Hotspot & AOT

Now it's time to compile

Dmitry Chuyko

Java SE Performance Team

September 22, 2016

Contents

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.

Introduction

Reminder: It's 2016

- JDK 9 Early Access
  https://jdk9.java.net/
- JDK 8u
- JDK 7 End of Public Updates in April 2015

Overview: Computing

A long time ago in a galaxy far, far away...

- Pre-computer machines appeared
- Computers and their machine codes
- Languages and compilers
- Scripts
- Computer science

Overview: Java

- Is a language
- Set of specifications
- Used to be called *slow*
  ´Because it's interpreted"
  (not true)
- "Write once, run anywhere"
  (true)

Overview: JVM

- Is a code itself
- Can dynamically execute arbitrary correct bytecode

Overview: JVM

- Is a code itself
- Can dynamically execute arbitrary correct bytecode
- May be written in anything
- May produce native code and re-use the result

The Current Situation

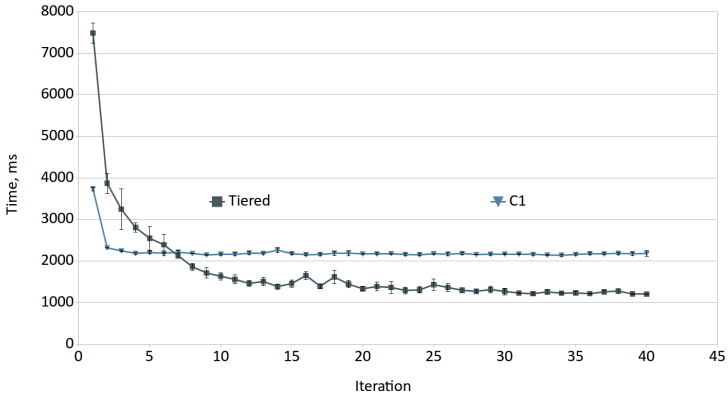
Overview: Hospot

- Is a JVM
- Written in C++
- Native shared libraries (libjvm)
- Produces bytecode dynamically for its own purposes
- Does just-in-time compilation
- Supports many modes
  - Garbage collectors
  - Pointers encoding
  - etc.

Overview: JIT in Hotspot

- Tiered compilation
  - Level 0. Interpreter
  - Level 1. C1 without profiling (optimized), terminal
  - Level 2. C1 with basic profiling
  - Level 3. C1 with full profiling
  - Level 4. C2, terminal, expensive
- Unused method versions are thrown away to save footprint
- Optimizations, resource constraints
  - $\Rightarrow$ de-optimizations to level 0
- All modes (if not switched off), CPU instruction set
  - Custom code

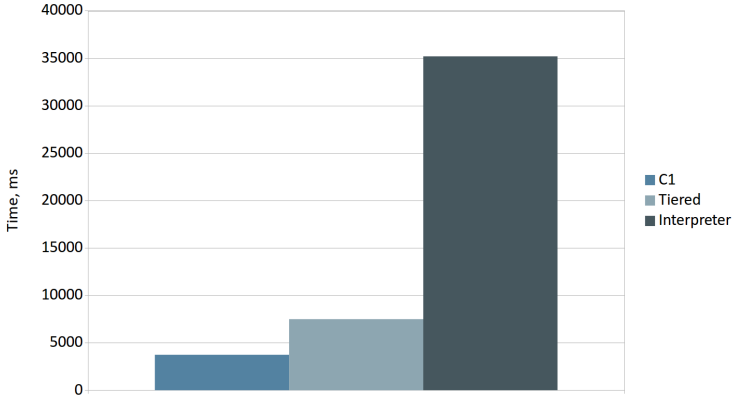
Problem: Application Warm-up



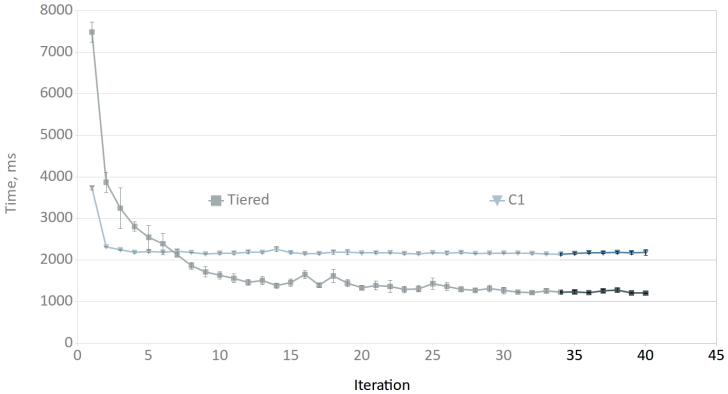
Iterative workload

- Startup time
- Time to performance

Problem: Startup Time

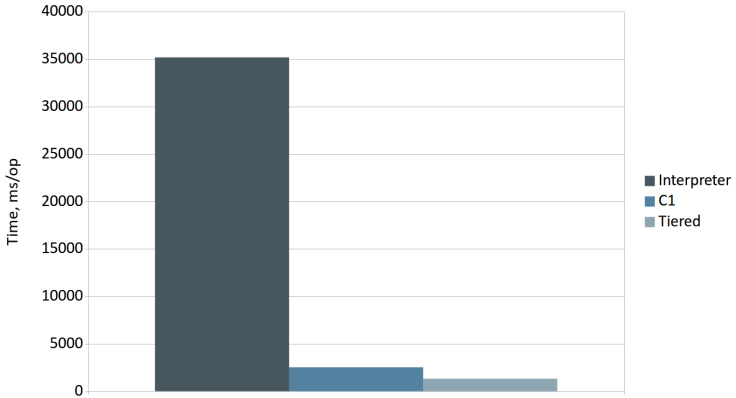


Problem: Time to Performance



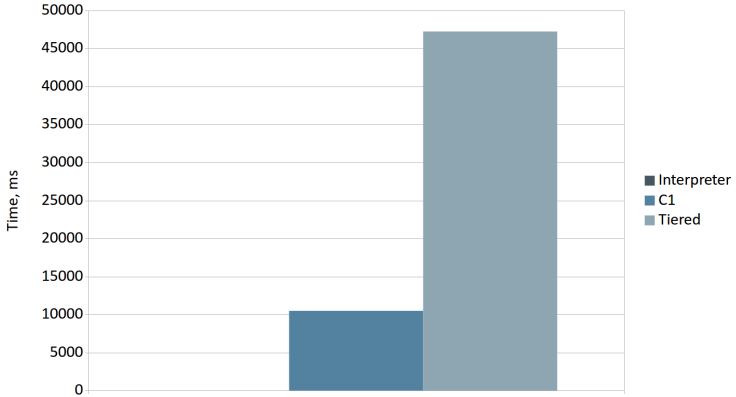
Problem: Time to Performance

Peak Performance

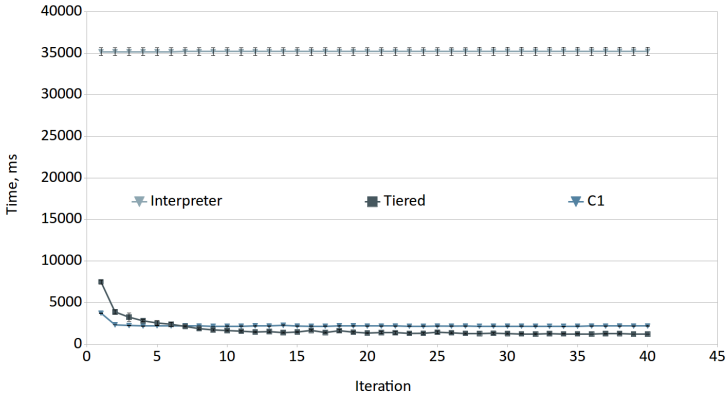


Problem: Time to Performance

Sum of Iterations



Problem: Application Latency



Iterative workload

- Interpreter is slow
- Level 1 (C1) is relatively also slow

Problem: Application Latency

- Wish it to be HFT...

```
@Transactional void buyOrSell(Quote quote)
```

Problem: Application Latency

- Wish it to be HFT...

```
@Transactional void buyOrSell(Quote quote)
```

  - De-optimization when flow changes
  - Training workloads

- And you meet

```
void buy_or_sell [[db:transactional]] (Quote* quote)
```

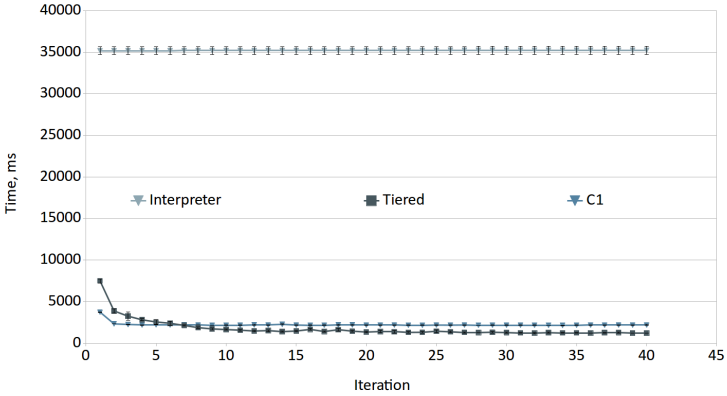
```
CFLAGS_ALL += -O3
```

Problem: Bootstrapping

Meta-circular implementations

- It's possible to write JVM in Java, Scala or JavaScript
- "My dear JVM existing as bytecode image, please start and make yourself efficient in execution of bytecode. Quickly"
- Actually the 3 problems above but doubled

Problem: Bootstrapping



Ahead-of-time Compilation

Solution: Startup time

- Pre-compile initialization code
  - No interpreter for class loading, reflection etc.
  - No resources for compilation

Solution: Time to performance

- Pre-compile critical code
  - Start with much better than interpreter performance
  - No resources for compilation
- Reach peak performance
  - Collect same profiling info
  - JIT with profile-guided optimizations

Solution: Latency

- Pre-compile critical code
  - High and stable performance
    - Optimizations
  - No de-optimization (almost)
  - No re-compilation (almost)

Solution: Density, Power Consumption

For free

- Some critical code is pre-compiled
- Share it

Pre-compilation: Different Solutions Exist

- AOT whole application to native executable
  - Native exe/elf
  - Trial runs for better image layout
  - Bundled or shared VM
  - Deep dependency analysis
  - Pre-defined mode
  - JIT is secondary
- VM with JIT and AOT compilers
  - Optional cache for class data and code
  - Trial runs for methods filtering
- Replay recorded compilations and optimizations

Pre-compilation: For Hotspot

- Need to generate code
  - Mostly no de-optimizations
  - Better than C1
- No tight time budget
- Need to resolve and load generated code

Pre-compilation: For Hotspot

- Need to generate code
  - Mostly no de-optimizations
  - Better than C1
- No tight time budget
- Need to resolve and load generated code
- How about one more compiler?

Graal

Graal: Project

- Experimental dynamic compiler written in Java
- Supports Java
- OpenJDK project
  http://openjdk.java.net/projects/graal/
- Oracle Labs team
- GraalVM based on Hotspot
  http://www.oracle.com/technetwork/oracle-labs/program-languages/overview/index.html

Graal: For AOT

- It proven to work
  - SubstrateVM
- Flexible and handy
  - Modular
  - Annotation based way
- Possible to avoid most de-optimizations
  - No speculative optimizations
  - Compile all paths
- Focused on performance

Graal: For AOT

- It proven to work
  - SubstrateVM
- Flexible and handy
  - Modular
  - Annotation based way
- Possible to avoid most de-optimizations
  - No speculative optimizations
  - Compile all paths
- Focused on performance
- How does it interact with Hotspot?

JVM Compiler Interface

JEP 243: Java-Level JVM Compiler Interface

- OpenJDK feature, already in 9
  http://openjdk.java.net/jeps/243
- Experimental feature

JEP 243: Goals

- Allow the JVM to load Java plug-in code to examine and intercept JVM JIT activity.
- Record events related to compilation, including counter overflow, compilation requests, speculation failure, and deoptimization.
- Allow queries to relevant metadata, including loaded classes, method definitions, profile data, dependencies (speculative assertions), and compiled code cache.
- Allow an external module to capture compilation requests and produce code to be used for compiled methods.

JVMCI: Graal as C2 Replacement

```
-XX:+UnlockExperimentalVMOptions -XX:+EnableJVMCI -XX:+UseJVMCICompiler
[-Djvmci.Compiler=graal]
```

JVMCI: Details

- Not used for C1, C2
- Special module jdk.vm.ci
- Familiar extension patterns
  - CompilerFactory, StartupEventListener, HotSpotJVMCIBackendFactory, HotSpotVMEventListener...

JVMCI: How it works

Hotspot

- Compilation Queue
- Metaspace
- Code Cache

JVMCI Compiler

- Compilation Request
- jdk.vm.ci.meta
- byte[]

JVMCI: How about this?

Hotspot

- Queue
- Metaspace
- Code Cache

Proxy

Network

Proxy

Compilation Server

- Request
- jdk.vm.ci.meta
- byte[]

Artifacts

Code: AOT Modes

- Targeted at problem
  - Tiered. Similar to Level 2
  - Non-Tiered – Latency
- Targeted at VM mode
- Defined by Graal/AOT options (profiling, thresholds etc.)

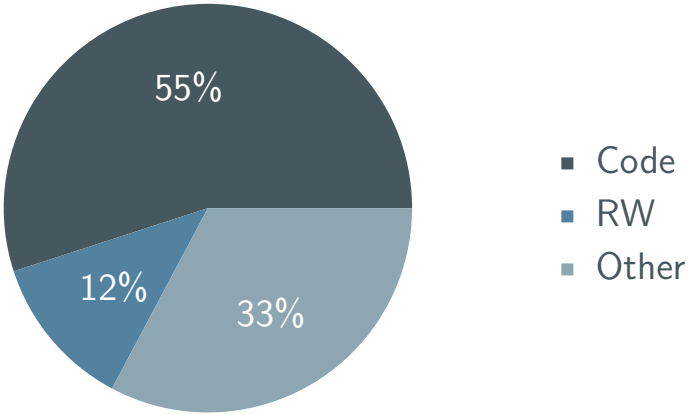
Code: AOT & Tired

- Tiered
  - AOT → level 3 → AOT → level 4
- Non-Tiered
  - AOT

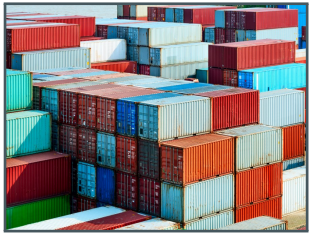
Code: Libraries

- Native shared library (ELF DSO)
  - OS knows how to treat it right
  - Compatible with tools
  - Specific to mode
  - Same runtime
- Modified Hotspot that works with compiled methods from shared libraries
- New `jaotc` tool for compilation
  - Modules
  - Jars
  - Classes

Code: libjava.base.so, 240 MB



Packaging: Self-contained Apps



Packaging: Self-contained Apps

- Java Packager
  - Prepares fancy .dmg for shiny Mac
  - Bundled with 100 Mb JRE
- JEP 275: Modular Java Application Packaging
  http://openjdk.java.net/jeps/275
  - jlink helps to generate a JRE image with the required modules only
  - Extensions
  - AOT libs can be created and added