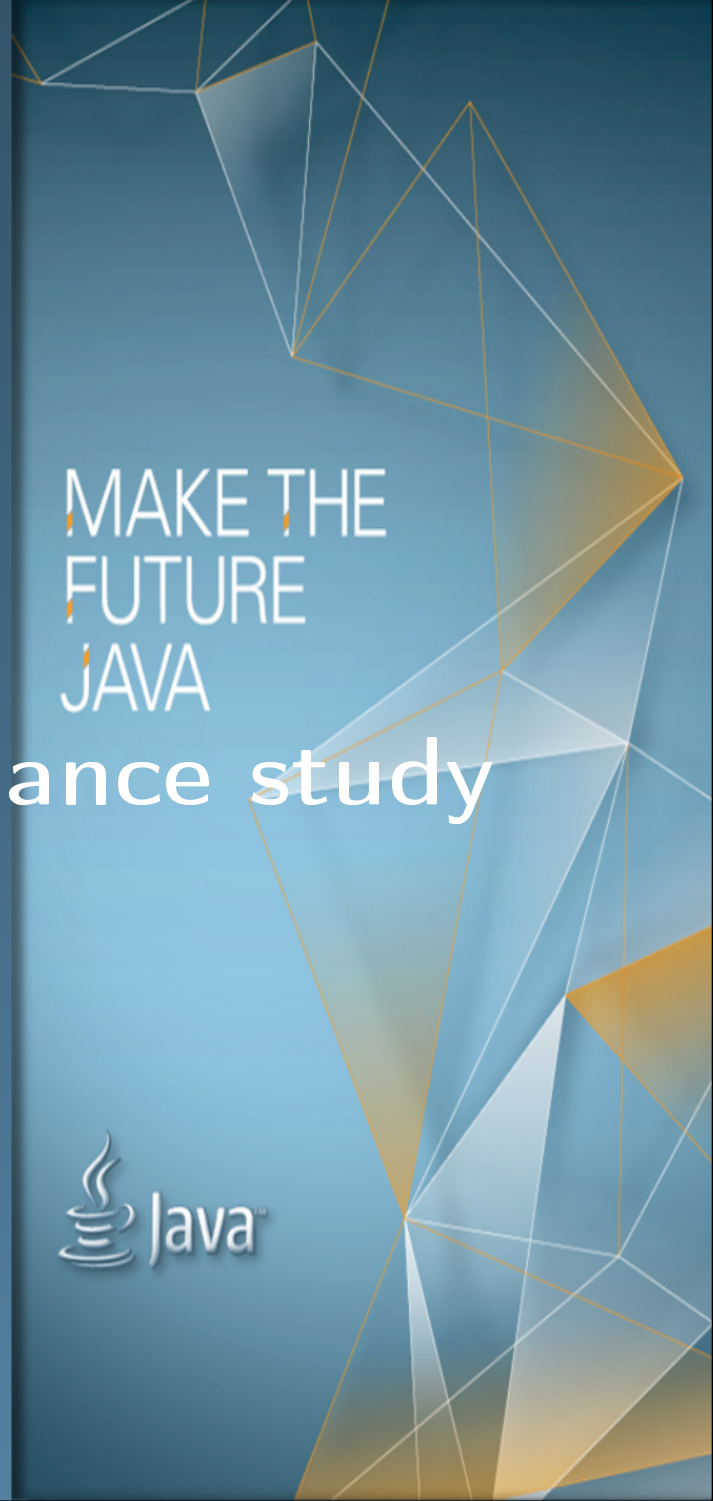
ORACLE



JDK 8: Lambda Performance study

Sergey Kuksenko
sergey.kuksenko@oracle.com, @kuksenk0

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.

Lambda

Lambda: performance

Lambda

vs

Anonymous Class

Lambda: performance

Lambda

- linkage

vs

Anonymous Class

- class loading

Lambda: performance

Lambda

- linkage
- capture

vs

Anonymous Class

- class loading
- instantiation

Lambda: performance

Lambda

- linkage
- capture
- invocation

vs

Anonymous Class

- class loading
- instantiation
- invocation

Lambda: SUT[1]

- Intel® Core™ i5-520M (Westmere) [2.0 GHz] 1x2x2
  - Xubuntu 11.10 (64-bits)
- HDD Hitachi 320Gb, 5400 rpm

[1] System Under Test

Linkage

Linkage: How?

```
@GenerateMicroBenchmark
@BenchmarkMode(Mode.SingleShotTime)
@OutputTimeUnit(TimeUnit.SECONDS)
@Fork(value = 5, warmups = 1)
public static Level link() {
    ...
};
```

Linkage: How?

```
@GenerateMicroBenchmark
@BenchmarkMode(Mode.SingleShotTime)
@OutputTimeUnit(TimeUnit.SECONDS)
@Fork(value = 5, warmups = 1)
public static Level link() {
    ...
};
```

Linkage: How?

```
@GenerateMicroBenchmark
@BenchmarkMode(Mode.SingleShotTime)
@OutputTimeUnit(TimeUnit.SECONDS)
@Fork(value = 5, warmups = 1)
public static Level link() {
    ...
};
```

Linkage: How?

```
@GenerateMicroBenchmark
@BenchmarkMode(Mode.SingleShotTime)
@OutputTimeUnit(TimeUnit.SECONDS)
@Fork(value = 5, warmups = 1)
public static Level link() {
    ...
};
```

Linkage: What?

Required:

- lots of lambdas

Linkage: What?

Required:

- lots of different lambdas

Linkage: What?

Required:

- lots of different lambdas

e.g. `()->()->()->()->()->...->()->null`

Linkage: What?

Required:

- lots of different lambdas

e.g. `()->()->()->()->()->...->()->null`

```
@FunctionalInterface
public interface Level {
    Level up();
}
```

Linkage: lambda chain

```
...

public static Level get1023(String p) {
    return () -> get1022(p);
}

public static Level get1024(String p) {
    return () -> get1023(p);
}

...
```

Linkage: anonymous chain

```
...

public static Level get1024(final String p){
    return new Level() {
        @Override
        public Level up() {
            return get1023(p);
        }
    };
}

...
```

Linkage: benchmark

```
@GenerateMicroBenchmark
...
public static Level link() {
    Level prev = null;
    for(Level curr = Chain0.get1024("str");
        curr != null;
        curr = curr.up() ) {
        prev = curr;
    }
    return prev;
}
```

Linkage: results (hot)

	-TieredCompilation		+TieredCompilation	
	anonymous	lambda	anonymous	lambda
1K	0.47	0.80	0.35	0.62
4K	1.58	2.16	1.12	1.58
16K	4.96	5.62	4.22	4.67
64K	16.51	17.53	15.68	16.21

time, seconds

Linkage: results (cold)

	-TieredCompilation		+TieredCompilation	
	anonymous	lambda	anonymous	lambda
1K	7.24	0.95	6.98	0.77
4K	16.64	2.46	16.16	1.84
16K	22.44	5.92	21.25	4.90
64K	34.52	18.20	33.34	16.33

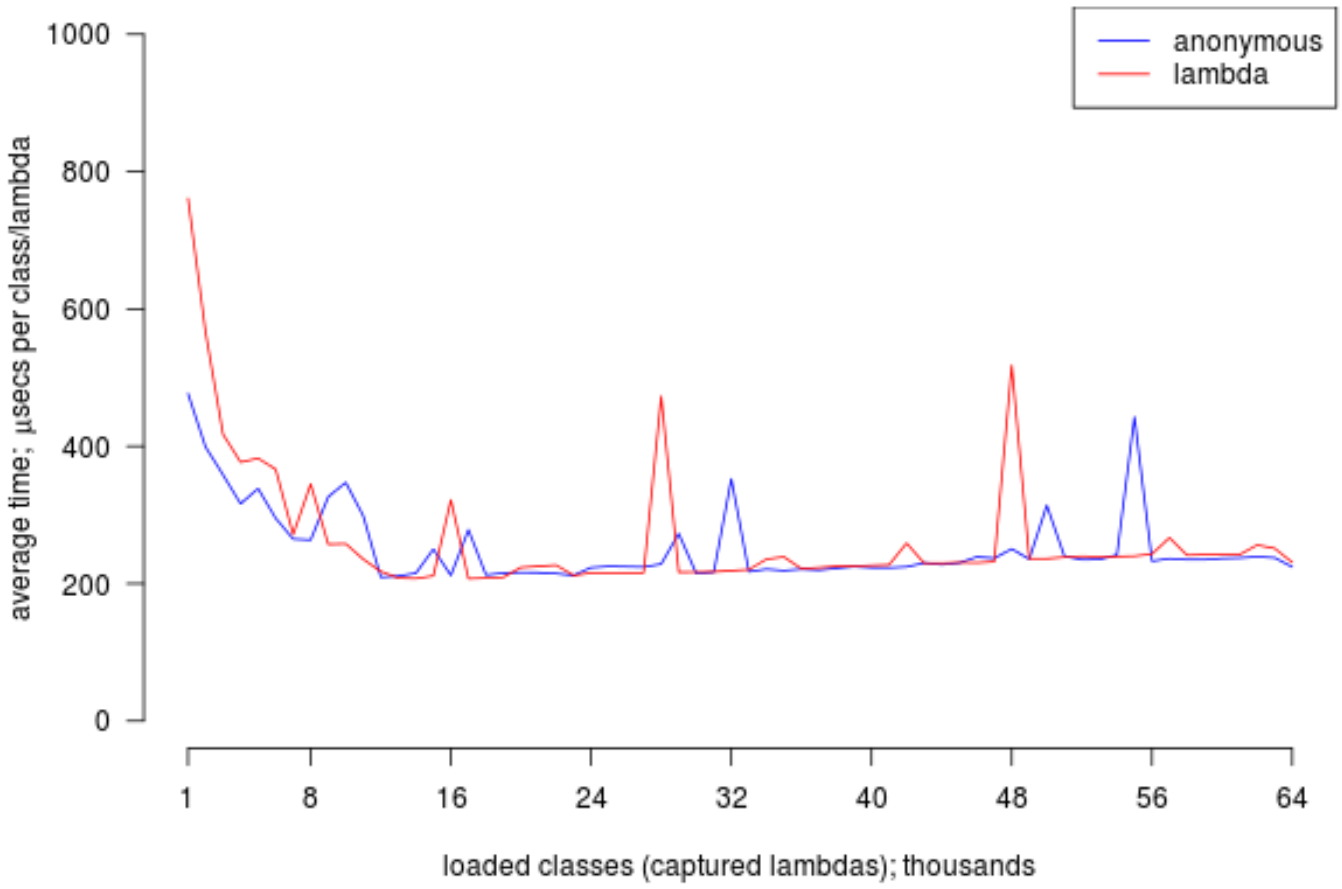
time, seconds

Linkage: results (cold)

	-TieredCompilation		+TieredCompilation	
	anonymous	lambda	anonymous	lambda
1K	1440%	19%	1894%	24%
4K	953%	14%	1343%	16%
16K	352%	5%	404%	5%
64K	109%	4%	113%	1%

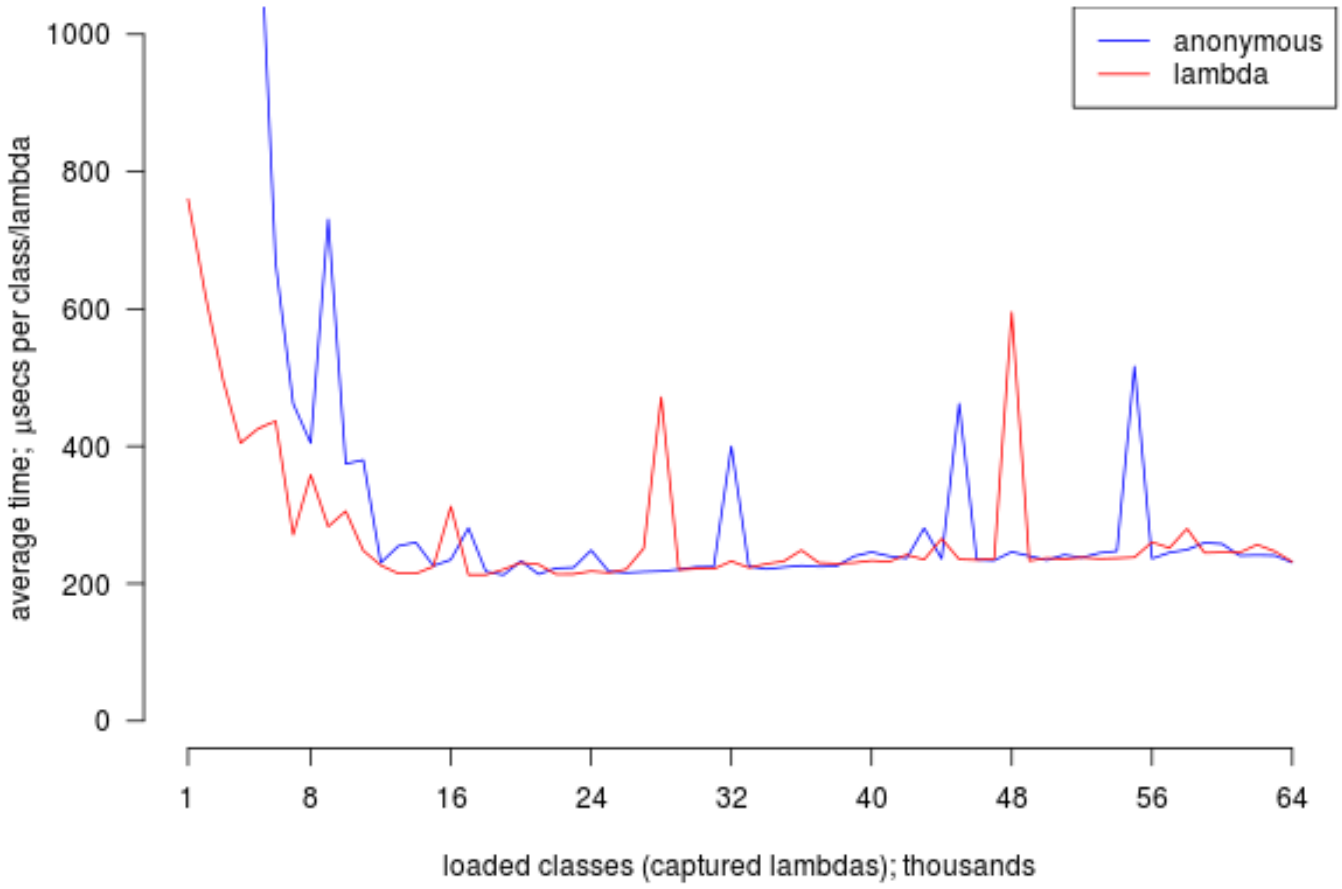
performance hit

Linkage: results (hot)



ORACLE

Linkage: results (cold)



ORACLE

Linkage: Main contributors (lambda)

25% - resolve_indy
    13% - link_MH_constant
44% - LambdaMetaFactory
    20% - Unsafe.defineClass

Capture

Non-capture lambda: benchmarks

```
public static Supplier<String> lambda(){
    return () -> "42";
}
```

Non-capture lambda: benchmarks

```
public static Supplier<String> lambda(){
    return () -> "42";
}

public static Supplier<String> anonymous(){
    return new Supplier<String>() {
        @Override
        public String get() {
            return "42";
        }
    };
}
```

Non-capture lambda: benchmarks

```
public static Supplier<String> lambda(){
    return () -> "42";
}

public static Supplier<String> anonymous(){
    return new Supplier<String>() {
        @Override
        public String get() {
            return "42";
        }
    };
}

public static Supplier<String> baseline(){
    return null;
}
```

Non-capture lambda: results

	single thread	
baseline	5.29 ± 0.02	
anonymous	6.02 ± 0.02	
cached anonymous	5.36 ± 0.01	
lambda	5.31 ± 0.02	

average time, nsecs/op

Non-capture lambda: results

	single thread	max threads (4)
baseline	5.29 ± 0.02	5.92 ± 0.02
anonymous	6.02 ± 0.02	12.40 ± 0.09
cached anonymous	5.36 ± 0.01	5.97 ± 0.03
lambda	5.31 ± 0.02	5.93 ± 0.07

average time, nsecs/op

Capture: lambda

```
public Supplier<String> lambda() {
    String localString = someString;
    return () -> localString;
}
```

Instance size = 16 bytes [2]

[2] 64-bits VM, -XX:+CompressedOops

Capture: anonymous (static context)

```
public static Supplier<String> anonymous() {
    String localString = someString;
    return new Supplier<String>() {
        @Override
        public String get() {
            return localString;
        }
    };
}
```

Instance size = 16 bytes [3]

[3] 64-bits VM, -XX:+CompressedOops

Capture: anonymous (non-static context)

```
public Supplier<String> anonymous() {
    String localString = someString;
    return new Supplier<String>() {
        @Override
        public String get() {
            return localString;
        }
    };
}
```

Instance size = 24 bytes [3]

[3] 64-bits VM, -XX:+CompressedOops

Capture: results

	single thread	max threads
anonymous(static)	6.94 ± 0.03	13.4 ± 0.33
anonymous(non-static)	7.88 ± 0.09	18.7 ± 0.17
lambda	8.29 ± 0.04	16.0 ± 0.28

average time, nsec/op

Capture: results

	single thread	max threads
anonymous(static)	6.94 ± 0.03	13.4 ± 0.33
anonymous(non-static)	7.88 ± 0.09	18.7 ± 0.17
lambda	8.29 ± 0.04	16.0 ± 0.28

average time, nsec/op

Capture: exploring asm

```
...
mov     0x68(%r10),%ebp
                  ; *getstatic someString
mov     $0xefe53110,%r10d
                  ; metadata('Capture1$$Lambda$1')
movzbl 0x186(%r12,%r10,8),%r8d
add     $0xfffffffffffffffc,%r8d
test    %r8d,%r8d
jne     allocation_slow_path
mov     0x60(%r15),%rax
mov     %rax,%r11
add     $0x10,%r11
cmp     0x70(%r15),%r11
jae     allocation_slow_path
mov     %r11,0x60(%r15)
prefetchnta 0xc0(%r11)
mov     0xa8(%r12,%r10,8),%r10
mov     %r10,(%rax)
movl    $0xefe53110,0x8(%rax)
                  ; {metadata('Capture1$$Lambda$1')}
mov     %ebp,0xc(%rax)
                  ;*invokevirtual allocateInstance
...
```

Capture: exploring asm

```
...
mov     0x68(%r10),%ebp
                  ; *getstatic someString
mov     $0xefe53110,%r10d
                  ; metadata('Capture1$$Lambda$1')
movzbl 0x186(%r12,%r10,8),%r8d
add     $0xfffffffffffffffc,%r8d
test    %r8d,%r8d
jne     allocation_slow_path
mov     0x60(%r15),%rax
mov     %rax,%r11
add     $0x10,%r11
cmp     0x70(%r15),%r11
jae     allocation_slow_path
mov     %r11,0x60(%r15)
prefetchnta 0xc0(%r11)
mov     0xa8(%r12,%r10,8),%r10
mov     %r10,(%rax)
movl    $0xefe53110,0x8(%rax)
                  ; {metadata('Capture1$$Lambda$1')}
mov     %ebp,0xc(%rax)
                  ;*invokevirtual allocateInstance
...
```

← check
if class was initialized
(Unsafe.allocateInstance from jsr292 LF's)

Capture: benchmark

Can we find a benchmark or/and JVM environment where allocation size difference is significant?

Capture: benchmark

```
@GenerateMicroBenchmark
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@OperationsPerInvocation(SIZE)[4]
public Supplier<Supplier> chain_lambda() {
    Supplier<Supplier> top = null;
    for (int i = 0; i < SIZE; i++) {
        Supplier<Supplier> current = top;
        top = () -> current;
    }
    return top;
}
```

[4] SIZE==1048576

Capture: chain results

- out of the box

	1 thread	
anonymous	8.4 ± 1.1	
lambda	6.7 ± 0.6	

- -Xmx1g

anonymous	11 ± 1.2	
lambda	7.6 ± 0.4	

- -Xmx1g -Xmn800m

anonymous	8.1 ± 0.9	
lambda	6.0 ± 0.7	

average time, nsecs/op

Capture: beware of microbenchmarks

- out of the box

	1 thread	4 threads
anonymous	8.4 ± 1.1	47 ± 16
lambda	6.7 ± 0.6	28 ± 10

- -Xmx1g

anonymous	11 ± 1.2	84 ± 9
lambda	7.6 ± 0.4	47 ± 20

- -Xmx1g -Xmn800m

anonymous	8.1 ± 0.9	123 ± 18
lambda	6.0 ± 0.7	28 ± 14

average time, nsecs/op

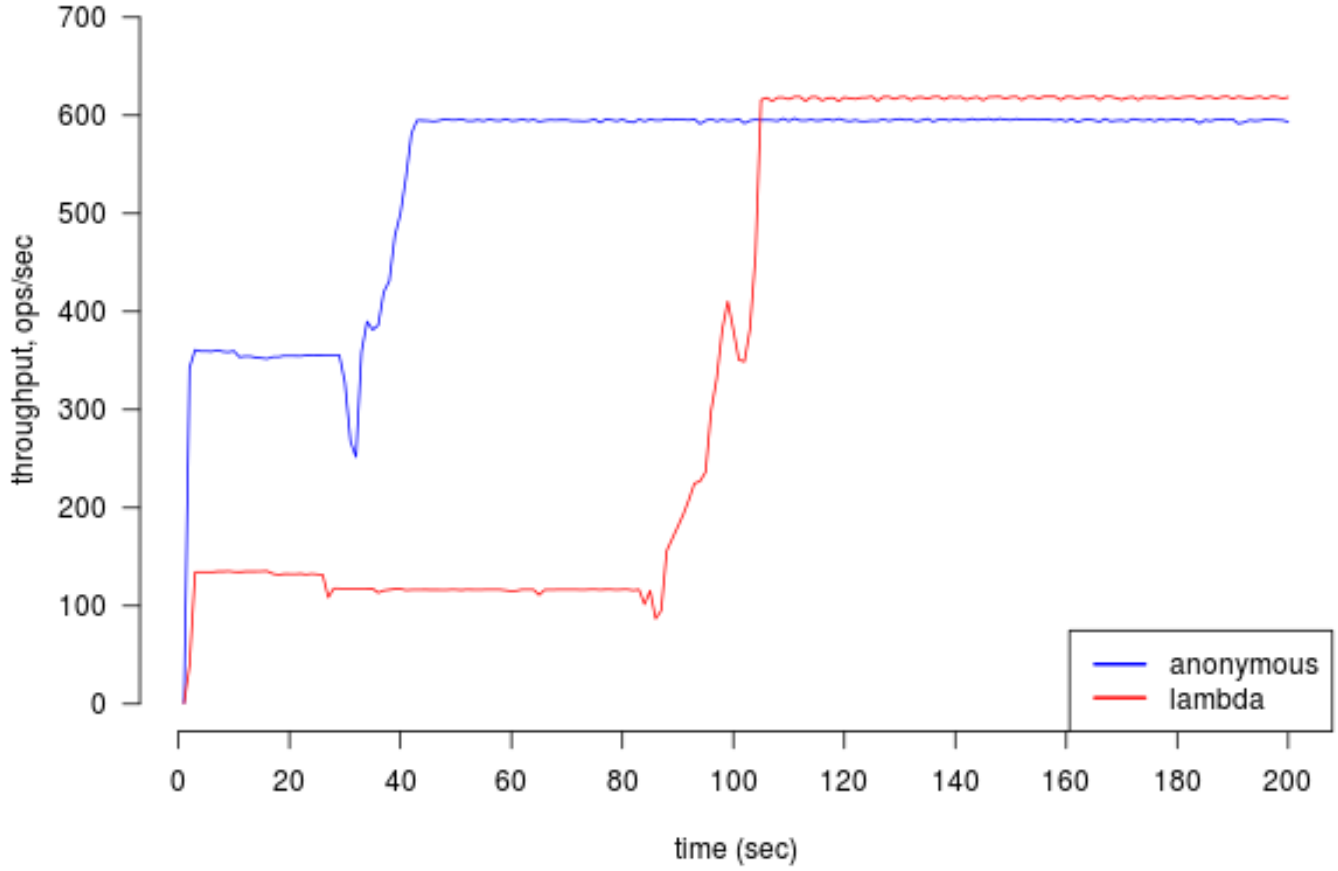
Capture warmup (time-to-performance)

Capture: time-to-performance

- lots of different lambdas (e.g. linkage benchmark)
- throughput (-bm Throughput)
- no warmup (-wi 0)
- get throughput each second (-r 1)
- large amount of iterations (-i 200)

Capture: time-to-performance

4K chain; -XX:-TieredCompilation



Capture: lambda slow warmup

Main culprits:

- jsr292 LF implementation
  - layer of LF's generated methods

Capture: LF's inline tree

```
@ 1   oracle.micro.benchmarks.jsr335.lambda.chain.lamb.cap1.common.Chain3:get3161
  @ 1   java.lang.invoke.LambdaForm$MH/1362679684::linkToCallSite
    @ 1   java.lang.invoke.Invokers::getCallSiteTarget
      @ 4   java.lang.invoke.ConstantCallSite::getTarget
    @ 10   java.lang.invoke.LambdaForm$MH/90234171::convert
      @ 9   java.lang.invoke.LambdaForm$DMH/1041177261::newInvokeSpecial_L_L
        @ 1   java.lang.invoke.DirectMethodHandle::allocateInstance
          @ 12   sun.misc.Unsafe::allocateInstance (0 bytes)   (intrinsic)
        @ 6   java.lang.invoke.DirectMethodHandle::constructorMethod
        @ 16   ...$$Lambda$936::<init>
          @ 1   java.lang.invoke.MagicLambdaImpl::<init> (5 bytes)
            @ 1   java.lang.Object::<init> (1 bytes)
```

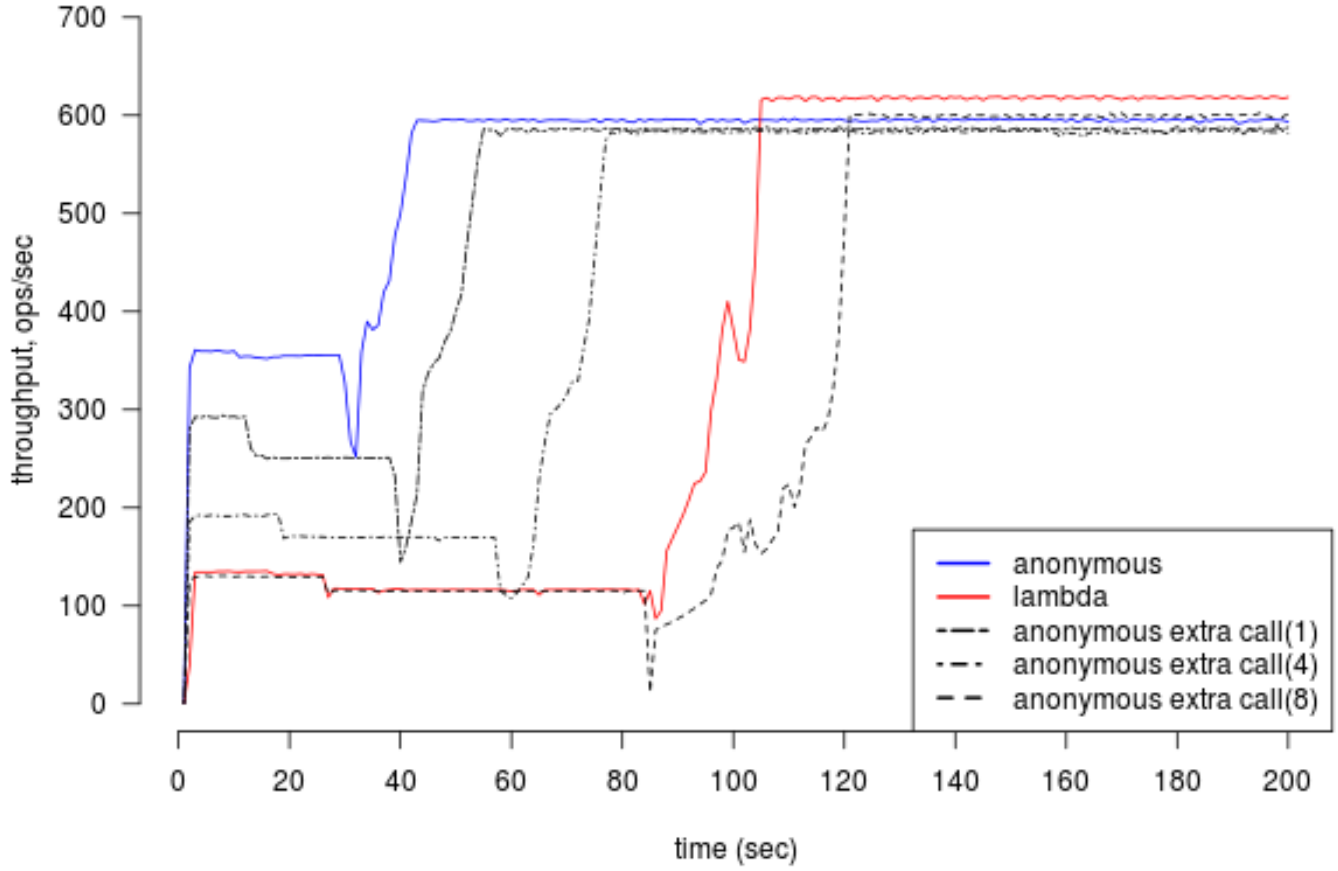
Capture: lambda slow warmup

Main culprits:

- jsr292 LF implementation
  - layer of LF's generated methods
- HotSpot (interpreter)
  - calling a method is hard (even simple delegating methods)

Capture: time-to-performance

`extra invocations for anonymous`



ORACLE

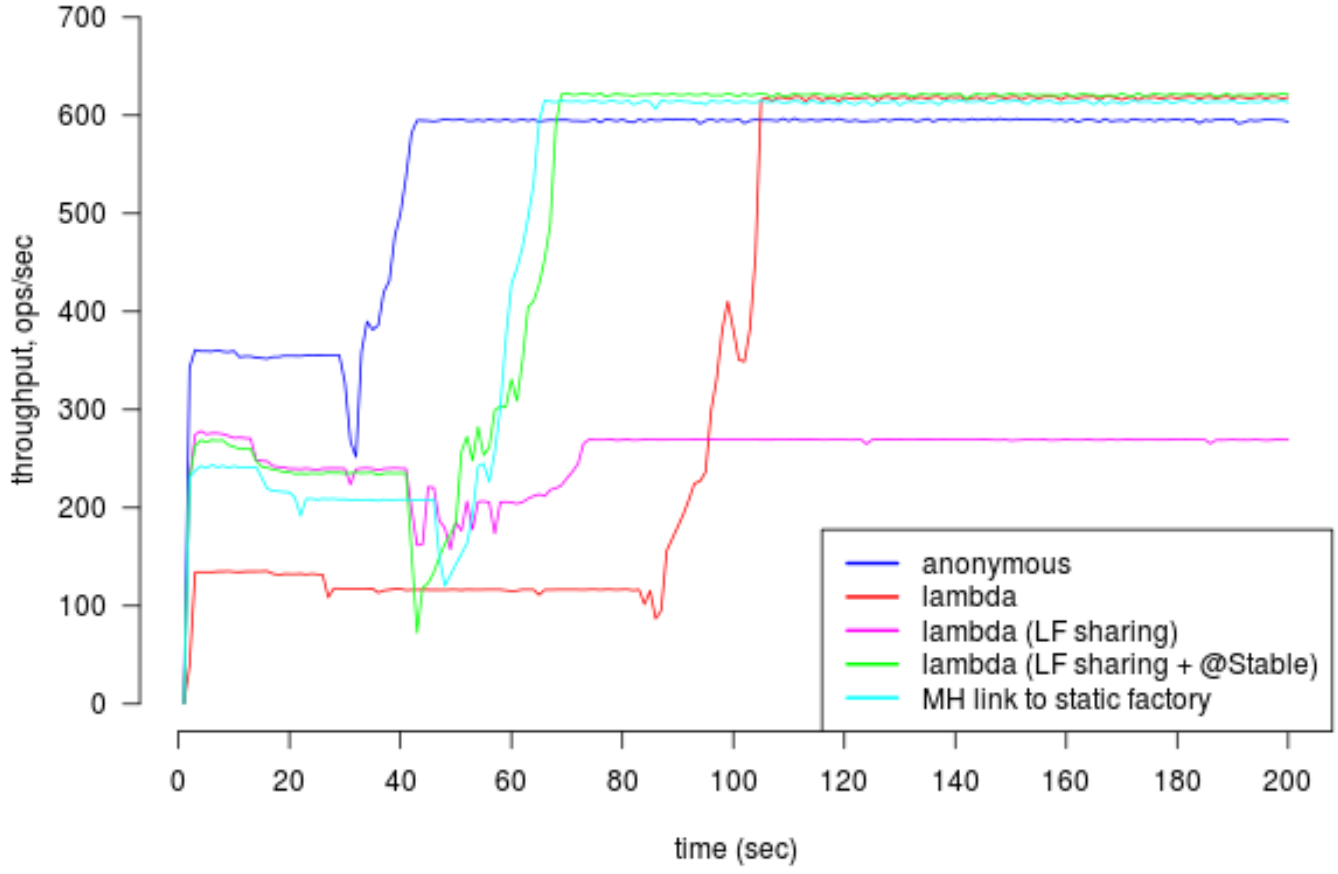
Capture: lambda slow warmup

Areas for improvement:

- Lambda runtime representation?
- jsr292 LF implementation?
- Tiered Compilation?
- HotSpot (interpreter)?

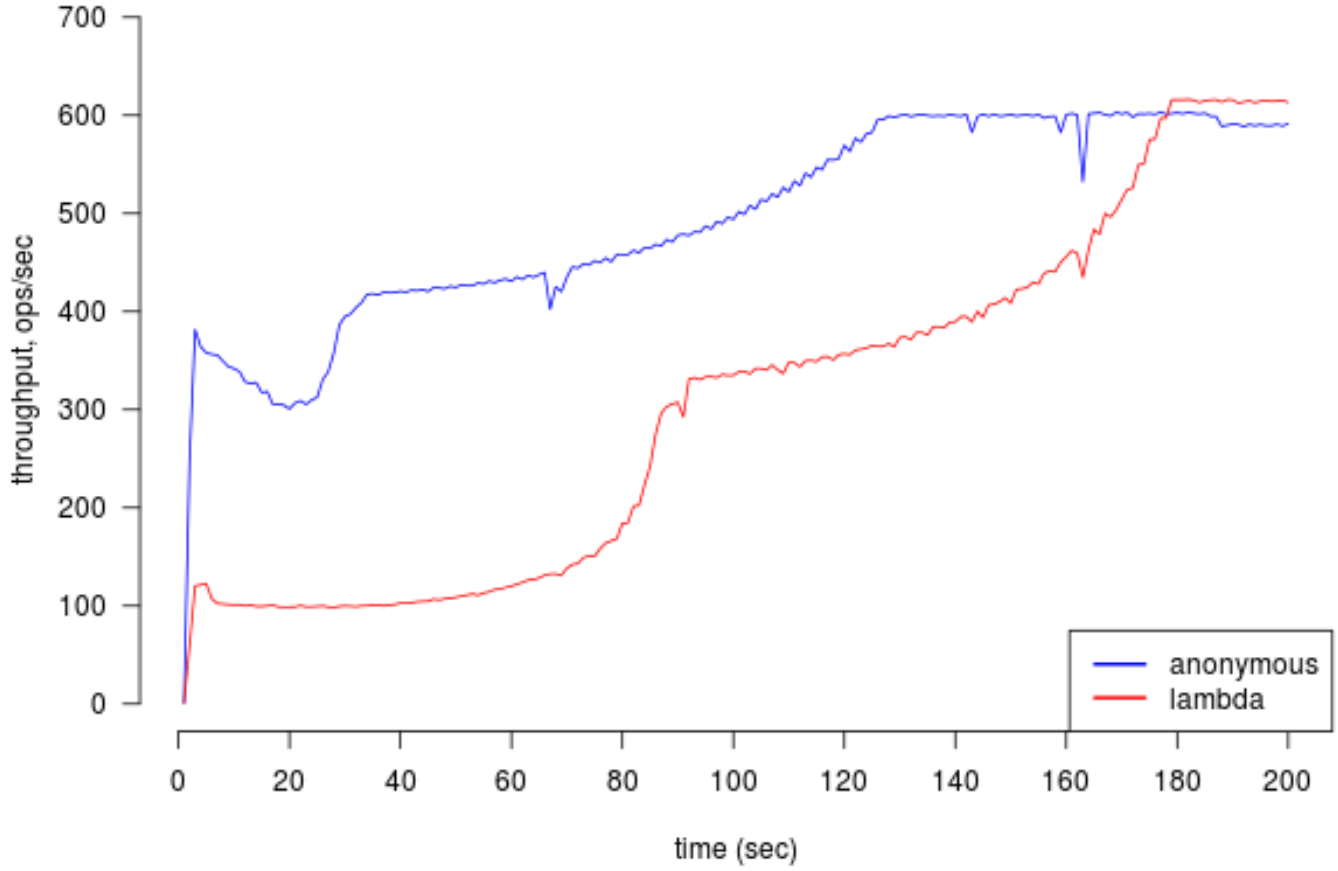
Capture: time-to-performance

`4K chain; -XX:-TieredCompilation`



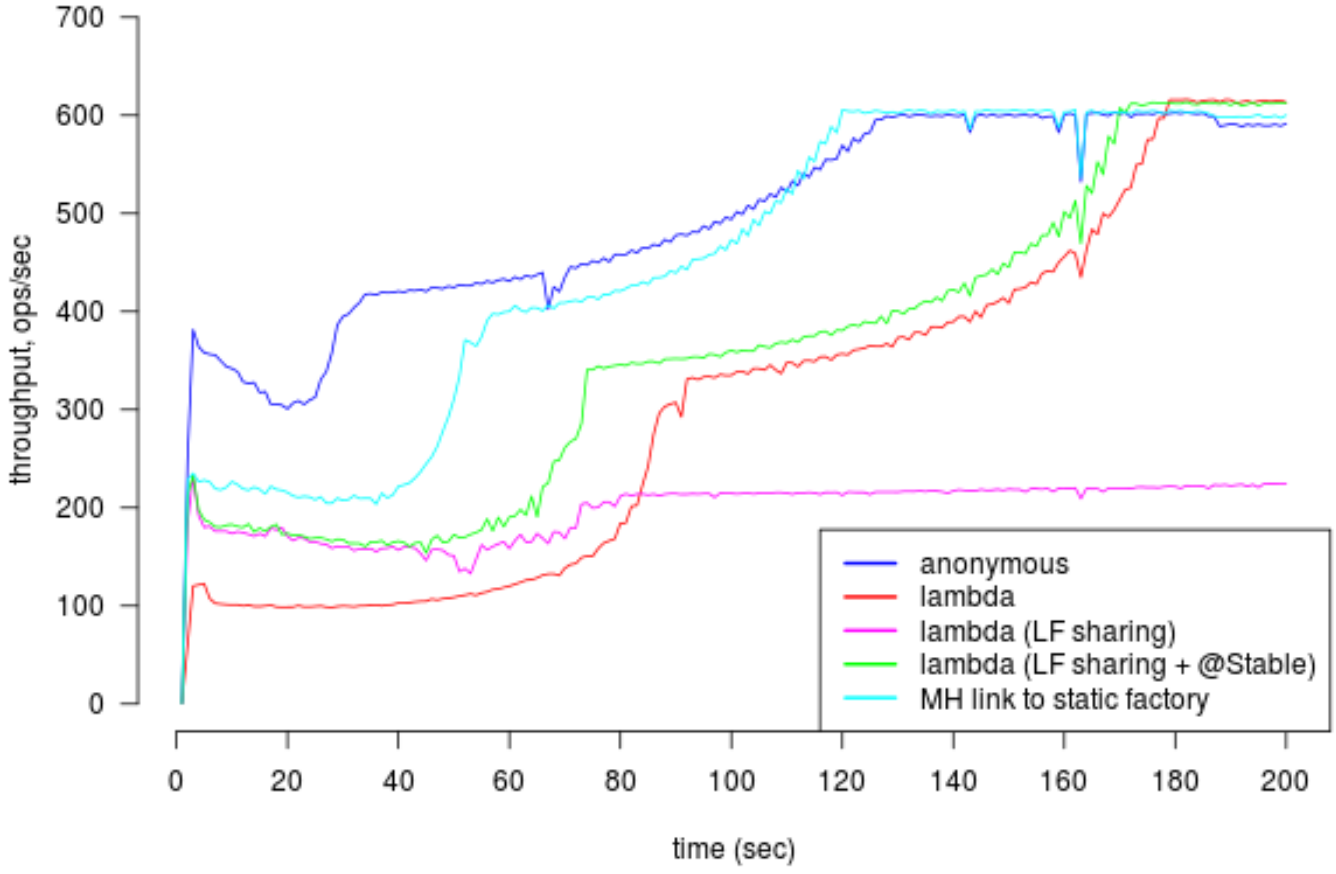
Capture: time-to-performance

`4K chain; -XX:+TieredCompilation`



Capture: time-to-performance

`4K chain; -XX:+TieredCompilation`



Invocation

Invocation: performance

Lambda invocation behaves
exactly as anonymous class invocation

Invocation: performance

Lambda[5] invocation behaves exactly as anonymous class invocation

[5] current implementation

Lambda and optimizations

Inline: benchmark

```
public String id_lambda(){
    String str = "string";
    Function<String, String> id = s -> s;
    return id.apply(str);
}
```

Inline: benchmark

```
public String id_lambda(){
    String str = "string";
    Function<String, String> id = s -> s;
    return id.apply(str);
}

public String id_ideal(){
    String str = "string";
    return str;
}
```

Inline: results

ideal	5.38 ± 0.03
anonymous	5.40 ± 0.02
cached anonymous	5.37 ± 0.03
lambda	5.38 ± 0.02

average time, nsecs/op

Inline: asm

ideal, anonymous, cached anonymous:

```
...
mov   $0x7d75cd018,%rax   ;   {oop("string")}
...
```

lambda:

```
...
mov   $0x7d776c8b0,%r10   ;   {oop(a 'TestOpt0$$Lambda$1')}
mov   0x8(%r10),%r11d
cmp   $0xefe56908,%r11d   ;   {metadata('TestOpt0$$Lambda$1')}
jne   <invokeinterface_slowpath>
mov   $0x7d75cd018,%rax   ;   {oop("string")}
...
```

Scalar replacement: benchmark

```
public String sup_lambda(){
    String str = "string";
    Supplier<String> sup = () -> str;
    return sup.get();
}
```

Scalar replacement: benchmark

```
public String sup_lambda(){
    String str = "string";
    Supplier<String> sup = () -> str;
    return sup.get();
}

public String sup_ideal(){
    String str = "string";
    return str;
}
```

Scalar replacement: results

ideal	5.49 ± 0.03
anonymous	5.52 ± 0.02
lambda	5.53 ± 0.02

average time, nsecs/op

Scalar replacement: asm

ideal, anonymous, lambda:

```
...
mov   $0x7d75cd018,%rax   ;   {oop("string")}
...
```

Streams

Lazy vs Eager: benchmark

```
List<Integer> list = new ArrayList<>();

@GenerateMicroBenchmark
public int forEach_4filters() {
    Counter c = new Counter();
    list.stream()
        .filter(i -> (i & 0xf) == 0)
        .filter(i -> (i & 0xff) == 0)
        .filter(i -> (i & 0xfff) == 0)
        .filter(i -> (i & 0xffff) == 0)
        .forEach(c::add);
    return c.sum;
}
```

Lazy vs Eager: benchmark

```
List<Integer> list = new ArrayList<>();

@GenerateMicroBenchmark
public int forEach_3filters() {
    Counter c = new Counter();
    list.stream()

        .filter(i -> (i & 0xff) == 0)
        .filter(i -> (i & 0xfff) == 0)
        .filter(i -> (i & 0xffff) == 0)
        .forEach(c::add);
    return c.sum;
}
```

Lazy vs Eager: benchmark

```
List<Integer> list = new ArrayList<>();

@GenerateMicroBenchmark
public int forEach_2filters() {
    Counter c = new Counter();
    list.stream()


        .filter(i -> (i & 0xfff) == 0)
        .filter(i -> (i & 0xffff) == 0)
        .forEach(c::add);
    return c.sum;
}
```

Lazy vs Eager: benchmark

```
@GenerateMicroBenchmark
public int iterator_4filters() {
    Counter c = new Counter();
    Iterator<Integer> iterator = list
            .stream()
            .filter(i -> (i & 0xf) == 0)
            .filter(i -> (i & 0xff) == 0)
            .filter(i -> (i & 0xfff) == 0)
            .filter(i -> (i & 0xffff) == 0)
            .iterator();
    while (iterator.hasNext()) {
            c.add(iterator.next());
    }
    return c.sum;
}
```

Lazy vs Eager: benchmark

```
@GenerateMicroBenchmark
public int for_4filters() {
    Counter c = new Counter();
    for(Integer i : list) {
        if((i & 0xf) == 0 &&
           (i & 0xff) == 0 &&
           (i & 0xfff) == 0 &&
           (i & 0xffff) == 0) {
            c.add(i);
        }
    }
    return c.sum;
}
```

Lazy vs Eager: results

	2 filters	3 filters	4 filters
forEach	3.0	1.8	1.7
iterator	1.1	0.7	0.6
for	2.4	2.4	2.3

throughput, ops/sec

Q&A?