

ORACLE

# The Art of Java Benchmarking ...building harnesses, ugh

Aleksey Shipilev

aleksey.shipilev@oracle.com, @shipilev

MAKE THE  
FUTURE  
JAVA



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: Speaker's Credentials

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

# 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:

- interpreter vs. compiler: which is simpler to benchmark?

# Basics: Evil Optimizations

Optimizations distort the performance models!

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

Examples:

- interpreter vs. compiler: which is simpler to benchmark?
- `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/](http://openjdk.java.net/projects/code-tools/jmh/)

- Works around pitfalls tailored to HotSpot/OpenJDK specifics
- Bug fixes 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:

```
@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:

- \_\_\_\_\_: 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
- \_\_\_\_\_: 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
- \_\_\_\_\_: 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.

- **Milli**benchmarks are not really hard
- **Micro**benchmarks are challenging, but OK
- **Nano**benchmarks are the damned beasts!
- **Pico**benchmarks...

# Basics: Warmup

## Definition

# 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



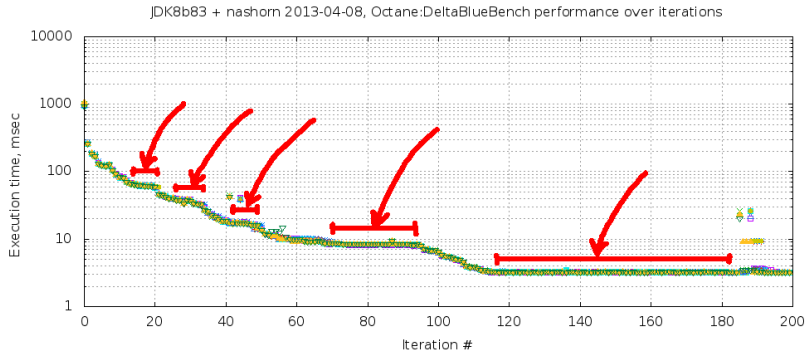
# Basics: Warmup

## 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



# 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

# System: Optimization Quiz (A)

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

Threads	Ops/nsec	Scale
1	3.06 ± 0.10	
2	5.72 ± 0.10	1.87 ± 0.03
4	5.87 ± 0.02	1.91 ± 0.03

# System: Optimization Quiz (A)

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

Threads	Ops/nsec	Scale
1	3.06 ± 0.10	
2	5.72 ± 0.10	1.87 ± 0.03
4	5.87 ± 0.02	1.91 ± 0.03

- Q(1): No change going for 2 → 4 threads?

# System: Optimization Quiz (A)

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

Threads	Ops/nsec	Scale
1	3.06 ± 0.10	
2	5.72 ± 0.10	1.87 ± 0.03
4	5.87 ± 0.02	1.91 ± 0.03

- Q(1): No change going for 2 → 4 threads?
- Q(2): Only 1.87x change for 1 → 2 threads?

# System: Power management

Running dummy benchmark,  
+ Down-clocking to 2.0 GHz

Threads	Ops/nsec	Scale
1	1.97 ± 0.02	
2	3.94 ± 0.05	2.00 ± 0.02
4	4.03 ± 0.04	2.04 ± 0.02



# 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

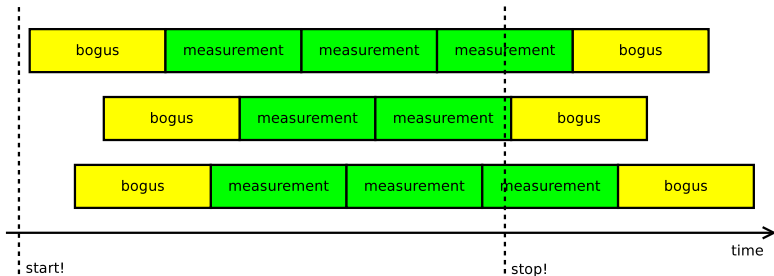
Time sharing systems balance utilization

(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:



# VM: Optimization Quiz (B)

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

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

```
@GenerateMicroBenchmark  
public double measureRight() { 34.0 ± 1.0 ns  
    return Math.log(x);  
}
```

# 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)

```
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  $\Rightarrow$  reference heap write  $\Rightarrow$  store barrier



# VM: DCE, Blackholes

JMH provides «Blackholes».  
Blackhole consumes the value.

```
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  $\Rightarrow$  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 (C)

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

```
@GenerateMicroBenchmark  
public double measureWrong() { 1.0 ± 0.1 ns  
    return Math.log(42);  
}
```

```
private double x = 42;  
@GenerateMicroBenchmark  
public double measureRight() { 34.0 ± 1.0 ns  
    return Math.log(x);  
}
```

# VM: Constant folding, etc.

Compilers are good at partial evaluation<sup>1</sup>

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

---

<sup>1</sup>All right, all right! It is not really *the* PE.

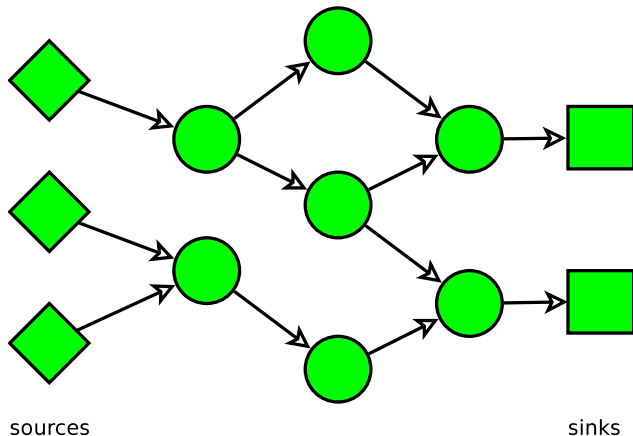
# VM: CSE

JMH prevents load commoning across @GMB calls

```
double x;
@GMB
double doWork() {
    doStuff(x);
}
|
volatile boolean done;
void doMeasure() {
    while (!done) {
        doWork();
    }
}
```

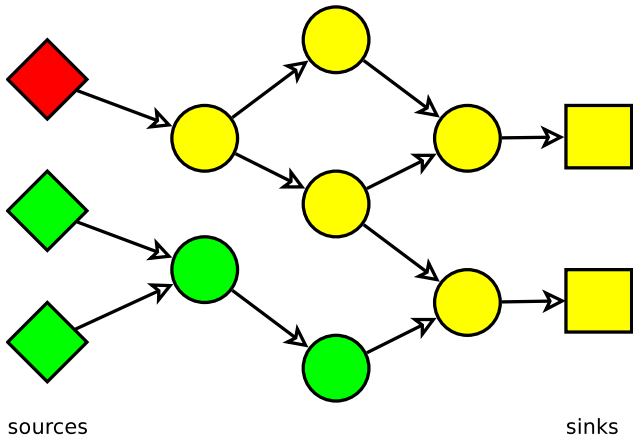
(i.e. read everything from heap  $\Rightarrow$  you are good!)

# VM: DCE, CSE... Same thing!



Losing either a source or a sink loses the part of the benchmark. Silently.

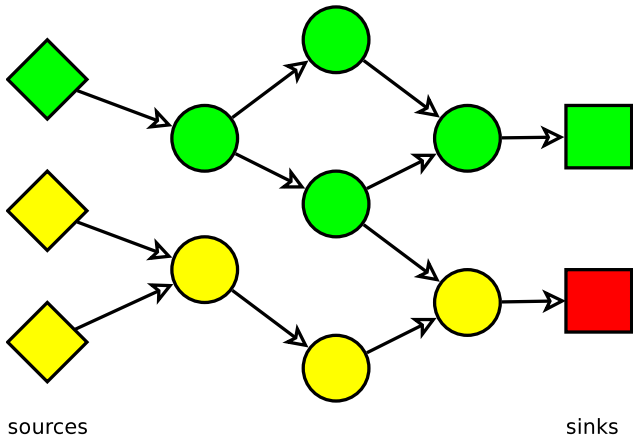
# VM: DCE, CSE... Same thing!



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 (D)

```
// 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 (D), #2

N	ns/call	ns/add
1	1.5 ± 0.1	1.5 ± 0.1
10	2.0 ± 0.1	0.1 ± 0.01
100	2.7 ± 0.2	0.05 ± 0.02
1000	68.8 ± 0.9	0.07 ± 0.01
10000	410.3 ± 2.1	0.04 ± 0.01
100000	3836.1 ± 40.6	0.04 ± 0.01

## VM: Optimization Quiz (D), #2

N	ns/call	ns/add
1	1.5 ± 0.1	1.5 ± 0.1
10	2.0 ± 0.1	0.1 ± 0.01
100	2.7 ± 0.2	0.05 ± 0.02
1000	68.8 ± 0.9	0.07 ± 0.01
10000	410.3 ± 2.1	0.04 ± 0.01
100000	3836.1 ± 40.6	0.04 ± 0.01

Which one to believe?

0.04 ns/add  $\Rightarrow$  25 adds/ns  $\Rightarrow$  GTFO!

# VM: Loop unrolling

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

# VM: Optimization Quiz (E)

```
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 (E), #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 (E), #3

test	ns/op
testM1	4.6 ± 0.1
testM2	36.0 ± 0.4



# VM: Optimization Quiz (E), #3

test	ns/op
testM1	4.6 ± 0.1
testM2	36.0 ± 0.4
repeat testM1	35.8 ± 0.4

# VM: Optimization Quiz (E), #3

test	ns/op
testM1	4.6 ± 0.1
testM2	36.0 ± 0.4
repeat testM1	35.8 ± 0.4
forked testM1	4.5 ± 0.1
forked testM2	4.5 ± 0.1

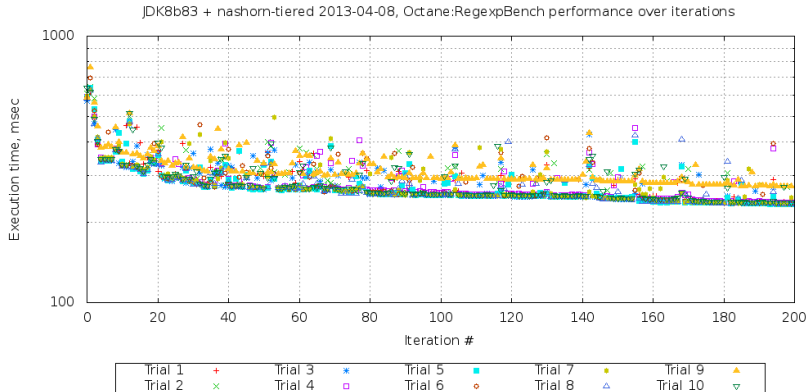
# VM: Profile feedback

Dynamic optimizations  
can use runtime information

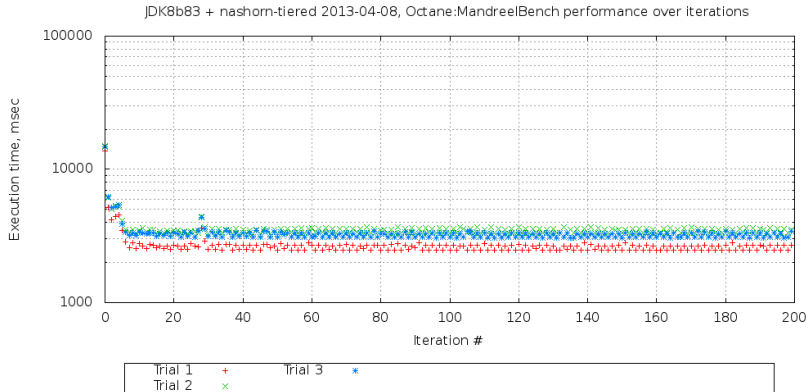
(Ex.: call profile, type profile, CHA info)

- **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 (F)



# VM: Optimization Quiz (F), #2



# 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 withing every subsystem, multiple JVM runs
- **JMH Remedy:** multiple forks



ORACLE

# VM: Inlining budgets

Inlining is the uber-optimization

- **Downside:** You can not inline everything ⇒ 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.

```
public void testLong_loop
    (Loop loop, Result r, MyBench bench) {
    long ops = 0;
    r.start = System.nanoTime();
    do {
        bench.testLong(); // @GMB
        ops++;
    } while(!loop.isDone);
    r.end = System.nanoTime();
    r.ops = ops;
}
```



# CPU: Optimization Quiz (G)

```
@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 (G), #2

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

	Exclusive	Shared
Throughput, op/sec	615 ± 12	828 ± 21

# CPU: Optimization Quiz (G), #2

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

	Exclusive	Shared
Throughput, op/sec	615 ± 12	828 ± 21
Threads	4	4

# CPU: Optimization Quiz (G), #2

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

	Exclusive	Shared
Throughput, op/sec	615 ± 12	828 ± 21
Threads	4	4
Maps	4	1

# CPU: Optimization Quiz (G), #2

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

	Exclusive	Shared
Throughput, op/sec	615 ± 12	828 ± 21
Threads	4	4
Maps	4	1
Footprint, Kb	1024	256

# 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 (G)

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]++;  
}
```

# CPU: Optimization Quiz (G), #2

Threads	Average ns/call	Hit
1	2.0 ± 0.1	
2	18.5 ± 2.4	9x
4	32.9 ± 6.2	16x
8	85.4 ± 13.4	42x
16	208.9 ± 52.1	104x
32	464.2 ± 46.1	232x

Why?



# CPU: Bulk method transfers

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

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

# CPU: Optimization Quiz (H)<sup>2</sup>

## Exhibit B.

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

## Exhibit P.

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

Which one is faster?

<sup>2</sup>Credits: Sergey Kuksenko (@kuksen0)

# CPU: Optimization Quiz (H)

## E. Branched

```
L0: mov 0xc(%ecx,%ebp,4),%ebx
     test %ebx,%ebx
     jl  L1
     add %ebx,%eax
     jmp L2
L1: sub %ebx,%eax
L2: inc %ebp
     cmp %edx,%ebp
     jl  L0
```

## E. Predicated

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

Which one is faster?

# CPU: Optimization Quiz (H)

Regular Pattern = (+, -)\*

	NHM	Bldzr	C-A9 <sup>3</sup>	SNB
branch_regular	0.9	0.8	5.0	0.5
branch_shuffled	6.2	2.8	9.4	1.0
branch_sorted	0.9	1.0	5.0	0.6
predicated_regular	2.0	1.0	5.3	0.8
predicated_shuffled	2.0	1.0	9.3	0.8
predicated_sorted	2.0	1.0	5.7	0.8

time, nsec/op

---

<sup>3</sup>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!

- Spending a day on benchmark is better than wasting a week implementing the wrong suggestion
- Never trust anything (unless checked before)
- Ever challenge everything (including 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
- @Parameters

# Conclusion: JMH



**Java Plumb**

@JavaPlumb



"Any Java benchmarking framework contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of JMH" via @shipilev



Reply



Retweeted



Favorite



More

7

RETWEETS

3

FAVORITES



2:19 AM - 29 Oct 13

# Thanks!

