Java Microbenchmark Harness (the lesser of two evils)

Aleksey Shipilev
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: Why would we even listen to this guy?

- ex-«Intel, Apache Harmony performance geek»
- ex-«SPEC tech. representative for Oracle»
- in-«Oracle/OpenJDK performance geek»
- Guilty of:
  1. Lots of shameful internal stuff
  2. SPECjbb2013
  3. Concurrency improvements (e.g. @Contended)
  4. Java Concurrency Stress Tests (jcstress)
  5. Java Microbenchmark Harness (jmh)
Intro: Obligatory JVMLS reference

This talk was also well received at JVMLS 2013.
Intro: Obligatory JVMLS reference

This talk was also well received at JVMLS 2013.
Basics
Basics: Benchmarks are experiments

- Computer Science → Software Engineering
  - Build software to meet functional requirements
  - Mostly don’t care about HW and data specifics
  - Abstract and composable, «formal science»

- Software Performance Engineering
  - «Real world strikes back!»
  - Exploring complex interactions between hardware, software, and data
  - Based on empirical evidence, i.e. «natural science»
Basics: Experimental Control

Any experiment requires the control

- Sometimes, just a few baseline measurements
- Sometimes, vast web of support experiments
Basics: Experimental Control

Any experiment requires the control

- Sometimes, just a few baseline measurements
- Sometimes, vast web of support experiments
- Software-specific: peek under the hood!
Basics: Experimental Control

Any experiment requires the control

- Sometimes, just a few baseline measurements
- Sometimes, vast web of support experiments
- Software-specific: peek under the hood!

Experiments assume the hypothesis (model), against which we do the control
Basics: Common Wisdom

Microbenchmarks are bad
Basics: Common Wisdom

Microbenchmarks are bad
Basics: The Root Cause

«Premature optimization is the root of all evil»
(Khuth, 1974)
Basics: The Root Cause

«Premature Optimization
is the root of all evil»
(Shipilev, 2013)
Basics: Evil Optimizations

Optimizations distort the performance models!

- Applied in «common» (>50%) cases
- Unclear interdependencies
- «Black box» abstraction fails big time
Basics: Evil Optimizations

Optimizations distort the performance models!

- Applied in «common» (>50%) cases
- Unclear interdependencies
- «Black box» abstraction fails big time

Examples:

- «new MyObject()»
Basics: Evil Optimizations

Optimizations distort the performance models!
- Applied in «common» (>50%) cases
- Unclear interdependencies
- «Black box» abstraction fails big time

Examples:
- «new MyObject()»: allocated in TLAB? allocated in LOB? scalarized? eliminated?
Basics: Know Thy Optimizations

Understanding the performance model is the road to awe

- This is the endgame result for benchmarking
- Benchmarking is for exploring the performance models (which also helps to get better at benchmarking)
- Every new optimization $\Rightarrow$ new hassle for everyone
Basics: Benchmarks vs. Optimization

**Ground Rule**

Benchmarking is the (endless) fight against the optimizations

**Collorary**

Benchmarking harness #1 priority: managing the optimizations
Basics: JMH

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

- Works around pitfalls common to HotSpot/OpenJDK
- Bugs are fixed as VM evolves, or we discover more
- We (perfteam) validate benches by rewriting them with JMH
- Facilitates peer review
Basics: JMH API Sneak Peek

Let users declare the benchmark body:

```java
@GenerateMicroBenchmark
generateMicroBenchmark
public void helloWorld() {
    // do something here
}
```

...then generate lots of supporting synthetic code around that body.

(At this point, simply generating the auxiliary subclass works fine, but it is limiting for some cases)
Basics: Getting the units right

*Benchmarks:

- micro:
Basics: Getting the units right

*Benchmarks:*

- **micro:** 1...1000 us, single webapp request
Basics: Getting the units right

*Benchmarks:

- **micro**: 1...1000 us, single webapp request
- **nano**: 1...1000 ns, single operations
Basics: Getting the units right

*Benchmarks:

- milli: 1...1000 ms, SPECjvm98, SPECjbb2005
- micro: 1...1000 us, single webapp request
- nano: 1...1000 ns, single operations
Basics: Getting the units right

*Benchmarks:

- **kilo**: 1...1000 s, SPECjvm2008, SPECjbb2013
- **milli**: 1...1000 ms, SPECjvm98, SPECjbb2005
- **micro**: 1...1000 us, single webapp request
- **nano**: 1...1000 ns, single operations
Basics: Getting the units right

*Benchmarks:*

- **kilo:** > 1000 s, Linpack
- **milli:** 1...1000 ms, SPECjvm98, SPECjbb2005
- **micro:** 1...1000 us, single webapp request
- **nano:** 1...1000 ns, single operations
Basics: Getting the units right

*Benchmarks:*

- **kilo:** > 1000 s, Linpack
- **_____:** 1...1000 s, SPECjvm2008, SPECjbb2013
- **milli:** 1...1000 ms, SPECjvm98, SPECjbb2005
- **micro:** 1...1000 us, single webapp request
- **nano:** 1...1000 ns, single operations
- **pico:** 1...1000 ps, pipelining
Basics: ...increaseth sorrow

Benchmarks amplify all the effects visible at the same scale.

- Millibenchmarks are not really hard
- Microbenchmarks are challenging, but OK
- Nanobenchmarks are the damned beasts!
- Picobenchmarks...
Basics: Warmup

Definition

«Warmup» = waiting for the transient responses to settle down

Every online optimization requires warmup

JIT compilation is *NOT* the only online optimization

Ok, «Watch -XX:+PrintCompilation»?
Basics: Warmup

Definition

«Warmup» = waiting for the transient responses to settle down
Basics: Warmup

Definition

«Warmup» = waiting for the transient responses to settle down

- Every online optimization requires warmup
Basics: Warmup

Definition

«Warmup» = waiting for the transient responses to settle down

- Every online optimization requires warmup
- JIT compilation is NOT the only online optimization
Basics: Warmup

Definition

«Warmup» = waiting for the transient responses to settle down

- Every online optimization requires warmup
- JIT compilation is **NOT** the only online optimization
- Ok, «Watch -XX:+PrintCompilation»?
Basics: Warmup plateaus

|DK8b83 + nashorn 2013-04-08, Octane:DeltaBlueBench performance over iterations|
Major pitfalls
Major pitfalls: The Goal

The goal for this section is to **scare you away from**:

- (blindly) building the benchmark harnesses
- (blindly) trusting the benchmark harnesses
- (blindly) trusting the benchmarks
- (blindly) being generally blind about benchmarks
Let us run the empty benchmark.
System reports 4 online CPUs.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Ops/nsec</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.06 ± 0.10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5.72 ± 0.10</td>
<td>1.87 ± 0.03</td>
</tr>
<tr>
<td>4</td>
<td>5.87 ± 0.02</td>
<td>1.91 ± 0.03</td>
</tr>
</tbody>
</table>
System: Optimization Quiz (A)

Let us run the empty benchmark.
System reports 4 online CPUs.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Ops/nsec</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.06 ± 0.10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5.72 ± 0.10</td>
<td>1.87 ± 0.03</td>
</tr>
<tr>
<td>4</td>
<td>5.87 ± 0.02</td>
<td>1.91 ± 0.03</td>
</tr>
</tbody>
</table>

Question 1: Why no change for 2 → 4 threads?
System: Optimization Quiz (A)

Let us run the empty benchmark.
System reports 4 online CPUs.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Ops/nsec</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.06 ± 0.10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5.72 ± 0.10</td>
<td>1.87 ± 0.03</td>
</tr>
<tr>
<td>4</td>
<td>5.87 ± 0.02</td>
<td>1.91 ± 0.03</td>
</tr>
</tbody>
</table>

- Question 1: Why no change for 2 → 4 threads?
- Question 2: Why only 1.87x change for 1 → 2 threads?
System: Power management

Running dummy benchmark,
+ Down-clocking to 2.0 GHz

<table>
<thead>
<tr>
<th>Threads</th>
<th>Ops/nsec</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.97 ± 0.02</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3.94 ± 0.05</td>
<td>2.00 ± 0.02</td>
</tr>
<tr>
<td>4</td>
<td>4.03 ± 0.04</td>
<td>2.04 ± 0.02</td>
</tr>
</tbody>
</table>
System: Power management

Many subsystems balance power-vs-performance
(Ex.: cpufreq, SpeedStep, Cool&Quiet, TurboBoost)

- **Downside**: breaks the homogeneity of time
- **Remedy**: disable power management, fix CPU clock frequency
- **JMH Remedy**: run longer, do not park threads
System: OS Schedulers

OS schedulers balance affinity-vs-power
(Ex.: Solaris schedulers, Linux power-efficient taskqueues)

- **Downside**: breaks the processing symmetry
- **Remedy**: tight up scheduling policies
- **JMH Remedy**: run longer, do not park threads
System: Time Sharing

Time sharing systems balance utilization
(Ex.: everywhere)

- **Downside**: thread start/stop is not instantaneous, thread run time is non-deterministic, the load is non-uniform
- **Remedy**: make sure everything runs before measuring
- **JMH Remedy**: «bogus iterations»
System: Time Sharing, #2

JMH provides the remedy – bogus iterations:
public void measure() {
    long startTime = System.nanoTime();
    while (!isDone) {
        work();
    }
    println(System.nanoTime() - startTime);
}
«Is there a problem, officer?»

```java
public void measure () {
    long realTime = 0;
    while (!isDone) {
        setup(); // skip this
        long time = System.nanoTime();
        work();
        realTime += (System.nanoTime() - time);
    }
    println(realTime);
}
```
System: Time Sharing, Quiz (B)

Measuring the reciprocal throughput via total/iteration time:

The throughput grows past the CPU count – WTF?!
public void measure() {
  long startTime = System.nanoTime();
  long realTime = 0;
  while(!isDone) {
    setup(); // skip this
    long time = System.nanoTime();
    work();
    realTime += (System.nanoTime() - time);
    ...WHOOPS, WE DE-SCHEDULE HERE...
  }
  println(realTime);
  println(System.nanoTime() - startTime);
}
System: Time Sharing

Time sharing gives the illusion of running multiple threads simultaneously

- **Downside**: this illusion is broken for performance
- **Remedy**: do **NOT** overload the system!
- **JMH Remedy**: big red warning sign
VM: Optimization Quiz (C)

```java
@GenerateMicroBenchmark
public void baseline() {
    0.5 ± 0.1 ns
}

@GenerateMicroBenchmark
public void measureWrong() {
    Math.log(x);
    0.5 ± 0.1 ns
}

@GenerateMicroBenchmark
public double measureRight() {
    return Math.log(x);
    34.0 ± 1.0 ns
}
```
VM: Dead-code elimination

Compilers are good at eliminating the redundant code.

- **Downside**: can remove (parts of) the benchmarked code
- **Remedy**: consume the results, depend on the results, provide the side effect
- **JMH Remedy**: API support
VM: Avoiding dead-code elimination

DCE is somewhat easy to avoid for primitives:

- Primitives have binary combinators!
- Caveat #1: Combinator cost?
- Caveat #2: Low-range primitives enable speculation (boolean)

```java
int sum = 0;
for (int i = 0; i < 100; i++) {
    sum += op(i);
}
return sum; // consume in caller
```
VM: Avoiding dead-code elimination

DCE is hard to avoid for references:

- Caveat #1: Fast object combinator, anyone?
- Caveat #2: Need to escape object to limit thread-local optimizations.
- Caveat #3: Publishing the object ⇒ reference heap write ⇒ store barrier
VM: DCE, Blackholes

JMH provides «Blackholes». Blackhole consumes the value.

```java
class Blackhole {
    void consume(int v) { doMagic(v); }
    void consume(Object o) { doMagic(o); }
}
```

- Returns are implicitly fed into the blackhole
- User can request additional blackhole ⇒ heap writes again, dammit!
VM: Avoiding dead-code elimination, Blackholes

Relatively easy for primitives:

class Blackhole {
    static volatile Wrapper NULL;
    volatile int g1 = 1, g2 = 2;

    void consume(int v) {
        if (v == g1 & v == g2) {
            NULL.field = 0; // implicit NPE
        }
    }
}
VM: DCE, Blackholes

Harder for references:

class Blackhole {
    Object sink;
    int prngState;
    int prngMask;

    void consume(Object v) {
        if ((next(prngState) & prngMask) == 0) {
            sink = v; // store barrier here
            prngMask = (prngMask << 1) + 1;
        }
    }
}
VM: Optimization Quiz (D)

```java
@GenerateMicroBenchmark
class Benchmark {
    public void baseline() {
        // Benchmark baseline method
    }
}

@GenerateMicroBenchmark
class Benchmark {
    public double measureWrong() {
        return Math.log(42);
    }
}

private double x = 42;
@GenerateMicroBenchmark
class Benchmark {
    public double measureRight() {
        return Math.log(x);
    }
}
```

0.5 ± 0.1 ns

1.0 ± 0.1 ns

34.0 ± 1.0 ns
VM: Constant folding, etc.

Compilers are good at partial evaluation\textsuperscript{1}

- **Downside**: can remove (parts of) the benchmarked code
- **Remedy**: make the sources unpredictable
- **JMH Remedy**: API support

\textsuperscript{1}All right, all right! It is not really the PE.
JMH prevents load commoning across @GMB calls

double x;

@GenerateMicroBenchmark
double doWork() {
    doStuff(x);
}

volatile boolean done;

void doMeasure() {
    while (!done) {
        doWork();
    }
}

(i.e. read everything from heap ⇒ you are good!)
VM: DCE, CSE... Same thing!

Losing either a source or a sink loses the part of the benchmark. Silently.
VM: DCE, CSE… Same thing!

Losing either a source or a sink loses the part of the benchmark. Silently.
VM: DCE, CSE… Same thing!

Losing either a source or a sink loses the part of the benchmark. Silently.
VM: Optimization Quiz (E)

// changing N, will performance differ?
static int N = 100;

@GenerateMicroBenchmark
public int test() { return doWork(N); }

int x = 1, y = 2;

private int doWork(int reps) {
    int s = 0;
    for (int i = 0; i < reps; i++)
        s += (x + y);
    return s;
}
### VM: Optimization Quiz (E), #2

<table>
<thead>
<tr>
<th>N</th>
<th>ns/call</th>
<th>ns/add</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$1.5 \pm 0.1$</td>
<td>$1.5 \pm 0.1$</td>
</tr>
<tr>
<td>10</td>
<td>$2.0 \pm 0.1$</td>
<td>$0.1 \pm 0.01$</td>
</tr>
<tr>
<td>100</td>
<td>$2.7 \pm 0.2$</td>
<td>$0.05 \pm 0.02$</td>
</tr>
<tr>
<td>1000</td>
<td>$68.8 \pm 0.9$</td>
<td>$0.07 \pm 0.01$</td>
</tr>
<tr>
<td>10000</td>
<td>$410.3 \pm 2.1$</td>
<td>$0.04 \pm 0.01$</td>
</tr>
<tr>
<td>100000</td>
<td>$3836.1 \pm 40.6$</td>
<td>$0.04 \pm 0.01$</td>
</tr>
</tbody>
</table>

Which one to believe?

0.04 ns/add $\Rightarrow$ 25 adds/ns $\Rightarrow$ GTFO!
VM: Optimization Quiz (E), #2

<table>
<thead>
<tr>
<th>N</th>
<th>ns/call</th>
<th>ns/add</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.5 ± 0.1</td>
<td>1.5 ± 0.1</td>
</tr>
<tr>
<td>10</td>
<td>2.0 ± 0.1</td>
<td>0.1 ± 0.01</td>
</tr>
<tr>
<td>100</td>
<td>2.7 ± 0.2</td>
<td>0.05 ± 0.02</td>
</tr>
<tr>
<td>1000</td>
<td>68.8 ± 0.9</td>
<td>0.07 ± 0.01</td>
</tr>
<tr>
<td>10000</td>
<td>410.3 ± 2.1</td>
<td>0.04 ± 0.01</td>
</tr>
<tr>
<td>100000</td>
<td>3836.1 ± 40.6</td>
<td>0.04 ± 0.01</td>
</tr>
</tbody>
</table>

Which one to believe?

0.04 ns/add ⇒ 25 adds/ns ⇒ GTFO!
Loop unrolling greatly expands the scope of optimizations

- **Downside**: assume the single loop iteration is $M$ ns. After unrolling the effective cost is $\alpha M$ ns, where $\alpha \in [0; +\infty)$
- **Remedy**: avoid unrollable loops, limit the effect of unrolling
- **JMH Remedy**: proper handling for CSE/DCE nils loop unrolling effects
interface M {
    void inc();
}

abstract class AM implements M {
    int c;
    void inc() {
        c++;
    }
}

class M1 extends AM {}
class M2 extends AM {}/
VM: Optimization Quiz (F), #2

M m1 = new M1();
M m2 = new M2();

@GenerateMicroBenchmark
public void testM1() { test(m1); }

@GenerateMicroBenchmark
public void testM2() { test(m2); }

void test(M m) {
    for (int i = 0; i < 100; i++)
        m.inc();
}
VM: Optimization Quiz (F), #3

<table>
<thead>
<tr>
<th>test</th>
<th>ns/op</th>
</tr>
</thead>
<tbody>
<tr>
<td>testM1</td>
<td>4.6 ± 0.1</td>
</tr>
<tr>
<td>testM2</td>
<td>36.0 ± 0.4</td>
</tr>
</tbody>
</table>
### VM: Optimization Quiz (F), #3

<table>
<thead>
<tr>
<th>test</th>
<th>ns/op</th>
</tr>
</thead>
<tbody>
<tr>
<td>testM1</td>
<td>4.6 ± 0.1</td>
</tr>
<tr>
<td>testM2</td>
<td>36.0 ± 0.4</td>
</tr>
<tr>
<td>repeat testM1</td>
<td>35.8 ± 0.4</td>
</tr>
</tbody>
</table>
VM: Optimization Quiz (F), #3

<table>
<thead>
<tr>
<th>test</th>
<th>ns/op</th>
</tr>
</thead>
<tbody>
<tr>
<td>testM1</td>
<td>4.6 ± 0.1</td>
</tr>
<tr>
<td>testM2</td>
<td>36.0 ± 0.4</td>
</tr>
<tr>
<td>repeat testM1</td>
<td>35.8 ± 0.4</td>
</tr>
<tr>
<td>forkted testM1</td>
<td>4.5 ± 0.1</td>
</tr>
<tr>
<td>forkted testM2</td>
<td>4.5 ± 0.1</td>
</tr>
</tbody>
</table>
Dynamic optimizations can use runtime information

(Ex.: call profile, type profile, CHA info)

- **Downside**: Big difference in running multiple benchmarks, or a single benchmark in the VM
- **Remedy**: Warmup all benchmarks together; OR fork the JVMs
- **JMH Remedy**: Bulk warmup support; forking
VM: Optimization Quiz (G)

JDK8b83 + nashorn-tiered 2013-04-08, Octane:RegExpBench performance over iterations

Execution time, msec

Iteration #

Trial 1  Trial 2  Trial 3  Trial 4  Trial 5  Trial 6  Trial 7  Trial 8  Trial 9  Trial 10

Copyright © 2013, Oracle and/or its affiliates. All rights reserved.
VM: Optimization Quiz (G), #2

JDK8b33 + nashorn-tiered 2013-04-08, Octane:MandrellBench performance over iterations

Execution time, msec

Iteration #

Trial 1 +  Trial 2 ×  Trial 3 *

Slide 54/75. Copyright © 2013, Oracle and/or its affiliates. All rights reserved.
VM: Run-to-run variance

Many scalable algos are inherently non-deterministic!
(Ex.: memory allocators, profiler counters, non-fair locks, concurrent data structures, some other intelligent tricks up our sleeve...)

- **Downside**: (potentially) (devastatingly) large run-to-run variance
- **Remedy**: replays within every subsystem, multiple JVM runs
- **JMH Remedy**: multiple forks
VM: Inlining budgets

Inlining is the über-optimization

- **Downside**: You can not inline everything \(\Rightarrow\) subtle inlining budget considerations
- **Remedy**: Smaller methods, smaller loops, examining -XX:+PrintInlining, forcing inlining
- **JMH Remedy**: Generated code peels potentially hot loops, user-friendly @CompileControl
VM: Inlining example

Small hot method: inlining budget starts here.

```java
public void testLong_loop
    (Loop loop, Result r, MyBenchmark bench) {
    long ops = 0;
    r.start = System.nanoTime();
    do {
        bench.testLong(); // @GenerateMicroBenchmark
        ops++;
    } while (!loop.isDone);
    r.end = System.nanoTime();
    r.ops = ops;
}
```
@State
public class TreeMapBench {
    Map<String, String> map = new TreeMap<>();

    @Setup
    public void setup() { populate(map); }

    @GenerateMicroBenchmark
    public void test(BlackHole bh) {
        for (String key : map.keySet()) {
            String value = map.get(key);
            bh.consume(value);
        }
    }
}
CPU: Optimization Quiz (H), #2

```java
@GenerateMicroBenchmark
public void test(BlackHole bh) {
    for (String key : map.keySet()) {
        String value = map.get(key);
        bh.consume(value);
    }
}
```

<table>
<thead>
<tr>
<th>Throughput, op/sec</th>
<th>Exclusive</th>
<th>Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>615 ± 12</td>
<td>828 ± 21</td>
</tr>
</tbody>
</table>
@GenerateMicroBenchmark
public void test(BlackHole bh) {
    for (String key : map.keySet()) {
        String value = map.get(key);
        bh.consume(value);
    }
}

<table>
<thead>
<tr>
<th>Throughput, op/sec</th>
<th>Exclusive</th>
<th>Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>615 ± 12</td>
<td>828 ± 21</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Slide 59/75. Copyright © 2013, Oracle and/or its affiliates. All rights reserved.
@GenerateMicroBenchmark
public void test(BlackHole bh) {
    for (String key : map.keySet()) {
        String value = map.get(key);
        bh.consume(value);
    }
}

<table>
<thead>
<tr>
<th></th>
<th>Exclusive</th>
<th>Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput, op/sec</td>
<td>615 ± 12</td>
<td>828 ± 21</td>
</tr>
<tr>
<td>Threads</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Maps</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
@GenerateMicroBenchmark
public void test(BlackHole bh) {
    for (String key : map.keySet()) {
        String value = map.get(key);
        bh.consume(value);
    }
}

<table>
<thead>
<tr>
<th></th>
<th>Exclusive</th>
<th>Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput, op/sec</td>
<td>615 ± 12</td>
<td>828 ± 21</td>
</tr>
<tr>
<td>Threads</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Maps</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Footprint, Kb</td>
<td>1024</td>
<td>256</td>
</tr>
</tbody>
</table>
CPU: Cache capacity

DRAM memory is too far and too slow. Cache the hottest stuff on-die SRAM cache!

- **Downside**: Remarkably different performance for memory accesses, depending on your luck
- **Remedy**: Track the memory footprint; multiple experiments with different problem sizes; shared/distinct data for the worker threads
- **JMH Remedy**: @State scopes
CPU: Optimization Quiz (I)

How scalable is this?

@State(Scope.Benchmark) class Shared {
    final int[] c = new int[64];
}

@State(Scope.Thread) class Local {
    static final AtomicInteger COUNTER = ...;
    final int index = COUNTER.incrementAndGet();
}

@GenerateMicroBenchmark
void work(Shared s, Local l) {
    s.c[l.index]++;
}
<table>
<thead>
<tr>
<th>Threads</th>
<th>Average ns/call</th>
<th>Hit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.0 ± 0.1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>18.5 ± 2.4</td>
<td>9x</td>
</tr>
<tr>
<td>4</td>
<td>32.9 ± 6.2</td>
<td>16x</td>
</tr>
<tr>
<td>8</td>
<td>85.4 ± 13.4</td>
<td>42x</td>
</tr>
<tr>
<td>16</td>
<td>208.9 ± 52.1</td>
<td>104x</td>
</tr>
<tr>
<td>32</td>
<td>464.2 ± 46.1</td>
<td>232x</td>
</tr>
</tbody>
</table>

Why?
CPU: Bulk method transfers

Memory subsystem tracks data in cache-line quantums. Cache lines are 32, 64, 128 bytes long.

- **Downside**: the dense inter-thread accesses are hard on memory subsystem (false sharing)
- **Remedy**: padding, subclass juggling, @Contended
- **JMH Remedy**: control structures are heavily padded, auto-padding for @State
Exhibit B.

```java
int sum = 0;
for (int x : a) {
    if (x < 0) {
        sum -= x;
    } else {
        sum += x;
    }
}
return sum;
```

Exhibit P.

```java
int sum = 0;
for (int x : a) {
    sum += Math.abs(x);
}
return sum;
```

Which one is faster?

---

Credits: Sergey Kuksenko (@kuksenk0)
### CPU: Optimization Quiz (J)

E. Branched

<table>
<thead>
<tr>
<th>L0: mov 0xc(%ecx,%ebp,4),%ebx</th>
<th>test %ebx,%ebx</th>
</tr>
</thead>
<tbody>
<tr>
<td>jl L1</td>
<td>add %ebx,%eax</td>
</tr>
<tr>
<td>jmp L2</td>
<td>cmp %edx,%ebp</td>
</tr>
<tr>
<td>L1: sub %ebx,%eax</td>
<td>jl L0</td>
</tr>
<tr>
<td>L2: inc %ebp</td>
<td></td>
</tr>
<tr>
<td>cmp %edx,%ebp</td>
<td></td>
</tr>
</tbody>
</table>

E. Predicated

| L0: mov 0xc(%ecx,%ebp,4),%ebx | mov %ebx,%esi |
| test %ebx,%ebx                | neg %esi      |
| jl L1                         | test %ebx,%ebx|
| add %ebx,%eax                 | cmovl %esi,%ebx|
| jmp L2                        | add %ebx,%eax |
|                               | inc %ebp      |
|                               | cmp %edx,%ebp |
|                               | jl Loop       |

Which one is faster?
**CPU: Optimization Quiz (J)**

Regular Pattern = (+, –)*

<table>
<thead>
<tr>
<th></th>
<th>NHM</th>
<th>Bldzr</th>
<th>C-A9(^3)</th>
<th>SNB</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch_regular</td>
<td>0.9</td>
<td>0.8</td>
<td>5.0</td>
<td>0.5</td>
</tr>
<tr>
<td>branch_shuffled</td>
<td>6.2</td>
<td>2.8</td>
<td>9.4</td>
<td>1.0</td>
</tr>
<tr>
<td>branch_sorted</td>
<td>0.9</td>
<td>1.0</td>
<td>5.0</td>
<td>0.6</td>
</tr>
<tr>
<td>predicated_regular</td>
<td>2.0</td>
<td>1.0</td>
<td>5.3</td>
<td>0.8</td>
</tr>
<tr>
<td>predicated_shuffled</td>
<td>2.0</td>
<td>1.0</td>
<td>9.3</td>
<td>0.8</td>
</tr>
<tr>
<td>predicated_sorted</td>
<td>2.0</td>
<td>1.0</td>
<td>5.7</td>
<td>0.8</td>
</tr>
</tbody>
</table>

\(^3\)Using client compiler
CPU: Branch Prediction

Out-of-Order engines speculate a lot.
Most of the time (99%+) correct!

- **Downside**: Vastly different performance when speculation fails
- **Remedy**: Realistic data! Multiple diverse datasets
Conclusion
Conclusion: not as simple as it sounds

You should be scared by now!

Resist the urge to:

- believe the pleasant results
- reject the unpleasant results
- write the throw-away benchmarks
- write the «generic» benchmark harnesses
- believe the fancy reports and beautiful APIs
- trust the code
Conclusion: Benchmarking is serious

More rigor is never a bad thing!

- The intuition is almost always wrong (unless you rock)
- Never trust anything (unless checked before)
- Ever challenge everything (especially these slides)
- Embrace failure (especially your failures)
- Grind your teeth, and redo the tests (especially yours)
Conclusion: Things on list to do

JMH does one thing and does it right: gets you less «back to square one» moments

Other things to improve usability:

- Java API (in progress)
- Bindings to reporters (in progress)
- Bindings to the other JVM languages
- @Param-eters
Thanks!
Conclusion: But wait...

Java Microbenchmark Harness
(the lesser of two evils)

Aleksey Shipilev
aleksey.shipilev@oracle.com, @shipilev
Conclusion: Alternative Evil

Don’t do any performance assessments at all

You should already know why it is far worse. ...right?
Thanks!