
ORACLE®


Java™
ORACLE®

Startup Challenges

Non-lethal bathwater removal techniques for a snappier OpenJDK

Claes Redestad
Java SE Performance
Oracle

Safe Harbor Statement

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.

Startup challenges - at a glance…

- Intrinsic trade-off between startup and other concerns such as peak performance and footprint

  Startup often draws the shortest straw

- Death by a thousand cuts

  Inefficiencies - many of which individually escape detection - accumulate over time

- New language features tend towards dynamic setup

  Generating code lazily (lambdas, indified string concatenation, etc..) help performance at a startup cost

OpenJDK startup optimization… why bother?

- Broadens the range of applications the JVM can be a good fit for

  CLI tools, function-as-a-service …

- Consolidate efforts targetting embedded systems

- Fight the misconception that "Java is slow"

- Improved quality of life

Startup techniques

- CDS
- HotSpot AOT (jaotc)

No limits on applications; few - if any - drawbacks

- jlink

Gains from removing unused modules

- GraalVM native-image

"Native" startup times; several limitations

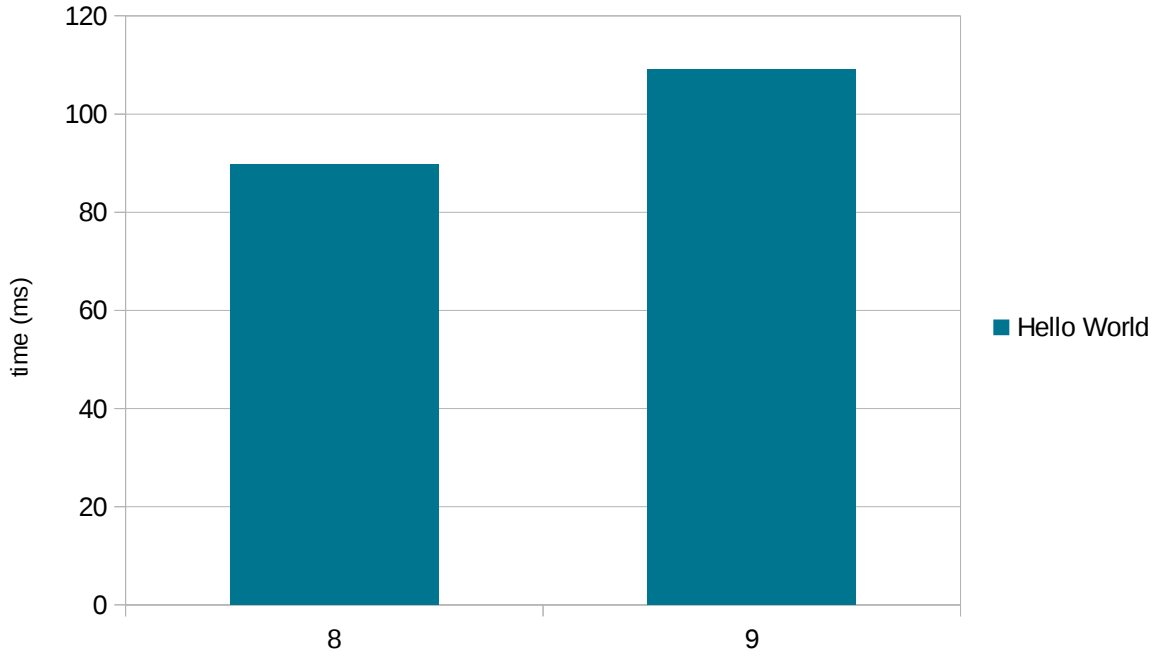
But let's start from the beginning…

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
}
```

Let's ignore that running a program like this from start to finish includes more than "startup" for now…

Hello World: JDK 9

- While the Java module system enables certain speed-ups, it does come with some initial overhead
- Regressions also due making G1 default, segmenting the code cache, implementing VM flag constraint checks(!) etc…
- Numerous startup optimizations softened the blow

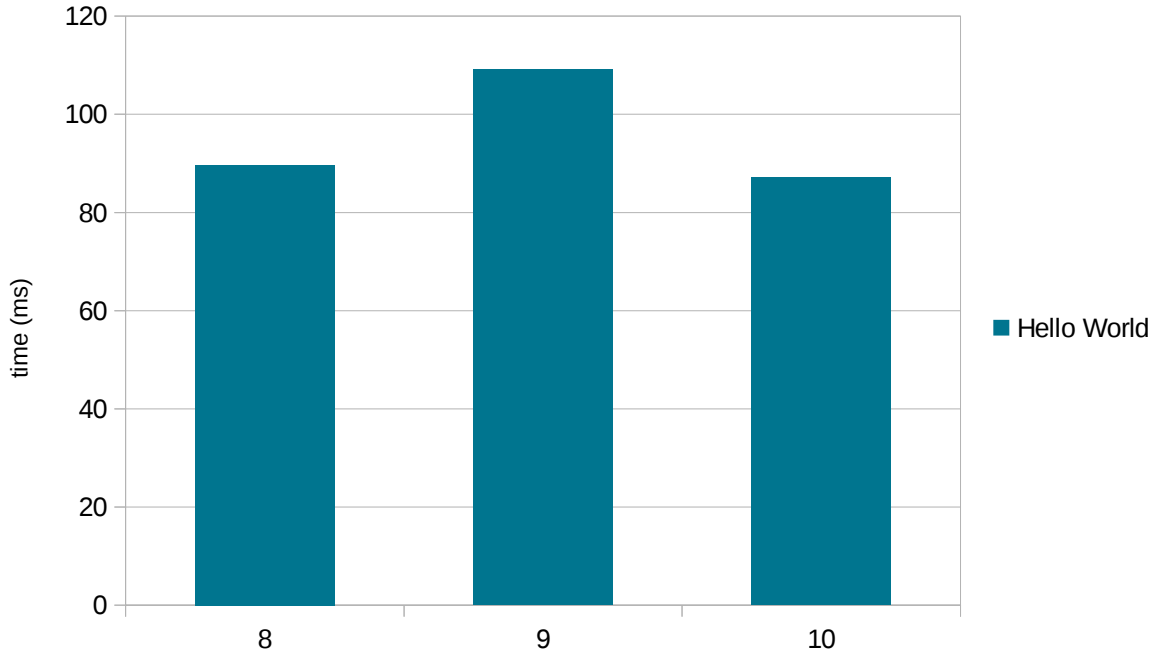


Unless otherwise stated: -Xshare:auto -Xmx32m
Intel(R) Xeon(R) E5-2630 v3 @ 2.40GHz
Ubuntu 16.04.3 LTS

Hello World: JDK 10

Turning the tide…

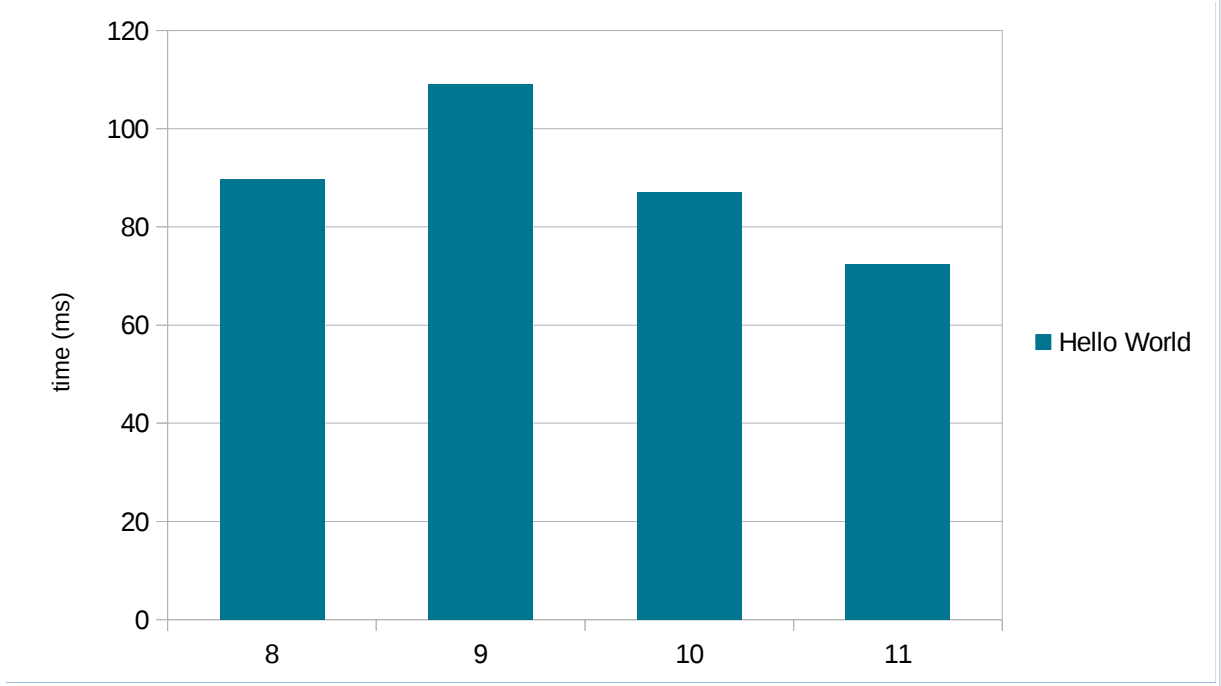
- Optimized module resolution
- Numerous small library and runtime optimizations
- G1 startup and footprint improvements
- CDS support for pre-resolving constant pool entries (e.g. String literals)



Hello World: JDK 11

Even faster…

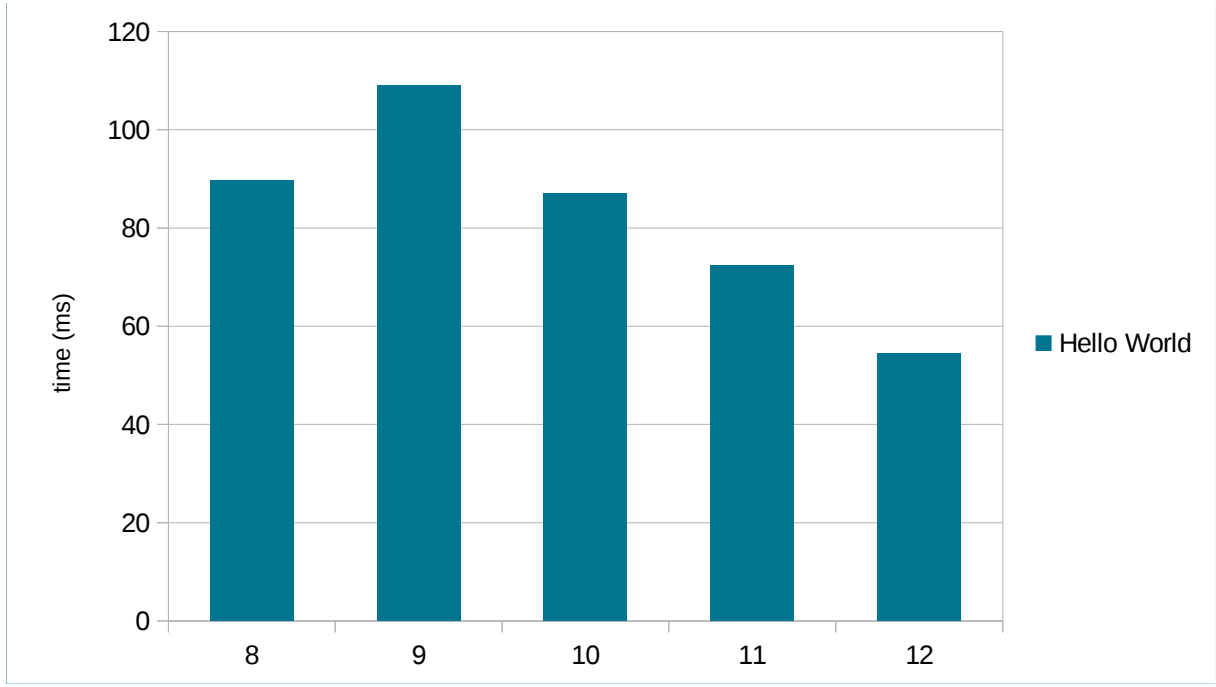
- Removal of Java EE and Java FX modules from the (Oracle) JDK brings a leaner standard image, which means less work needed to bootstrap modules
- Library optimizations continued



Hello World: JDK 12(?)

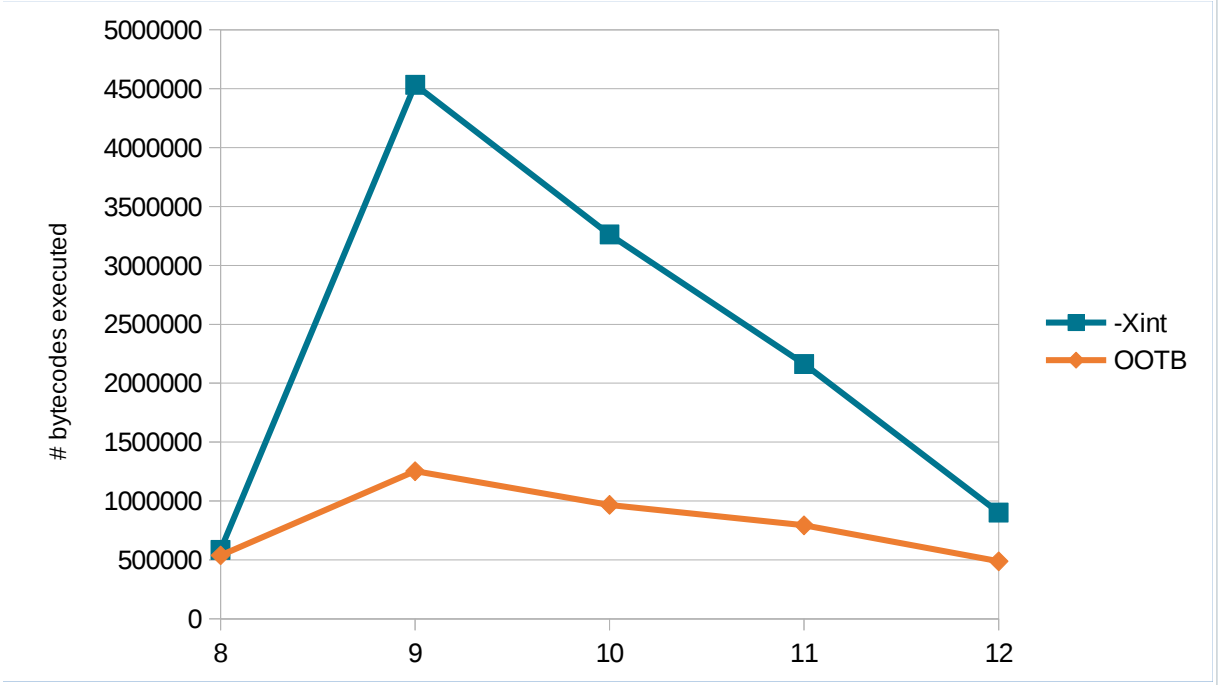
Early stage of development, but startup is already set to improve:

- Using CDS to serialize the default module graph (and more) allows a large cut in startup
- Dialing up the strictness of module encapsulation (`--illegal-access=deny`) is being considered, which would cut off another 2-3ms



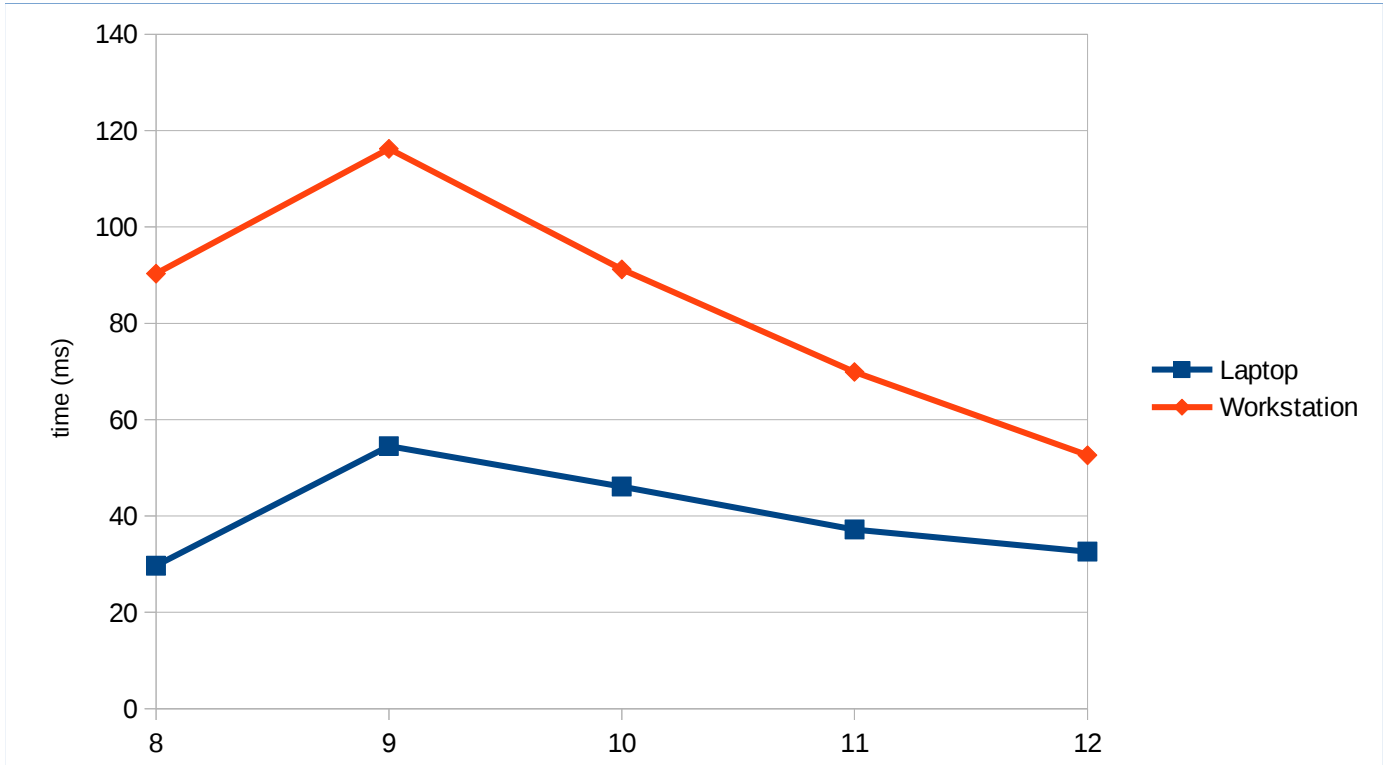
Hello World: Bytecodes executed

- Bootstrapping the module system (JDK 9) caused a ~9x increase in the number of "raw" bytecodes executed (`-Xint`)
- … but allowed us to do less work before JIT threads can be activated (System.initPhase1), meaning JIT kick in earlier
- … which means we quickly shift from the interpreter to more optimized code (mainly C1 at this early stage)



Hello form factors

- The shift to execute more logic in java during bootstrap means a higher dependency on our system's ability to quickly JIT oft-used methods
- On a dual-core laptop[1] we might struggle to break even with JDK 8



[1] Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz

Getting ahead of ourselves

JDK 9 introduced `jaotc`, an experimental tool to compile Java code into native executable code ahead-of-time.

Improving startup time and reduce time-to-performance is one of the goals.

However, AOT will typically be a loss for something as short-running as a Hello World…

Hello Lambda

Will the following program run as fast as HelloWorld?

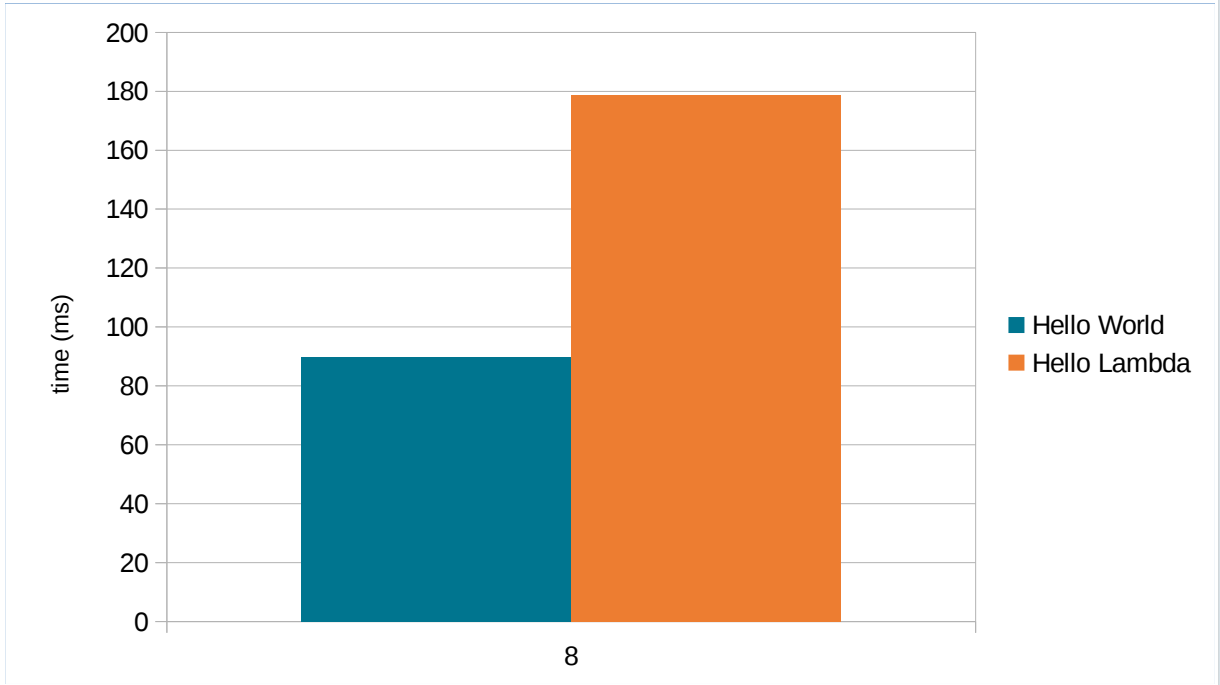
```
import java.util.function.Consumer;

public class HelloLambda {
  public static void main(String[] args) {
    Consumer<String> println = System.out::println;
    println.accept("Hello World!");
  }
}
```

Hello Lambda: JDK 8

On JDK 8, the cost of initializing a single lambda is almost 90ms on my workstation (8u172).

So during JDK 9 development, we started seeing similar startup regressions in various benchmarks, and every so often nice improvements had to be backed out…

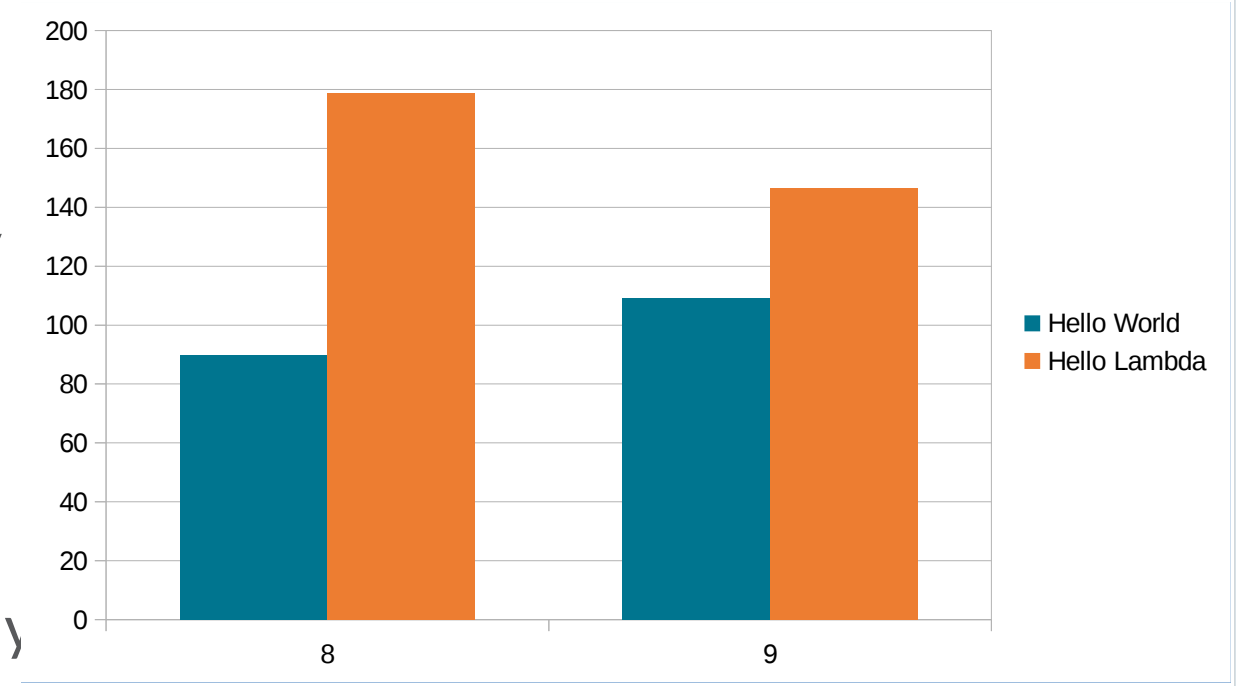
"Let's try to do better!" we exclaimed in unison. Probably.



Hello Lambda: JDK 9

Thus for JDK 9, an effort was made to reduce the one-off overheads (JDK-8086045):

- Various minor improvements to make the JSR-292 implementation lazier: don't eagerly initialize the LambdaForm interpreter (not used by default since 8u40) etc…
- Implement an experimental jlink plugin to generate a number of dynamically generated but runtime invariant classes ahead-of-time, mainly LFs

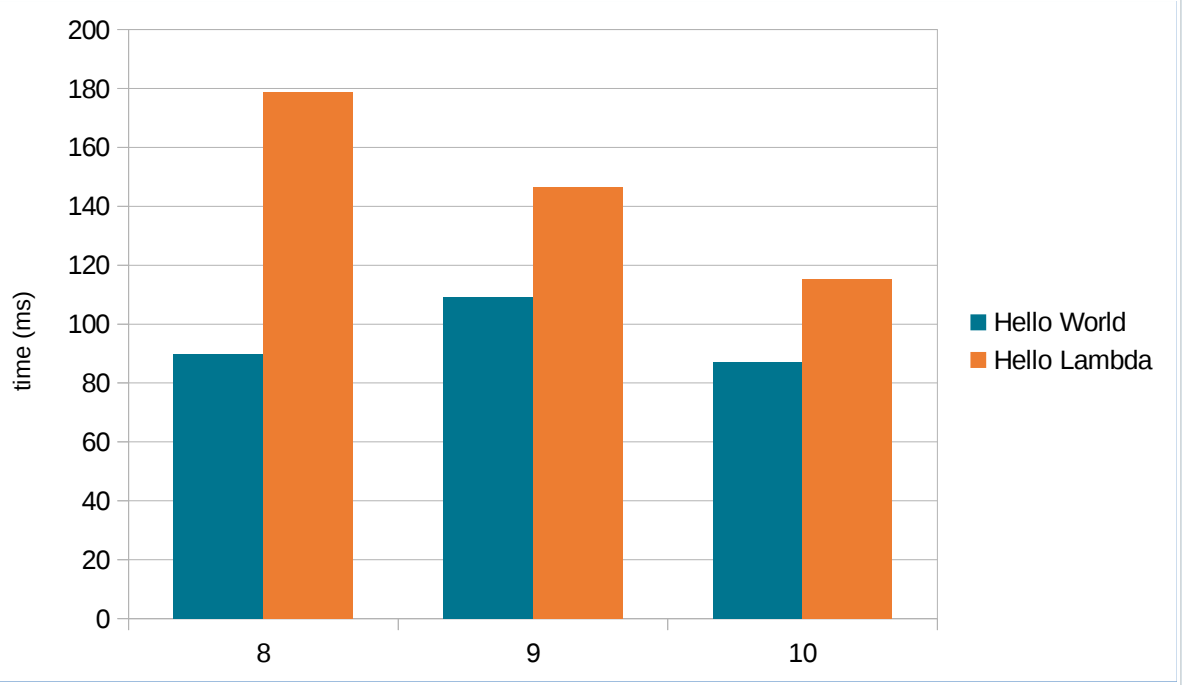


All-in-all 50-60% of initial overhead was removed.

Hello Lambda: JDK 10

JDK 10 saw only a few incremental improvements as engineering time was mainly spent on other things…

(6 months go by so quickly!)



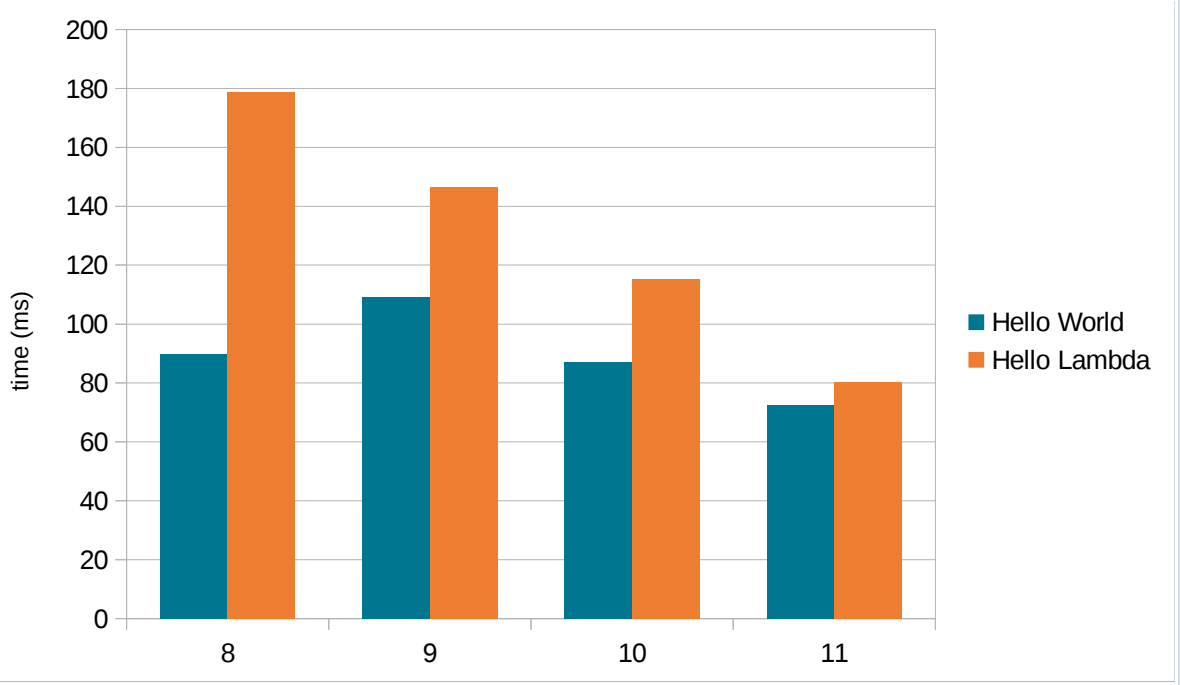
Hello Lambda: JDK 11

An unexpected breakthrough brought the initial Lambda setup cost down to a few milliseconds!

This comes from special-casing the `LambdaMetafactory::metafactory` BSM to be invoked exactly from BootstrapMethodInvoker.

Doing so removes the need to dynamically type-check arguments, see JDK-8194818

Can we use lambdas everywhere now?!



Scaling up: LambdaN

Let's generate a few simple test programs that create and execute an arbitrary amount of similar lambdas:

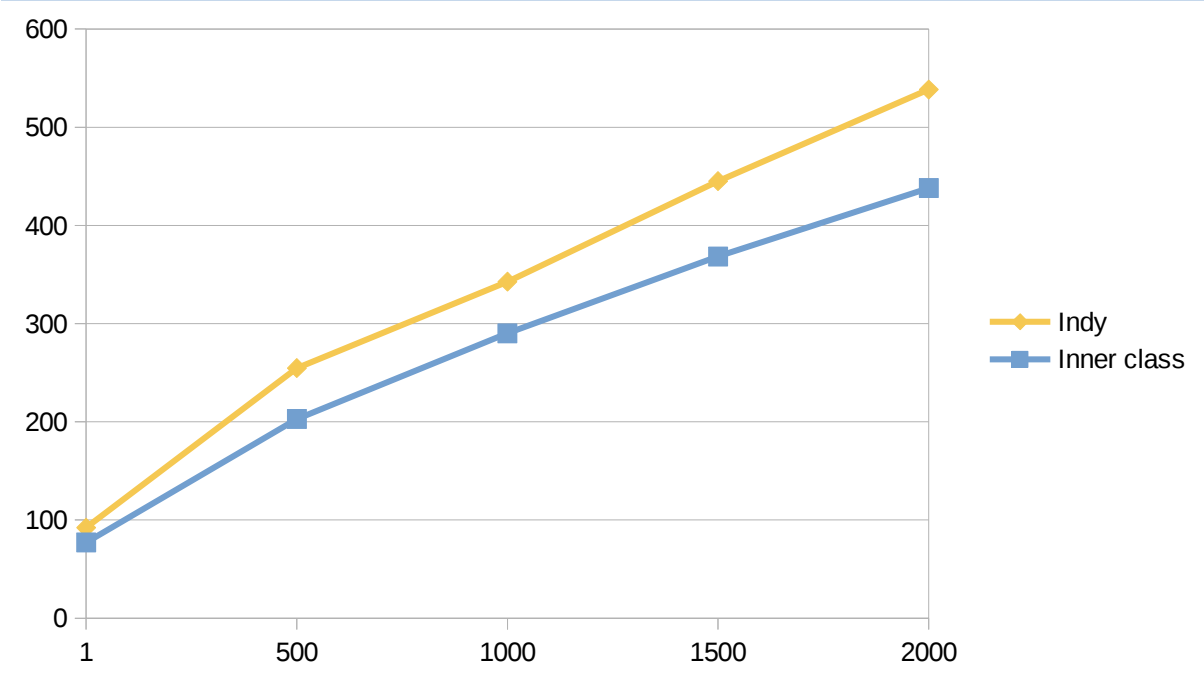
```
public class Lambda10 {
  public static void main(String[] args) {
    System.out.println(
      ((IntConsumer<Integer>)i -> i + 1).apply(0));
    ...
    System.out.println(
      ((IntConsumer<Integer>)i -> i + 10).apply(9));
  }
}
```

Scaling up: Lambda1-2000

Stepping from 1 to 2000 for this test of non-capturing lambdas we end up at ~0.187ms/lambda after warmup.

An equivalent test using anonymous inner classes executes at a rate of ~0.140ms/class.

This means there is a slightly higher scaling overhead



Lambda translation

Today, lambda expressions like the following:

```
System.out.println(((IntFunction<Integer>)i -> i + 1).apply(2));
```

.. is translated by javac into a private static method:

```
private static Integer lambda$main$0(int i) { return i + 1; }
```

... and a `invokedynamic` to retrieve an object implementing the functional interface (in this case `IntFunction`), which is then invoked using typical means (`invokeinterface`):

```
getstatic     #2                  // Get System.out
invokedynamic #3,  0              // References the static method above and a
                                  // bootstrap method used to link on first call
iconst_2
invokeinterface #4,  2            // IntFunction.apply
invokevirtual #5                  // out.println
```

Lambda linking

On first invocation of the `invokedynamic`, it will do some just-once bootstrapping.

The runtime first retrieves method handles for the target method and the bootstrap method (`MethodHandleNatives.linkMethodHandleConstant`)

The runtime does an upcall into java (`MethodHandleNatives.linkCallSite`), which invokes the provided bootstrap method, in this case `LambdaMetafactory.metafactory`.

The bootstrap method creates and returns a `CallSite` object, which is ultimately linked into place by the runtime.

Lambda class generation

To create a `CallSite` to link, `LambaMetafactory.metafactory` spins a class that implement the functional interface.

Main purpose of the generated class will be to call the static method generated by javac.

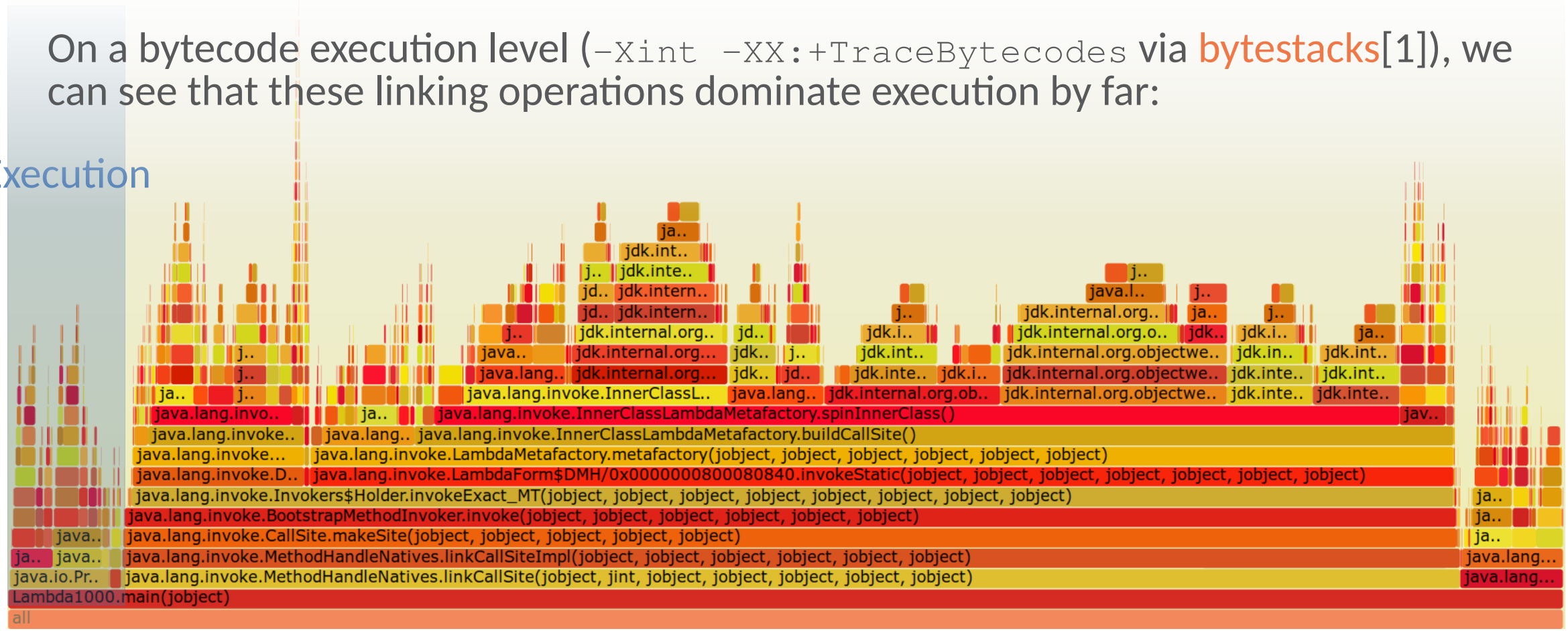
If the lambda is capturing, the `CallSite` will wrap a `MethodHandle` to the constructor of the lambda class.

If the lambda is non-capturing the lambda class doesn't carry any state, we don't really need to create an instance, so a single instance of the newly generated class is created and returned:

```
return new ConstantCallSite(MethodHandles.constant(...));
```

Profiling Lambda bootstrapping

On a bytecode execution level (`-Xint -XX:+TraceBytecodes` via bytestacks[1]), we can see that these linking operations dominate execution by far:



[1] https://github.com/cl4es/bytestacks

Introducing Condy

JDK 11 adds Constant Dynamic: Condy.

In essence, condy enhances the `ldc` bytecode so that creation of class-level constants can be delegated to a bootstrap method on first use.

Quite similar to how `invokedynamic` sets up its callsite on first use.

Using Condy to bootstrap Lambdas

If what your indy does is basically creating and returning an instance of something wrapped in a `ConstantCallSite` wrapped in a `MethodHandles.constant`...

... might as well use condy!

Making all non-capturing lambdas bootstrap using condy instead of indy is straightforwards.

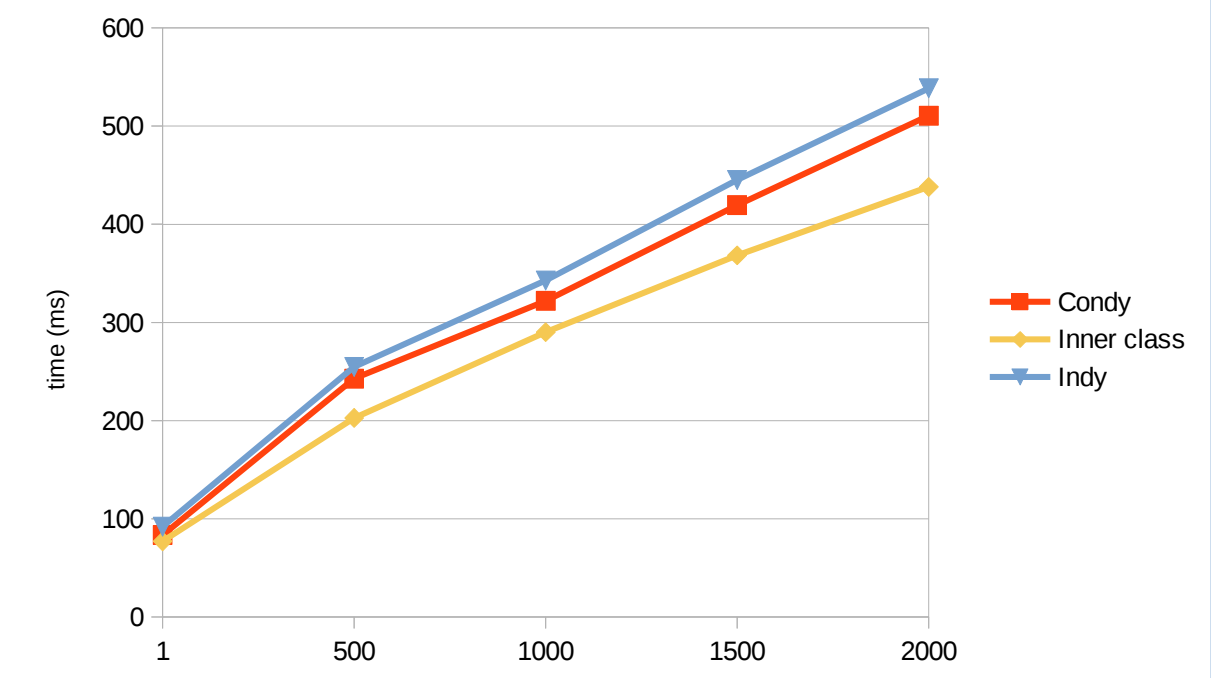
Doing so removes the creation of a few objects and removes a simple invocation to get hold of the singleton.

Lambda1-2000 - Condy Ed.

Using condy instead of indy for non-capturing lambda instantiation is a linearly scaling optimization:

- Around 10% total reduction in startup time at scale
- Removing 25% of the overhead relative to inner classes

This enhancement has been baking in amber for a while, and should make it into the mainline soon (JDK-8186216)



Indify String concatenation

JDK 9 introduced the ability to indify String concatenation (JEP 280): ISC for short.

Instead of desugaring to a series of `StringBuilder` operations, javac may emit an `invokedynamic` which delegates to a bootstrap method to set up the code necessary to perform the concatenation on demand.

By default, ISC generates a tree of MethodHandles that enables some very substantial optimizations, but there are some known startup overheads that have been linked to the overheads of initializing `java.lang.invoke`[1]

[1] https://shipilev.net/talks/jfokus-Feb2016-lord-of-the-strings.pdf

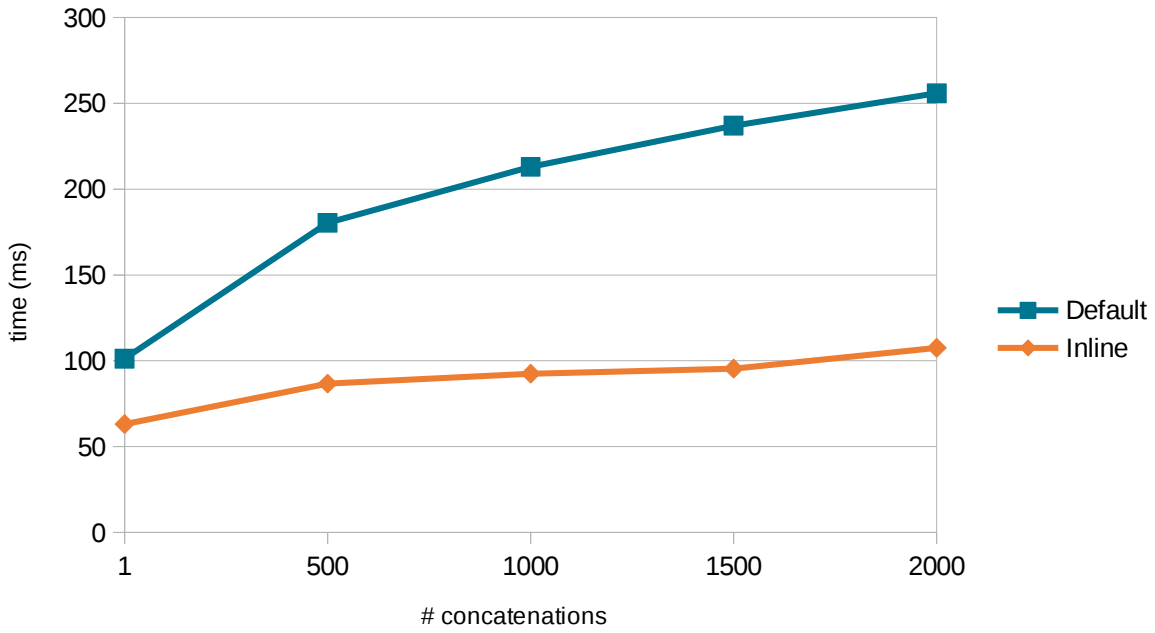
Hello String concatenation

Running code to test many repeated but similar String concatenations:

```
int i = 0;
out.println("Hello " + (i++));
out.println("Hello " + (i++));
...
```

.. using both the new default and the old javac strategy (inline), we get some interesting results…

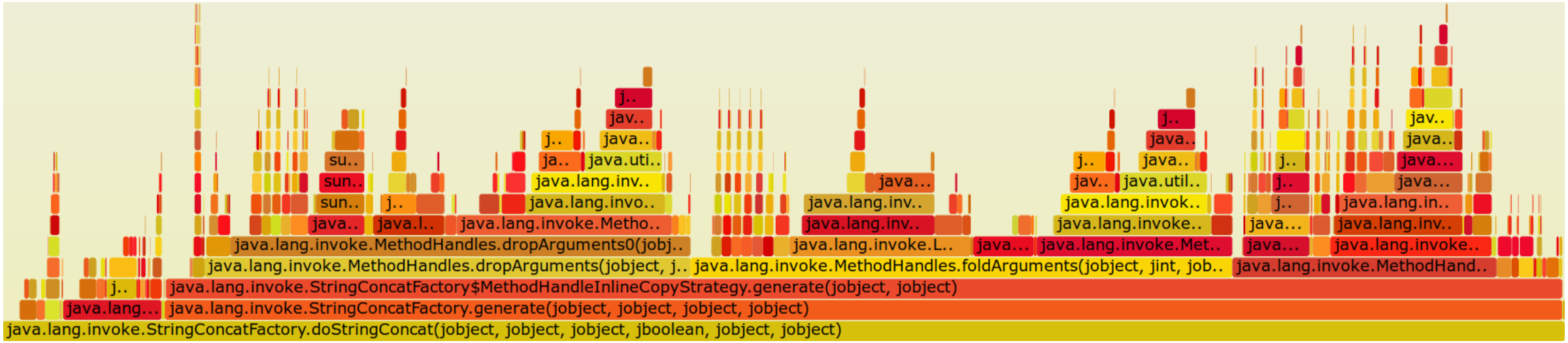
It seems there's more to it than merely paying for the shared infrastructure of `invokedynamic`…



Profiling ISC initialization

- Time mainly spent in `StringConcatFactory.doStringConcat`, doing various `MethodHandles.drop-/fold-/insertArguments` operations.

  This is the creation of the MethodHandle combinator tree making up the String concatenation. Creation will reuse cached internal primitives, e.g., `LambdaForms`, but even so there's a lot of ceremony involved…

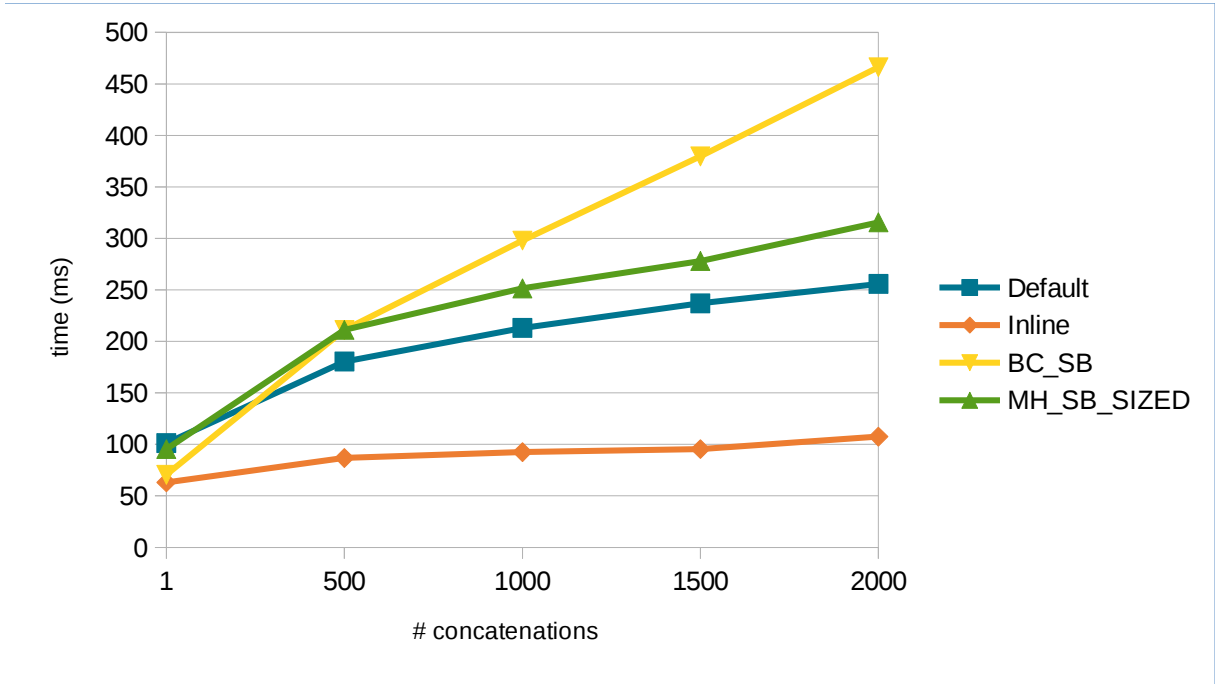


Are the alternative concatenation strategies any better?

ISC implements a few different strategies for string concatenation, for example `BC_SB` and `MH_SB_SIZED`.

The `BC_SB` strategy spin bytecode per call site that resembles the code emitted by javac before ISC. This is closer to the legacy code in initial costs, but scales poorly.

The simpler `MethodHandle`-based strategies provided, e.g. `MH_SB_SIZED` seem to scale worse than the default, too.



More indyfied, more overheads!

Several JDK projects aim to leverage indy in ways similar to ISC:

- `String.format` can effectively reuse much of the ISC mechanics to implement a routine that is up to 40x faster
- Valhalla and Amber explore routinely using indy to generate methods like `equals` and `hashCode` to allow better laziness and semantic coupling

Challenge: Setup is bound to have both one-off and per-callsite overheads in line with lambdas and/or ISC

Opportunity: *Optimizing* the underlying machinery will yield greater benefits.

Towards a better MethodHandle API

- Adding more high-level and custom MethodHandle combinators can be profitable

  JEP 274 (JDK 9) implemented a number of combinators for common patterns such as loops and try/finally blocks to ease composing more powerful expressions with fewer building blocks - there might be opportunities to improve on this

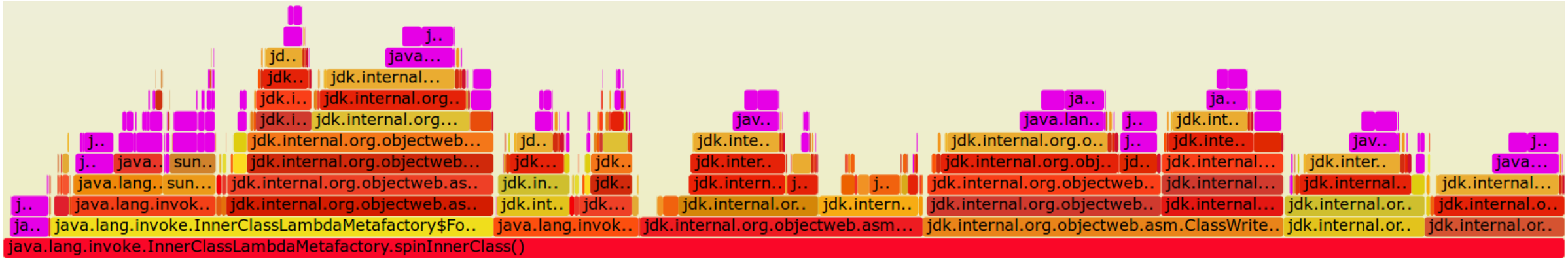
- Better support for spinning dynamically generated code ahead-of-time

- Isolated methods? http://openjdk.java.net/jeps/8158765

Towards better bytecode generation

To generate bytecode, `java.lang.invoke` spend a lot of time performing String transforms, e.g., `java.lang.String` -> `Ljava/lang/String.`

In total ~50% of the bytecode executed in the spinInnerClass method is spent in java.lang.String and friends, which seems a bit of a waste…



Better constants

JEP 334 would introduce an API to model methods, classes and dynamic constants descriptively, e.g., `java.lang.invoke.constant.ClassDesc`

This will allow us to simplify how we go from a descriptive model to bytecode, mainly remove some intermediate String representations.

Properly leveraged it seems hopeful that such an API will enable some wins.

The constants API is also a building block for adding language support to emit ldc and invokedynamic instructions directly (JEP 303). This effort is motivated partly by simplifying testing of increasingly complex usage of indy and condy, but might also unlock some startup improvements…

Why this obsession with startup?

In our little performance team we run a lot of benchmarks.

Startup benchmarks were often seen to regress for obscure reasons.

So I started scratching that itch …
… developed some crude tools to help analyze …
… helped develop better benchmarks …
… started fixing what I could.

Soon I was being asked to help out different projects to ensure startup was on track.

It's been a lot of fun.