jmh-gradle-plugin

https://gradle.org/

Coming up...

Resist the Temptation

Misleading Measurements

Introducing JMH

Live Demo

Examples & Discussion

So say we all

"We should forget about small efficiencies, say about **97%** of the time: **premature optimization** is the root of all evil. Yet we should not pass up our opportunities in that critical **3%**."

– Donald Knuth
"Structured Programming with Goto Statements"
ACM Journal Computing Surveys, Vol 6, No. 4, Dec. 1974. p.268.

That critical 3%

- Someone complained, “It's slow.”
- You can reproduce the scenario.
- The scenario is critical enough.
- You profiled it.
- Profiling showed it is actually slow.
- You identified the worst bottleneck.
- And you have a possible alternative.

Full disclosure

Novice

A little knowledge is a dangerous thing

Here be dragons

* not to scale

Expert

Stay tuned...

Resist the Temptation

Misleading Measurements

Introducing JMH

Live Demo

Examples & Discussion

This should be easy

Measure how long it takes to compute:

$$\ln(\pi)$$

in Java: `Math.log(Math.PI)`

This should be easy

```
@Test
public void singleShotMillis() {
  long start = System.currentTimeMillis();
  Math.log(Math.PI);
  long end = System.currentTimeMillis();
  System.out.printf(
    “%,d ms per computation%n”,
    end - start);
}
```

1 operation
0 milliseconds

This should be easy

```
@Test
public void singleShotMillisNanos() {
  long start = System.nanoTime();
  Math.log(Math.PI);
  long end = System.nanoTime();
  System.out.printf(
    “%,d ns per computation%n”,
    end - start);
}
```

1 operation
220,394 nanoseconds
0.22 ms

Loop to the rescue

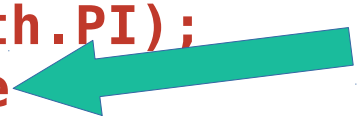
```
@Test
public void loopItNanos() {
  int iterations = 9_000;
  long start = System.nanoTime();
  for (int i = iterations; i > 0; i--) {
    Math.log(Math.PI);
  }
  long end = System.nanoTime();
  System.out.printf(
    “%,d ns for %,d iterations = %4.1f ns/computation%n”,
    /* … */);
}
```

442,613 nanoseconds
9,000 iterations
49.2 ns/iteration

Amortize the loop

```
@Test
public void loopAmortized() {
  int iterations = 9_000;
  long totalNanos = doLoopOf15(iterations);
  System.out.printf(/* … */);
}

private long doLoopOf15(int iterations) {
  long start = System.nanoTime();
  for (int i = iterations; i > 0; i--) {
    Math.log(Math.PI);
    // … 13 more
    Math.log(Math.PI);
  }
  long end = System.nanoTime();
  return end - start;
}
```



also arbitrary

```
4,904,175  nanoseconds
    9,000  iterations, 15 ops/iter
   36.3 ns/op
```

Repeat multiple times

```
@Test
public void loopTheLoop() {
  int runs = 30;
  int iterationsPerRun = 9_000;
  long totalRunNanos = 0;
  for (int r = runs; r > 0; r--) {
    totalRunNanos += doLoopOf15(iterationsPerRun);
  }
  System.out.printf(/* … */);
}
```

```
27,108,442  nanoseconds
   270,000  iterations, 15 ops/iter
       6.7  ns/op
```

Wait, what???

```
@Test
public void loopTheLoopALot() {
  int runs = 1_000_000;
  int iterationsPerRun = 1_000;
  long totalRunNanos = 0;
  for (int r = runs; r > 0; r--) {
    totalRunNanos += doLoopOf15(iterationsPerRun);
  }
  System.out.printf(/* … */);
}
```

47,096,670 nanoseconds
1.0e9 iterations, 15 ops/iter
0.003 ns/op

This should be easy, *but*

hand-writing
meaningful benchmarks

is hard.

This should be easy, *but*

hand-writing
~~meaningful benchmarks~~
benchmarks that *mean anything at all*

is hard.

This approach fails to consider

- Interpreted vs JIT-compiled execution
- Dead code elimination
- Constant folding
- Method inlining
- Instruction reordering
- Loop unrolling & other tricks
- JVM code cache
- Poly/mega-morphic call sites
- Multi-threading / contention
- Modern CPU hardware behavior

On next...

Resist the Temptation

Misleading Measurements

Introducing JMH

Live Demo

Examples & Discussion

About JMH

- **Written by the OpenJDK maintainers**
- **Magically hides the complicated part of JVM measurement**
- **Generates the *real* benchmarks from simple, annotated code**
- **Benchmark runner, computes statistics**
- **Can support any JVM language**
- **Programmatic control via API**
- **Pluggable profilers (e.g. GC)**

Gradle project structure

```
./ build.gradle
   gradle.properties
   settings.gradle

   src/
        main/
              java/
              resources/
        test/
              java/
              resources/
        jmh/
              java/
              resources/
```

A minimal build.gradle

```
plugins {
  id 'me.champeau.gradle.jmh' version '0.3.0'
}

apply plugin: 'java'

repositories {
  mavenCentral()
}

jmh {
  jmhVersion '1.12'
  if (project.hasProperty('jmh.benchmark')) {
    include = \
      '.*' + project.property('jmh.benchmark') + '.*'
  }
}
```

Gradle tasks

```
$ cd $PROJECT_DIR
$ ./gradlew clean
$ ./gradlew build    # compile, jar, test main code
$ ./gradlew jmh      # runs ALL benchmarks
$ ./gradlew jmh -Pjmh.benchmark=BenchmarkClassName
```

Hello, benchmark

```
// src/jmh/java/org/hjug/jmhdemo/JmhPerfTest.java
package org.hjug.jmhdemo;

import java.util.concurrent.TimeUnit;
import org.openjdk.jmh.annotations.*;

@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
public class JmhPerfTest {

    @Benchmark
    public void version1() {
        Math.log(Math.PI);
    }
}
```

Hmmm...

```
// src/jmh/java/org/hjug/jmhdemo/JmhPerfTest.java
package org.hjug.jmhdemo;

import java.util.concurrent.TimeUnit;
import org.openjdk.jmh.annotations.*;

@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
public class JmhPerfTest {

    @Benchmark
    public void nop() {
        // nothing to do
    }
}
```

Benchmark attempt #2

```
// src/jmh/java/org/hjug/jmhdemo/JmhPerfTest.java
package org.hjug.jmhdemo;

import java.util.concurrent.TimeUnit;
import org.openjdk.jmh.annotations.*;

@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
public class JmhPerfTest {

    @Benchmark
    public double version2() {
        return Math.log(Math.PI);
    }
}
```

Hmmm...

```
// src/jmh/java/org/hjug/jmhdemo/JmhPerfTest.java
package org.hjug.jmhdemo;

import java.util.concurrent.TimeUnit;
import org.openjdk.jmh.annotations.*;

@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
public class JmhPerfTest {

    @Benchmark
    public double constant() {
        return 0.0;
    }
}
```

Benchmark attempt #3

```
// src/jmh/java/org/hjug/jmhdemo/JmhPerfTest.java
package org.hjug.jmhdemo;

import java.util.concurrent.TimeUnit;
import org.openjdk.jmh.annotations.*;

@State(Scope.Benchmark)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
public class JmhPerfTest {

    private double pi = Math.PI;

    @Benchmark
    public double version3() {
        return Math.log(pi);
    }
}
```

Alternative to #3

```
// src/jmh/java/org/hjug/jmhdemo/JmhPerfTest.java
package org.hjug.jmhdemo;

import java.util.concurrent.TimeUnit;
import org.openjdk.jmh.annotations.*;
import org.openjdk.jmh.infra.Blackhole;

@State(Scope.Benchmark)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
public class JmhPerfTest {

    private double pi = Math.PI;

    @Benchmark
    public void version3a( Blackhole bh ) {
        bh.consume( Math.log(pi) );
    }
}
```

You're watching...

Resist the Temptation

Misleading Measurements

Introducing JMH

Live Demo

Examples & Discussion

Common annotations

```
@BenchmarkMode(Mode)
@BenchmarkMode({ mode1, mode2, … modeN })
  // Mode.Throughput      Mode.SingleShotTime
  // Mode.AverageTime     Mode.SampleTime

@OutputTimeUnit(TimeUnit) // sec, us, ns, etc.

@State(Scope)
  // Scope.Benchmark  Scope.Thread  Scope.Group

@Param({ “val1”, “val2”, … “valN” })

@Setup
@TearDown
```

Common annotations

```
@Fork(count, warmupCount, jvmArgs, ...)
@Measurement(batchSize, iterations, duration, unit)
@Warmup(batchSize, iterations, duration, unit)
@Threads(int)
@Group(String tag)
@GroupThreads(int)
```

and more, of course!

JMH “DO”s

- Return results or consume with a Blackhole
- Read inputs from state fields, and use @Param for variations with different values
- Limit @Benchmark methods to exactly the code under test
- Use @Setup and @TearDown methods to manage state outside of the measured code
- Run with multiple threads if multi-thread performance is a concern

JMH “DON'T”s

- **Don't assume the JVM runs code in the same order or exactly as written in a source file**
- **Don't put loops in your @Benchmark unless the loop is part of the code under test**
- **Don't run too few warm-up iterations such that the target code never gets JIT-compiled**
- **Don't measure non-memory I/O (too many factors outside of JVM)**
- **Don't forget that results will vary by machine, architecture, OS, kernel, JVM, etc.**

Later tonight...

Resist the Temptation

Misleading Measurements

Introducing JMH

Live Demo

Examples & Discussion

From last month's meeting

```
// original:

for (int i = 0; i < limit; i++) {
  for (int j = 0; j < limit; j++) {
    if (i < j) {
      // do some work
    }
  }
}

// optimized:

for (int j = 0; j < limit; j++) {
  for (int i = 0; i < j; i++) {
    // do some work
  }
}
```

1293.7 runs/sec
limit = 1024

So why was this faster in the real application?

6069.8 runs/sec
limit = 1024

Because the benchmark was measuring different “work”!

Mistake: oversimplifying

```
// optimized benchmark:
long sum = 0;
for (int j = 0; j < limit; j++) {
  for (int i = 0; i < j; i++) {
    sum += i + j;
  }
}
return sum;
```

Commutative operation, only involves local vars

```
// optimized production code:
double sum = 0;
for (int j = 0; j < limit; j++) {
  for (int i = 0; i < j; i++) {
    sum += matrixDoubleArray[i][j];
  }
}
return sum;
```

Heap memory access, could be very large?

Optimization swapped inner/outer loop variables

Side effect: accessing row-major array in column-major order

Looped 2-D array access

Access / Size	256 x 256	1024 x 1024	20480 x 20480
Original	18,817.3	1,076.6	3.2
Column-major	**30,938.8**	164.6	0.1
Row-major	**35,414.6**	**1,851.3**	**5.3**

outer loops per second, higher is better

Fact check

Common knowledge states that Java reflection is slow.

Various ways to invoke a method

Java 8	Normal	Exceptional
Direct (baseline)	**43.870**	**3.208**
getMethod().invoke()	226.993	not measured
Method.invoke()	48.999	1406.453
MethodHandle	48.165	291.574
Lambda	44.423	3.732

nanoseconds per call, lower is better

Thank you for attending.

/* DO NOT READ THIS LINE */