

# Choose your own performance; a Project Leyden update

Dan Heidinga (@danheidinga)

Software Architect, Java Platform Group

Oracle

# Application performance

300

Typical

Time per task

225

150

75

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

Task repetitions

# Project Leyden

- Project Leyden is about improving the *startup* and *warmup* of Java applications
  - *Startup* is the time it takes to get to the first useful unit of work
  - *Warmup* is the time it takes for the application to reach peak performance
- Both include work done by the application (e.g., parsing configuration files), and work done *on behalf* of the application (e.g., loading and compiling classes)
- Startup and warmup are an issue for Java applications because Java is *highly dynamic*

# Java is highly dynamic

- Wait, what? I thought Java was a “static” language!
  - The terms “static” and “dynamic” are complex and frequently abused
- Assuming “static” means “at compile time” and dynamic means “at run time” (already a bad assumption), is Java statically typed, or dynamically typed?
  - Answer: yes!
  - Java has a static type system, enforced at compile time
  - Java also has a dynamic type system, enforced at run time
- The two have much overlap (e.g., both obey the same subtyping rules), but also significant differences in both directions
  - Some things are only part of the static type system: generics, checked exceptions, definite assignment analysis, overload resolution, etc
  - Others are dynamic but not static: `ArrayStoreException` for array writes
- *Java is both statically and dynamically typed*

# Static vs dynamic

## Static (sta-tic)

1. exerting force by reason of weight alone without motion
2. of or relating to bodies at rest or forces in equilibrium
3. showing little change (a *static* population)
4. characterized by a lack of movement, animation, or progression
5. standing or fixed in one place (see *stationary*)

## Dynamic (dy-nam-ic)

1. marked by usually continuous and productive activity or change (a *dynamic* city)
2. Energetic, forceful (a *dynamic* personality)
3. of or relating to physical force or energy

# Static vs dynamic

- The main distinction here is *changing* vs *unchanging*
  - One aspect is *what* is changing or not
  - In the context of languages, we often think only about static vs dynamic *type checking*
- But, Java has
  - Dynamic typing (array store checks, casting)
  - Dynamic class loading and verification
  - Dynamic class redefinition
  - Dynamic compilation (JITting)
  - Dynamic recompilation (deoptimization)
  - Dynamic linkage and access control
  - Dynamic dispatch (virtual methods)
  - Dynamic introspection (instanceof, reflection)
  - ...
- Java is pretty dynamic!

# Static vs dynamic

- Another important distinction is *over what period* is something changing or not
  - In the context of languages, we often think only about *compile vs run time*
  - In reality, there are many interesting phase transitions, often fine-grained ones

```
static final int seed = CesiumSample.getRandomNumber();
```

- Is this static or dynamic?
  - Well, it says “static”
  - But, the initializer runs at run time, which sounds dynamic
  - But but, the variable’s value is held constant for its visible lifetime (from class initialization to JVM exit), which sounds static
  - But but but, this lifetime does not coincide with “compile vs run time”
- The JVM optimizes static final fields aggressively, inlining them into compiled code, even though their values were “born dynamic”

# Static and dynamic: choose both

- The Java answer is rarely “choose one, lose one”
  - Instead, static and dynamic reasoning are intertwined and balanced
- Optimizing entities that “look static” but were “born dynamic” is a core JVM competency
  - Static field initialization
  - Class loading
  - Many, many JIT optimizations
- As is hiding the evidence if the thing that “looks static” eventually changes
  - CHA-based speculative optimization
  - Profile-driven deoptimization



# What happens during startup?

- Activities that are part of your program or framework (and any libraries you might use)
  - Reading config files
  - Scanning for annotations
  - Opening sockets, registering listeners
  - Creating loggers
- JVM activities on behalf of your program
  - Class loading
    - Reading classes from disk
    - Classfile validation and metadata construction
    - Running static initializers
  - Interpretation
  - Profile gathering
  - Callsite linkage, constant pool resolution

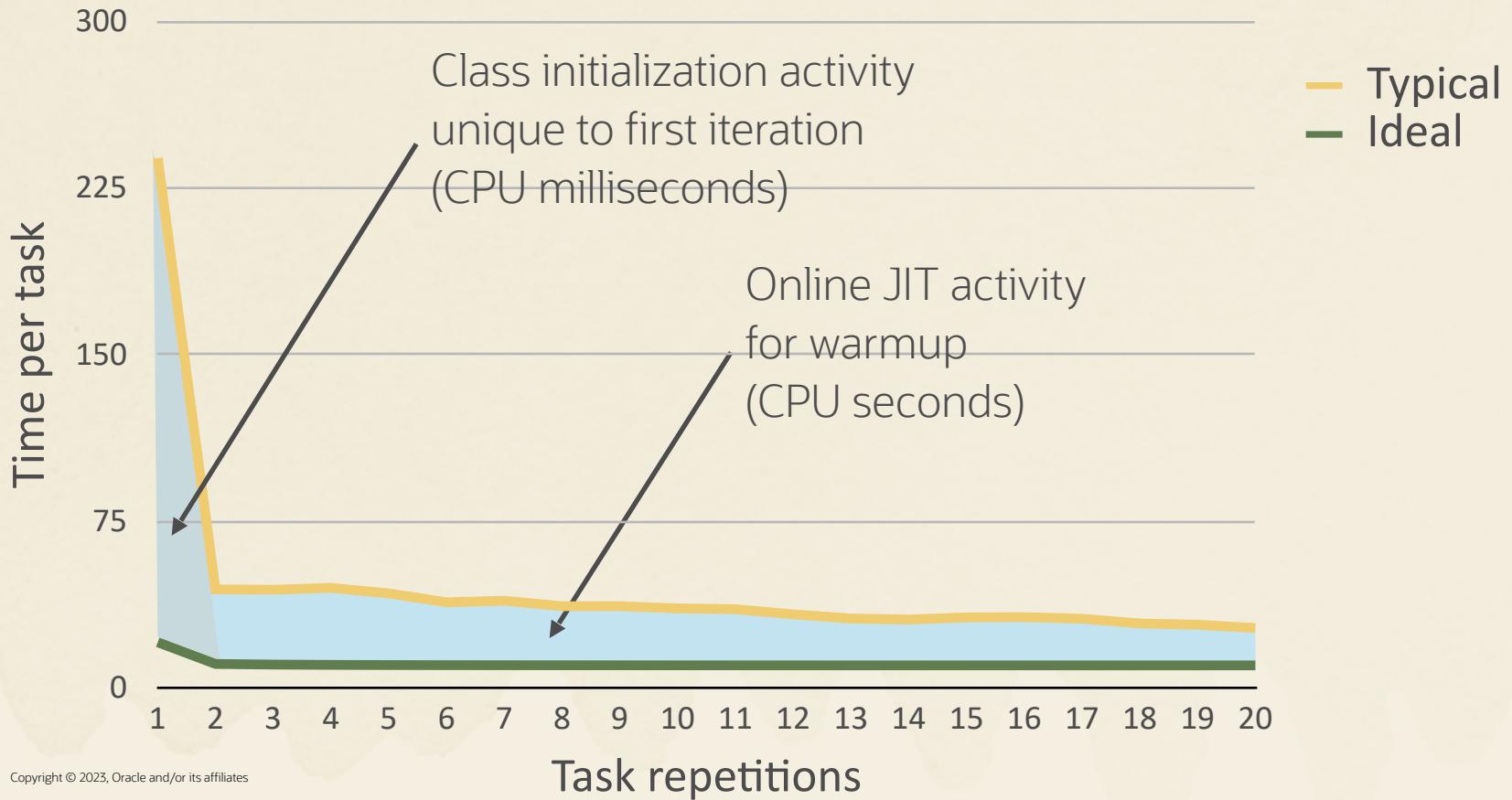
# What happens during warmup?

- Activities that are part of your program or framework
  - Populating caches
- JVM activities on behalf of your program
  - JIT compilation of hot code
    - Tiered – C1 vs C2/Graal

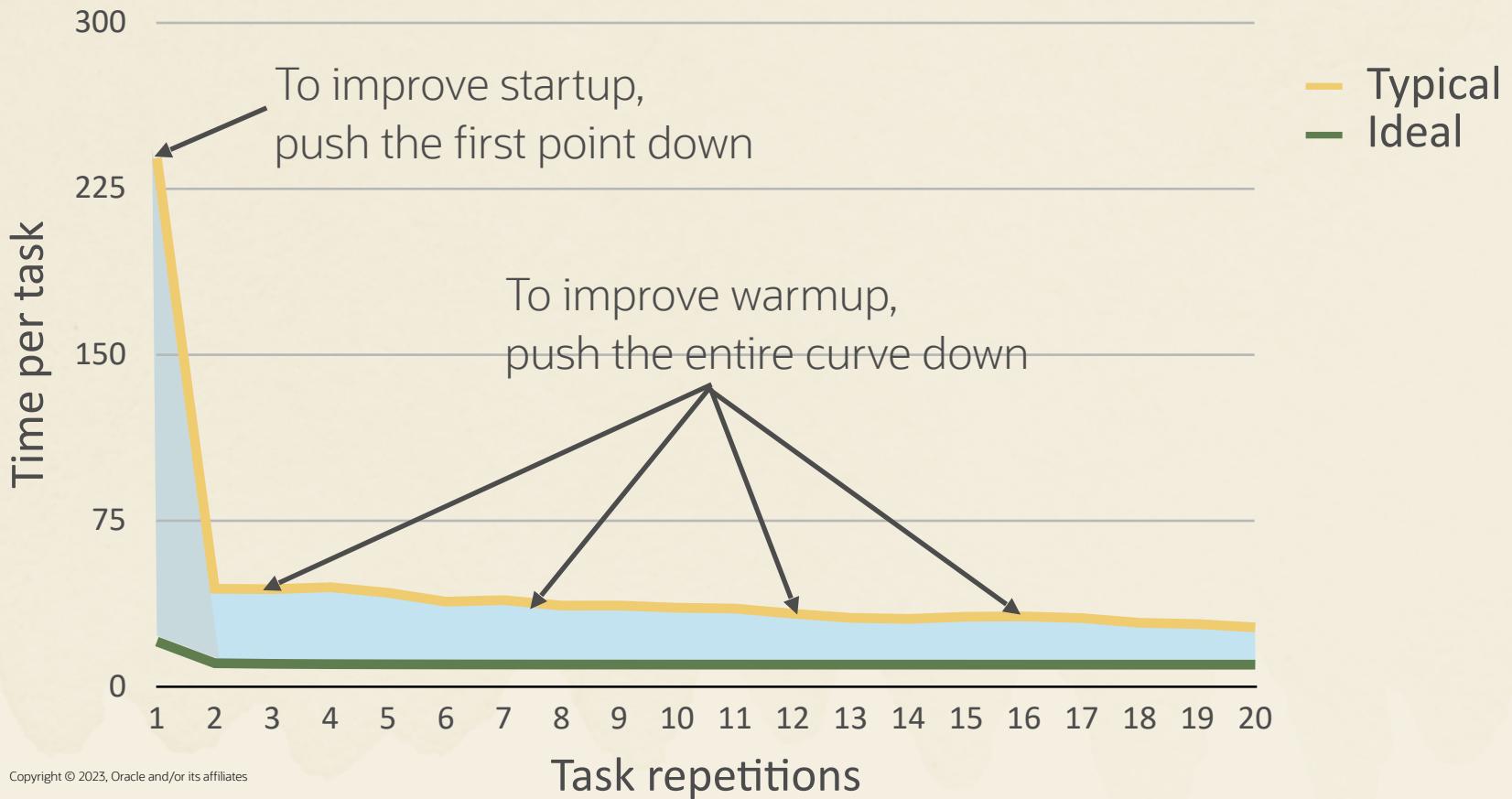
# Why do we do it this way?

- It may sound kind of inefficient to do all this *every time the program starts*
- So, why do we do it this way?
  - Dynamic features like on-the-fly class loading and reflection makes Java programs *more expressive*
  - Dynamic compilation produces better code quality, because the compiler has more information with which to make *better optimization decisions*
    - Current hardware configuration, including exact processor model
    - Profiling data about how has *this* run of the program behaved
    - Can (re)optimize programs based on their observed behavior, not just their code
- All this dynamism makes for excellent peak performance and user convenience
  - The cost of all this dynamism is slower *startup and warmup*
  - But this is still a good tradeoff for many applications!

# Startup and warmup



# Startup and warmup



# Improving startup and warmup

- To push these curves down, we have to shift work off the critical path
  - Could shift work later in time, such as by laziness
  - Could shift work earlier in time, from run time to build time
- We can shift work that is part of the application (e.g., running application or framework code), as well as work on behalf of the application (e.g., compiling application code)
- The JDK already employs a number of computation-shifting techniques, and we can do more

# Shifting computation

- Java already has plenty of features that can shift computation
- Many fall out of the normal semantics of Java programs
  - Compile-time constant folding (shifts earlier)
  - Garbage collection (later)
  - Lazy class loading and initialization (later)
- Some require work by the user to enable
  - Experimental ahead-of-time (AOT) compilation (earlier)
  - Pre-digested class-data archives (CDS) (earlier)
- Shifting work is fair game *as long as the program's meaning is preserved*

# Leyden will expand options for shifting

- Some kinds of shifting will likely require no specification changes
  - E.g., expand lambdas into ordinary bytecode at build time (earlier)
  - Speculatively compiling code ahead-of-time
- Others will definitely require specification changes
  - E.g., eliminate dead code (stripping) (earlier)
- Yet others may involve new platform features to allow developers to express temporal constraints directly in the programming model
  - E.g., lazy static final fields, or computed constants (later)

# Constraining dynamism

- Some forms of shifting may not be practical or safe without accepting additional constraints on the program's execution
  - E.g., “Class X won't change”
- Some programs may be willing to accept some constraint on Java's natural dynamism, if there is sufficient benefit
  - Different programs may want to make different tradeoffs here
- Some programs can even tolerate the ultimate constraint on dynamism, the “closed-world assumption”
  - Forbid dynamic class loading and severely limits reflection
- But, many applications (and developers) have problems with this constraint
- Leyden will explore a broad range of potential constraints
  - E.g., “no class redefinition for module M”
- *Developers can then choose how to trade functionality for performance*

# Project Leyden – key values and techniques

- *Shift* computation temporally, both later and earlier in time
- *Constrain* Java's natural dynamism, if necessary to enable more and better shifting
- *Selectively*, per the needs of each particular program
- *Compatibly*, to preserve program meaning

# Cached Data Storage

## Shifting today: ~~Class Data Sharing~~ (AppCDS)

- Cached Data Storage (previously Class Data Sharing) was introduced in JDK 5
  - Cache data (parsed classfile bytes) and metadata for common system classes
  - Improves startup since every run starts with loading the same core classes
- Evolved significantly over time
  - Cache dynamically generated lambda proxy classes (JDK 8)
  - Cache application classes as well as system classes (“AppCDS”, JDK 10)
  - Cache selected “pure” heap objects (e.g., the default module graph, JDK 12)
- CDS will continue to be a key tool for improving startup, and will be enhanced through Leyden
  - Cache dynamically collected profile data, compiled code, and more ...

# Using AppCDS

- AppCDS uses an *archive file* which contains cached class metadata
  - Archive file is memory-mapped at runtime, is used to accelerate class loading
- The JDK ships with a *default archive*, which is always used unless disabled
  - Generated as part of the JDK build
  - Contains metadata on JDK classes that are likely to be loaded at startup
- Alternate archive file can be specified with command line flags
  - You can create your own archive and specify which classes to pre-process
  - This can include both JDK classes and application classes
- Most people don't use AppCDS today
  - But probably should, can get a reasonable startup improvement for relatively little work

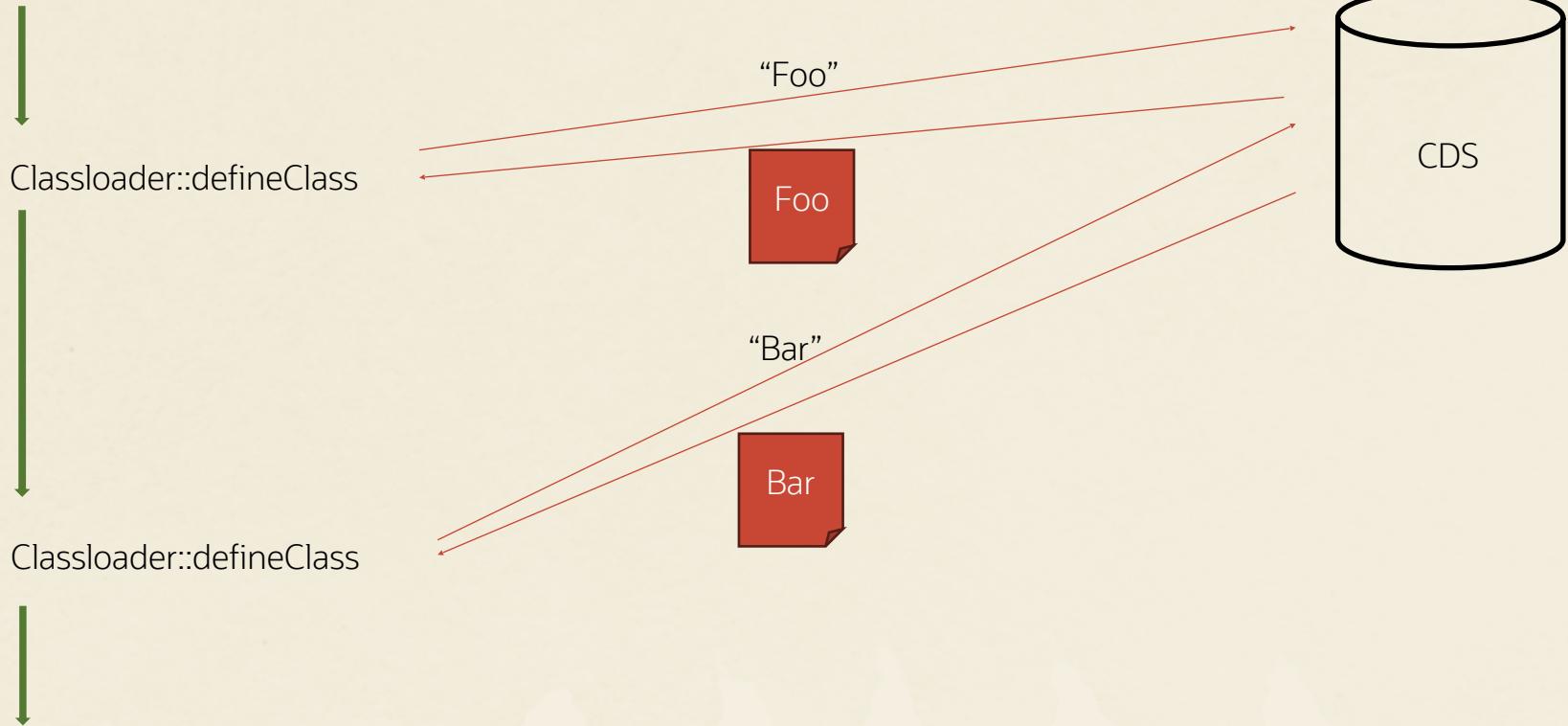
# Using AppCDS

- To use AppCDS today, we typically do the following steps
  - Perform one or more “training runs” of the program, using the **-XX:DumpLoadedClassList** flag, which dumps out a list of loaded classes
    - Multiple lists can be merged, they’re just text files
  - Create an AppCDS archive, using the **-Xshare:dump**, **-XX:SharedClassListFile**, and **-XX:SharedArchiveFile** flags
  - Run program with the archive, using the **-XX:SharedArchiveFile** flag
- Example: javac compiling one file
  - CDS explicitly disabled: 296ms
  - Default CDS: 259ms
  - AppCDS (requires training run): 152ms
- AppCDS functionality may eventually be packaged as a condenser

# Training runs

- AppCDS generally relies on a *training run*
  - Exercises the startup and warmup code paths, under observation
  - Allows us to discover ahead-of-time what we'd otherwise discover in the early phases of run time
  - Usually requires writing a small driver program (like an integration test)
  - Runs at build time (like an integration test)
- Training runs will show up elsewhere in our startup story, as we'll see
  - Training runs are effective for the same reason dynamic compilation is effective – it allows us to base analysis on what the program *actually does*

# CDS class loading



# Background: what happens before main?

```
$ java -cp ...  
MyMainClass
```

- The “java” launcher starts and initializes the Java Virtual Machine
  - Uses JNI\_CreateJavaVM to set up the JVM:
    - Allocates and configures the native parts of Hotspot
    - Initializes the garbage collector, the JIT, etc
    - Loads and initializes the classes need to bootstrap Java - Object, Class, String, ClassLoader, Module, Thread, ...
    - Attaches the caller thread as the initial Thread
  - Uses JNI FindClass to load the main class in the application classloader
  - Uses JNI to locate and execute the main method

# Premain and CDS

- “Premain” is a conceptual phase of JVM execution that consists of the set of actions occurring prior to running the main method
  - Class loading and initialization
  - JIT compilation
  - Garbage collection
- CDS shifts assets from training runs to deployment runs
  - Class metadata and (some) heap objects today
  - More things in the future
- What happens if we combine premain and CDS?

## JEP draft: Loaded Classes in CDS Archives

*Authors* Ioi Lam, John Rose  
*Owner* Ioi Lam  
*Type* Feature  
*Scope* Implementation  
*Status* Draft  
*Component* hotspot/runtime  
*Reviewed by* Vladimir Kozlov  
*Created* 2023/09/06 04:07  
*Updated* 2024/03/21 22:02  
*Issue* 8315737

### Summary

Enhance CDS to store classes in the loaded state, rather than merely pre-parsed.

### Goals

- Enable “live” classes to be adopted from CDS directly into the running application, for classes loaded by the three main class loaders (bootstrap, platform, and system).
- Simplify future storage of additional CDS assets (metaspace, Java heap, code cache), by stabilizing addresses within the CDS archive.

Both of these improvements to CDS (a HotSpot optimization mechanism) indirectly serve the larger goals of Project Leyden, which include better startup, warmup, and footprint for Java applications.

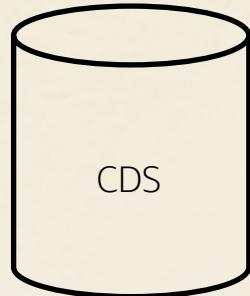
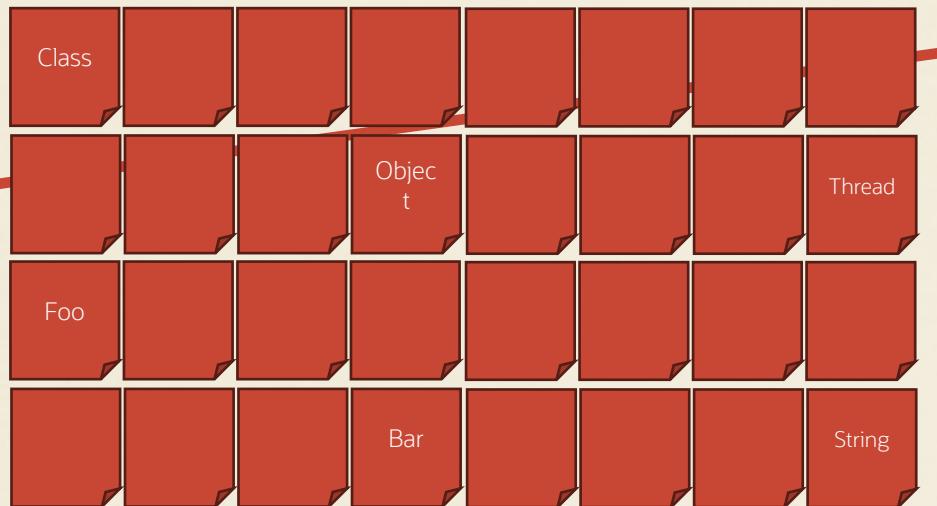
<https://openjdk.org/jeps/8315737>

# Adopting CDS assets in Premain



JNI\_CreateJavaVM

MyMainClass::main



Adopting live classes with stable addresses enables:

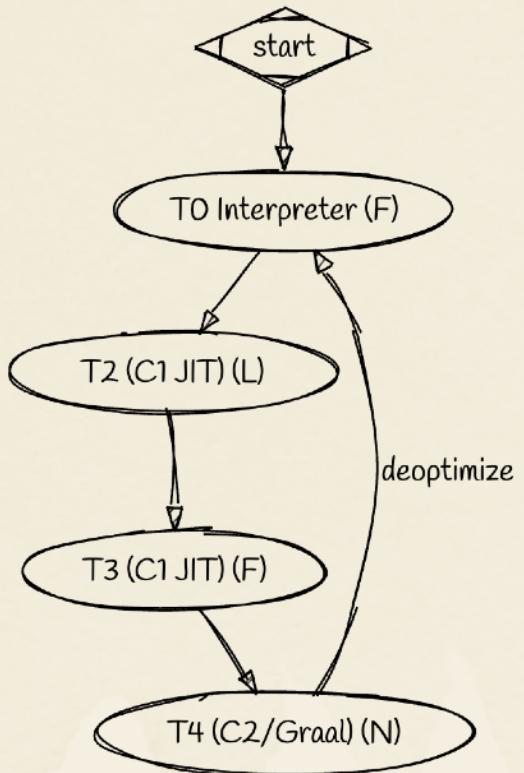
- Persisting profile data from training runs
- Persisting AOT code from training runs



*How far can we get  
without imposing new constraints on existing code,  
without making any specification changes, and  
without sacrificing any of Java's natural dynamism?*

*How far can we get  
simply by leveraging  
existing HotSpot components?*

# The Circle of Life (for Java Code) (Simplified)



# Background: Tiered compilation in HotSpot

- Tier 0: JVM bytecode interpreter
  - Collects full profile information (execution paths and types)
- Tier 1: Simplest possible code
  - No profiling; use is rare
- Tier 2: Simple code with profiling at method entry only
  - Limited use
- Tier 3: Simple code with full profiling
  - Spins up quickly
- Tier 4: Optimized code which benefits from profiling, but collects none
  - Assumes all required classes have been initialized
  - Can de-optimize on awkward inputs (lower tiers cannot)
  - De-optimization is followed by further profiling, and re-optimization

# Tiered compilation: Startup, warmup, and peak

- Startup is handled by slower tiers 0..3, starting with the interpreter (0)
  - Startup resolves symbols, runs class initializers, etc.
- Warmup happens as code shifts from lower tiers to higher ones
  - First, lower tiers must gather profiles
  - The JIT then uses those profiles to optimize Tier 4 code
    - This takes time!
- Peak is reached when all hot code stabilizes in the highest tier (4)

# JIT compilation is speculative

- All dynamic compilation done by the JVM is “speculative”
- We are always free to toss the compiled code and fall back to a lower tier
  - Maybe the environment changed (e.g., a new class was loaded that invalidates an assumption used in compilation)
  - Maybe profiling data told us that the code was no longer optimal
  - Maybe we just ran out of space in the code cache
  - Maybe the debugger caused some code to deoptimize
- So JIT compilation is a pure optimization, one we can freely do and undo at will

# Teaching CDS new tricks

- CDS primarily caches pre-parsed class data and metadata today
- The “Loaded Classes in CDS Archives” JEP draft extends this to
  - Class objects (reflective mirrors), not just class metadata
  - Resolved constant pool references to classes, methods, and fields
- And it can be extended further
  - Profiles gathered during tiers 0-3
  - Compiled methods from tiers 1-4
  - Resolved invokedynamic linkage states
  - Pre-initialization of most enum and hidden classes
- Plus load-time checks to ignore cached metadata if something changed

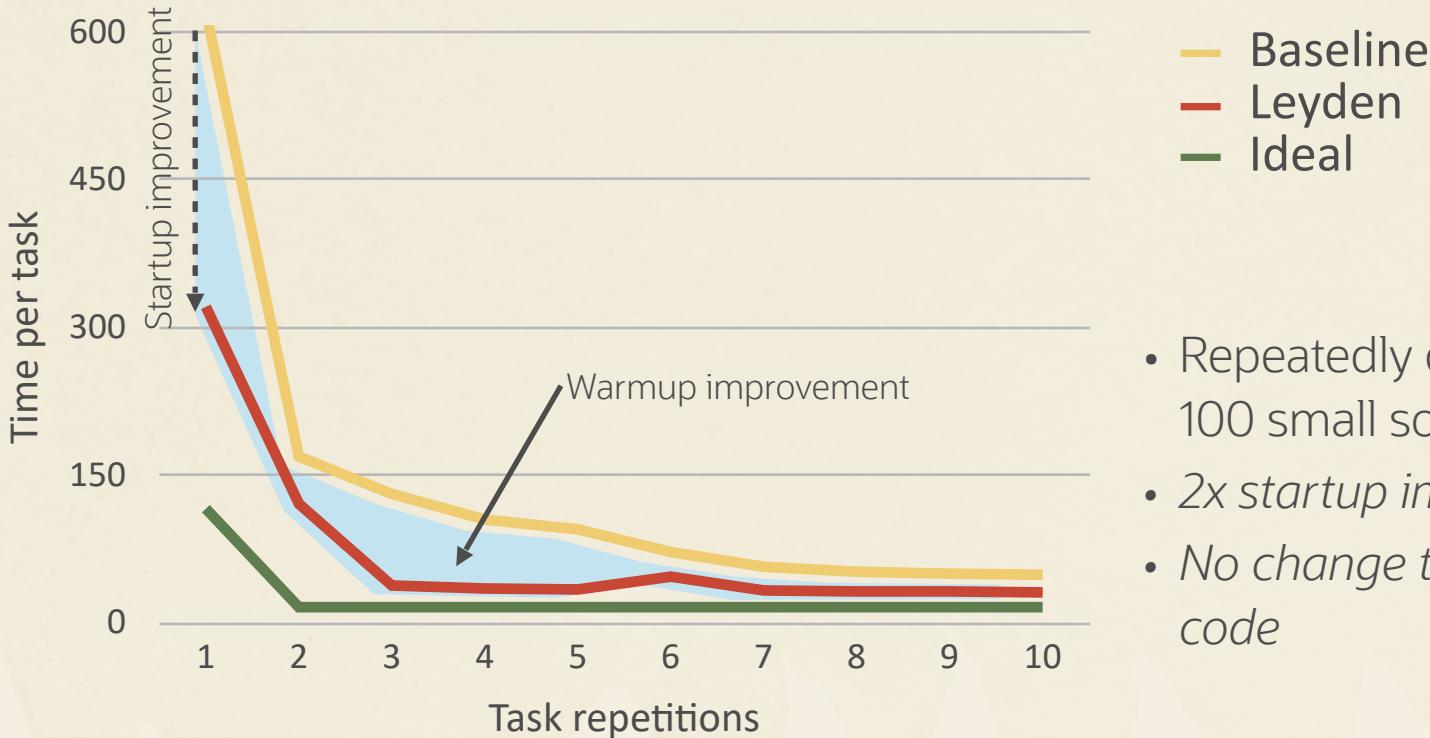
# Speculative AOT: JIT compilation shifted earlier

- Cache profiles and compiled code from training runs, for use in later runs
- Many new options
  - At startup, we could schedule JIT activity earlier based on cached profiles
  - Or (even more quickly), load pre-compiled code from archive
- Install the compiled code for a method after all the classes upon which it depends have been initialized
  - Even better, we can cache two kinds of code ahead-of-time
  - With class-initialization checks, and without class-initialization checks
  - Install the former initially, then swap in the latter after the required classes have been initialized

# AOT-compiled code remains speculative

- Java has always been both static and dynamic
  - Locally static, globally dynamic
- Cached profiles and code are records of dynamic observations of an application
  - The flip side of static application analysis — but requires no new constraints!
- They are used speculatively by HotSpot
  - Just as they always have been: “Success is a habit, but failure is an option”
  - If an assumption is violated then we de-optimize, re-profile, and re-optimize
- This approach copes well with surprises at run time
  - Code sometimes changes between training and production
  - Applications sometimes have distinct phases of activity
- Yet this is not surprising: On-the-fly adaptation is one of Java’s distinct strengths!
  - “Something changed in the application since the training run” is just one more reason we can deoptimize and reoptimize
- Key challenge: Optimizing the policies that govern execution-mode transitions

# Case study: javac

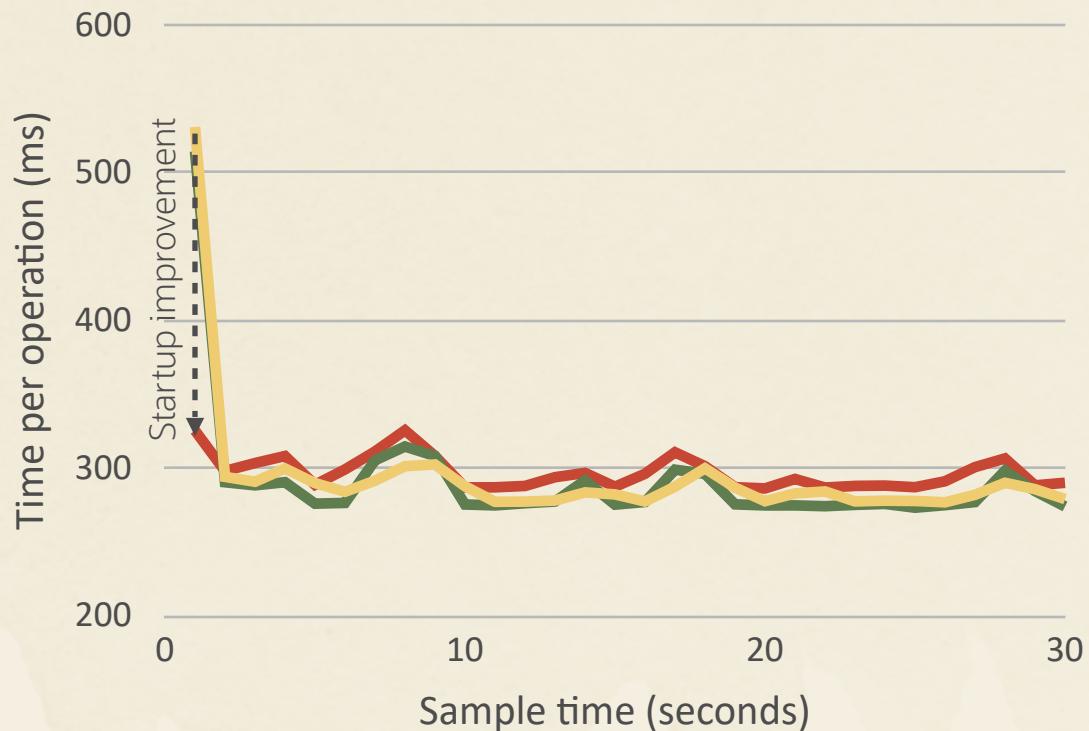


- Repeatedly compile 100 small source files
- *2x startup improvement*
- *No change to existing code*

# Case study: javac

- 2x startup improvement for free!
  - We can shift optimization work earlier in time, via CDS
  - No new constraints, no changes to existing code
- There was no one “magic bullet” technique
  - We used several already, to good effect, and we’ll keep looking for more
  - AOT compilation, linkage, constant pool resolution
- This machinery doesn’t tune itself
  - Many tuning adjustments needed to craft ideal compilation policies
  - Policy tuning is business-as-usual in the JDK

# Case study: XML validation (SPECjvm 2008)

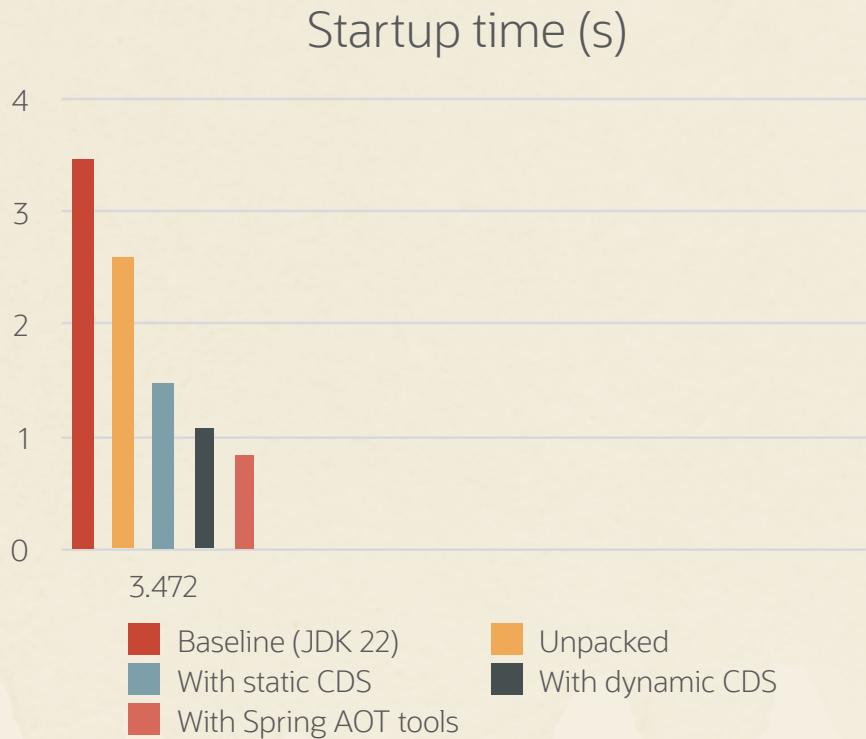


- *8x startup improvement*
- *Little warmup improvement*
- *No change to existing code*

# Case study: XML validation (SPECjvm 2008)

- Sometimes startup is the only interesting win
  - Warmup is often already okay for smaller applications
- In this case, we could decisively improve startup compared to the baseline policy
- Benchmark noise can make it hard to decide when we've reached peak performance

# Case study: Spring Boot “Pet Clinic”



*Spring Boot “Pet Clinic”*  
*4.1x startup improvement*  
*with no change to existing*  
*code*

# Case study: Spring Boot “Pet Clinic”

- There are many tactics which can improve startup
  - We win big because the tactics work in synergy
- Early class loading, via CDS, is a big win
- Caching compiled code is a big win
- Resolving invokedynamic call sites earlier is a lesser win
- Clever tier-4 code that contains class-initialization checks is a smaller win
- Using Spring Boot’s ahead-of-time configuration tool is a big win
  - The tool scans for configuration annotations at build time and generates code to wire up components quickly at run time
  - It is, in effect, a Spring-specific condenser

# Time-shifting speculative optimizations works!

- This overall approach shows great promise
  - Significant gains
  - Full compatibility
  - No new constraints
  - No changes to the programming model
  - No changes to the specification
  - Retains all of Java's natural dynamism
- Largely a rearrangement of existing JVM components, plus some new policies
- Next steps
  - Improve ergonomics of training runs
  - Further case studies to improve both mechanisms and policies

# Condensers

- New tool of Leyden is *condensers*
- A condenser is a tool in the JDK that
  - Performs some of the computation encoded in a program image
    - Thereby shifting it earlier in time
  - Transforms the image into a new image, possibly containing
    - New code (e.g., ahead-of-time compiled methods)
    - New data (e.g., serialized heap objects)
    - New metadata (e.g., pre-loaded classes)
    - New constraints (e.g., no class redefinition)

# Condensers

- Condensers are *meaning-preserving*
  - The resulting image has the same meaning as the original, under the selected set of constraints
- Condensers are *composable*
  - The image output by one condenser can be the input to another
  - A particular condenser can be applied multiple times, if needed
- Condensers are *selectable*
  - Developers choose how to condense, and when
    - If you're testing or debugging, then don't bother — just run normally
  - Insofar as shifting computation requires accepting constraints, you can trade functionality for performance via the condenser configuration you choose

# Computed constants

- For some computations, shifting forward or backwards in time is “obviously safe”  

```
static final int x = 1 + 2;
```
- But for interesting computations, we may need some hints from the user
  - What form should those hints take?
- The JVM contains some optimizations for mutable state that is known to eventually converge to a steady state
  - Within the JDK, these are marked with `@Stable`, and can be constant folded / inlined into generated code as if it were final
  - But, `@Stable` is an internal annotation – user code can’t do this (yet)

# Computed constants

- There is a draft JEP for “Computed Constants” that aims to expose the `@Stable` optimizations to user code

```
public static final ComputedConstant<String> name
    = ComputedConstant.of(() -> readNameFromFile("/path/to/file.txt"));
```

- Looks like just a library for lazy initialization, but engages JVM optimizations for stable values
- Allows fine-grained selection of “shift this later in time”, without giving up the performance of static final fields
  - May eventually be extended to permit shifting *earlier* in time, but more research is needed here

# Lots more to do...

- Explore new ways to shift computation and constrain dynamism
  - E.g., Speculative ahead-of-time compilation, linkage, and resolution
  - To be delivered incrementally
  - “Loaded Classes in CDS Archives” Draft JEP is a first step in this path
- Improve the ergonomics of training runs
- Introduce condensers into the Java Platform
  - Evolve the Java Platform Specification and JDK tooling to support condensation
  - Evolve the run-time image format to accommodate new code, data, and metadata, as necessary
- Find the right way to bring shiftability into the programming model
- Lots of experiments underway; lots more to come!



```
graph TD
graph [fontname = "Handlee"];
node [fontname = "Handlee"];
edge [fontname = "Handlee"];

bgcolor=transparent;
start -> t0;
t0 -> t2
t2 -> t3
t3 -> t4
t4 -> t0 [label="deoptimize"]

t0 [label="T0 Interpreter (F)"]
t2 [label="T2 (C1 JIT) (L)"]
t3 [label="T3 (C1 JIT) (F)"]
t4 [label="T4 (C2/Graal) (N)"]
start [shape=Mdiamond];
}
```