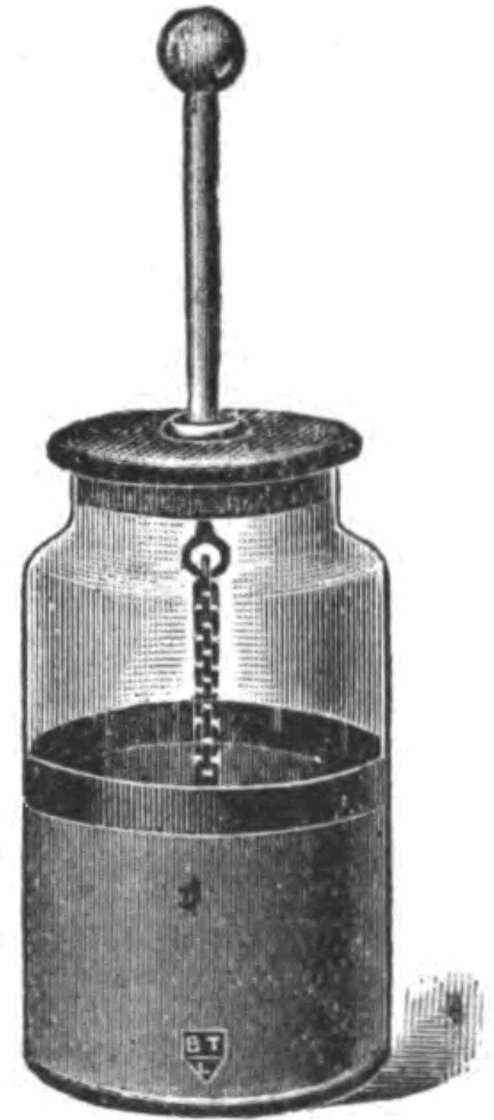
Project Leyden

Capturing Lightning in a Bottle



Mark Reinhold
Chief Architect, Java Platform Group, Oracle

JCP Executive Committee
2023/9/14

Copyright © 2023, Oracle and/or its affiliates

Leyden: Goal

Improve the startup time, warmup time, and footprint of Java programs

Copyright © 2023, Oracle and/or its affiliates

Leyden: Means

Shift computation temporally,
later and earlier in time

Constrain Java’s natural dynamism,
to enable more and better shifting

Selectively, per the needs of each particular program

Compatibly, to preserve program meaning

Copyright © 2023, Oracle and/or its affiliates

Shifting computation

- We can shift two kinds of computation
  - Work expressed directly by a program (*e.g.*, invoke a method)
  - Work done on behalf of a program (*e.g.*, compile a method to native code)
- Java implementations already have features that can shift computation
  - Automatically: Compile-time constant folding (shifts EARLIER in time)
Garbage collection (LATER)
  - Or optionally: Ahead-of-time (AOT) compilation (EARLIER)
Pre-digested class-data archives (CDS) (EARLIER)
Lazy class loading and initialization (LATER)
  - Either way, always preserving program meaning per the Specification
    - So as to ensure compatibility

Copyright © 2023, Oracle and/or its affiliates

Leyden will explore new ways to shift computation

- Some kinds of shifting will likely require no specification changes
  - *E.g.*, expand lambdas into ordinary bytecode (EARLIER)
- Others will definitely require specification changes
  - *E.g.*, eliminate dead code (stripping) (EARLIER)
- Yet others will be new platform features that allow developers to express temporal shifting directly in source code
  - *E.g.*, lazy static final fields (LATER)

Copyright © 2023, Oracle and/or its affiliates

Constraining dynamism

- Shifting computation often requires code analysis
  - But: Java's dynamic features make code analysis difficult
- We could simplify code analysis by imposing a ***closed-world constraint***
  - Forbids dynamic class loading and severely limits reflection
  - Many applications don't work under this constraint
  - Many developers aren't willing to live with this constraint
- Leyden will therefore explore a spectrum of constraints, up to and including the closed-world constraint
  - **Selectively degrade Java's natural dynamism** to enable more and better shifting of computation
  - Developers can choose how to trade functionality for performance

Copyright © 2023, Oracle and/or its affiliates

Condensers: Tools for shifting & constraining computation

The key new concept of Leyden

- 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, faster image that may contain:
    - 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)

Copyright © 2023, Oracle and/or its affiliates

Key properties of condensers

- Condensers are **meaning-preserving**
  - The resulting program image has the same meaning as the original
- 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 condensers that you choose

Copyright © 2023, Oracle and/or its affiliates

Performance is an emergent property

- The performance of your program depends upon the condensers that you choose
- Given sufficiently powerful condensers:
  - If you shift enough computation earlier or later in time, you might even be able to produce a fully-static native image
  - This will likely require accepting many constraints
- Leyden need not specify fully-static native images directly
  - Instead, it will enable sufficient shifting of computation and constraining of dynamism
  - Fully-static native images can fall out as an emergent property

Copyright © 2023, Oracle and/or its affiliates

Leyden roadmap

- Introduce condensers into the Java Platform
  - Evolve the Java Platform Specification to allow meaning-preserving whole-program transformations
  - Evolve the run-time image format to accommodate new code, data, and metadata
- Explore new ways to shift computation and constrain dynamism
- Explore related improvements

Copyright © 2023, Oracle and/or its affiliates

Leyden progress!

openjdk.org/projects/leyden

- Introduce condensers
  - *Toward Condensers* (prototype and design note, Goetz, Reinhold, & Sandoz)
- Shift computation and constrain dynamism
  - Pre-generate lambda classes (prototype branch, Heidinga)
  - *Condensing Indy Bootstraps* (design note, Goetz)
  - *Computed Constants* (prototype and draft JEP, Minborg & Cimadamore)
  - Experiments in shifting speculative compilation (prototype branch, Rose *et al.*)
- Related improvements
  - Hermetic application packaging (prototype branch, Zhou)
  - JMOD-less linking (prototype, Gehwolf)

Copyright © 2023, Oracle and/or its affiliates

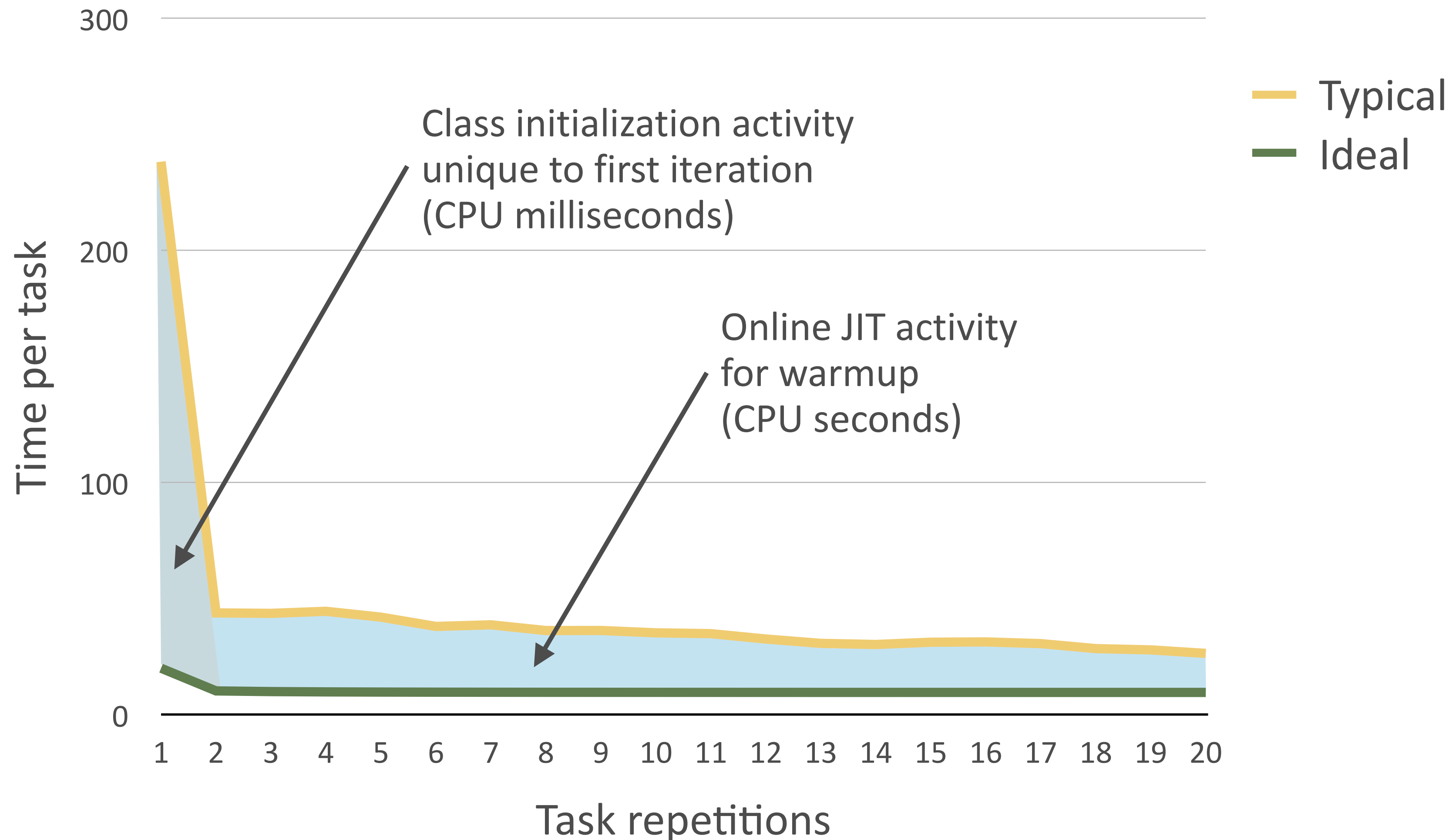
Experiments in time-shifting speculative optimizations

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?

Copyright © 2023, Oracle and/or its affiliates

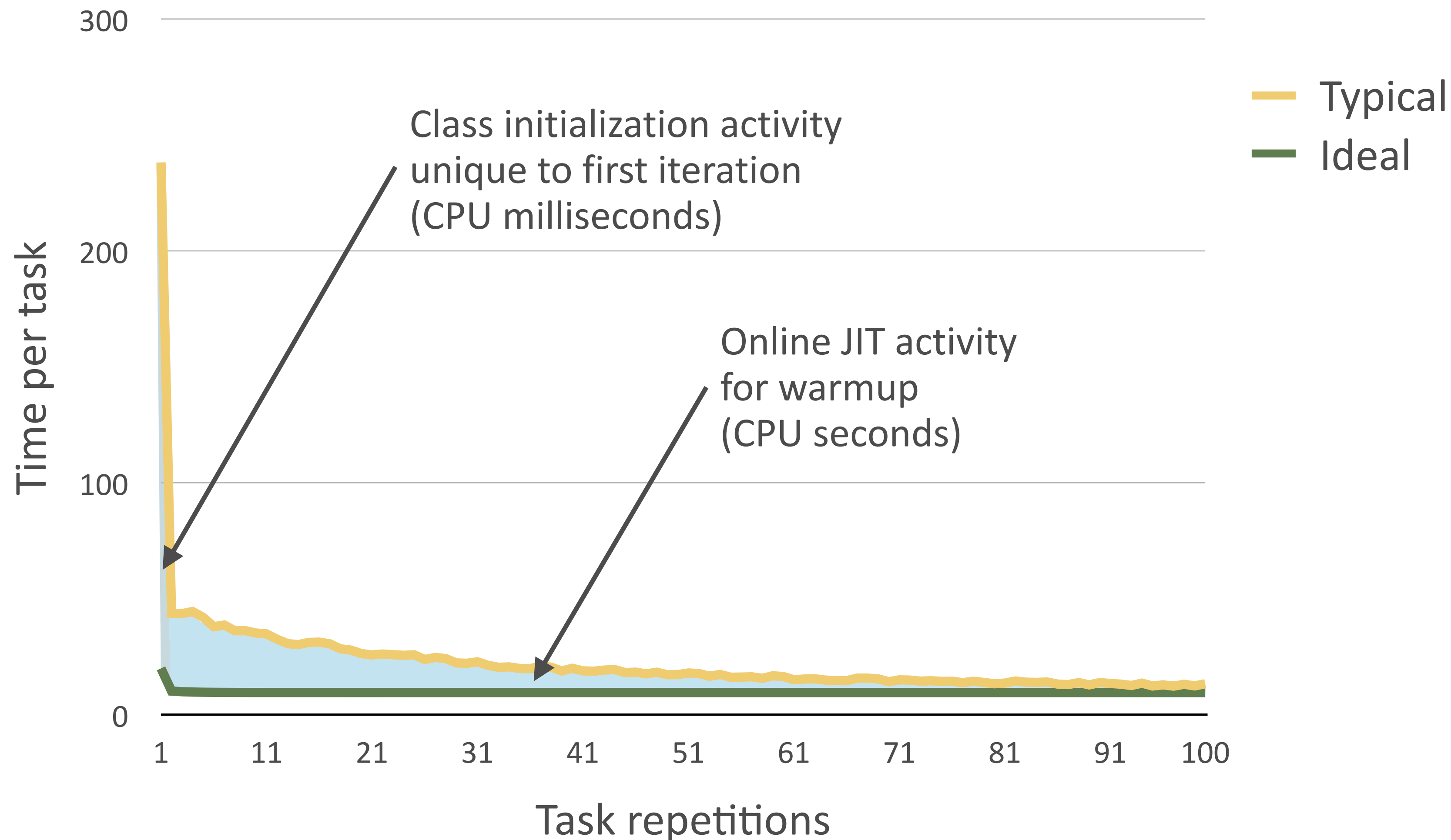
A tale of two graphs: What startup looks like today



HYPOTHETICAL MODEL

Copyright © 2023, Oracle and/or its affiliates

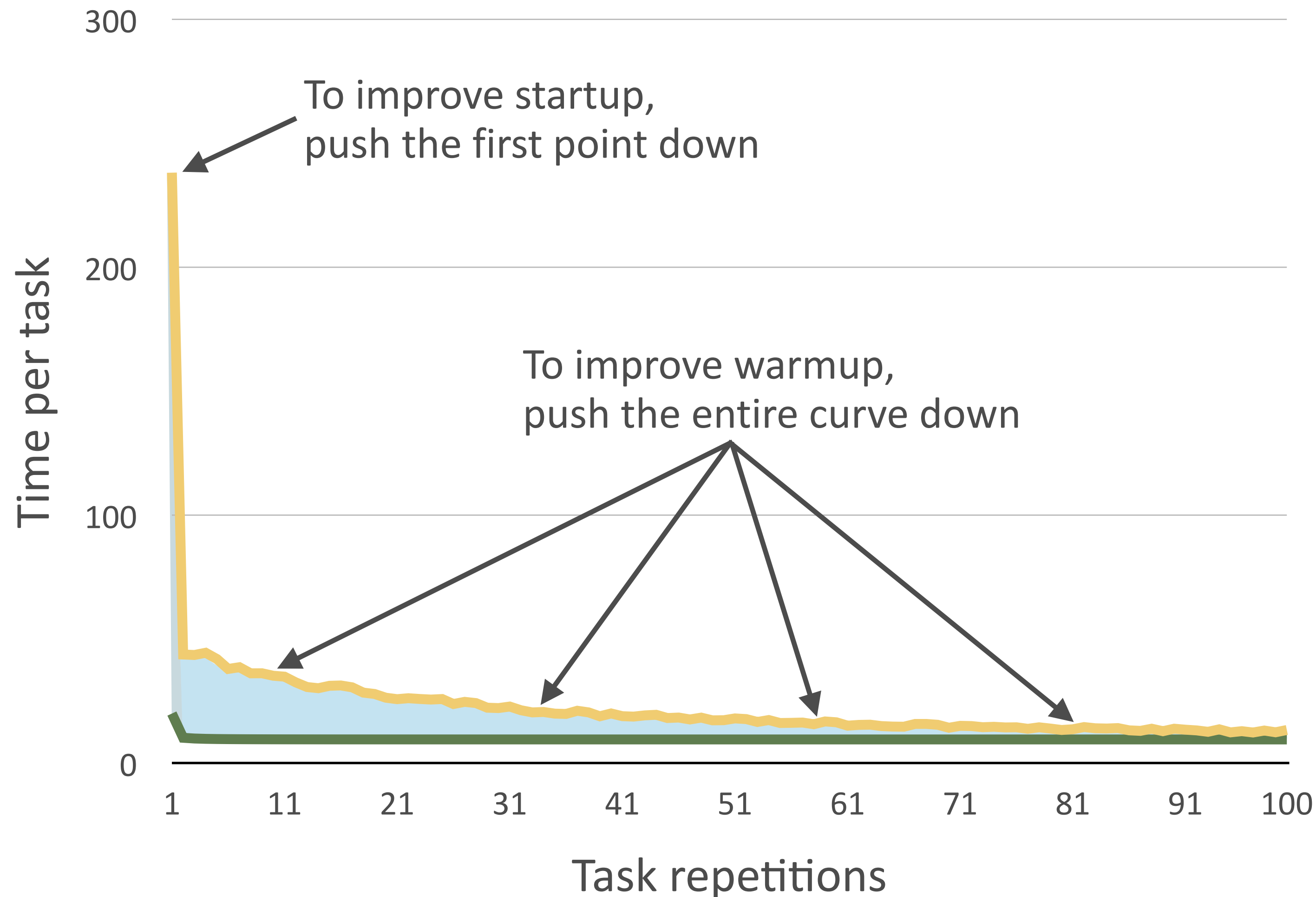
A tale of two graphs: At larger scales it's all warmup



HYPOTHETICAL MODEL

Copyright © 2023, Oracle and/or its affiliates

Challenge: Make startup/warmup faster at multiple scales



HYPOTHETICAL MODEL

Copyright © 2023, Oracle and/or its affiliates

Static vs. dynamic optimization: Choose two

- The Java answer is never “Choose One, Lose One”
  - Java balances static and dynamic reasoning
- HotSpot optimizes dynamic computations
  - Converting them, in effect, to static computations
  - But **speculatively** so: Can dynamically de-optimize and re-optimize when things change
- We can shift these optimizations earlier in time
  - Speculatively optimizing before application startup
- Goal: Drive startup time and warmup time into the noise, while **maintaining compatibility**
  - No new constraints, no code change required

Copyright © 2023, Oracle and/or its affiliates

Background: The four tiers of 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

Copyright © 2023, Oracle and/or its affiliates

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 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)

Copyright © 2023, Oracle and/or its affiliates

AOT compilation = Dynamic compilation shifted earlier

- Key idea: Cache profiles and compiled code from earlier application runs for use in later runs
  - Earlier runs could be **training runs**, which synthetically exercise the application along typical paths
  - Or they could be actual runs in production
- At startup, we can quickly JIT code based on cached profiles
- Alternatively, we can even more quickly (5–500x) load cached code
  - 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

Copyright © 2023, Oracle and/or its affiliates

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!
- Key challenge: Optimizing the policies that govern execution-mode transitions

Question: Where do we cache profiles and compiled code?

Copyright © 2023, Oracle and/or its affiliates

Application class-data sharing (CDS)

dev.java/learn/jvm/cds-appcds

Shifting computation backward in time since 2004

- Introduced in JDK 5 (2004) for system classes and serial GC only
  - Focus was on client application startup
  - Cache data (*i.e.*, parsed class-file bytes) and metadata for commonly-used system classes
- Evolved significantly since then
  - Support all GCs
  - Cache commonly-used application classes (JDK 10, 2018)
  - Cache select “pure” heap objects (*e.g.*, the default module graph) (JDK 12, 2019)
- Enhancements for Leyden
  - Cache dynamically-collected compilation profile data
  - Cache compiled code

Copyright © 2023, Oracle and/or its affiliates

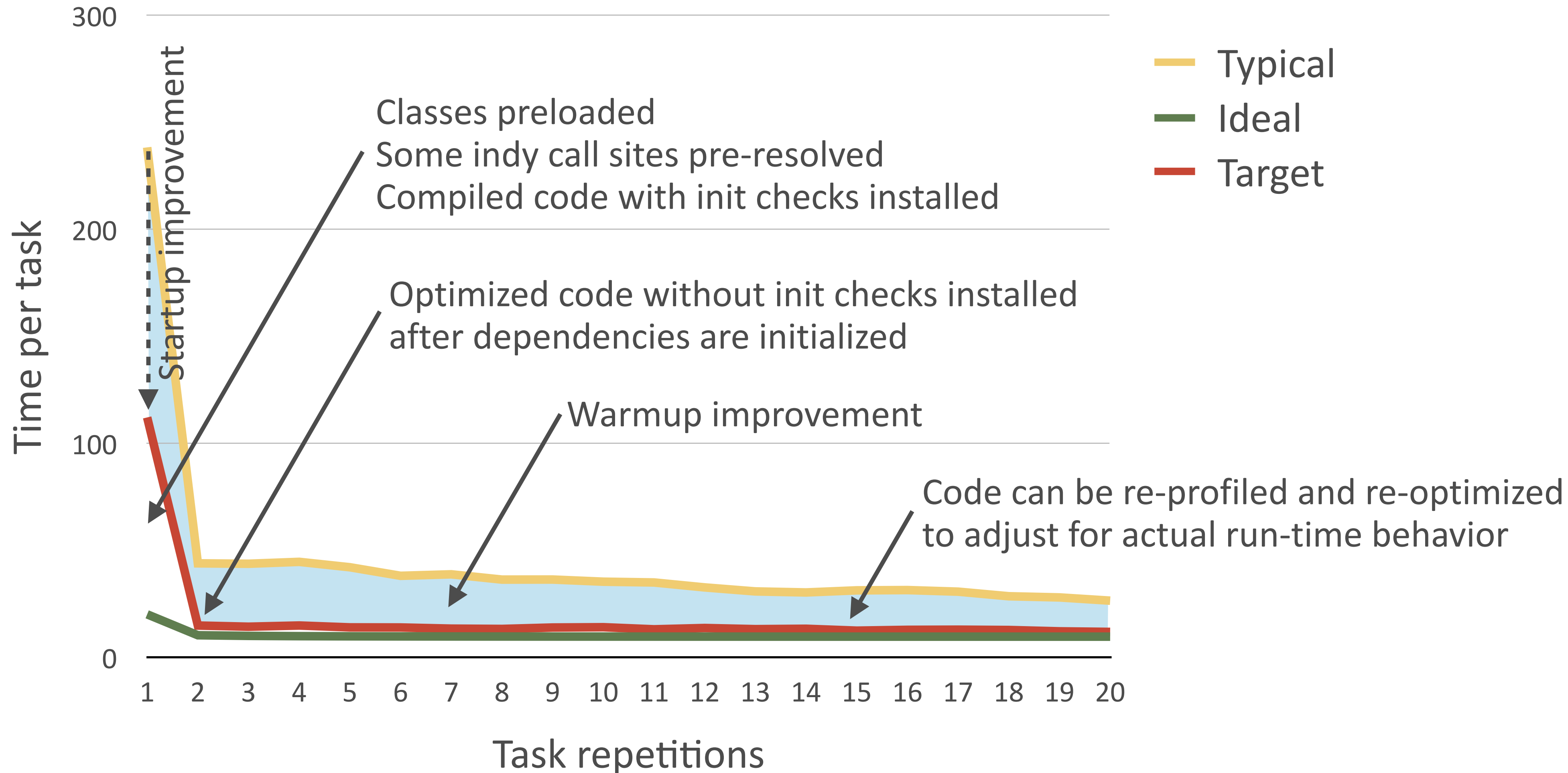
New CDS tricks: Early loading, resolution, and initialization

- Preload cached classes
  - Cache actual class/interface objects, not just pre-parsed data and metadata
- Resolve symbolic field, method, and class references in constant pools
- Resolve `invokedynamic` call sites
  - Expand lambdas and string-concatenation operations into their dynamic form
- Initialize `enum` classes and hidden classes
  - We'll eventually explore more-extensive early initialization

"CDS" has never been a great name — we'll likely rename this to the "startup cache"

Copyright © 2023, Oracle and/or its affiliates

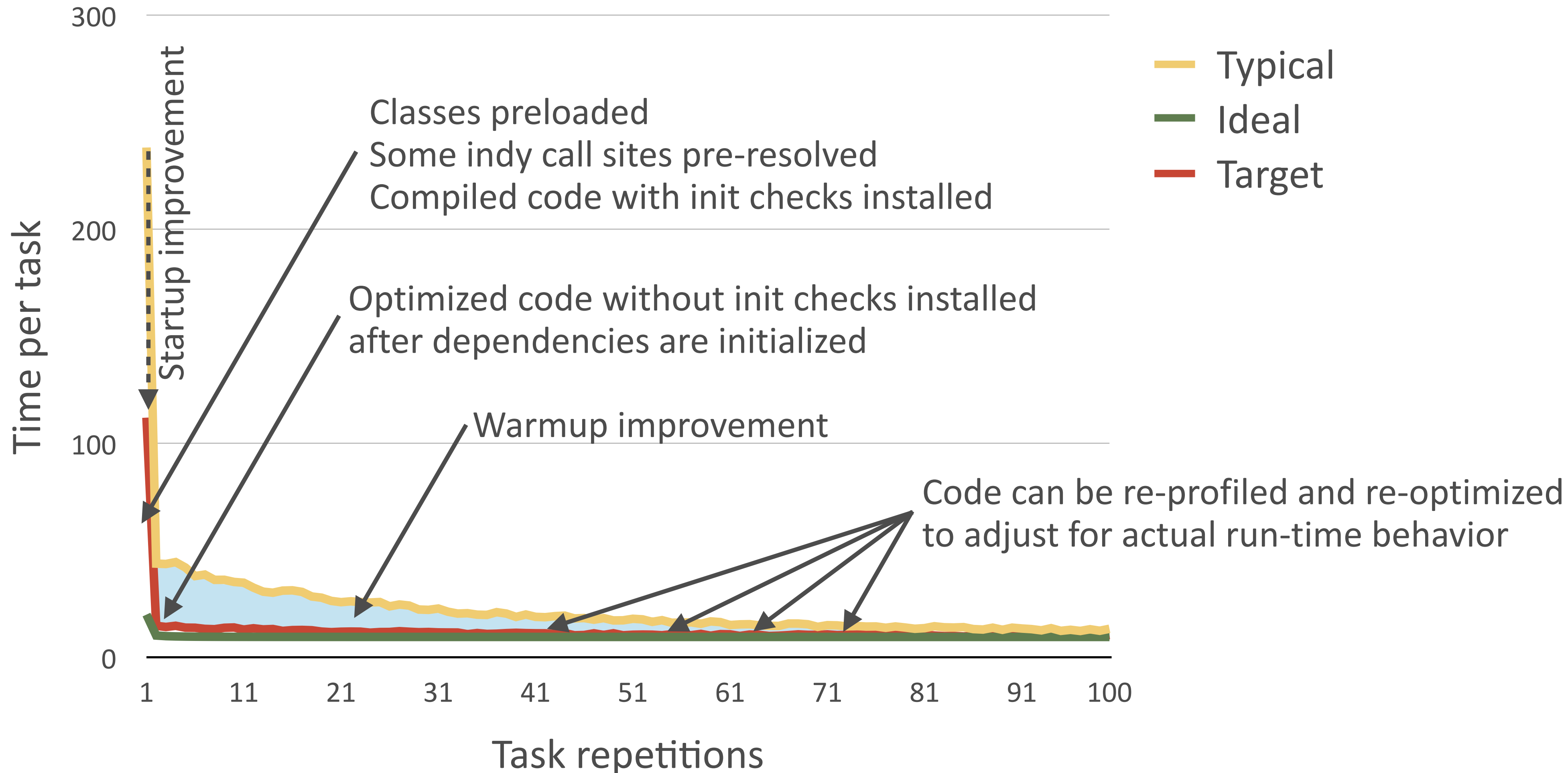
A tale of two graphs: Better startup and early warmup



HYPOTHETICAL MODEL

Copyright © 2023, Oracle and/or its affiliates

A tale of two graphs: Better long-term warmup



HYPOTHETICAL MODEL

Copyright © 2023, Oracle and/or its affiliates

Case studies

javac

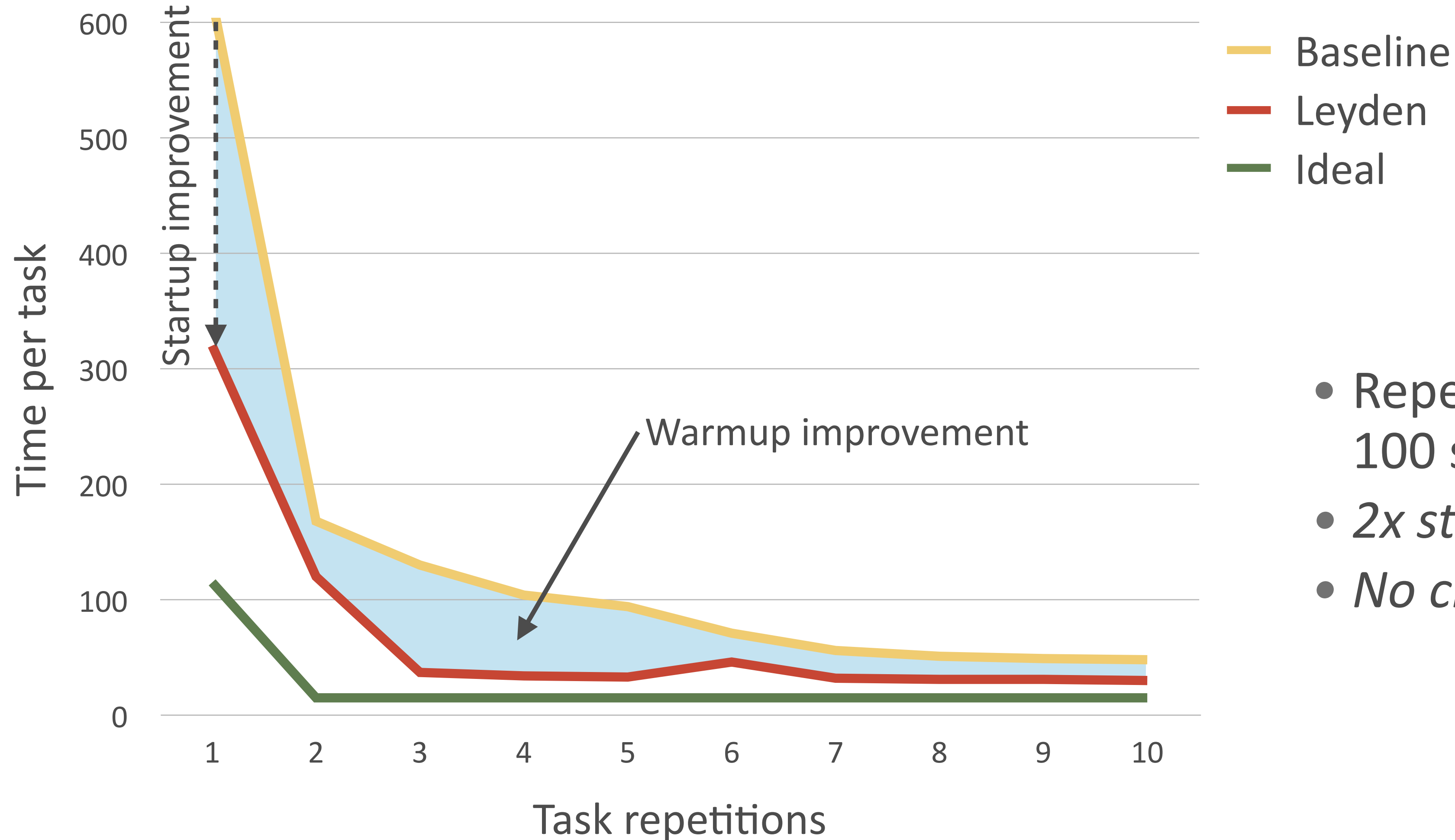
XML validation

Spring Boot

WARNING: WORK IN PROGRESS

Copyright © 2023, Oracle and/or its affiliates

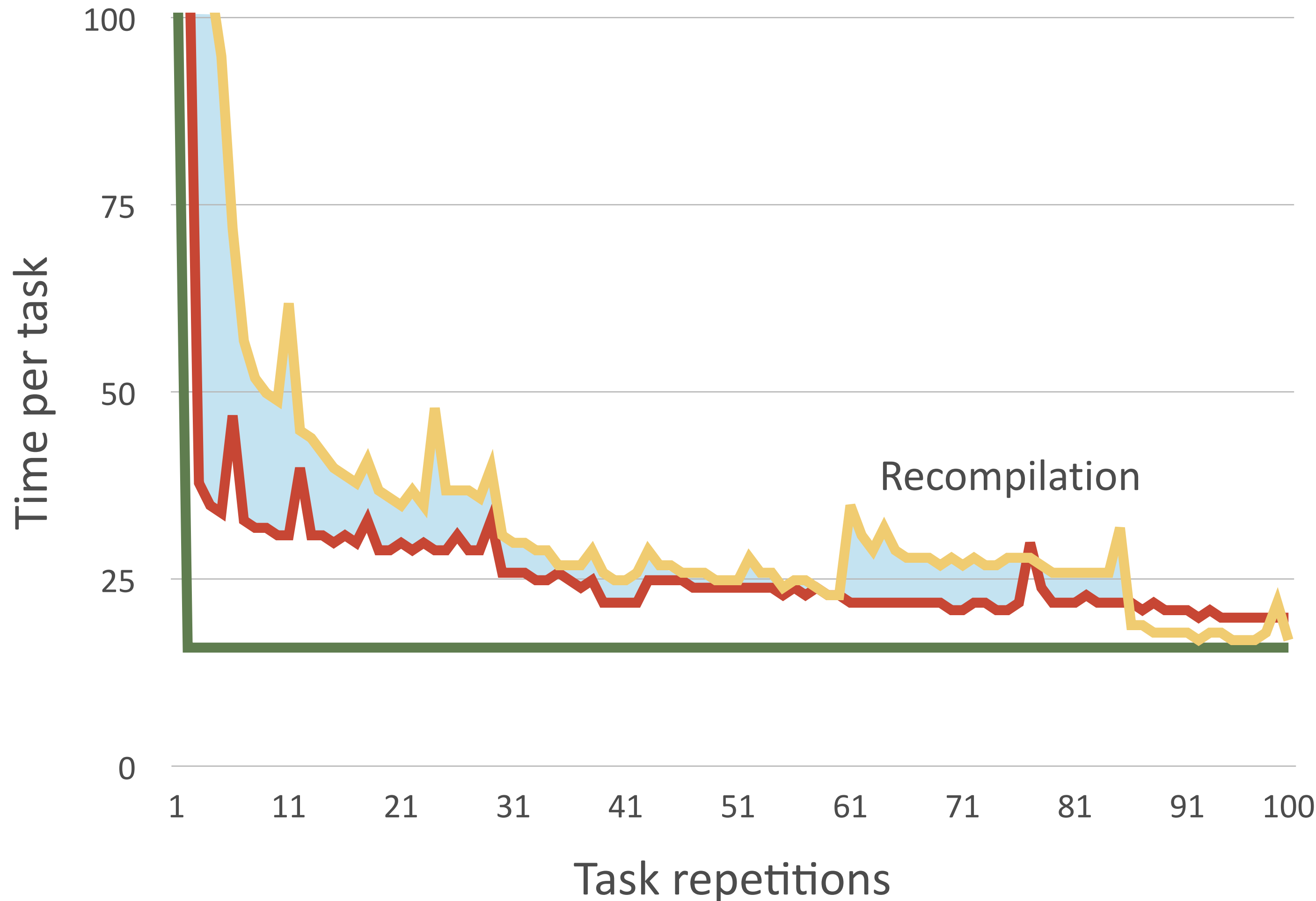
javac: The first few iterations



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

Copyright © 2023, Oracle and/or its affiliates

javac: More iterations, start of recompilation



Tuning and policy work required to bring the red line down further

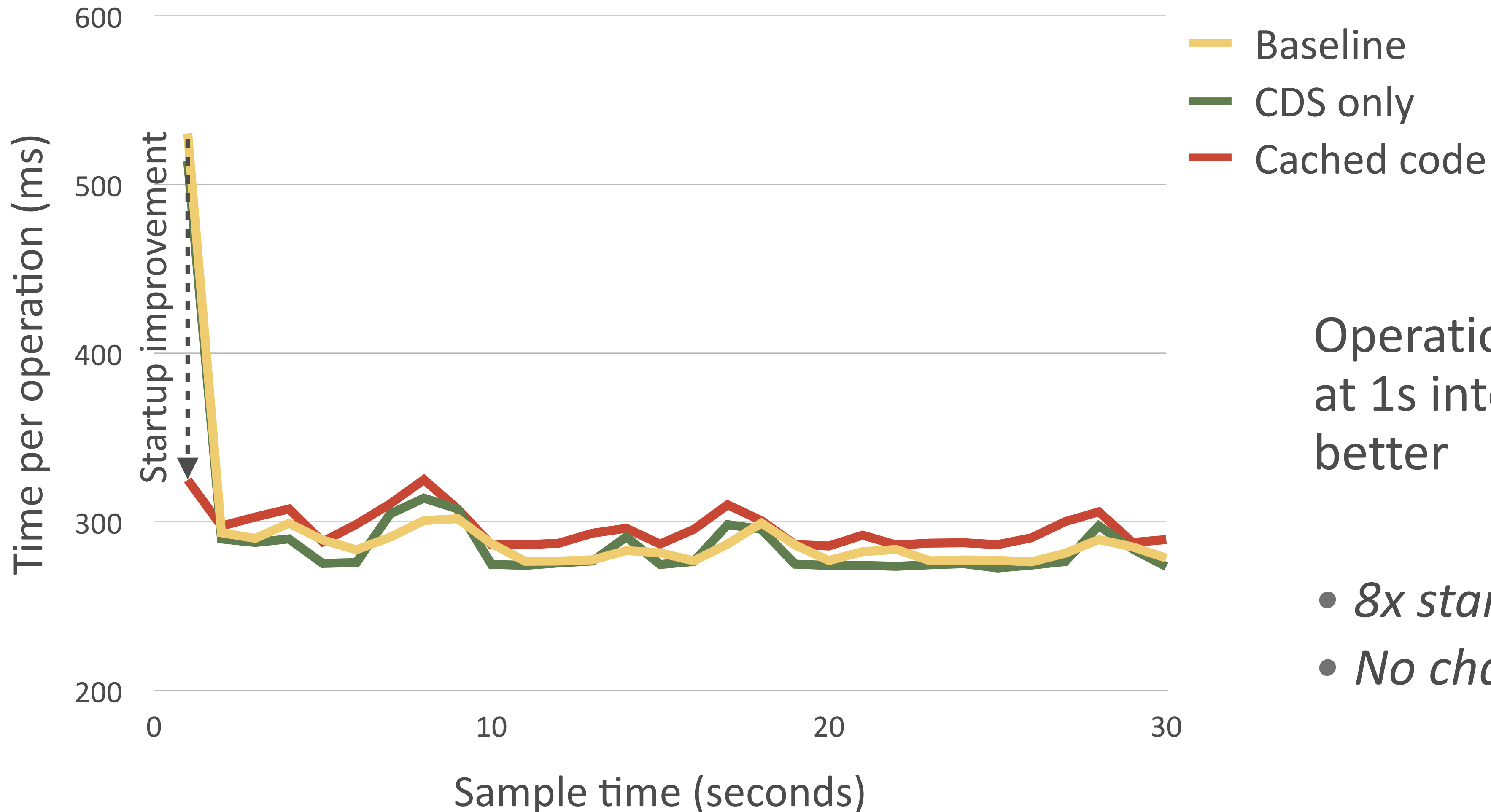
Copyright © 2023, Oracle and/or its affiliates

Lessons from javac case study

- It works — 2x startup improvement for free!
  - We can shift optimization work earlier in time, via CDS
  - No new constraints, no changes to existing code
- There is no one "magic bullet" technique
  - We have several, we'll keep hunting for more
- Multiple time scales of warmup are important
  - We'll try to chase them all
- This machinery doesn't tune itself
  - Well-tuned policy is a way of life, not a possession

Copyright © 2023, Oracle and/or its affiliates

SPECjvm2008 XML validation benchmark



Operation time sampled at 1s intervals; lower is better

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

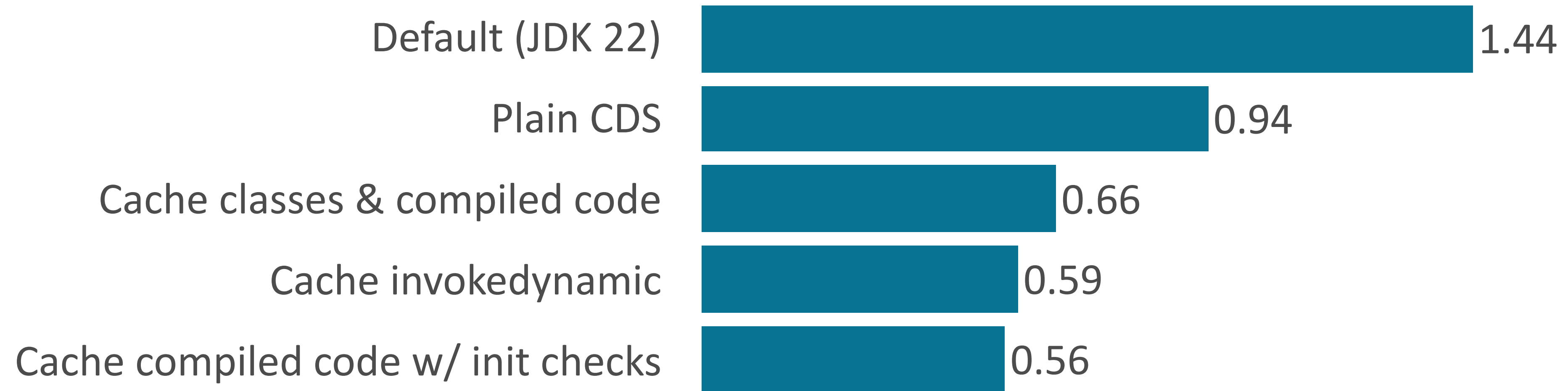
Copyright © 2023, Oracle and/or its affiliates

Lessons from XML validation case study

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

Copyright © 2023, Oracle and/or its affiliates

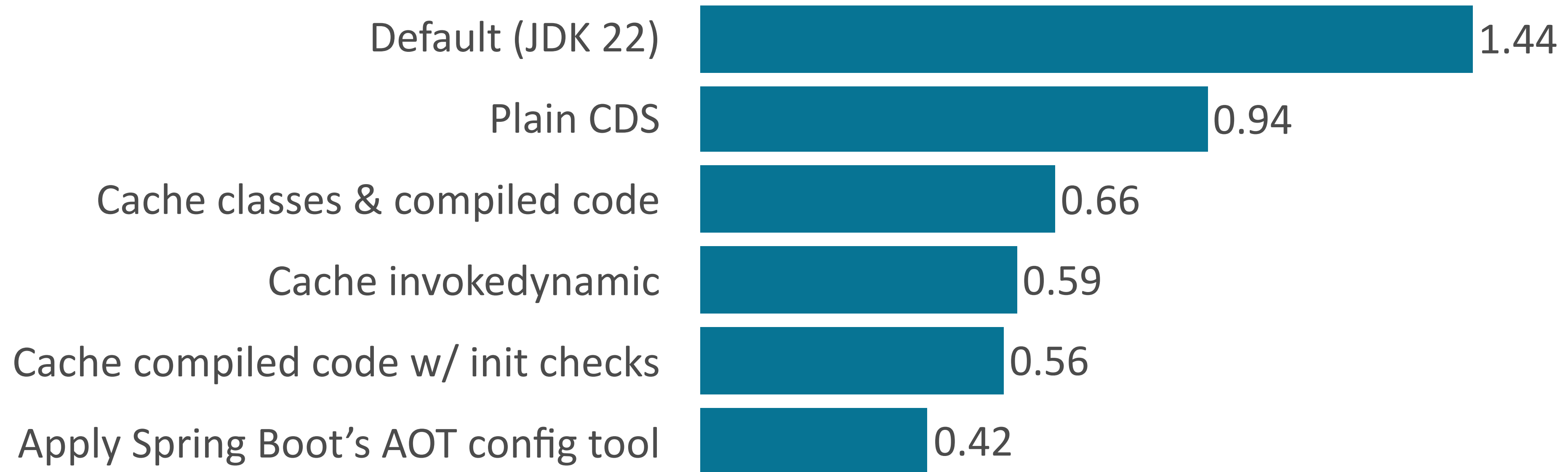
Spring Boot "Hello, world" startup



2.6x improvement with no change to existing code

Copyright © 2023, Oracle and/or its affiliates

Spring Boot “Hello, world” startup



~~2.6x~~ 3.4x improvement with no change to existing code

Copyright © 2023, Oracle and/or its affiliates

Lessons from Spring Boot case study

- 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

Copyright © 2023, Oracle and/or its affiliates

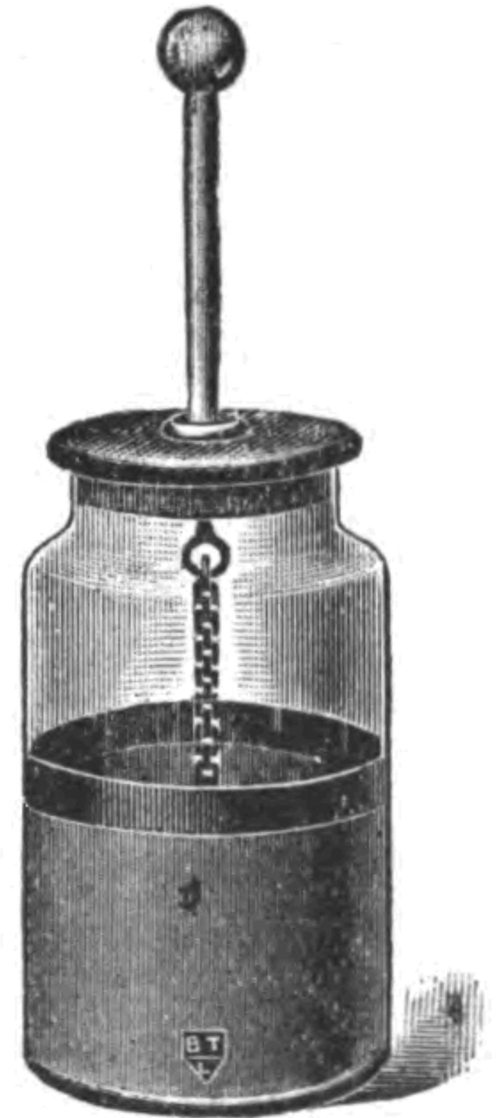
Summary: Time-shifting speculative optimizations works!

- 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
- This is largely a rearrangement of existing JVM components, plus some new policies
  - Current patch is just 23K lines
- We're just getting started!
  - A small team has been working on this only since March
- Next steps …
  - Simplify the developer workflow into condensers (prototype is heavy on CLI flag soup)
  - Further case studies to improve both mechanisms and policies

Copyright © 2023, Oracle and/or its affiliates

Project Leyden

Capturing Lightning in a Bottle



Mark Reinhold

Chief Architect, Java Platform Group, Oracle

JCP Executive Committee
2023/9/14

Copyright © 2023, Oracle and/or its affiliates