Ahead of Time Compilation

HotSpot Compiler offsite

Igor Veresov
Vladimir Kozlov
2018

Java
Your
Next
(Cloud)

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.

AOT motivations

- Needed for longer term strategy of supporting Future Java based JIT compiler.
- Provide faster startup for applications since hot methods and class Initializers will be readily available. Expected to be important for Cloud.
- Provide quicker time to peak performance. Statically generated code could be 1st pass in a multi-tiered compilation system.
- Support platforms where executing dynamically generated code is prohibited e.g. iOS.
- Density improvement - sharing AOT'd code, app dependent.

AOT key features

- AOT is Open feature.
- New JDK tool **jaotc** is added as part of java installation. **jaotc** is java static compiler which produces native code for compiled java methods.
- Hotspot JVM supports AOT by default (controlled by -XX:+UseAOT flag).
- AOT is only experimental feature. Though experimental, we'll work to timely address bugs or provide workaround.
- It is experimental in JDK 9 and 10 only on linux-x64 and running G1 or Parallel GC with and without Compressed oops.

AOT components

- JAOTC - Java static compiler tool. JEP 295: Ahead-of-Time Compilation.
- JVMCI - JVM compiler interface for Java based JIT compilers. JEP 243: Java-Level JVM Compiler Interface.
- Graal - java based java compiler. https://github.com/oracle/graal
- Hotspot JVM.

New java modules to provide AOT functionality

- **jdk.aot** - AOT compiler tool.
- **jdk.internal.vm.ci** - JVMCI module provides interface for Java based JIT compilers.
- **jdk.internal.vm.compiler** - clone of Graal-core Java based Java compiler.
- **jdk.aot** and **jdk.internal.vm.compiler** are built on MacOS, Linux and Win x64. The build is controlled by ‘configure’ flag —enable-aot{=yes|no|auto}
- **make/common/Modules.gmk** checks if these AOT modules should be build.
- **make/CompileJavaModules.gmk** specify flags to build module.
- **make/autoconf/hotspot.m4** adds aot to JVM_FEATURES list which is checked in **make/hotspot/lib/JvmFeatures.gmk** to set -DINCLUDE_AOT

jdk.aot module

jdk.aot module is used by **jaotc** tool for AOT Java static compilation.

It has only 3 packages:

jdk.tools.jaotc

jdk.tools.jaotc.binformat

which are located in AOT repository:

src/jdk.aot/share/classes/

Hotspot AOT code

- Code which loads and verifies AOT libraries, verifies AOT classes, publish AOT code, iterates over oops and metadata is located in Hotspot repository:
  src/hotspot/share/aot/

- Code which calls into JVM runtime from AOT code to resolve classes, methods, Java strings is located in JVMCI runtime repository:
  src/hotspot/share/jvmci/compilerRuntime.*

- Code which uses JVMCI to extract compilation metadata (including classes fingerprint) for compiled code during AOT compilation is part of JVMCI runtime code:
  src/hotspot/share/jvmci/jvmciCodeInstaller.*

AOT functionality overview

New JDK tool **jaotc** is used for AOT compilation. It uses Graal-core as the code-generating backend to produce AOT shared libraries in ELF format.

To use **jaotc** user have to specify .class, .jar files or java module name as input and resulting AOT library name as output (unnamed.so is used if name is not specified):

```
jaotc --output libHelloWorld.so HelloWorld.class
jaotc --output libjava.base.so --module java.base
```

User can specify which methods to compile or exclude with —compile-commands flag

```
jaotc --output libjava.base.so —compile-commands base.txt —module java.base
```

The command file can have 2 commands: *exclude or compileOnly*

AOT functionality overview (cont.)

During JVM startup the AOT initialization code looks for well-known shared libraries in a well-known location ($JAVA_HOME/lib) or libraries specified by -XX:AOTLibrary option. If shared libraries are found, these are loaded and used. If no shared libraries can be found, AOT will be turned off for this JVM instance run.

```
java -XX:AOTLibrary=./libHelloWorld.so,./libjava.base.so HelloWorld
```

The same JVM version and the same java runtime configuration should be used during AOT compilation:

```
jaotc -J-XX:+UseParallelGC -J-XX:-UseCompressedOops --output libHelloWorld.so HelloWorld.class
java -XX:+UseParallelGC -XX:-UseCompressedOops -XX:AOTLibrary=./libHelloWorld.so HelloWorld
```

The runtime configuration is recorded in AOT library and verified when the library is loaded during execution. If verification failed this AOT library will not be used and JVM will continue run or exit if flag -XX:+UseAOTStrictLoading is specified.

AOT functionality overview (cont.)

The set of AOT libraries could be generated for different execution environment. JVM knows next *well-known* names for AOT libraries generated for specific runtime configuration. It will look for them in $JAVA_HOME/lib directory and load the one which correspond to current run-time configuration:

```
-XX:-UseCompressedOops -XX:+UseG1GC :         libjava.base.so
-XX:+UseCompressedOops -XX:+UseG1GC :         libjava.base-coop.so
-XX:-UseCompressedOops -XX:+UseParallelGC :   libjava.base-nong1.so
-XX:+UseCompressedOops -XX:+UseParallelGC : libjava.base-coop-nong1.so
```

JVM knows AOT libraries names for next Java modules:

```
java.base,  jdk.compiler,  jdk.scripting.nashorn,  jdk.internal.vm.ci,  jdk.internal.vm.compiler
```

Note, user himself have to compile and install into $JAVA_HOME/lib directory *well-known* AOT libraries.

AOT functionality overview (cont.)

AOT-compiled code in AOT libraries is treated by JVM as **extension of existing CodeCache**. When a Java class is loaded JVM looks if corresponding AOT-compiled methods exist in loaded AOT libraries and add links to them from java methods descriptors. AOT-compiled code follows the same invocation/ deoptimization/unloading rules as normal JIT-compiled code.

src/hotspot/share/aot/aotLoader.*
src/hotspot/share/aot/aotCodeHeap.*

AOT functionality overview (cont.)

Since class bytecodes can change over time, either through changes to the source code or via class transformation and redefinition, the JVM needs to detect such changes and reject corresponding AOT-compiled code. This is achieved with ***class fingerprinting***. During AOT compilation a fingerprint for each class is generated and stored in AOT library. During execution, when a class is loaded and AOT-compiled code is found for this class, the fingerprint for the class is compared to the one stored in AOT library. If there is a mismatch then the AOT code for that particular class is not used.

src/hotspot/share/classfile/classFileStream.cpp
src/hotspot/share/oops/instanceKlass.*

JAOTC tool flags

src/jdk.aot/share/classes/jdk.tools.jaotc/src/jdk/tools/jaotc/Options.java

jaotc tool is part of java installation and can be used the same way as **javac**:

```
jaotc <options> <name or list>
```

Where name is class name or jar file. And list is a '**:**' separated list of class names, modules, jar files or directories which contain class files.

The following **jaotc** options are available:

--output <file>	Output file name. Default name is “unnamed.so".
--class-name <class names>	List of Java classes to compile
--jar <jar files>	List of jar files to compile
--module <modules>	List of Java modules to compile

JAOTC tool flags (cont.)

--directory <dirs>	List of directories where to search for files to compile
--search-path <dirs>	List of directories where to search for specified files
--compile-commands <file>	Name of file with compile commands:

```
exclude sun.util.resources..*.getContents\(\)\[\[Ljava/lang/Object;
exclude sun.security.ssl.*
compileOnly java.lang.String.*
```

AOT recognizes two compile commands currently:

```
exclude       - exclude compilation of specified methods
compileOnly   - compile only specified methods
```

Regular expressions are used to specify classes and methods.

JAOTC tool flags (cont.)

--compile-for-tiered
Generated profiling code for tiered compilation. By default profiling code is not generated (JDK 9, could be changed in future).

--compile-with-assertions
Generate code with java assertions. By default assertions code is not generated.

--compile-threads <number>
Number of compilation threads to be used. Default value is **min(16, available_cpus)**.

JAOTC tool flags (cont.)

--ignore-errors

Ignores all exceptions thrown during class loading. By default exit compilation if class loading throws exception.

--exit-on-error

Exit on compilation errors. By default failed compilation is skipped and compilation of other methods continues.

JAOTC tool flags (cont.)

--info Print information about compilation phases
--verbose Print more details about compilation phases including phases time
--debug Print even more details, including dump of compiled code (requires hsdis-<arch>.so library)

--help Print jaotc usage message and flags
--version Print version information
-J<flag> Pass **flag** directly to the JVM runtime system

Hotspot JVM runtime AOT flags

-XX:+/-UseAOT Use AOT-compiled files. By default it is ON.

-XX:AOTLibrary=<file> Specify a list of AOT library files. Separate libraries entries with colons (:) or comma (,).

-XX:+/-PrintAOT Print used AOT classes and methods.

-XX:+/-UseAOTStrictLoading Exit JVM if any of AOT libraries has run-time configuration not matching current run-time settings. Diagnostic flag is available (requires to specify -XX:+UnlockDiagnosticVMOptions flag). By default it is OFF.

Hotspot JVM Unified Logging AOT tags

aot+class+fingerprint

Create log in case class fingerprint does not match fingerprint recorded in AOT library

aot+class+load

Create log when corresponding class data is found in AOT library.

aot+class+resolve

Create log when a request from AOT compiled code to resolve class was successful or not.

AOT potential issues

- Out of memory (OOM) during AOT compilation when compiling big Java methods (JDK-8153214, JDK-8175214). Workarounds:
  - increase java heap: -J-Xmx8g
  - decrease number of compiling threads: --compile-threads 1
- Failed to compile java method for different reasons:
  - Graal issues
  - JAOTC issues (for huge methods)
- AOT code problems - could be Graal issue or JAOTC
- Hotspot AOT runtime - class/method resolution, oops/metadata iterations

AOT compilation phases

AOT compilation has next phases (main code: hotspot/src/jdk.aot/share/classes/jdk.tools.jaotc/src/jdk/tools/jaotc/Main.java):

- Parse flags
- Collect Java classes to compile based on flags
- Collect Java methods to compile based on collected classes and restrictions specified in —compile-commands file.
- Use Graal to compile collected classes.
- Record runtime configuration

AOT compilation phases (cont.)

- Parse compiled code:
  - record location of calls in compiled code
  - copy compiled code into code buffer and write call stubs after code.
  - replace calls destinations with calls to call stubs (trampolines) which will load destination address from RW section and jump there.

src/jdk.aot/share/classes/jdk.tools.jaotc/src/jdk/tools/jaotc/CodeSectionProcessor.java
src/jdk.aot/share/classes/jdk.tools.jaotc/src/jdk/tools/jaotc/amd64/AMD64ELFMacroAssembler.java

AOT compilation phases (cont.)

- Process compiled code metadata:
  - allocate memory for klass and method pointers (GOT cells) referenced in code
  - record location of calls, constant and oops (Strings) references
  - call into runtime through JVMCI to get class fingerprint and compiled metadata and store it in metadata sections

src/jdk.aot/share/classes/jdk.tools.jaotc/src/jdk/tools/jaotc/DataPatchProcessor.java

- Create AOT header section which contains general information about compilation.

AOT compilation phases (cont.)

- Create .o object file:
  - convert buffers with code and collected data into ELF sections
  - record referenced symbols
  - record relocation data for loader

- Generate shared AOT library using linked:

  ld -shared -z noexecstack -o lib.so lib.o

AOT specifics in Graal

- Graal changes

AOT library sections

AOT ELF sections have global names assigned to them to get their addresses during execution by using dlsym(). Sections global names contains 'JVM' as prefix of section name (see definitions in hotspot/src/share/vm/aot/aotCodeHeap.hpp):

JVM.header
AOTHeader - AOT data format version (1.0), counts of classes and methods, important arrays sizes, JVM version string.

JVM.config
AOTConfiguration - Runtime configuration: GC used, product or debug JVM, etc.

JVM.metaspace.names
Names of classes, methods, stubs and Java constant strings in UTF-8 format:
<u2_size>Ljava/lang/ThreadGroup;<u2_size>addUnstarted<u2_size>()V

AOT library sections (cont.)

JVM.method.metadata
AOTMethodData - compiled code metadata: scopes PCS, relocations, etc (similar to JIT code data)

JVM.klasses.offsets
AOTKlassData - classes related offsets, fingerprint, ID. JVM access this data using Java class name with dlsym(). Class names of these structures are global symbols in AOT library.

JVM.methods.offsets
AOTMethodOffsets - AOT code related offsets in other sections (name, code, metadata)

JVM.klasses.dependencies
Arrays of dependent methods - used for class unloading

JVM.text - read-only section with AOT code.

AOT library sections (cont.)

JVM.metaspace.got

Array for resolved and initialized (2 separate slots - GOT cells) class pointers referenced in code. At the beginning they contain 0. Class pointer is stored when class is resolved. AOT code does lazy class resolution by calling JVM runtime if loaded value from GOT cell is 0.

JVM.metadata.got

Array for resolved classes and methods referenced in AOT code metadata/debuginfo.
At the beginning they contain pointers to UTF-8 strings for class or method names. Name pointer is replaced with class or method pointer after resolution.

JVM.oop.got

Array for Java strings referenced in AOT code. At the beginning they 0 which is replaced during execution with pointer to String object.

AOT library sections (cont.)

JVM.stubs.offsets
AOTMethodOffsets - AOT runtime stubs descriptors.

JVM.code.segments
Table for fast AOT code search based on PC.

JVM.method.state
Table of AOT code state (not_set, in_use, invalid) which is set during AOT publishing and deoptimization. AOT code itself check it to skip execution of invalidated code.

AOT library sections (cont.)

Available ELF sections in AOT library can be looked with readelf:

readelf -S -W libHelloWorld.so

```
[ 5] .text             PROGBITS        0000000000002380 002380 005a80 00  AX  0   0 128
[ 6] .metaspace.names  PROGBITS        0000000000007e00 007e00 0007ea 00   A  0   0 128
[ 7] .klasses.offsets  PROGBITS        0000000000008600 008600 000150 00   A  0   0 128
[ 8] .methods.offsets  PROGBITS        0000000000008780 008780 000034 00   A  0   0 128
```

readelf -p .header libHelloWorld.so

```
String dump of section '.header':
  [     8]  )
  [    1d]  09-internal+0-2017-03-06-132808.vnkozlov.030617hs
```

Note, we specifically recorded JVM version into AOT library to make sure the same JVM will be used during execution (otherwise library will be rejected).

More details are on wiki page:
https://wiki.se.oracle.com/display/JPG/The+design+of+the+Shared+Object+files+used+by+the+VM

Java™

ORACLE®