Java Micro-Benchmarking

Constantine Nosovsky

SIGMA

Agenda

- Benchmark definition, types, common problems
- Tools needed to measure performance
- Code warm-up, what happens before the steady-state
- Using JMH
- Side effects that can affect performance
- JVM optimizations (Good or Evil?)
- A word about concurrency
- Full example, “human factor” included
- A fleeting glimpse on the JMH details

Benchmark definition, types, common problems

What is a “Benchmark”?

- Benchmark is a program for performance measurement

Requirements:

- Dimensions: throughput and latency
- Avoid significant overhead
- Test what is to be tested
- Perform a set of executions and provide stable reproducible results
- Should be easy to run

Benchmark types

- By scale
  - Micro-benchmark (component level)
  - Macro-benchmark (system level)

- By nature
  - Synthetic benchmark (emulate component load)
  - Application benchmark (run real-world application)

We’ll talk about

- Synthetic micro-benchmark
- Mimic component workload separately from the application
- Measure performance of a small isolated piece of code

- The main concern
- The smaller the Component we test – the stronger the impact of
  - Benchmark infrastructure overhead
  - JVM internal processes
  - OS and Hardware internals
  - … and the phases of the Moon
- Don’t we really test one of those?

When micro-benchmark is needed

- Most of the time it is not needed at all
- Does algorithm `A` work faster than `B`?
  (Consider equal analytical estimation)
- Does this tiny modification make any difference?
  (from Java, JVM, native code or hardware point of view)

Tools needed to measure performance

You had one job…

```java
final int COUNT = 100;

long start = System.currentTimeMillis();
for (int i = 0; i < COUNT; i++) {
        // doStuff();
}
long duration = System.currentTimeMillis() - start;

long avg = duration / COUNT;
System.out.println("Average execution time is " + avg + " ms");
```

Pitfall #0

- Using profiler to measure performance of small methods
  (adds significant overhead, measures execution “as is”)

- “You had one job” approach is enough in real life
  (not for micro-benchmarks, we got it already)

- Annotations and reflective benchmark invocations
  (you must be great at *java.lang.reflect* measurement)

Micro-benchmark frameworks

- **JMH** – Takes into account a lot of internal VM processes and executes benchmarks with minimal infrastructure (Oracle)
- **Caliper** – Allows to measure repetitive code, works for Android, allows to post results online (Google)
- **Japex** – Allows to reduce infrastructure code, generates nice HTML reports with JFreeChart plots
- **JUnitPerf** – Measure functionality of the existing JUnit tests

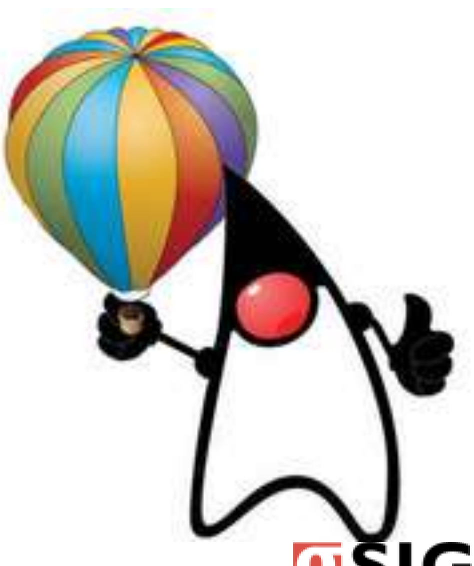
Java time interval measurement

- *System.currentTimeMillis()*
  - Value in milliseconds, but granularity depends on the OS
  - Represents a “wall-clock” time (since the start of Epoch)

- *System.nanoTime()*
  - Value in nanoseconds, since some time offset
  - The accuracy is not worse than *System.currentTimeMillis()*

- *ThreadMXBean.getCurrentThreadCpuTime()*
  - The actual CPU time spent for the thread (nanoseconds)
  - Might be unsupported by your VM
  - Might be expensive
  - Relevant for a single thread only

Code warm-up, what happens before the steady-state



Code warm-up, class loading

- A single warm-up iteration is NOT enough for Class Loading (not all the branches of classes may load on the first iteration)
- Sometimes classes are unloaded (it would be a shame if something messed your results up with a huge peak)

- Get help between iterations from
- *ClassLoadingMXBean.getTotalLoadedClassCount()*
- *ClassLoadingMXBean.getUnloadedClassCount()*
- -verbose:class

Code warm-up, compilation

- Classes are loaded, verified ~~and then being compiled~~
- *Oracle HotSpot* and *Azul Zing* run application in **interpreter**
- The hot method is being compiled after ~10k (server), ~1.5k (client) invocations
- Long methods with loops are likely to be compiled earlier

- Check *CompilationMXBean.getTotalCompilationTime*
- Enable compilation logging with
  - -XX:+UnlockDiagnosticVMOptions
  - -XX:+PrintCompilation
  - -XX:+LogCompilation -XX:LogFile=<filename>

SIGMA

Code warm-up, OSR

- Normal compilation and OSR will result in a similar code
- …unless compiler is not able to optimize a given frame
(e.g. inner loop is compiled before the outer one)

- In the real world normal compilation is more likely to happen, so it’s better to avoid OSR in your benchmark
- Do a set of small warm-up iterations instead of a single big one
- Do not perform warm-up loops in the steady-state testing method

Code warm-up, OSR example

Before:

```
public static void main(String… args) {
    loop1: if(P1) goto done1
        i=0;
        loop2: if(P2) goto done2
            A[i++];
        goto loop2; // OSR goes here
        done2:
    goto loop1;
    done1:
}
```

After:

```
void OSR_main() {
    A= // from interpreter
    i= // from interpreter
    loop2: if(P2) {
            if(P1) goto done1
            i=0;
        } else { A[i++]; }
    goto loop2
    done1:
}
```

- Now forget about array range check elimination

Reaching the steady-state, summary

- Always do warm-up to reach steady-state
  - Use the same data and the same code
  - Discard warm-up results
  - Avoid OSR
  - Don’t run benchmark in the “mixed modes” (interpreter/compiler)
  - Check class loading and compilation

Using JMH

- Provides Maven archetype for a quick project setup
- Annotate your methods with `@GenerateMicroBenchmark`
- `mvn install` will build ready to use runnable jar with your benchmarks and needed infrastructure

- `java -jar target/mb.jar <benchmark regex> [options]`
- Will perform warm-up following by a set of iterations
- Print the results

Side effects that can affect performance

SIGMA

Synchronization puzzle

8,244,087 usec

```
void testSynchInner() {
   synchronized (this) {
      i++;
   }
}
```

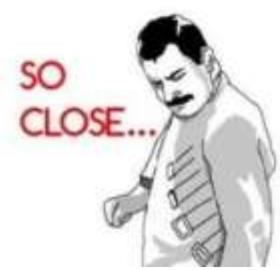
13,383,707 usec

```
synchronized void testSynchOuter() {
   i++;
}
```

Synchronization puzzle, side effect

- Biased Locking: an optimization in the VM that leaves an object as logically locked by a given thread even after the thread has released the lock (cheap reacquisition)

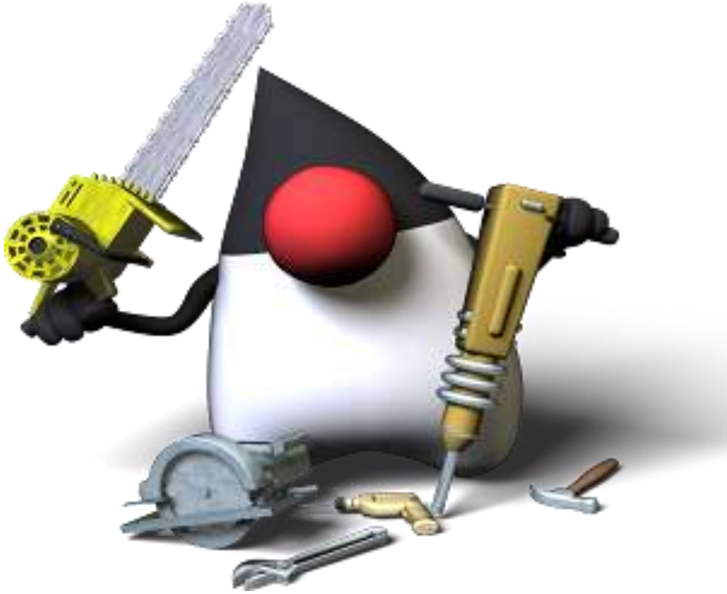
- Does not work on VM start up
  (4 sec in HotSpot)
  Use -XX:BiasedLockingStartupDelay=0



SIGMA

JVM optimizations (Good or Evil?)

- **WARNING:** some of the following optimizations
  will not work (at least for the given examples)
  in Java 6 (jdk1.6.0_26), consider using Java 7 (jdk1.7.0_21)



Dead code elimination

- VM optimization eliminates dead branches of code
- Even if the code is meant to be executed,
  but the result is never used and does not have any side effect

- Always consume all the results of your benchmarked code
- Or you'll get the "*over 9000*" performance level

- Do not accumulate results or store them in class fields that are never used either
- Use them in the unobvious logical expression instead

Dead code elimination, example

- Measurement: average nanoseconds / operation, less is better

```
private double n = 10;
```

1.008

```
public void stub() { }
```

1.017

```
public void dead() {
    @SuppressWarnings("unused")
    double r = n * Math.log(n) / 2;
}
```

48.514

```
public void alive() {
    double r = n * Math.log(n) / 2;
    if(r == n && r == 0)
        throw new IllegalStateException();
}
```

Constant folding

- If the compiler sees that the result of calculation will always be the same, it will be stored in the constant value and reused
- Measurement: average nanoseconds / operation, less is better

```
private double x = Math.PI;

public void stub() { }                    1.014

public double wrong() {
    return Math.log(Math.PI);             1.695
}

public double measureRight() {
    return Math.log(x);                   43.435
}
```

Loop unrolling

- Is there anything bad?
- Measurement: average nanoseconds / operation, less is better

```
private double[] A = new double[2048];

public double plain() {
    double sum = 0;
    for (int i = 0; i < A.length; i++)
        sum += A[i];
    return sum;
}

public double manualUnroll() {
    double sum = 0;
    for (int i = 0; i < A.length; i += 4)
        sum += A[i] + A[i + 1] + A[i + 2] + A[i + 3];
    return sum;
}
```

plain()	2773.883
manualUnroll()	816.791

Loop unrolling and hoisting

- Something bad happens when
  the loops of benchmark infrastructure code are unrolled
- And the calculations that we try to measure
  are hoisted from the loop

- For example, **Caliper** style benchmark looks like

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

Loop unrolling and hoisting, example

```
@GenerateMicroBenchmark
public int measureRight() {
    return (x + y);
}

@GenerateMicroBenchmark
@OperationsPerInvocation(1)
public int measureWrong_1() {
    return reps(1);
}

...

@GenerateMicroBenchmark
@OperationsPerInvocation(N)
public int measureWrong_N() {
    return reps(N);
}
```

Loop unrolling and hoisting, example

- Measurement: average nanoseconds / operation, less is better

Method	Result
Right	2.104
Wrong_1	2.055
Wrong_10	0.267
Wrong_100	0.033
Wrong_1000	0.057
Wrong_10000	0.045
Wrong_100000	0.043

A word about concurrency

- Processes and threads fight for resources (single threaded benchmark is a utopia)

Concurrency problems of benchmarks

- Benchmark states should be correctly
  - Initialized
  - Published
  - Shared between certain group of threads

- Multi threaded benchmark iteration should be synchronized and all threads should start their work at the same time

- No need to implement this infrastructure yourself,
just write a correct benchmark using your favorite framework

Full example, “human factor” included

List iteration

- Which list implementation is faster for the foreach loop?

- `ArrayList` and `LinkedList` sequential iteration is linear, `O(n)`
  - *ArrayList Iterator.next():* `return array[cursor++];`
  - *LinkedList Iterator.next():* `return current = current.next;`

- Let’s check for the list of 1 million `Integer`’s

List iteration, foreach vs iterator

- Measurement: average milliseconds / operation, less is better

23.659

```
public List<Integer> arrayListForeach() {
    for(Integer i : arrayList) {
    }
    return arrayList;
}
```

22.445

```
public Iterator<Integer> arrayListIterator() {
    Iterator<Integer> iterator = arrayList.iterator();
    while(iterator.hasNext()) {
        iterator.next();
    }
    return iterator;
}
```

List iteration, foreach < iterator, why?

- Foreach variant assigns element to a local variable
  ```
  for(Integer i : arrayList)
  ```
- Iterator variant does not
  ```
  iterator.next();
  ```
- We need to change Iterator variant to
  ```
  Integer i = iterator.next();
  ```
- So now it’s correct to compare the results, at least according to the bytecode ☺

List iteration, benchmark

```
@GenerateMicroBenchmark(BenchmarkType.All)
public List<Integer> arrayListForeach() {
    for(Integer i : arrayList) {
    }
    return arrayList;
}

@GenerateMicroBenchmark(BenchmarkType.All)
public Iterator<Integer> arrayListIterator() {
    Iterator<Integer> iterator = arrayList.iterator();
    while(iterator.hasNext()) {
        Integer i = iterator.next();
    }
    return iterator;
}
```

SIGMA

List iteration, benchmark, result

- Measurement: average milliseconds / operation, less is better

List impl	Iteration	Java 6	Java 7
ArrayList	foreach	24.792	5.118
	iterator	24.769	0.140
LinkedList	foreach	15.236	9.485
	iterator	15.255	9.306

- Java 6 ArrayList uses AbstractList.Itr,
  LinkedList has its own, so there is less abstractions
  (in Java 7 ArrayList has its own optimized iterator)

List iteration, benchmark, result

- Measurement: average milliseconds / operation, less is better

List impl	Iteration	Java 6	Java 7
ArrayList	foreach	24.792	5.118
	iterator	24.769	**0.140** WTF?!
LinkedList	foreach	15.236	9.485
	iterator	15.255	9.306

SIGMA

List iteration, benchmark, loop-hoisting

ListBenchmark.arrayListIterator()

```
Iterator<Integer> iterator = arrayList.iterator();
while(iterator.hasNext()) {
    iterator.next();
}
return iterator;
```

ArrayList.Itr<E>.next()

```
if (modCount != expectedModCount) throw new CME();
int i = cursor;
if (i >= size) throw new NoSuchElementException();
Object[] elementData = ArrayList.this.elementData;
if (i >= elementData.length) throw new CME();
cursor = i + 1;
return (E) elementData[lastRet = i];
```

List iteration, benchmark, BlackHole

```
@GenerateMicroBenchmark(BenchmarkType.All)
public void arrayListForeach(BlackHole bh) {
    for(Integer i : arrayList) {
        bh.consume(i);
    }
}

@GenerateMicroBenchmark(BenchmarkType.All)
public void arrayListIterator(BlackHole bh) {
    Iterator<Integer> iterator = arrayList.iterator();
    while(iterator.hasNext()) {
        Integer i = iterator.next();
        bh.consume(i);
    }
}
```

List iteration, benchmark, correct result

- Measurement: average milliseconds / operation, less is better

List impl	Iteration	Java 6	Java 7	Java 7 BlackHole
ArrayList	foreach	24.792	5.118	8.550
	iterator	24.769	0.140	8.608
LinkedList	foreach	15.236	9.485	11.739
	iterator	15.255	9.306	11.763

A fleeting glimpse on the JMH details

- We already know that JMH
  - Uses maven
  - Uses annotation-driven approach to detect benchmarks
  - Provides BlackHole to consume results (and CPU cycles)

JMH: Building infrastructure

- Finds annotated micro-benchmarks using reflection
- Generates infrastructure plain java source code around the calls to the micro-benchmarks
- Compile, pack, run, profit

- No reflection during benchmark execution

JMH: Various metrics

- Single execution time
- Operations per time unit
- Average time per operation
- Percentile estimation of time per operation

JMH: Concurrency infrastructure

- @State of benchmark data is shared across the benchmark, thread, or a group of threads
- Allows to perform Fixtures (`setUp` and `tearDown`) in scope of the whole run, iteration or single execution

- @Threads a simple way to run concurrent test if you defined correct @State

- @Group threads to assign them for a particular role in the benchmark

JMH: VM forking

- Allows to compare results obtained from various instances of VM
- First test will work on the clean JVM and others will not
- VM processes are not determined and may vary from run to run (compilation order, multi-threading, randomization)

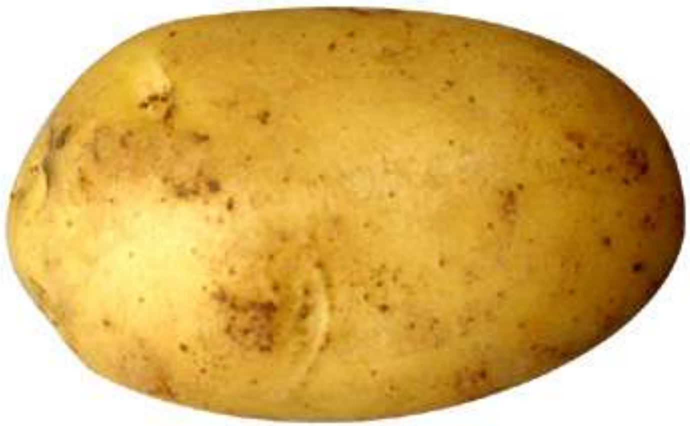
JMH: @CompilerControl

- Instructions whether to compile method or not
- Instructions whether to inline methods
- Inserting breakpoints into generated code
- Printing methods assembly

Conclusions

- Do not reinvent the wheel, if you are not sure how it should work (consider using existing one)
- Consider the results being wrong if you don’t have a clear explanation. Do not swallow that mystical behavior

Thanks for you attention



Questions?

SIGMA