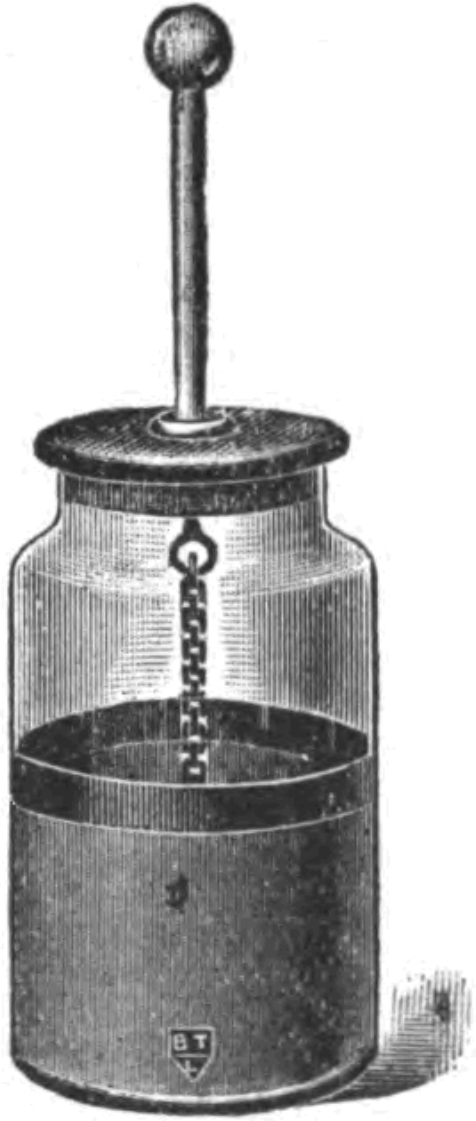
# Project Leyden

*Capturing Lightning in a Bottle*

Mark Reinhold
*Chief Architect, Java Platform Group, Oracle*

John Rose
*JVM Senior Architect, Java Platform Group, Oracle*

JVM Language Summit
2023/8/8



# Leyden: Goal

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

# 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

# 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

# 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)

# 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

# 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)

# Key properties of condensers

- Condensers are **meaning-preserving**
  - The resulting 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

# 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

# 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

# Leyden progress!

*openjdk.org/projects/leyden*

- Introduce condensers
  - *Toward Condensers* (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* (draft JEP, Minborg & Cimadamore)
  - Experiments in shifting speculative compilation (Rose *et al.*)
- Related improvements
  - Hermetic application packaging (prototype branch, Zhou)
  - JMOD-less linking (prototype, Gehwolf)

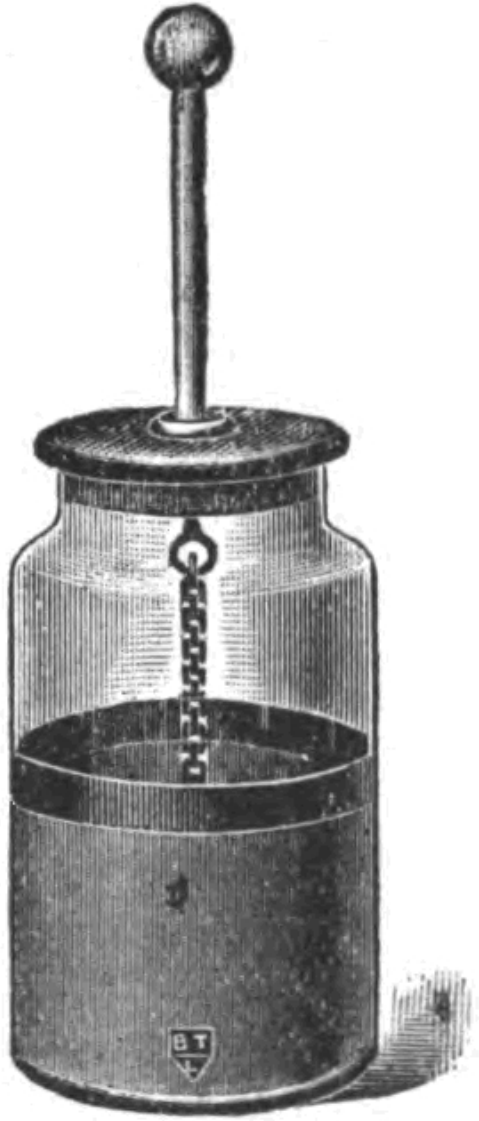
# Project Leyden

*Capturing Lightning in a Bottle*

Mark Reinhold
*Chief Architect, Java Platform Group, Oracle*

John Rose
*JVM Senior Architect, Java Platform Group, Oracle*

JVM Language Summit
2023/8/8



# Static AOT vs. dynamic JIT… a dilemma

- The Java answer is never “Choose One, Lose One” Java **balances** both static and dynamic reasoning.
- HotSpot speculatively **optimizes** dynamic states, in effect converting them to static states.
- In Leyden, such optimizations can be shifted, speculatively optimizing **before** app. startup.

(image from JVMLS 2010)

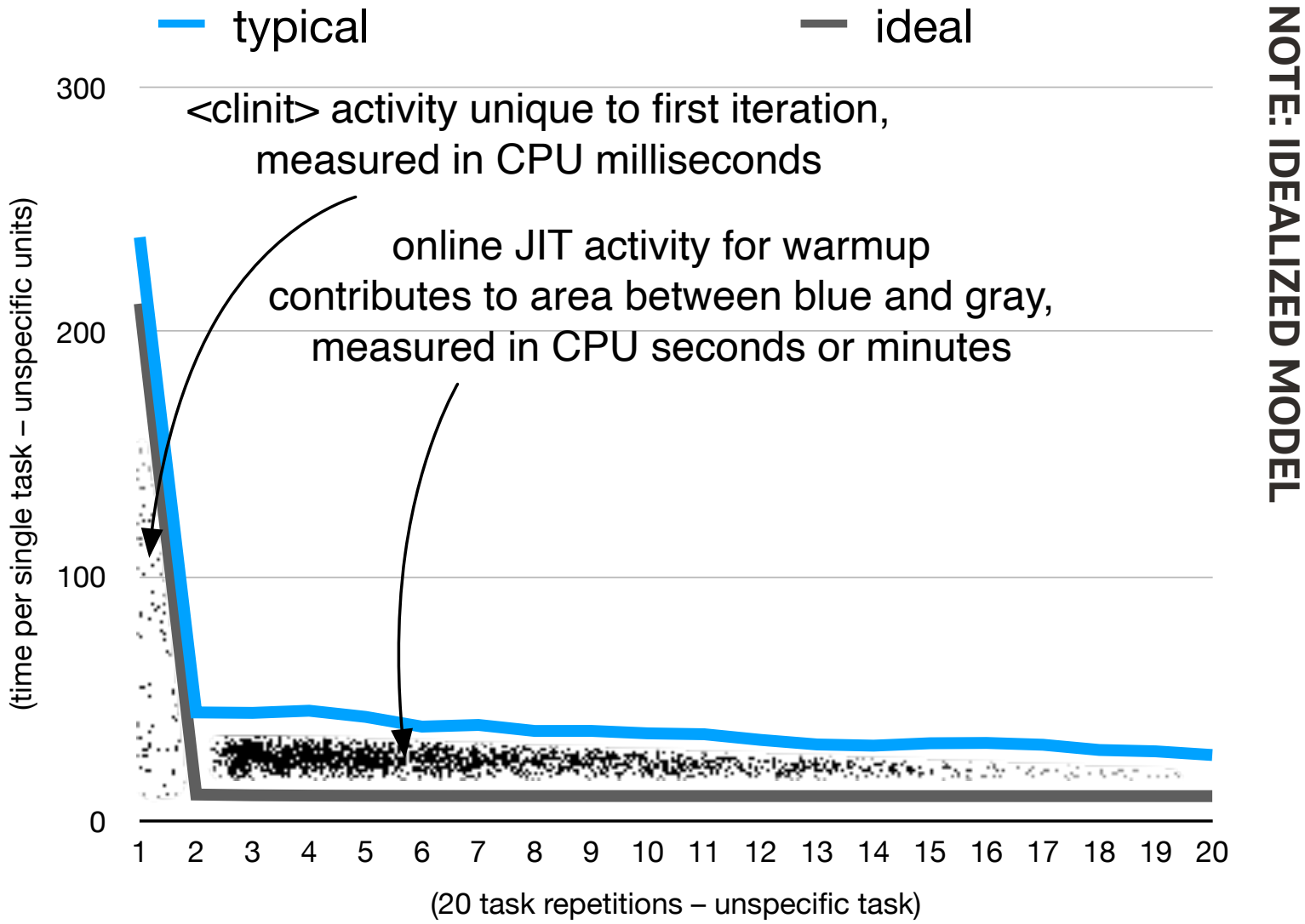
- Result: Users can drive startup time and warmup time into the noise, maintaining **compatibility**
  - No new constraints, no code change required

# Concepts and metrics (some definitions)

- **Startup activity** is setup computation to get through a first task, used for all tasks.
- **Startup time** is therefore (at most) the time of the first task, less other tasks.
  Time[Startup] ≤ Time[Task 1] - Time[Task 2]
  *Caveat: Not all apps have a repeatable representative task. Take with grains of salt…*
- **Warmup activity** is optimization effort (by JVM, not app) to reach peak performance.
- **Peak performance** may be defined as a statistical maximum, minus variance (noise).
  (Noise is often in the 3-5% range: say peak is reached at 95% throughput or better.)
- **Warmup time** is therefore the time it takes to reach 95% or higher of eventual peak.
- **Startup time** may also be measured as time to reach 80% (Pareto…) of eventual peak.

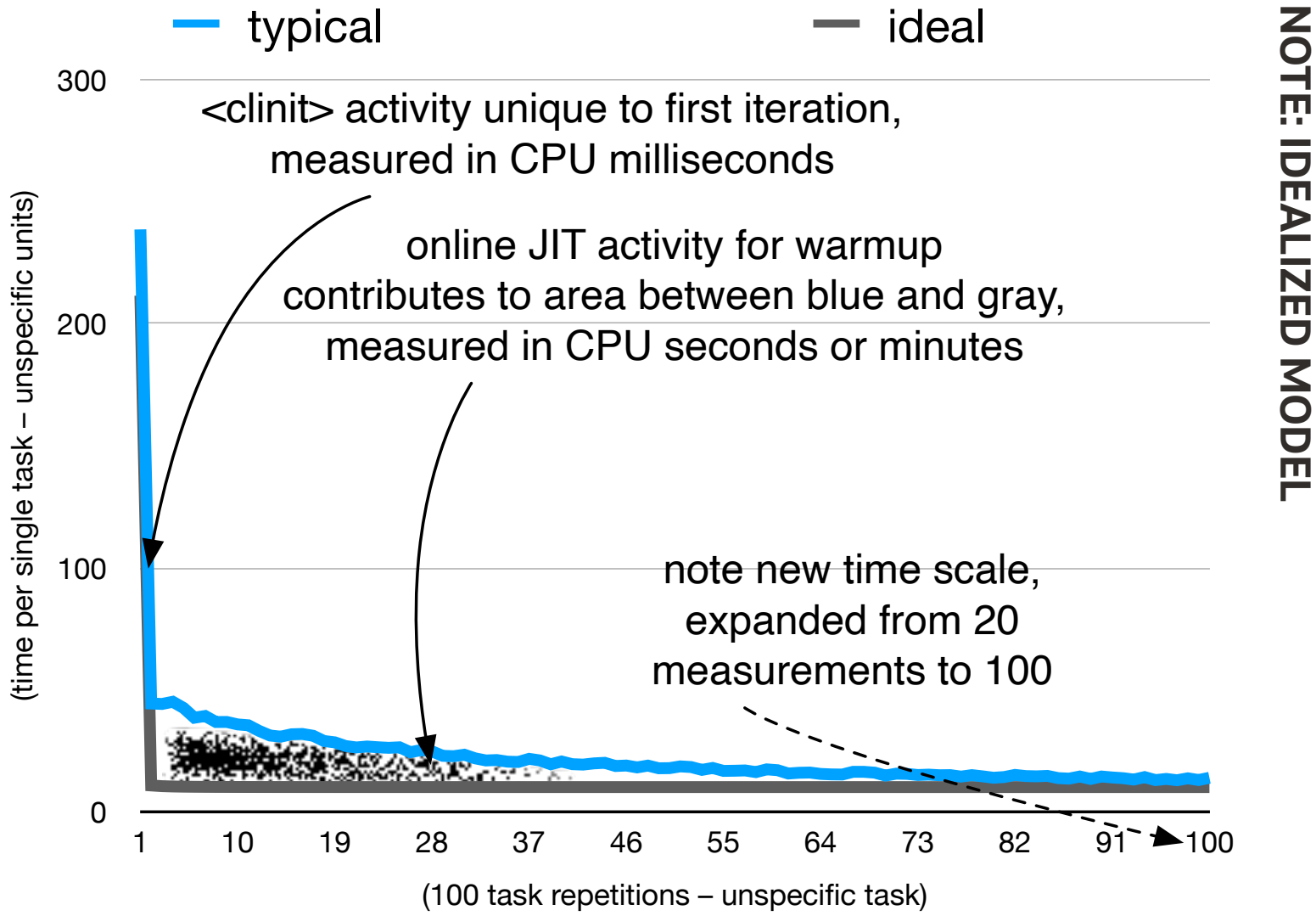
*(This is the startup time and warmup time we wish to drive into the noise.)*

# A tale of two graphs: What startup looks like today



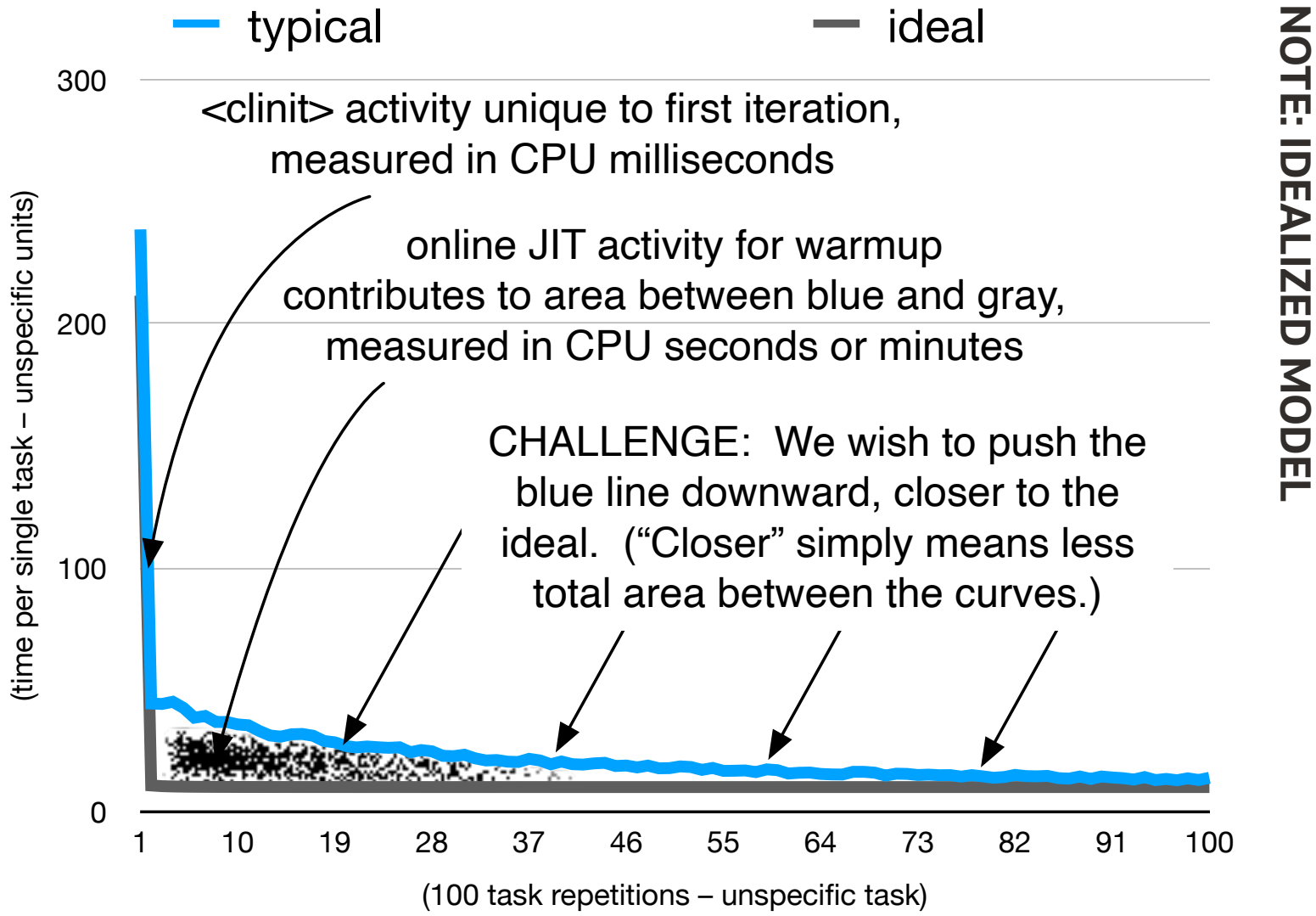

NOTE: IDEALIZED MODEL

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




NOTE: IDEALIZED MODEL

# Challenge: Make startup/warmup faster at multiple scales

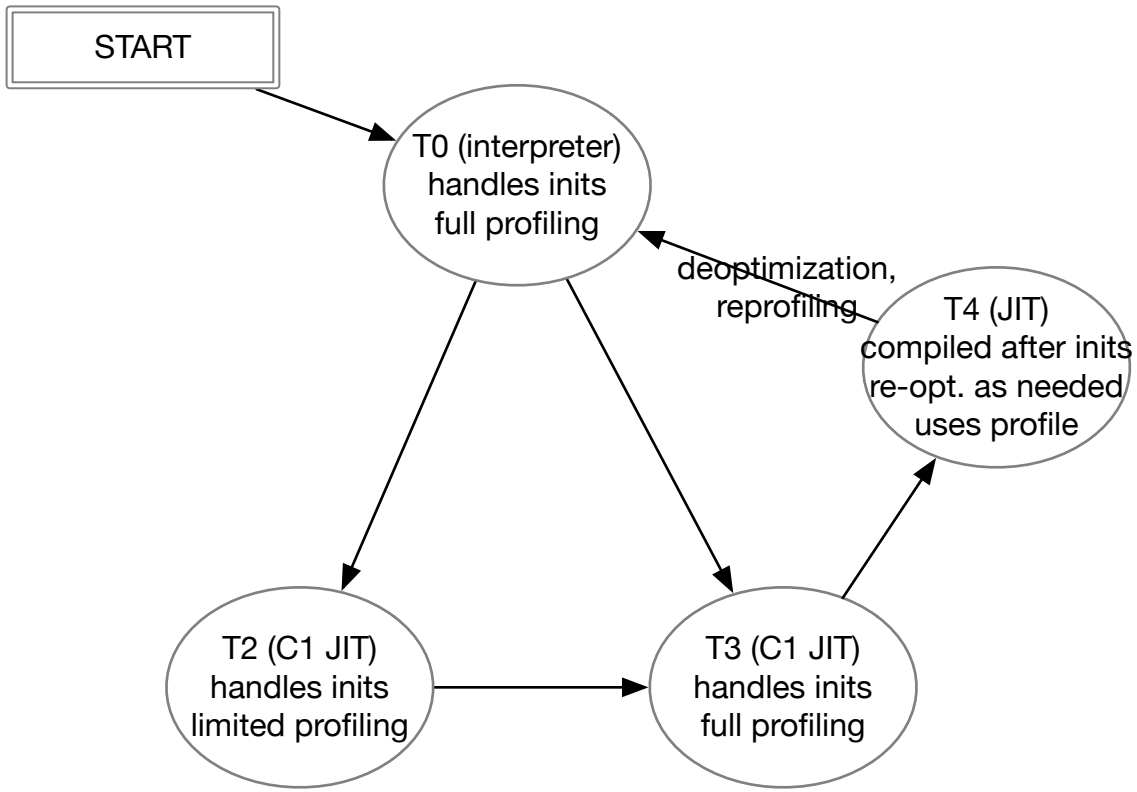



NOTE: IDEALIZED MODEL

# HotSpot Tiered Compilation: A primer

- Tiers 0..4 are execution modes, transparent except for performance.
- Tier 0 is the JVM bytecode interpreter.  It collects full profiling.
- Tier 1 is the simplest possible (C1) code.  No profiling.  Use is rare.
- Tier 2 is simple code with profiling at method entry (only).  Limited use.
- Tier 3 is simple code which also collects full profiling.  Spins up quickly.
- Tier 4 is optimized code which **benefits** from profiling, but collects none.
- Tier 4 may de-optimize on awkward inputs; lower Tiers may not.
- De-optimization is followed by further profiling, and re-optimization.

# HotSpot Tiered Compilation: A primer




JVM execution modes
and transitions in the standard
HotSpot execution policy.

(Rare T1 states are omitted.)

# Speedups are courtesy of HotSpot Tiered Compilation

- **Startup** is handled by slower Tiers 0..3, starting with the interpreter.
- Startup resolves symbols, runs class inits (<clinit>), runs indy BSMs.
- **Warmup** happens as code shifts from lower tiers to higher ones.
- First, lower tiers must gather **profiles** (execution paths and types).
- The JIT then uses those profiles to **optimize** Tier 4 code.  This takes time.
- Peak is reached when (most) code **stabilizes** in the highest Tier 4.

# Leyden can shift work to link, profile, initialize, & compile

- Leyden can shift the effort of collecting profiles and generating JIT code.
- An earlier run that gathers JVM information for Leyden is a **training run**.
- A later run that uses such information is called a **deployment run**.
- Some initialization states can be recorded for replay in deployment.
- Persistent profiles gathered in training can be applied in deployment.
- JIT code can be generated at startup from **persistent profiles**. (Helpful.)
- Or else, JIT code can be **archived** AOT for fast loading. (Very helpful!)

# Key concepts: How training prepares for deployment

- Definition: A **training run** is a representative execution of an application.
- Typical inputs and config. drive training run **startup** through expected paths and states.
- Preferably, training run **warmup** (repetitive tasks) leads to peak performance.
- During training, the JVM gathers initial states, profiles, JIT code, into **CDS** and/or **logs**. Optionally, multiple training runs are executed, and resulting logs of data are merged.
- The application is then **distilled** (terminal condensation step) into the optimized version.
- Executing the optimized application is called a **deployment run**.
- The deployment run starts with initial states, benefits from archived profiles and code.
- Optionally **auto-train**: Hide these steps “under the hood” for continuous improvement.

*If you can compose a system benchmark, you can compose a warmup training run.*

# The unreasonable effectiveness of training runs

- Java has always been both static and dynamic: Locally static, and globally dynamic.
- Training runs, which observe the app, are the dynamic “flip side” of static app analysis.
- We can use a Turing Machine (training) to exercise another T. M. (app), capturing reusable code and replayable profile and data states.
- Static views, though rich, tend to build models that are fragile, “needy” of constraints.
- Dynamic observations, if speculated, can be used as if they were statically deduced.
- This approach copes well with surprises during deployment. It is not surprising, since on-the-fly adaptation is Java’s distinct strength. (Speculative techniques allow for unplanned futures; this is a HotSpot core competency.)
- During deployment, we invisibly re-optimize JIT states captured from the training run.

# Our results are now promising enough to share

- We can now persist many states from training runs.  (See next slide.)
  - Using these states improves startup/warmup times for many use cases.
- Speculating on these states is robust, giving balance points between all-static and all-dynamic solutions.
  - Many recorded states can softly speculated, not firmly constrained.
  - With speculative optimizations, success is a habit, but failure is also an option.
- Online adaptive optimizations (JIT Tier 4) still get best performance.
  - We use HotSpot's JIT re-compilation to fill lingering performance gaps.
- Policy challenge: Combine the old and new tactics as needed, smoothly.

# States we can capture from training

- Class file events and other historical data: load, link, initialize (<clinit>), JIT compiles.
- Resolution of API points and indy (stored in constant pool images in the CDS archive).
- Execution profiles, code (all tiers).
- (And more later, such as lazy states…)

*Once captured, such data "looks static", helping optimizations. But it was "born dynamic". And it can change, triggering re-optimization.*


(image from JVMLS 2010)

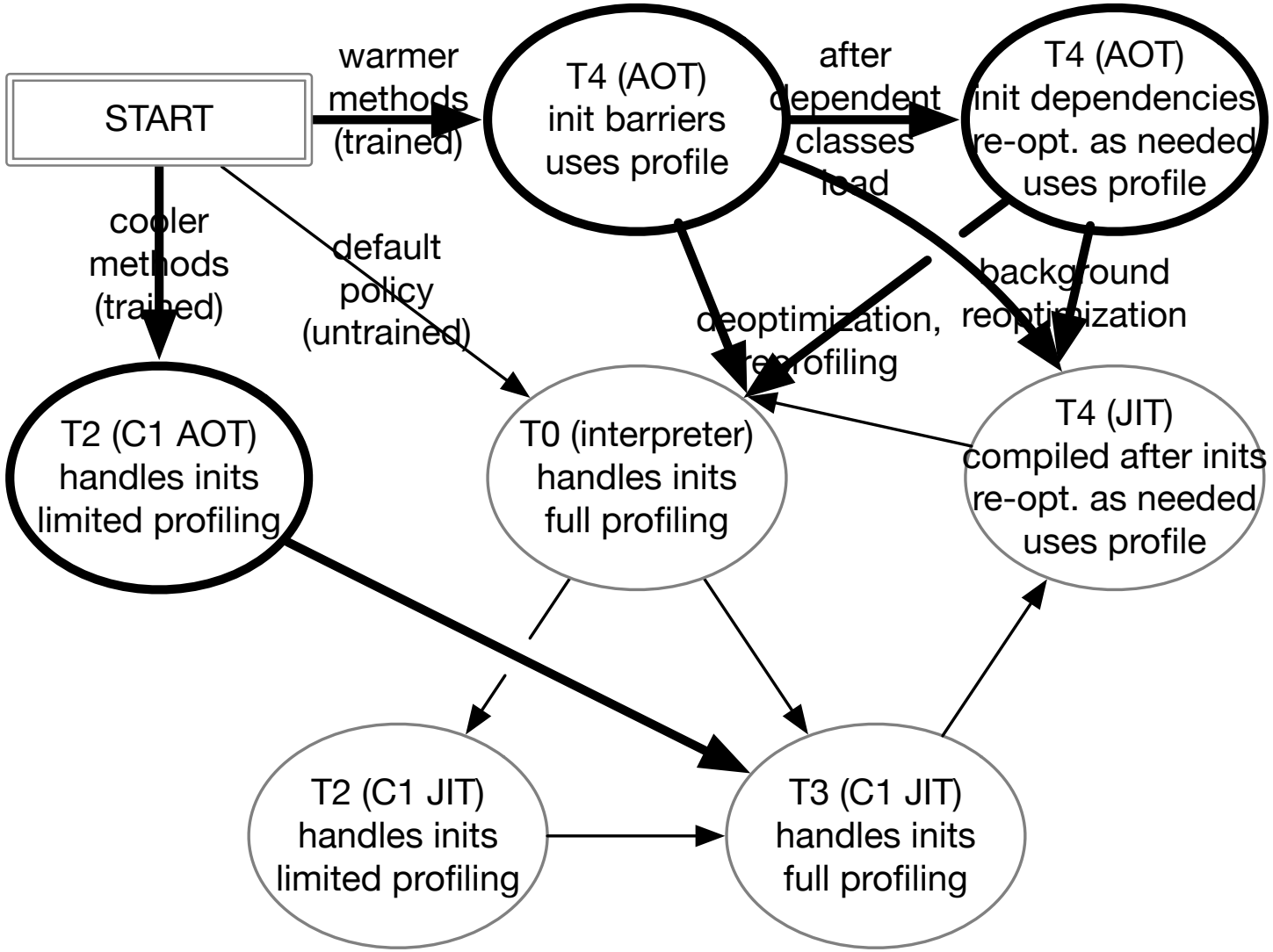
# Can this be made safe and sane?

- Key idea: **premain**, a specified location for re-executing recorded actions.
  - Java defines main as the first event of an application.
  - Leyden adds premain, an earlier event.
  - Premain warms up the application, rebuilding recorded states from training run.
- Some states which evolve from training runs are deemed safe to checkpoint. Other states are discarded.
  - The saved states from training are formalized as premain actions.
  - Some are irreversible, others are subject to speculation.
  - The JVM can *(or might not)* use CDS-like tech to set up these states before deployment. If it does, it appears that parts of premain “run super fast”.

# Archived code states arise from training run warmup

- Profiles from end of training run are preserved in CDS, for online re-optimization.
- Code is generated throughout training run and stored in archive for use with CDS.
- Code states include C1 (Tier 2 mainly) and optimized code (Tier 4).
- Code not classified as “hot” (i.e., used infrequently or only for setup) stays in Tier 2. Improves startup, as an alternative between the interpreter and aggressive optimization.
- Code is loaded from archive to online 5x to 500x faster than recompilation!
- Net savings can be seconds or even minutes, from **online JIT avoidance**.
- More savings from **interpreter avoidance**: The app. runs JIT code immediately. Archived code can de-opt into interpreter for corner cases, but is seldom discarded.
- Using the best available archived code improves startup and warmup (time to peak).
- **These savings are significant.** They can make the 2nd task run fast like the 100th.

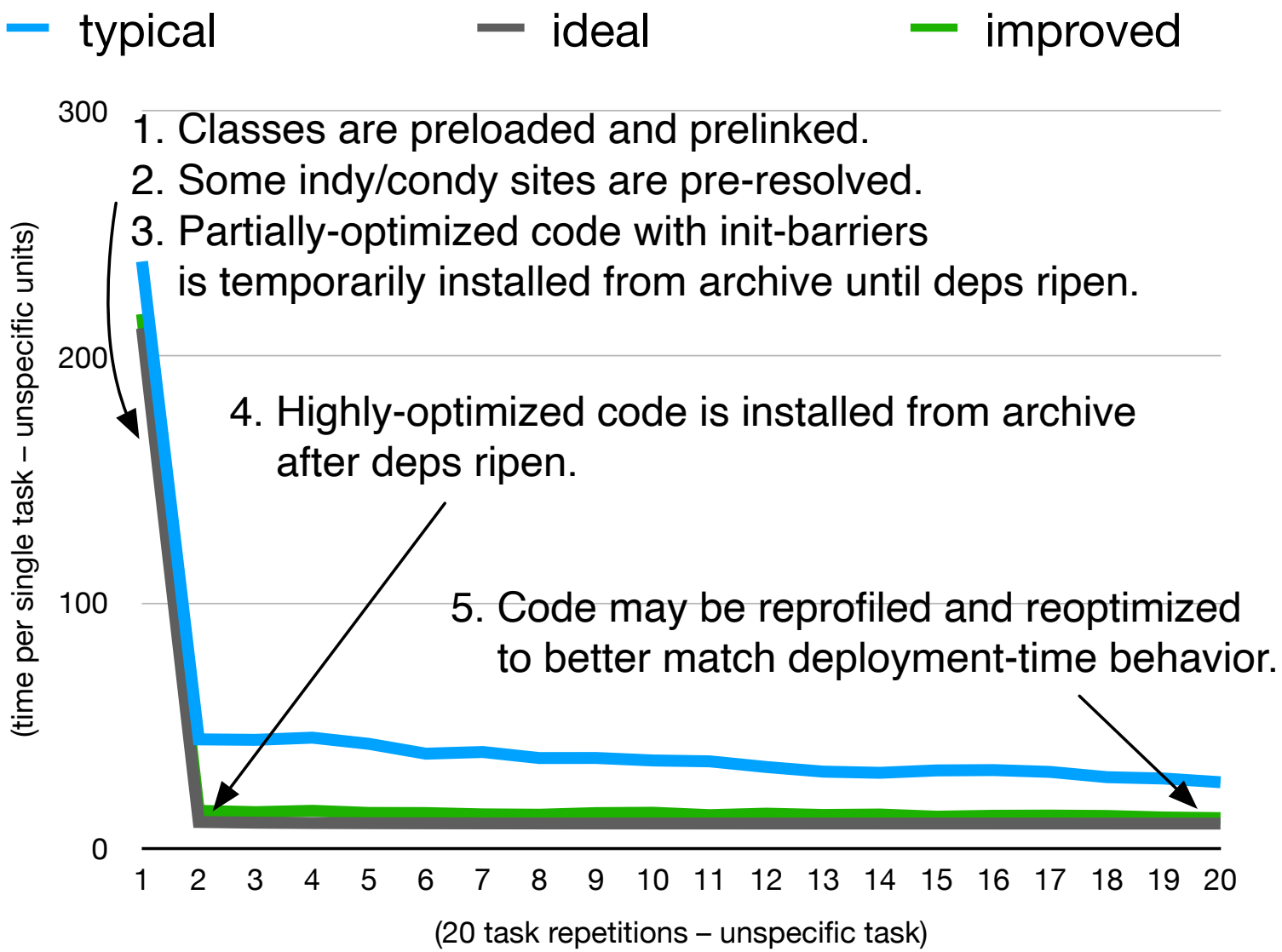
# JVM execution modes and transitions (new)




JVM execution modes and transitions, as extended by Project Leyden.

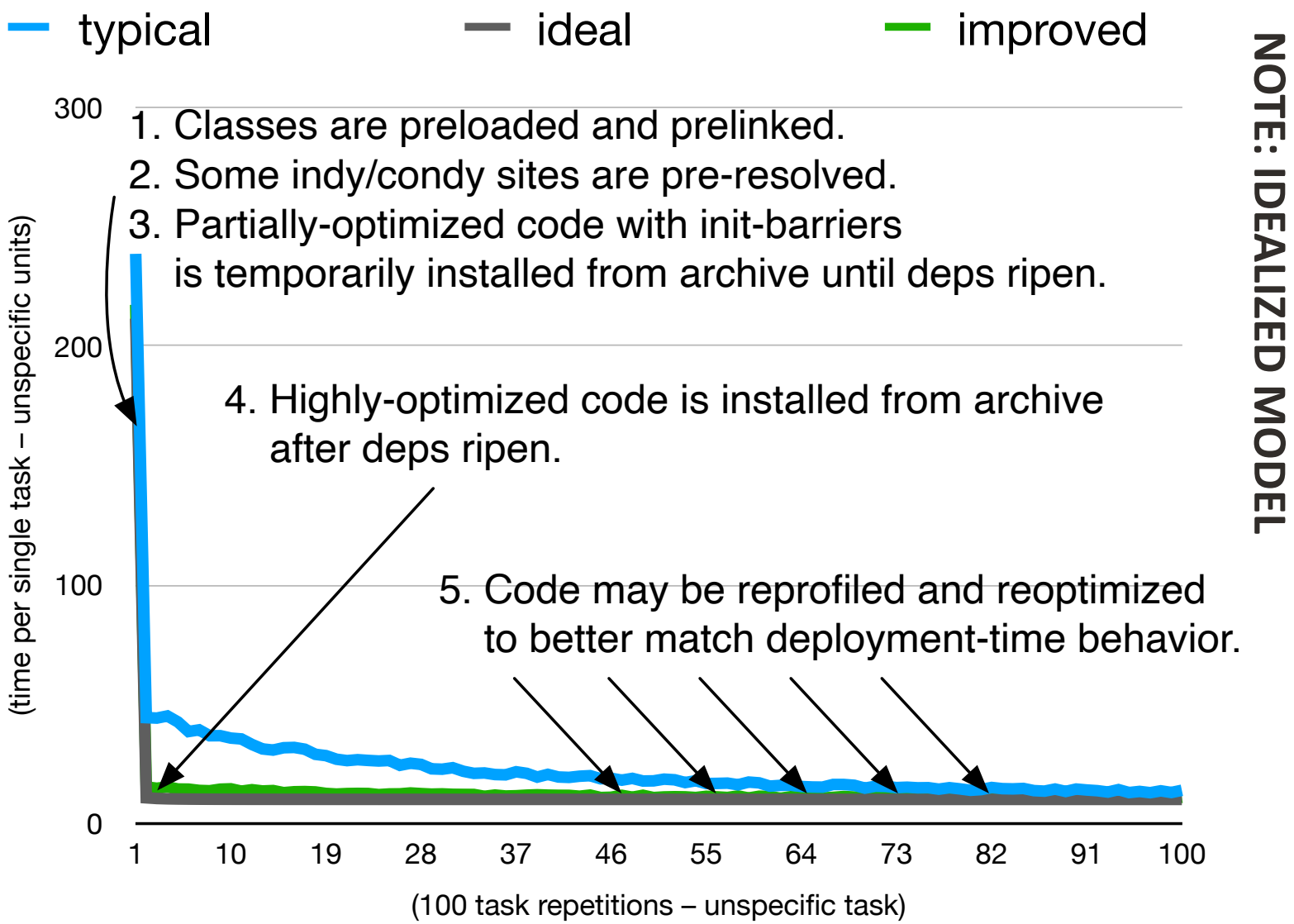
Note: Heavy lines mark new modes & transitions.

# A tale of three graphs: Better startup



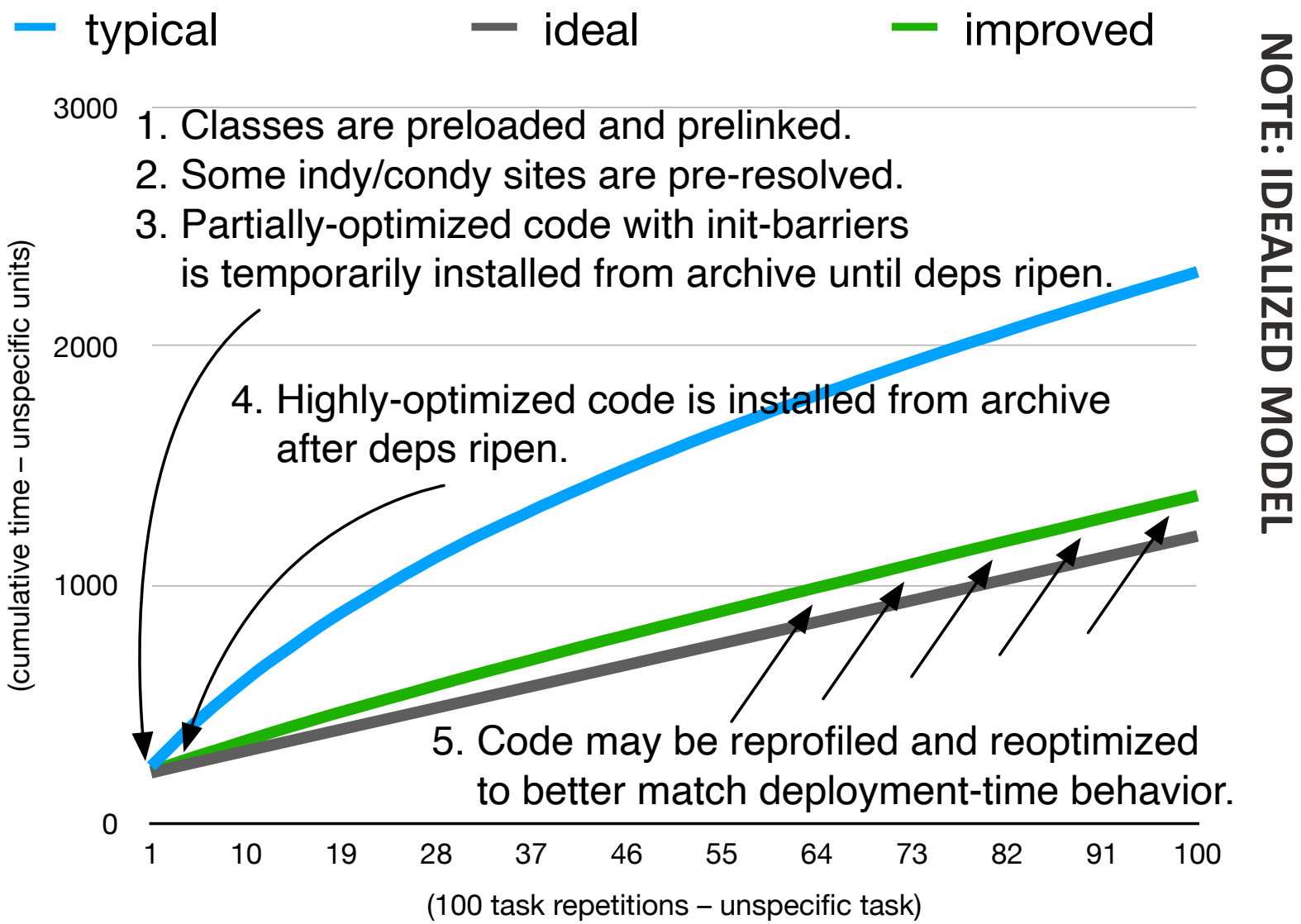

**NOTE: IDEALIZED MODEL**

# A tale of three graphs: Better warmup




NOTE: IDEALIZED MODEL

# A tale of three graphs: The long-term view

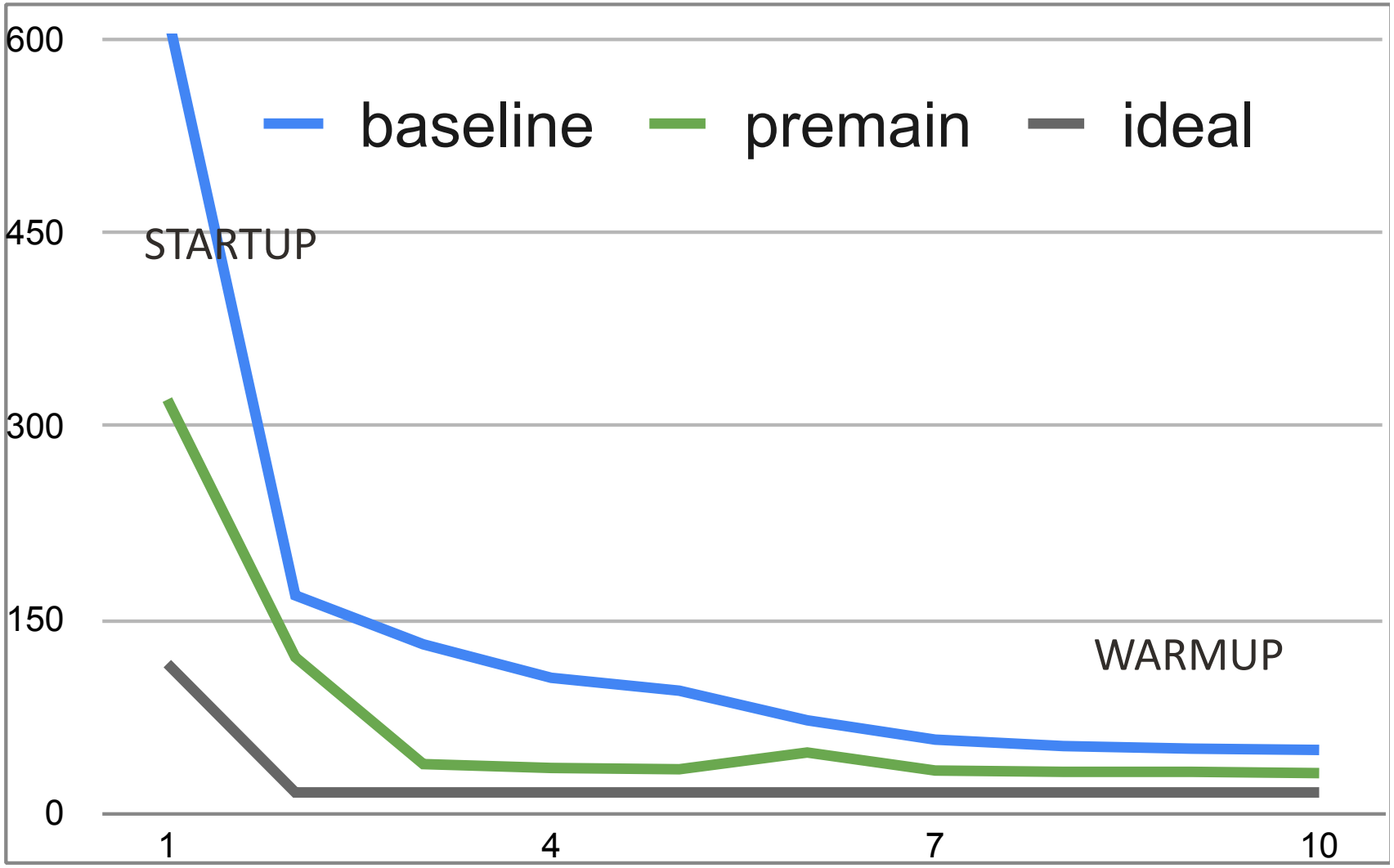



NOTE: IDEALIZED MODEL

# Case study: javac startup, warmup, peak

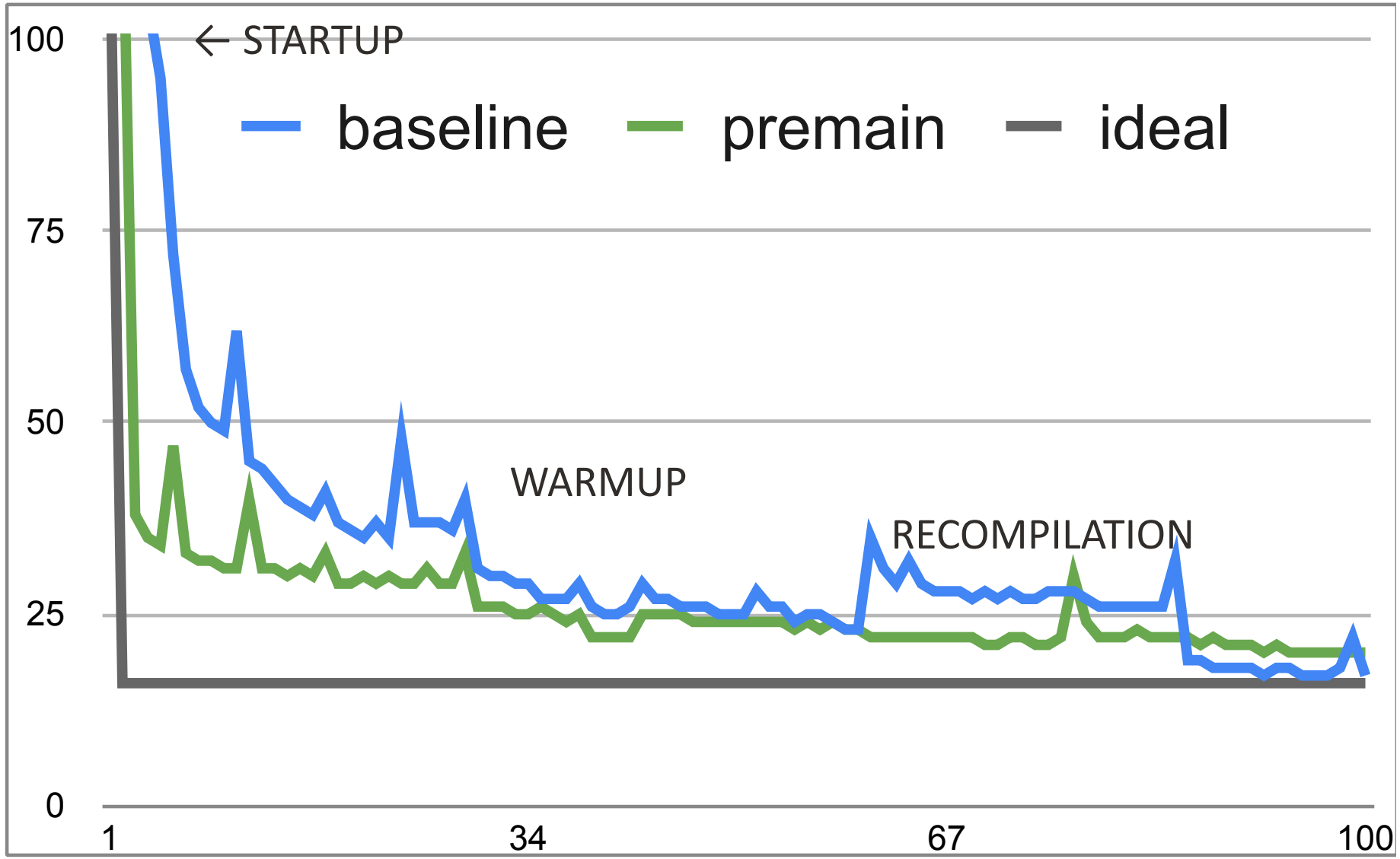
- Use javac to compile a series of 100 small source files. First task triggers startup. Each sub-task does it all again, and discards results. Repetition triggers warmup.
- States from training run are stored in CDS archive (with JIT cache).
  These AOT states include loaded metadata, selected resolution results.
  Indy resolutions include hidden class metadata and method handles in Java heap.
  *CDS does this already, but this use is broader and deeper — CDS has been underutilized!*
- Init-barriers code (Tier 4, C2) traps ONCE to interpreter to handle <clinit> events.
  Unlike default Tier 4, init-barriers code has a usable fast path after <clinit>.
- Regular archived code is loaded only when all <clinit> deps “ripen”.
- When most or all archived code is installed, Tier 4 recompilation begins.
  *CAVEAT: Work in progress. More tuning needed for machinery, policy, user model.*

# javac, the first few iterations

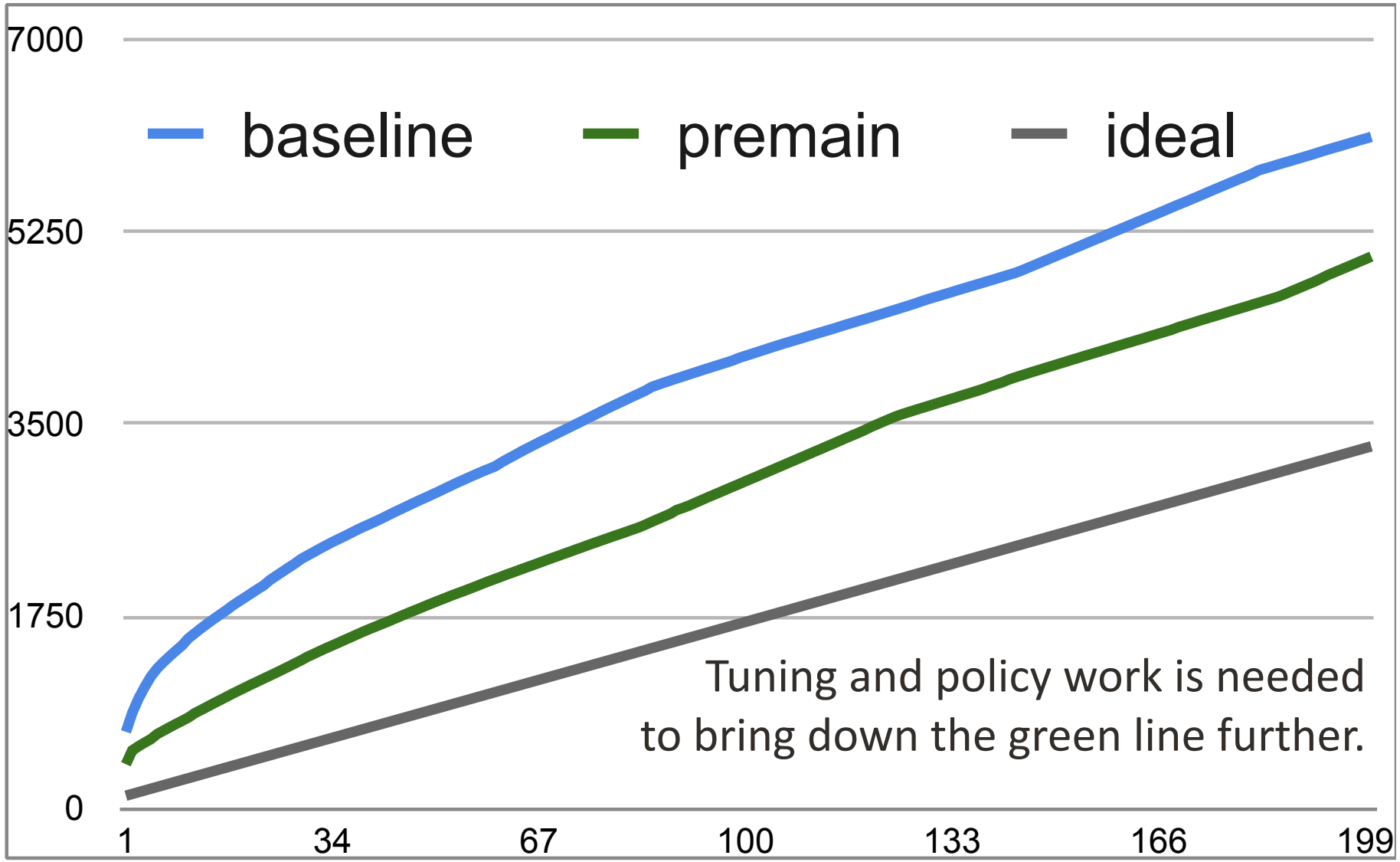



NOTE: ACTUAL MEASUREMENTS

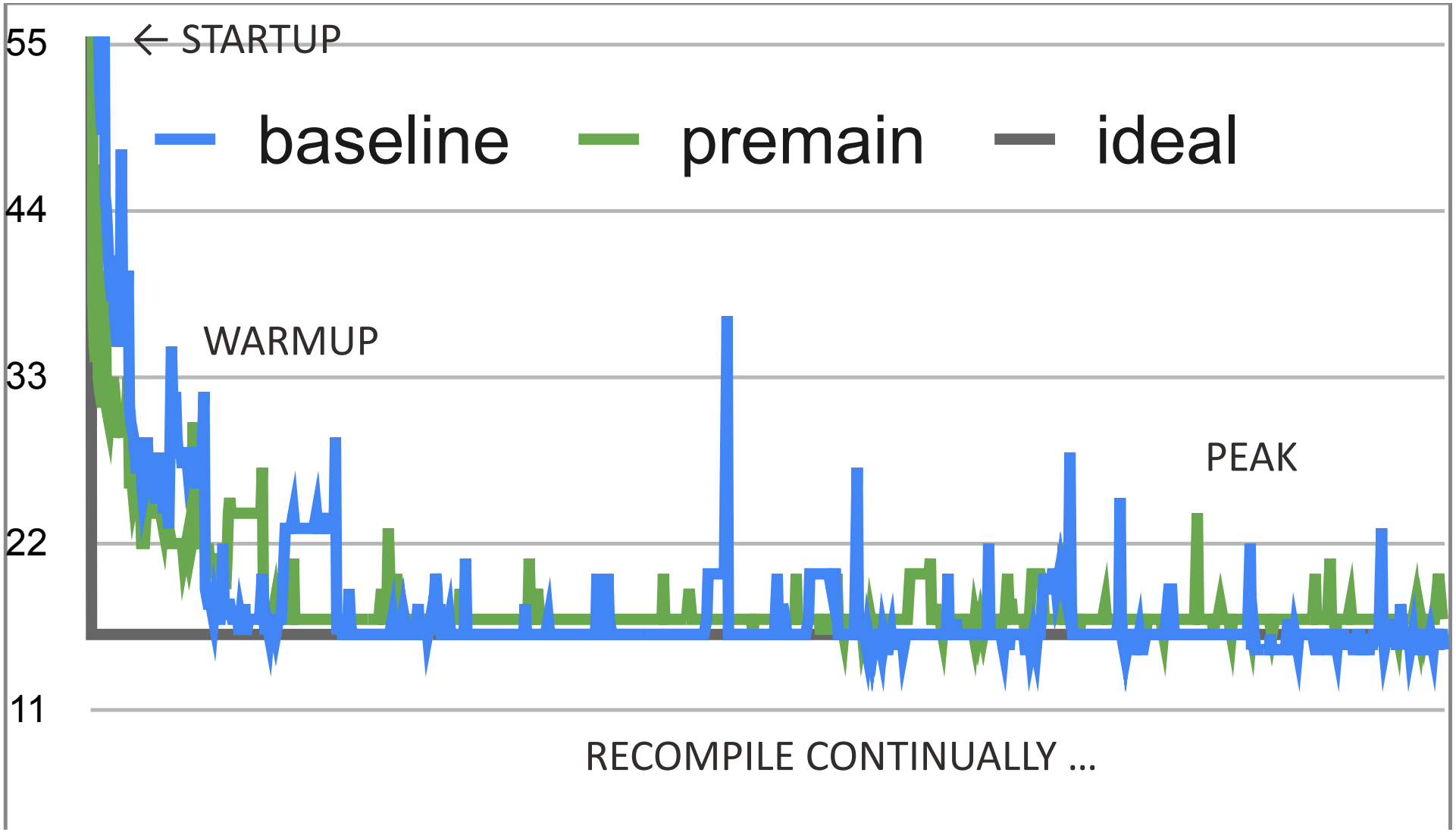
# javac, more iterations, showing start of recompilation



# javac, viewed cumulatively (total execution time)



# javac, in the longest time scales, experiences GC noise




Integral millisecond scale shows quantization noise also.

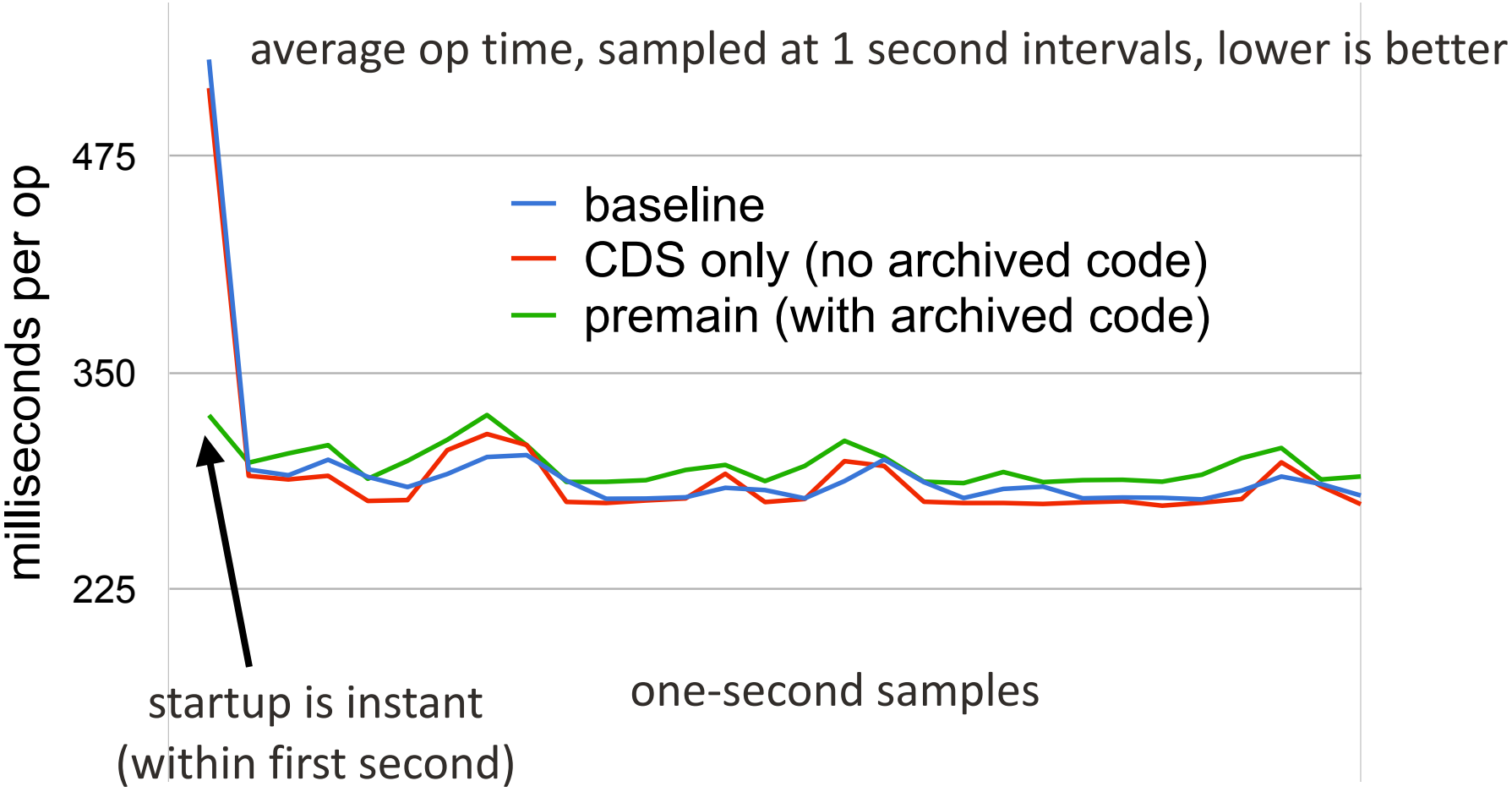
The baseline policy is still champion, for now.

But it is early days.

# Lessons from javac case study

- It works:  We can shift computation to premain, via CDS back to training runs.
- There are lots of startup and warmup states to push back to premain.
- There is no one “magic bullet” technique.  Let’s keep hunting for more.
- Multiple time-scales (of warmup) are important.  Let’s try to chase them all.
- When AOT stuff can go wrong, use JIT re-optimization as a fallback.
- Machinery doesn’t tune itself.  Well-tuned policy is a way of life, not a possession.

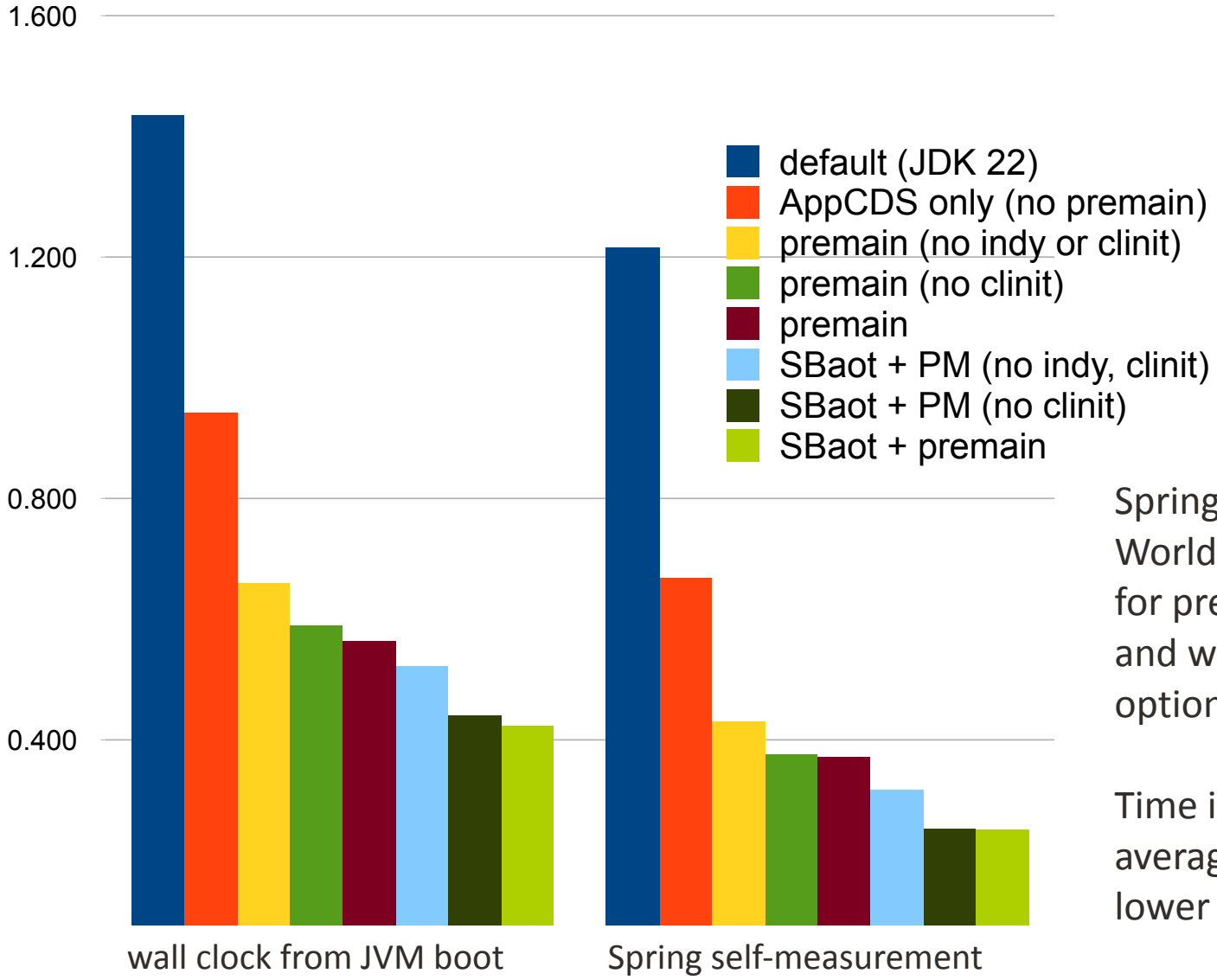
# Case study:  JVM2008 XML validation benchmark



# Lessons from XML validation case study

- Sometimes startup is the only interesting win; warmup is already OK for smaller apps.
- In this case, we can decisively improve startup, compared to the baseline policy.
- Benchmark noise can make it hard to decide where is the peak performance.
- Over time, the baseline policy still seems to win by a hair; this may need more work.

# Case study:  Spring Boot application framework startup




Spring v3.1.2 Hello World boot times for premain, with and without various options.

Time in seconds, average of 3 runs, lower is better

# Lessons from Spring Boot case study

- There are many tactics which can improve startup.
- We win big because the tactics all work in synergy (are not mutually exclusive).
- AppCDS is a win, back-shifting loading and resolution through premain.
  Code archiving is further win, back-shifting JIT work through premain.
- Back-shifting indy resolution states unlocks further optimizations.
  The clever Tier 4 code which checks for <clinit> wins, but only a little.
- Low-level JVM metrics give a little more data than Spring self-reported time.

# Current status of premain work: Fresh beginnings

- For now, premain activities are derived automatically from training runs.
- Optimizable states generated by premain are dumped into the CDS/JIT archive.
  (Other premain states are dropped, assumed to be reconstructed at deployment.)
- In the future, user-defined activities can march in this parade, as well.
  This requires work on characterizing which of those activities are trusted as pure.
- **No loss of Java's natural dynamism, no new constraints on the programming model.**
- Need user-friendly workflows (not flag soup): multi-run condensation, "auto-train".
  This requires more work on moving CDS/JIT states into log files and vice versa.
- Plenty of additional opportunity for performance tuning, policy integration, JIT work.

***There is so much more we can do. Join us!*** openjdk.org/projects/leyden

# Questions?

*openjdk.org/projects/leyden*

# Project Leyden

*Capturing Lightning in a Bottle*

Mark Reinhold
*Chief Architect, Java Platform Group, Oracle*

John Rose
*JVM Senior Architect, Java Platform Group, Oracle*

JVM Language Summit
2023/8/8