Techniques for a faster JVM start-up

Ionuţ Baloşin

IonutBalosin.com | IonutBalosin

Agenda

01 When the JVM start-up matters

02 App/Dynamic Class-Data-Sharing in HotSpot VM

03 Shared Class Cache in Eclipse OpenJ9

04 Ahead-of-Time Compilation with GraalVM native-image

About Me

Software Architect @ Raiffeisen Bank International
Technical Trainer | Security Champion | Blogger | Speaker

My Training Catalogue

- Software Architecture Essentials
- Java Performance Tuning
- Designing High-Performance, Scalable, and Resilient Applications
- Application Security for Java Developers

Training figures: 80+ sessions | 900+ trainees | 1300+ hours | 10+ clients | 4+ countries
Conference figures: 35+ sessions | 14+ countries

www.IonutBalosin.com

When the JVM start-up matters

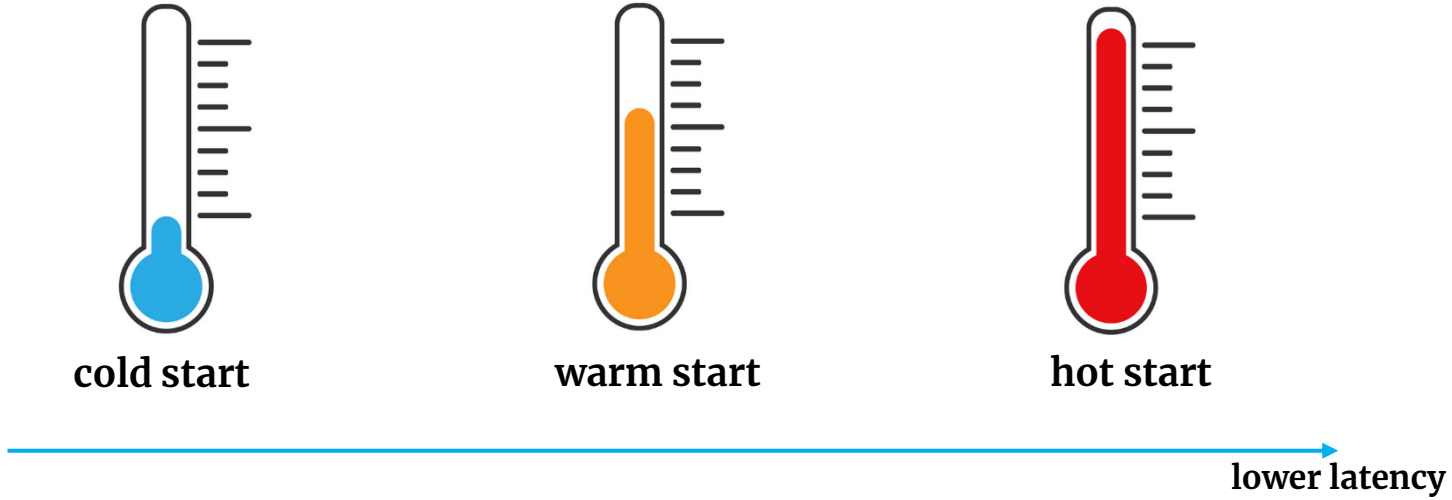
01

Applications where start-up time matters

- serverless applications (e.g., Function-as-a-Service)
- short-lived applications
- command-line applications

Benefits

- cost savings in cloud
- quick feedback loops during development cycles
- quick time-to-first-response in cold-start scenarios



cold start - start the app from scratch
warm start - prepare the app to reach a “steady” state
hot start - background to foreground transition (i.e., app is already in a “steady” state)

(Few) JVM (Cold) Start-up Optimization Techniques

App/Dynamic Class-Data-Sharing (CDS) **HotSpot VM**

Shared-Class-Cache (SCC) **Eclipse OpenJ9**

Ahead-of-Time (AOT) Compilation **GraalVM native-image**

OpenJDK CRaC (save the state of a JVM and restore it later)

OpenJDK Leyden (static application binaries with a faster start-up)

Alibaba Dragonwell JWarmUp

jlink/jpackage (possible improvements by removing modules)

Azul Prime ReadyNow!

(Few) JVM (Cold) Start-up Optimization Techniques

App/Dynamic Class-Data-Sharing (CDS) HotSpot VM

Shared-Class-Cache (SCC) Eclipse OpenJ9

Ahead-of-Time (AOT) Compilation GraalVM native-image

} current scope

OpenJDK CRaC (save the state of a JVM and restore it later)

OpenJDK Leyden (static application binaries with a faster start-up)

Alibaba Dragonwell JWarmUp

jlink/jpackage (possible improvements by removing modules)

Azul Prime ReadyNow!

App/Dynamic Class-Data Sharing in HotSpot VM

02

Class Data Sharing (CDS) - caches preprocessed metadata on disk (i.e., **default CDS archive** or **static base CDS archive**)

It contains 1300+ core library classes loaded by the **bootstrap class loader**

Classes are stored in a format that can be loaded very quickly (compared to classes stored in a JAR file), hence **improving the start-up time**

In most JDK distributions **CDS is enabled by default** unless *-Xshare:off* is specified

CDS location

```
Linux> $JAVA_HOME/lib/server/classes.jsa

Windows> %JAVA_HOME%\\bin\\server\\classes.jsa
```

AppCDS - extends **CDS** to built-in system class loader (i.e. app class loader) and custom class loaders (i.e., **static archive**)

AppCDS archive includes also core library classes (in the same archive) and it is a three-step procedure

Classes stored in the CDS are a few times larger (e.g. 3-5x) than classes stored in JAR files or the JDK runtime image

```
[info][cds] Shared spaces: preloaded 3959 classes

$ ls -l --block-size=1K
22392 -r--r--r-- 1 ionutbalosin ionutbalosin 22392 Mar 22 08:24 app-cds.jsa
```

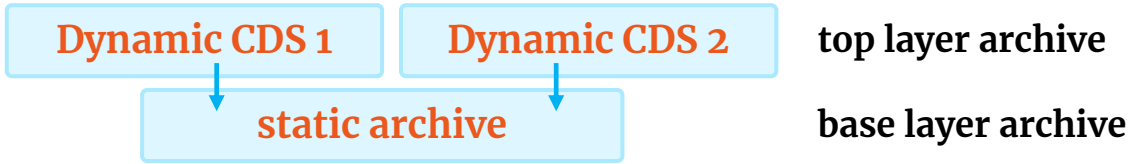
E.g., 3959 classes ~ 21.87 MiB (22392 KiB)

Dynamic CDS - extends **AppCDS** to allow dynamically archiving at the end of the Java process (i.e., **dynamic archive**)

It simplifies the **AppCDS** archive creation by eliminating the need to create the class list (i.e., the initial AppCDS step), hence it is a two-step procedure

By default, dynamic CDS archive is created on top of the static base CDS archive (e.g., *classes.jsa*) as a **top-layer archive**, and it implicitly uses less disk space

```
[info][cds] trying to map $JAVA_HOME/lib/server/classes.jsa
[info][cds] Opened archive $JAVA_HOME/lib/server/classes.jsa

[info][cds] trying to map dynamic-cds.jsa
[info][cds] Opened archive dynamic-cds.jsa
```



The **static archive** could be a default CDS archive (i.e., *classes.jsa*) or a static archive (i.e., AppCDS archive)

Chaining CDS archives

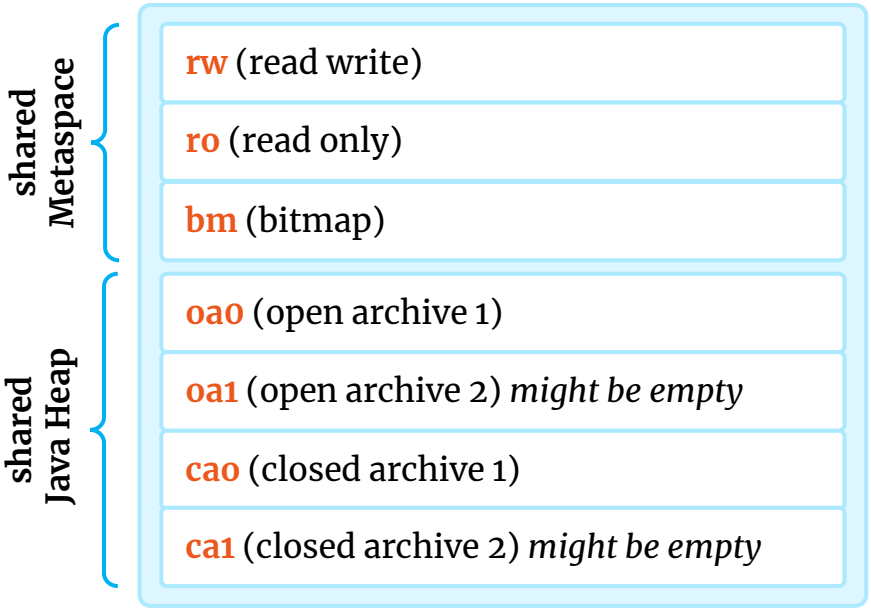
```
Linux> -XX:SharedArchiveFile=<static_archive>:<dynamic_archive>

Windows> -XX:SharedArchiveFile=<static_archive>\;<dynamic_archive>
```

Note: HotSpot VM does not allow more than two archives

Few CDS, App/Dynamic CDS enhancements in HotSpot

1.5	CDS was introduced (only C1 JIT and Serial GC)
9	CDS was extended to C2 JIT and other GCs (e.g., Parallel, ParallelOld, G1) but had limited shared Strings support (only with G1 GC for non-Win)
10	**JEP 310**: AppCDS
12	**JEP 341**: Default CDS Archives (generated at JDK build time)
13	**JEP 350**: Dynamic CDS Archives
15	**JDK-8232081**: Try to link all classes during dynamic CDS dump (i.e., not linked)
	JDK-8198698: Support Lambda proxy classes in dynamic CDS archive
	JEP 377: ZGC (production-ready) supports CDS
16	**JDK-8247666**: Support Lambda proxy classes in static CDS archive
17	**JDK-8261090**: Store old class files in static CDS archive
18	**JDK-8272331**: Automatically generate the CDS archive if necessary
19	**JDK-8261455**: Automatically generate the CDS archive if necessary

CDS Structure (7 regions)



Examples

rw – vtables

ro – SymbolTable, StringTable, SystemDictionary

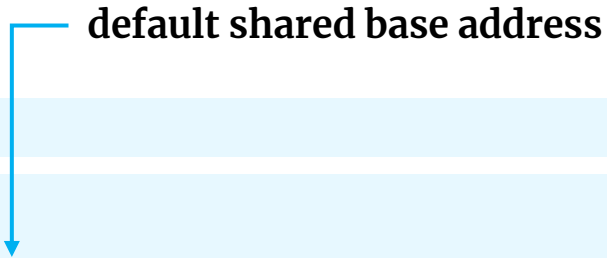
bm – bitmap that marks locations of all pointers across different regions within the archive

oa0, oa1 – java basic types (e.g., Boolean, Char, Float, etc.), Klass* objects (e.g., Instance*Klass*, TypeArrayKlass*, ObjArrayKlass*)

ca0, ca1 – interned strings

Archive is mapped at the default shared base address **0x800000000**
Address Space Layout Randomization (**ASLR**) might impact this

-XX:SharedBaseAddress=<new_address> overrides the default shared base address or use *-XX:SharedBaseAddress=0* to map it at an OS selected address




```
$ java -Xlog:cds -XX:ArchiveClassesAtExit=cds.jsa ...
```

```
[info][cds] Dumping shared data to file:
[info][cds] cds.jsa
[info][cds] Shared file region (rw ) 0: 8096560  bytes, addr 0x0000000800000000 offset 0x00001000 crc 0x60844844
[info][cds] Shared file region (ro ) 1: 13020552 bytes, addr 0x00000008007b9000 offset 0x007ba000 crc 0x01e83206
[info][cds] Shared file region (bm ) 2: 381560   bytes, addr 0x0000000000000000 offset 0x01425000 crc 0xdc5dfa66
[info][cds] Shared file region (ca0) 3: 925696   bytes, addr 0x00000007bfc00000 offset 0x01483000 crc 0xe5fe2616
[info][cds] Shared file region (oa0) 5: 724992   bytes, addr 0x00000007bf800000 offset 0x01565000 crc 0xf7881f99
```

Demo Time

@See https://github.com/ionutbalosin/faster-jvm-start-up-techniques

#section: App/Dynamic Class Data Sharing (CDS) in HotSpot JVM

Scenario: measure start-up time to first request[1] for the Spring PetClinit app

Trial run	default (elapsed in sec)	dynamic CDS (elapsed in sec)
1	7.167	5.907
2	7.165	5.968
3	7.034	5.689
4	6.867	5.764

hint: lower is better

Configuration
OpenJDK 19
Ubuntu 22.04.1 LTS / 5.15.0-47-generic
Intel i7-8550U Kaby Lake R
32GB DDR4 2400 MHz

[1] - accurate way to reflect how long a framework needs to start (i.e., avoiding lazy initialization techniques)

Summary

App/Dynamic CDS brings **noticeable start-up improvements** (if properly created)

Dynamic CDS archives **should be created after a broader usage of the application** (covering different use cases) and not by just starting and immediate stopping the application (i.e., classes are lazily loaded)

The more recent JDK version to use the better

Note: *App/Dynamic CDS also reduces the memory footprint if the same cache is shared across multiple JVMs (i.e., process resident set size) – not covered by this presentation*

Constraints

Running the CDS archive with a different JDK version than it was created with **does not work** (i.e. upgrading the JDK without regenerating the archive) - fixed in JDK 18 (JDK-8272331)

CDS archive is **not cross-platform reusable** (e.g., Linux, Windows, macOS)

Running the CDS archive with a modified jar timestamp that it was created with **does not work** (i.e., dynamic archive is disabled, just the base layer archive is used)

App/Dynamic CDS **omits all the jars referred** by other jars **as "c*lass-path*" attributes**

CDS archive **does not support pre JDK 5/6 classes** (JDK-8202556, JDK-8230413)

Shared-Class-Cache (SCC) in Eclipse OpenJ9

03

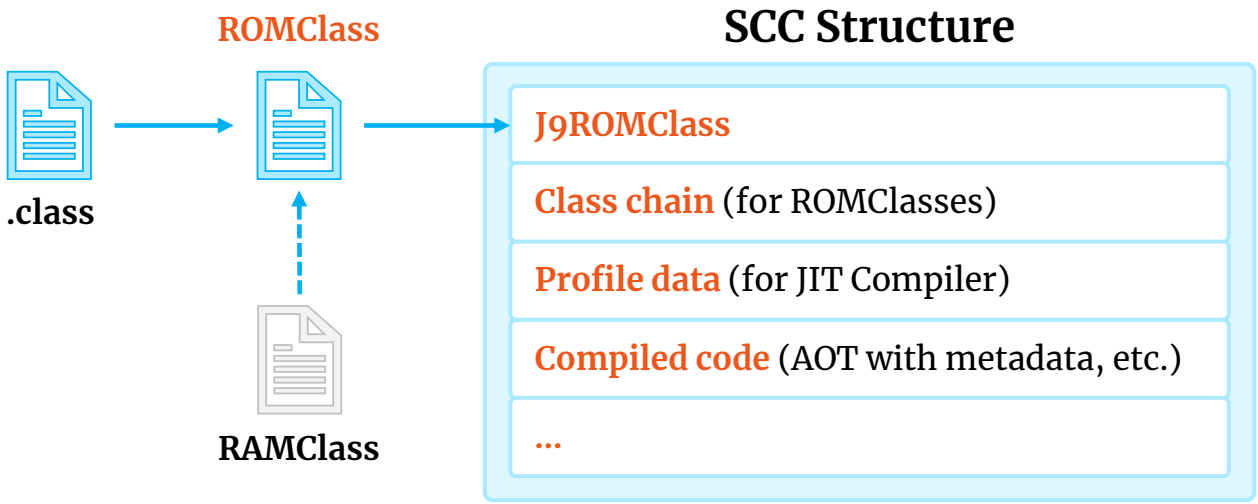
Shared Class Cache (SCC) is a memory-mapped file that stores mainly Java classes (e.g., ROMClass), profile data, compiled machine code (e.g., AOT), etc.

Introduced in 2007 (IBM SDK for Java 6)

Custom class loaders

- do not have class sharing support unless they extend *java.net.URLClassLoader*
- otherwise, helper APIs (e.g., *com.ibm.oti.shared* package) are provided

Every time a new class (not part of the cache) is loaded, it is dumped into the SCC



ROMClass - pointers to interfaces, superclass, inner classes, etc.
RAMClass - vtable, itable, Constant Pool, etc.

Only the **ROMClasses** are stored in SCC, **RAMClasses** are in the local memory of each JVM

Demo Time

@See https://github.com/ionutbalosin/faster-jvm-start-up-techniques

#section: Shared Classes Cache (SCC) and Dynamic AOT in Eclipse OpenJ9 JVM

Scenario: measure start-up time to first request[1] for the Spring PetClinit app

Trial run	no SCC (elapsed in sec)	SCC (elapsed in sec)
1	7.167	4.486
2	7.165	4.477
3	7.034	4.438
4	6.867	4.370

hint: lower is better

Configuration
OpenJ9 0.32.0 / OpenJDK 18.0.1
Ubuntu 22.04.1 LTS / 5.15.0-47-generic
Intel i7-8550U Kaby Lake R
32GB DDR4 2400 MHz

[1] - accurate way to reflect how long a framework needs to start (i.e., avoiding lazy initialization techniques)

Summary

SCC (combined with AOT) offers **great start-up performance improvement**

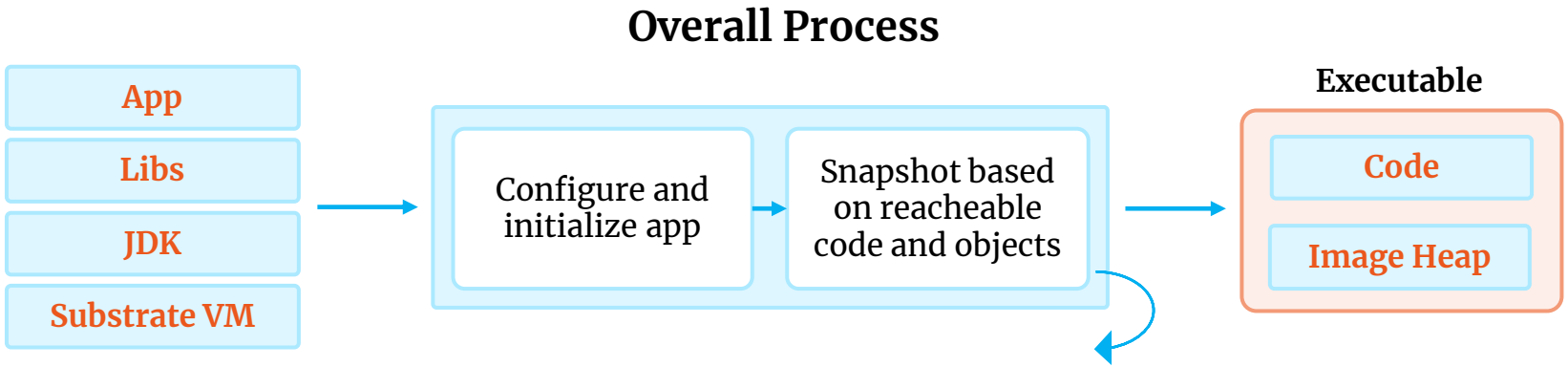
Highly customizable via command-line options

Note: *SCC also reduces the memory footprint if the same cache is shared across multiple JVMs (i.e., process resident set size) - not covered by this presentation*

Ahead-of-Time Compilation with GraalVM native-image

03

GraalVM native image compiles Java bytecode and generates native machine executables **Ahead-of-Time** (i.e., at build time). These executables start up almost instantly



- based on an iterative approach (intertwined by points-to analysis) parts of an application are run at build time, objects are allocated and snapshots are created
- **points-to analysis** results are used to AOT compile the reachable parts of an application

Source [“Initialize Once, Start Fast: Application Initialization at Build Time“ paper]

Demo Time

@See https://github.com/ionutbalosin/faster-jvm-start-up-techniques

#section: Ahead-of-Time (AOT) with GraalVM native-image

Scenario: measure start-up time to first request for the Spring PetClinit app

Trial run	Elapsed in sec
1	0.197
2	0.194
3	0.190
4	0.195

hint: lower is better

Configuration
GraalVM CE 22.0.0.2 native image
Ubuntu 22.04.1 LTS / 5.15.0-47-generic
Intel i7-8550U Kaby Lake R
32GB DDR4 2400 MHz

Summary

Use AOT with GraalVM native-image when start-up time is a **raw concern**

Dependencies

- **Java framework must have support for GraalVM native-image** (most of them already have such; e.g., *Quarkus*, *Spring*, *Micronaut*, *Helidon*, etc.)
- additionally, the **external dependencies must also be prepared for AOT** (quite a challenge at the moment)

Note: *replacing HotSpot VM with GraalVM native-image for long-running applications (e.g., microservices) is not the topic of this presentation*

Limitations

Some features require **additional configuration**, otherwise, a **fallback image** is generated (that launches the Java HotSpot VM)

- e.g., dynamic class loading, reflection, dynamic proxy, JNI , etc.

Some features are **not yet supported** with the closed-world optimization, and if used, lead to a fallback image

- e.g., invokedynamic, Security Manager, etc.

Features that **may operate differently** in the native image

- e.g., signal handlers, class initializers, finalizers, unsafe memory access, debugging, and, monitoring, etc.

See: [https://www.graalvm.org/$version/reference-manual/native-image/Limitations/]

Thank You

References

App/Dynamic Class Data Sharing In HotSpot JVM (Ionut Balosin)

Faster JMV Start-up Techniques (Ionut Balosin)

OpenJDK sources

OpenJ9 sources

Class data sharing in the HotSpot VM (Volker Simonis)

cl4cds (Volker Simonis)

Building Class Data Sharing Archives with Apache Maven (Gunnar Morling)

AppCDS for Spring Boot applications: first contact (Vladimir Plizga)

Startup Challenges (Claes Redestad)

Heap Archiving (Claes Redestad)

Java Ahead-of-Time Compilation with Oracle GraalVM (Christian Wimmer)

Improving GraalVM Native Image (Christian Wimmer)

It's always sunny with OpenJ9 (Dan Heidinga)

Optimize JVM start-up with Eclipse OpenJ9 (Marius Pirvu)

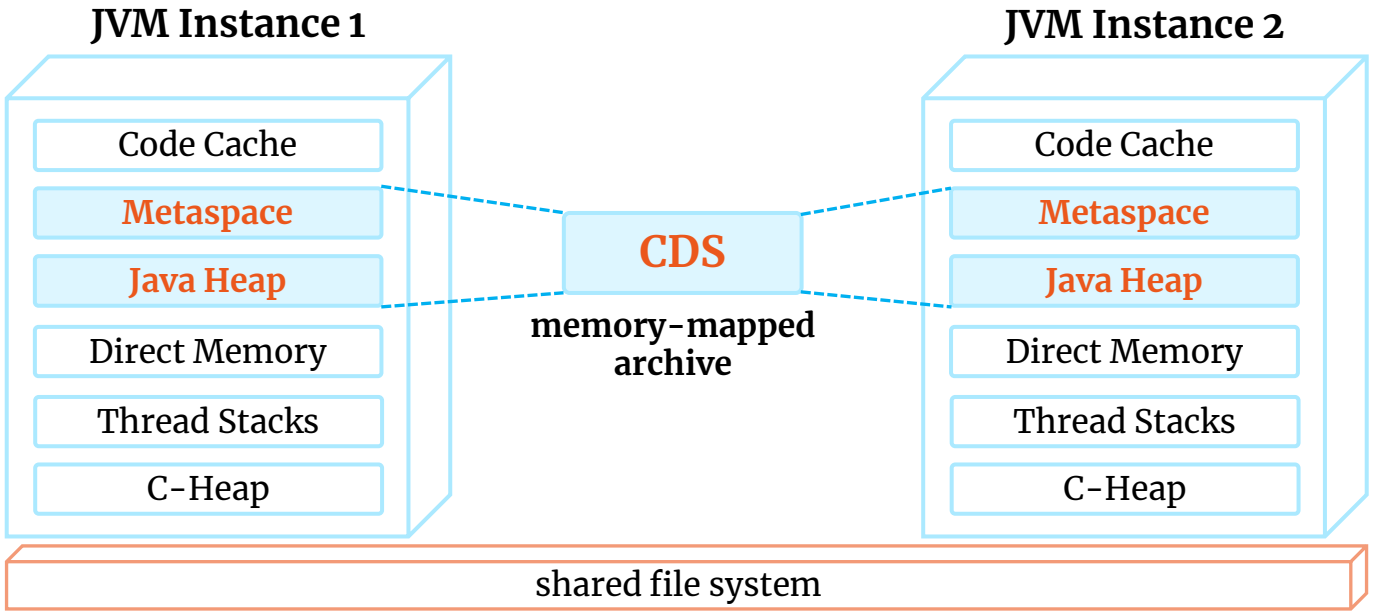
The multi-layer shared class cache for Docker (Younes Manton)

JIT and AOT in the JVM (Mark Stoodley)

Class sharing in Eclipse OpenJ9 (Ben Corrie, Hang Shao)

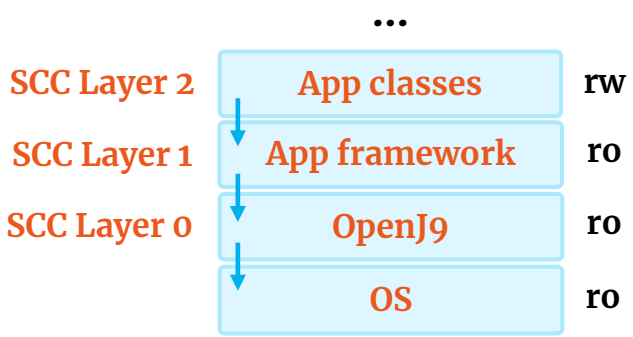
Appendix

CDS is memory-mapped at runtime and shared between multiple JVM instances (or Docker containers) on the same host (with a shared file system)



__Note:__ JVM #1 and JVM #2 also share the read-only parts of loaded libraries (dynamic linking; e.g., libjvm.so, libjava.so, etc.)

Multi-layer SCC – different (hierarchical) Docker images could be created with custom SCC layers on top of the others



Introduced in 2019 (Eclipse OpenJ9 0.17.0)

Characteristics

- each SCC layer is a separate file on disk
- every time a container starts it creates a new layer (*copy-on-write*). All below layers are untouched (*read-only*)
- each layer could independently size

A single (default) cache is equivalent to multi-layer SCC with layer number 0

App CDS in HotSpot JVM

```
$ java -Xshare:off -XX:DumpLoadedClassList=app-cds.lst ...
$ java -Xshare:dump -XX:SharedClassListFile=app-cds.lst -XX:SharedArchiveFile=app-cds.jsa ...
$ java -XX:SharedArchiveFile=app-cds.jsa ...
```

Dynamic CDS in HotSpot JVM

```
$ java -XX:ArchiveClassesAtExit=dynamic-cds.jsa ...
$ java -XX:SharedArchiveFile=dynamic-cds.jsa ...

$ java -XX:SharedArchiveFile=app-cds.jsa -XX:ArchiveClassesAtExit=dynamic-cds.jsa ...

$ java -XX:SharedArchiveFile=app-cds.jsa:dynamic-cds.jsa
```

SCC and Dynamic AOT in Eclipse OpenJ9 JVM

```
$ java -Xshareclasses:name=scc,cacheDir=. -Xscmx96m -XX:SharedCacheHardLimit=192m -Xquickstart ...
```

Source [https://github.com/ionutbalosin/faster-jvm-start-up-techniques]