ORACLE

Necessar(il)y Evil
dealing with benchmarks, ugh

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

MAKE THE FUTURE JAVA

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 Techrep for Oracle»
- in-«Oracle JDK performance guy»
- Guilty for:
  1. *cough*aurora*cough*
  2. SPECjbb2013
  3. Concurrency improvements (e.g. @Contended)
  4. Java Microbenchmark Harness (jmh)
  5. Java Concurrency Stress Tests (jcstress)

Basics

Basics: Naive question

What is benchmark?

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: Benchmarks are experiments

- Computer Science → Software Engineering
  - Way to construct 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!

Benchmarking assumes
the performance model

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» (= special) cases
- Unclear inter-dependencies
- «Black box» abstraction fails big time

Basics: Evil Optimizations

Optimizations distort the performance models!

- Applied in «common» (= special) cases
- Unclear inter-dependencies
- «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» (= special) cases
- Unclear inter-dependencies
- «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: Benchmarks vs. Optimization

Rule #1

Benchmarking is the fight against the optimizations

Level out the performance model:

- collapse the search space
- get predictable benchmarks
- contrast the optimizations we are after

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 ⇒ new hassle for everyone

Basics: JMH

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

- Works around many pitfalls tailored to HotSpot/OpenJDK specifics
- Bug fixes as VM evolves, or we discover more
- We (performance team) validate micros 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: ...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

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

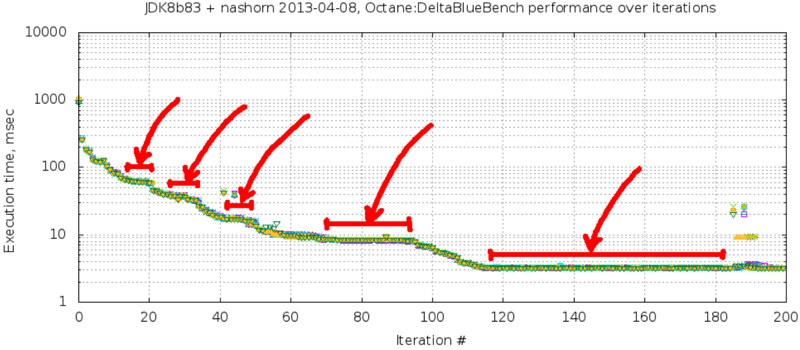
Basics: Warmup

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

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

- Why no change for $2 \rightarrow 4$ 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

- Why no change for $2 \rightarrow 4$ threads?
- Why 1.87x change for $1 \rightarrow 2$ threads?

System: Power management

Running dummy benchmark,
+ Down-clocking to 2.0 GHz
(effectively disable TurboBoost)

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. (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 the load vs. power

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

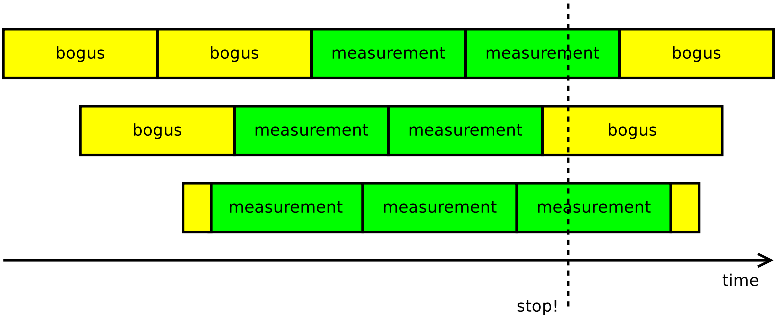
System: Time Sharing

Time sharing systems balance utilization.

- **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**: «synchronize 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() {
    Math.log(x);
}
```

0.5 ± 0.1 ns

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

34 ± 1 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: DCE, Avoiding

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: DCE, Avoiding

DCE is hard to avoid for references:

- Caveat #1: fast object combinator?
- Caveat #2: need to escape object to limit thread-local opto
- Caveat #3: publishing ⇒ heap write ⇒ 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 ⇒ pushes us for heap writes?

VM: DCE, 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
      prngMask = (prngMask << 1) + 1;
    }
  }
}
```


VM: Optimization Quiz (C)

```
@GenerateMicroBenchmark
public void baseline() {
}
```

0.5 ± 0.1 ns

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

1.0 ± 0.1 ns

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

34 ± 1 ns

VM: Constant folding, etc.

Compilers are good at partial evaluation[1]

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

[1] All right, @w7cook! It is not really *the* PE.

VM: CSE

JMH prevents load commoning across @GMB calls

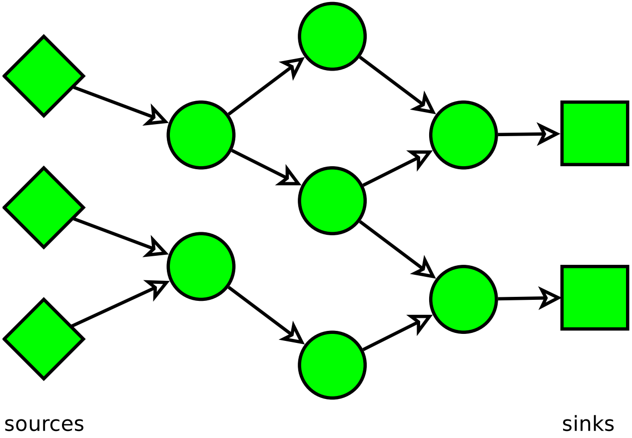
```
double x;

@GMB
double doWork() {
  doStuff(x);
}
```

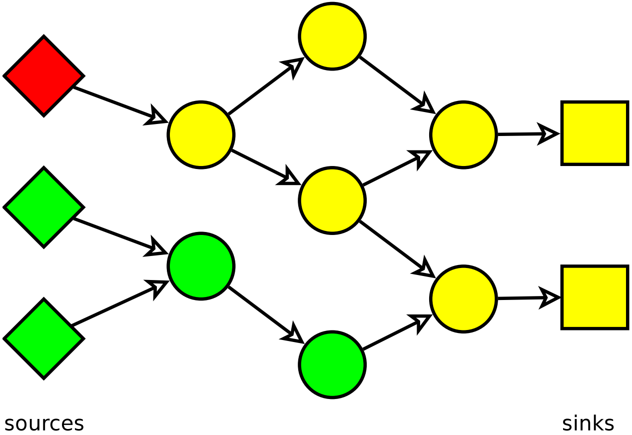
```
volatile boolean done;
void doMeasure() {
  while (!done) {
    doWork();
  }
}
```

(i.e. read everything from heap ⇒ you are good!)

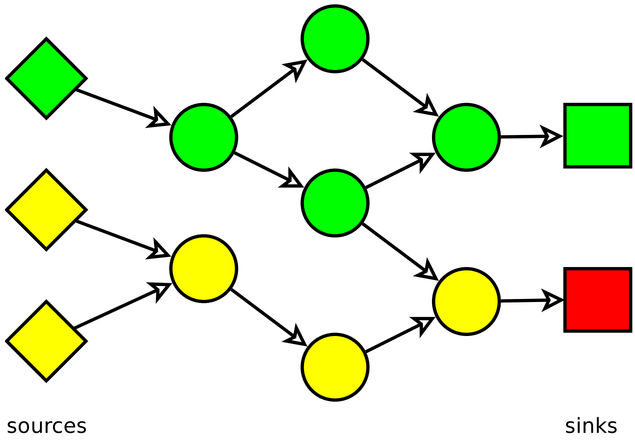
VM: DCE and CSE are brothers



VM: DCE and CSE are brothers



VM: DCE and CSE are brothers



VM: Optimization Quiz (D)

```
int x = 1, y = 2;

private int reps(int reps) {
    int s = 0;
    for (int i = 0; i < reps; i++) {
        s += (x + y);
    }
    return s;
}

@GenerateMicroBenchmark
public int test() { // Q: performance vs. N?
    return reps(N);
}
```


VM: Optimization Quiz (D), #2

N	ns/call		ns/add	
(jmh)	1.5	± 0.1	1.5	± 0.1
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	
(jmh)	1.5	± 0.1	1.5	± 0.1
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

0.04 ns/add ⇒ 25 adds/ns ⇒ 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 αM 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)

```java
interface Counter {
  void inc();
}

abstract class AC implements Counter {
  int c;
  void inc() {
    c++;
  }
}

class M1 extends AC {}
class M2 extends AC {}
```


VM: Optimization Quiz (E), #2

```
Counter m1 = new M1();
Counter m2 = new M2();

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

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

void test(Counter c) {
    for (int i = 0; i < 100; i++)
        c.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	

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

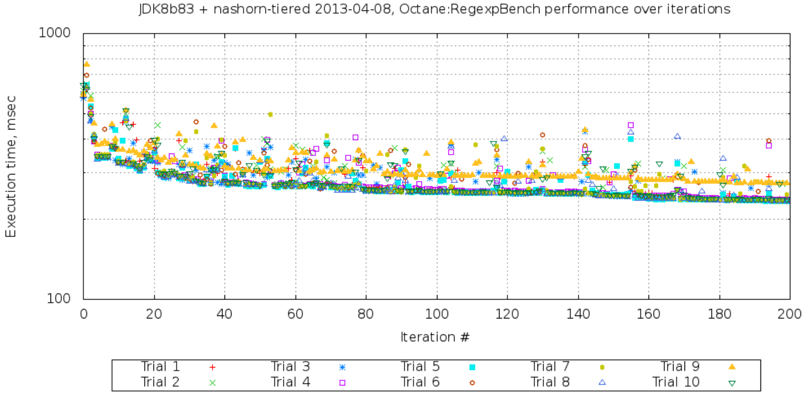
VM: Profile feedback

Dynamic optimizations

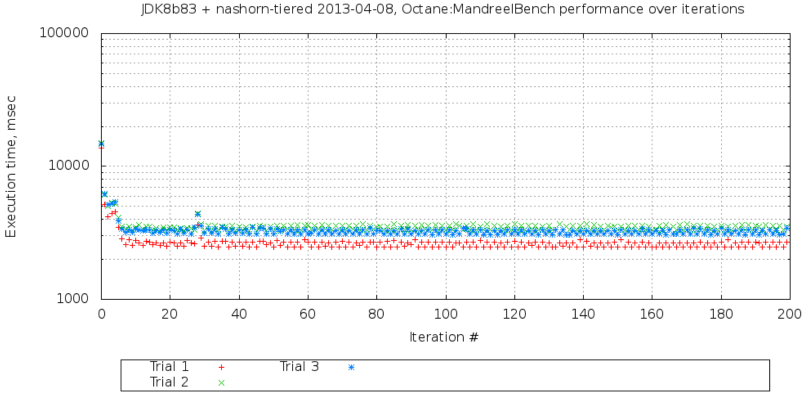
can use runtime information
(E.g. call/type profile)

- **Downside**: Big difference in running multiple benchmarks, or a single benchmark in the VM
- **Remedy**: Warmup everything together; fork JVMs
- **JMH Remedy**: Bulk warmup support; fork JVMs

VM: Optimization Quiz (F)



VM: Optimization Quiz (F), #2



VM: Run-to-run variance

Many scalable algos are inherently non-deterministic! (E.g. memory allocators, profiler counters, schedulers)

- **Downside**: (Potentially) large run to run variance
- **Remedy**: Replays withing every subsystem, multiple JVM runs
- **JMH Remedy**: multiple JVM runs

VM: Inlining budgets

Inlining is the uber-optimization

- **Downside**: You can't inline everything ⇒ subtle inlining budget considerations
- **Remedy**: Smaller methods, smaller loops, examining -XX:+PrintInlining, forcing inlining
- **JMH Remedy**: Generated code peels potentially hot loops, @CompileControl

VM: Inlining example

Small hot method: inlining budget starts here.

```
public RawResultPair testLong_loop
   (Loop loop, MyBenchmark bench) {
  long ops = 0;
  long start = System.nanoTime();
  do {
    bench.testLong(); // @GMB
    ops++;
  } while(!loop.isDone);
  long end = System.nanoTime();
  return new RawResultPair(ops, end - start);
}
```


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	$\sim$1024	$\sim$256

CPU: Cache capacity

DRAM memory is too far and too slow. Let's cache a lot of stuff near the CPUs in SRAMs!

- **Downside**: Severely different performance profiles for memory accesses
- **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)

Scalability for this code?

```
public class QuizG {
  @State(Scope.Benchmark) class Shared {
    int[] c = new int[64];
  }
  @State(Scope.Thread) class Local {
    static final AtomicInteger COUNTER = ...;
    int index = COUNTER.incrementAndGet();
  }
  @GenerateMicroBenchmark
  void work(Shared s, Local l) {
    s.c[l.index]++;
  }
}
```


CPU: Optimization Quiz (G), #2

Thread	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 is mostly dealing with high locality data. Cache lines are 32, 64, 128 bytes long.

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

CPU: Optimization Quiz (H)

Exhibit B.	Exhibit P.

<pre>int sum = 0;
for (int x : a) {
  if (x < 0) {
    sum -= x;
  } else {
    sum += x;
  }
}
return sum;</pre>
	<pre>int sum = 0;
	for (int x : a) {
	  sum += Math.abs(x);
	}
	return sum;</pre>

Which one is faster?

CPU: Optimization Quiz (H)

E. Branched	E. Predicated

<pre>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</pre>
	<pre>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</pre>

Which one is faster?

CPU: Optimization Quiz (H)

Regular Pattern = $(+, -)^*$

	NHM	Bldzr	C-A9[2]	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

[2] actually, not C2, but C1

CPU: Optimization Quiz (H)

Regular Pattern = $(+, +, -, +, -, -, +, -, -, +)^*$

	NHM	Bldzr	C-A9[3]	SNB
branch_regular	1.3	1.0	5.0	0.7
branch_shuffled	6.2	2.3	9.5	0.8
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	9.4	0.8
predicated_sorted	2.0	1.0	5.7	0.8

time, nsec/op

[3] actually, not C2, but C1

CPU: Branch Prediction

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

- **Downside**: vastly different performance on mispredicts
- **Remedy**: realistic data!

Conclusion

Conclusion: Benchmarking is serious

- Control the hardware optimizations
- Control the OS optimizations
- Control the JVM optimizations
- Control the language runtime optimizations
- Control the data
- Control the results

Conclusion: Things on list to do

JMH does one thing, and does it right

Other things to improve usability:

- `@Param`-eters
- Java API
- Bindings to the other JVM languages
- Bindings to more reporters

Thanks!

