GRAALVM, AOT, JIT and Spring Native

Motivations

- Spring team introduced Spring Native in 2019
- served as a research bed for several different approaches
- Spring 3 has **GraalVM Native Image Support**

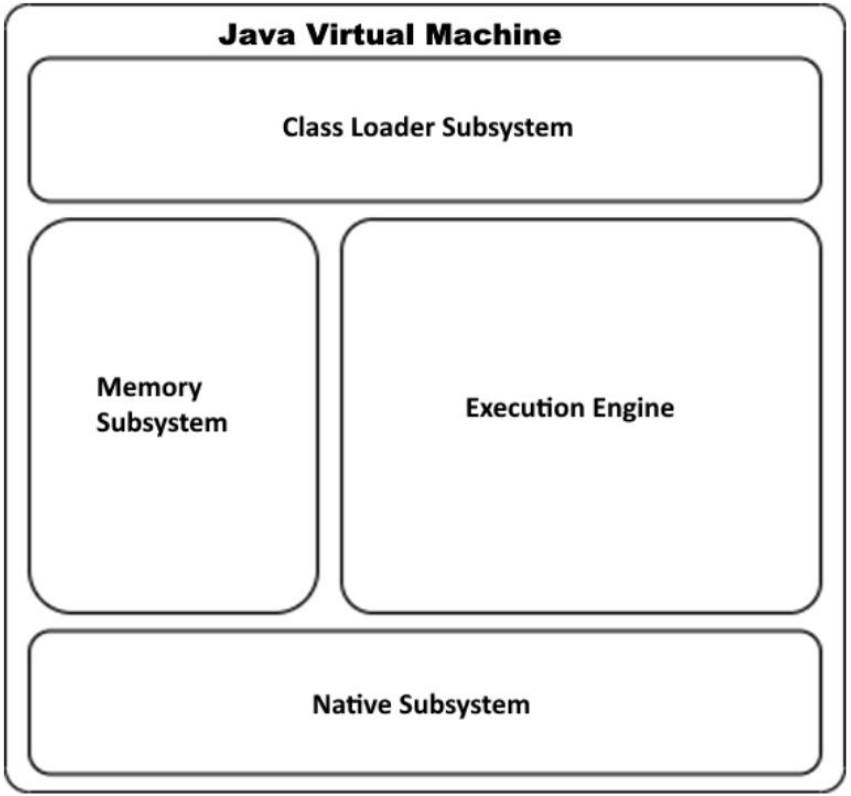
Why GraalVM?

- key requirements of a cloud-native application built on the microservice architecture:
  - Smaller footprint:
  - Quicker bootstrap
  - Polyglot and interoperability

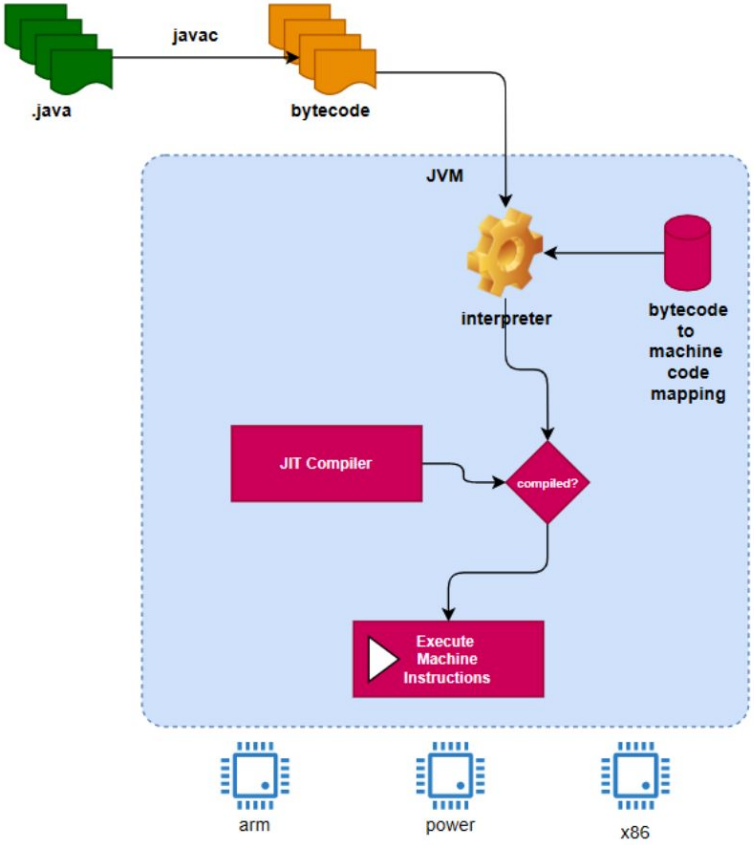
Graal Main functionalities

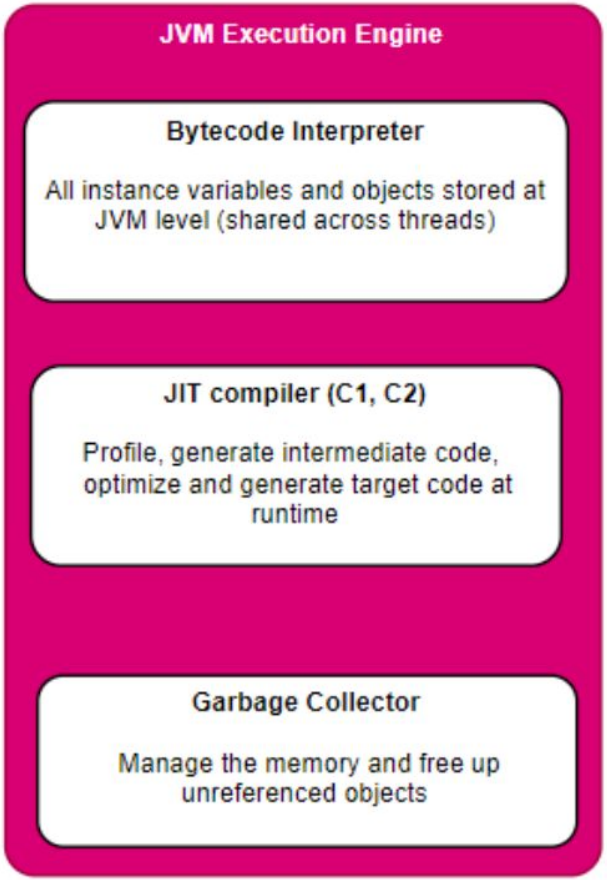
- Polyglot
- Just-in-time compilation
- Ahead-of-time compilation and native image

JIT- JVM architecture, 10000 feet view



JIT- JVM execution engine





JIT- C1 & C2

- JVM introduced two types of compilers, C1 (client) and C2 (server)
- **C1 compiler:**
  - counts the number of times a particular method/snippet of code is executed.
  - Once counter reaches a threshold, then that particular code snippet is compiled, optimized, and cached
- **C2 compiler:**
  - perform runtime code profiling
  - come up with code paths and hotspots.
  - also known as a hotspot.
- C1 is faster and good for short-running applications
- C2 is slower and heavy, ideal for long-running processes

JIT- Tiered Compilation

- Interpreted code (level 0)
- Simple C1 compiled code (level 1)
- Limited C1 compiled code (level 2)
- Full C1 compiled code (level 3)
- C2 compiled code (level 4)

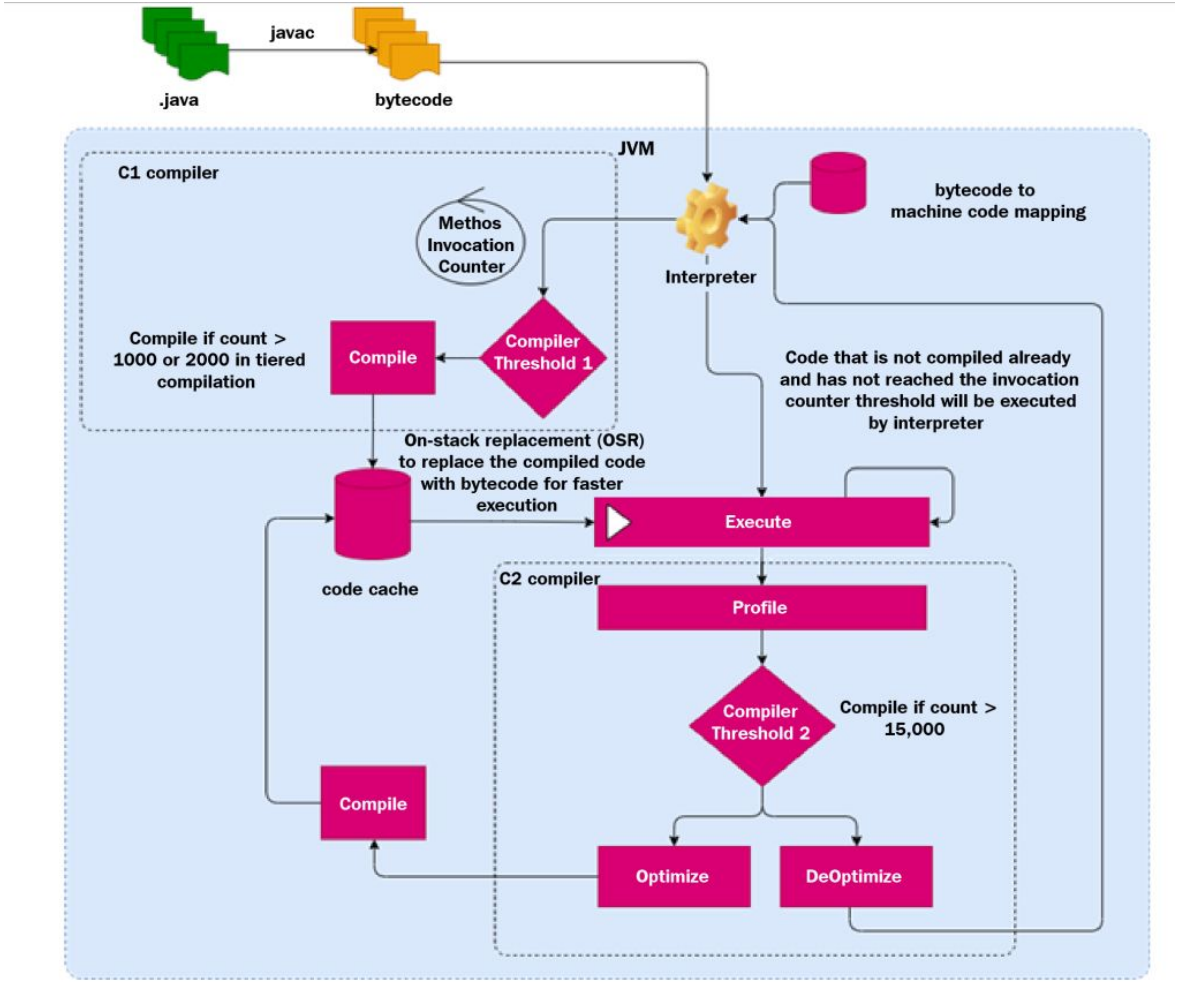
JIT- Code Optimizations

- **Inlining:**
  - Replaces invocation of hot methods with actual method body.
  - Like replacing variable names instead of getters and setters.
  - A hot method is detected based on its size and the number of invocations.
  - The size threshold used by JVM to decide inlining can be modified using the *-XX:MaxFreqInlineSize* flag (by default, it is 325 bytes)

JIT- Code Optimizations

- **Escape Analysis:**
  - Does a variable ever escape a local scope?
  - Local variables don’t need lock
  - Local variables are moved from heap to stack and no need to be garbage collected.
- **Deoptimization:**
  - Removing a compiled code from code cache
- **Monomorphic, bimorphic, and megamorphic dispatch**
  - Replaces child instance of a parent class
- **Loop optimization – Loop unrolling**

JIT- C2 PGO


javac
.java
bytecode
JVM
C1 compiler
Methos Invocation Counter
Interpreter
bytecode to machine code mapping
Compile if count > 1000 or 2000 in tiered compilation
Compile
Compiler Threshold 1
Code that is not compiled already and has not reached the invocation counter threshold will be executed by interpreter
On-stack replacement (OSR) to replace the compiled code with bytecode for faster execution
code cache
Execute
C2 compiler
Profile
Compiler Threshold 2
Compile if count > 15,000
Compile
Optimize
DeOptimize

JIT- Code cache settings

- **-XX:InitialCodeCacheSize**: The initial size of the code cache. The default size is 160 KB (varies based on the JVM version).
- **-XX:ReservedCodeCacheSize**: The maximum size the code cache can grow to. The default size is 32/48 MB. When the code cache reaches this limit, JVM will throw a warning: CodeCache is full. Compiler has been Disabled.
- **-XX:CodeCacheExpansionSize**: This is the expansion size when it scales up. Its default value is 32/64 KB.
- **-XX:+PrintCodeCache**: This option can be used to monitor the usage of the
- code cache.

JIT- XX:+PrintCompilation

```
<Timestamp> <CompilationID> <Flag> <Tier>
<ClassName::MethodName> <MethodSize> <DeOptimization Performed
if any>
```

JIT- XX:+PrintCompilation flags

- **On-Stack Replacement**: This is represented by the % character.
- **Exception Handler**: This is represented by the ! character. This indicates that the method has an exception handler.
- **Synchronized method**: This is represented by the s character. This indicates that the method is synchronized.
- **Blocking Mode**: This is represented by the b character. This means that the compilation did not happen in the background.
- **Native**: This is represented by the n character. This indicates that the code is compiled to the native method.

JIT- C2 shortcomings

- Written in C++
- No garbage collection
- Very complicated code

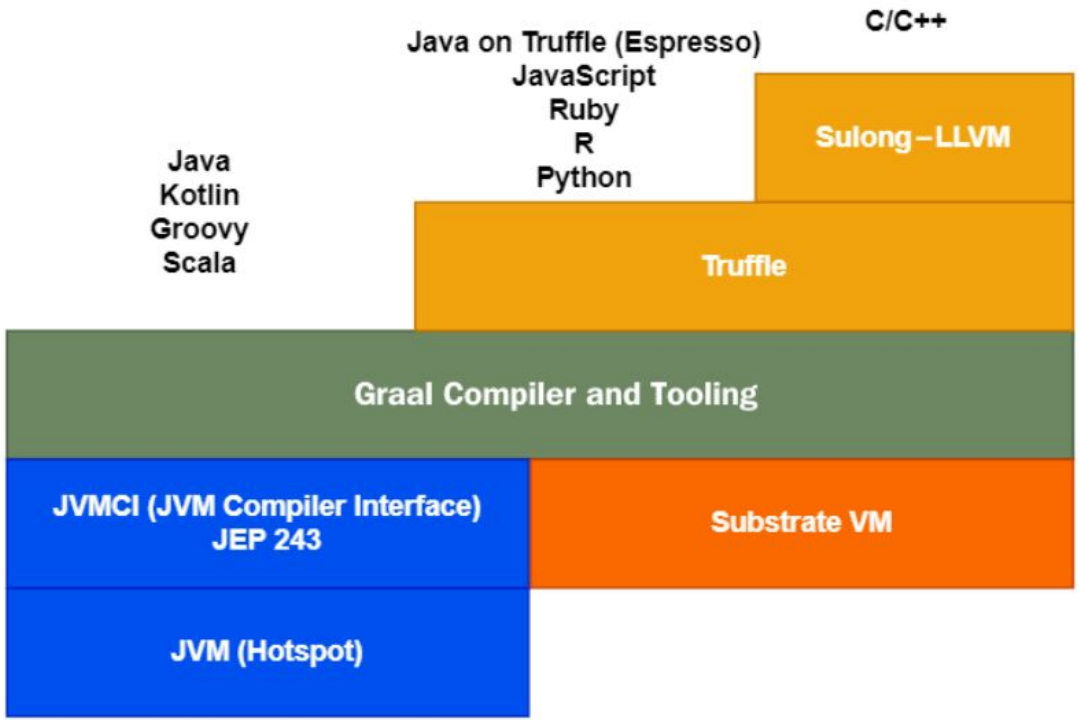
JVMCI

- JVMCI provides the API that is required to implement custom compilers and configure JVM to call these custom compiler implementations.
- The JVMCI API provides the following capabilities:
  - Access to VM data structures, which is required to optimize the code
  - Managing the compiled code following optimization and deoptimization
  - Callbacks from JVM to execute the compilation at runtime

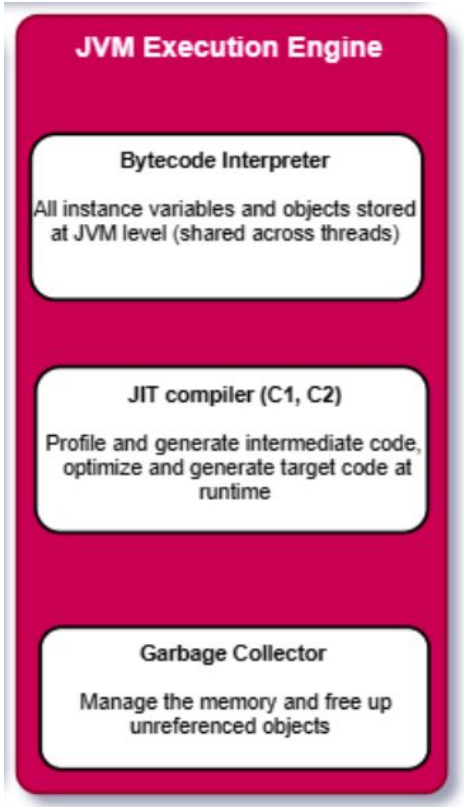
Graal JIT

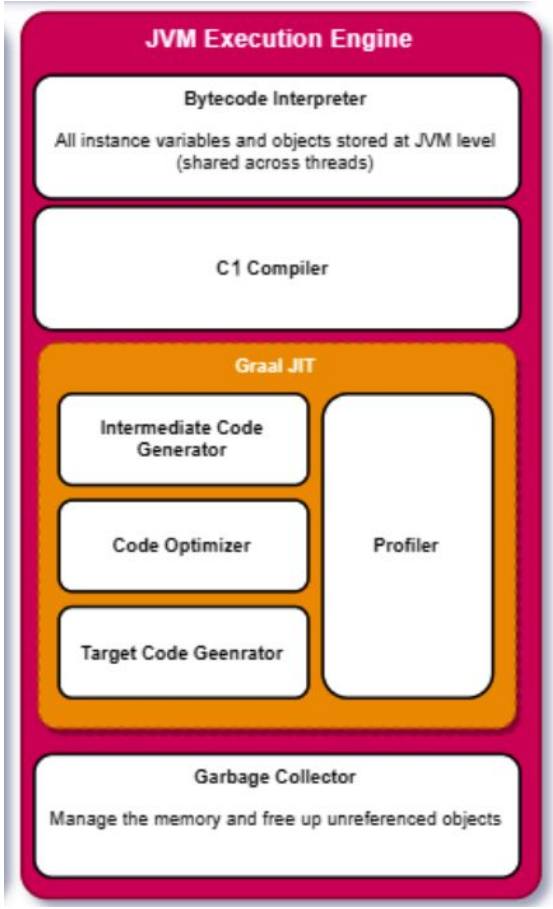
JVM JIT (C2)	Graal JIT
Implemented in C/C++ and has matured and hardened over time.	Implemented in Java, still in its infancy.
Production grade.	Production grade since 19.0.0.
Complex code and has become tougher to maintain. This has reached its end of life.	Highly modular and brings in all the sophisticated modern Java programming language features. Easy to enhance, maintain, and manage.
The focus is to optimize Java source code.	This focus of Graal is much more than just JIT compilation. Targeting the cloud-native programming requirements of Polyglot; a smaller footprint, faster build, and run. Graal does not only focus on JVM-based (Scala, Kotlin, and so on) languages, but also on dynamic languages (Ruby, Python, JavaScript, and suchlike) as well as native languages (C/C++).

Graal architecture



Graal execution engine

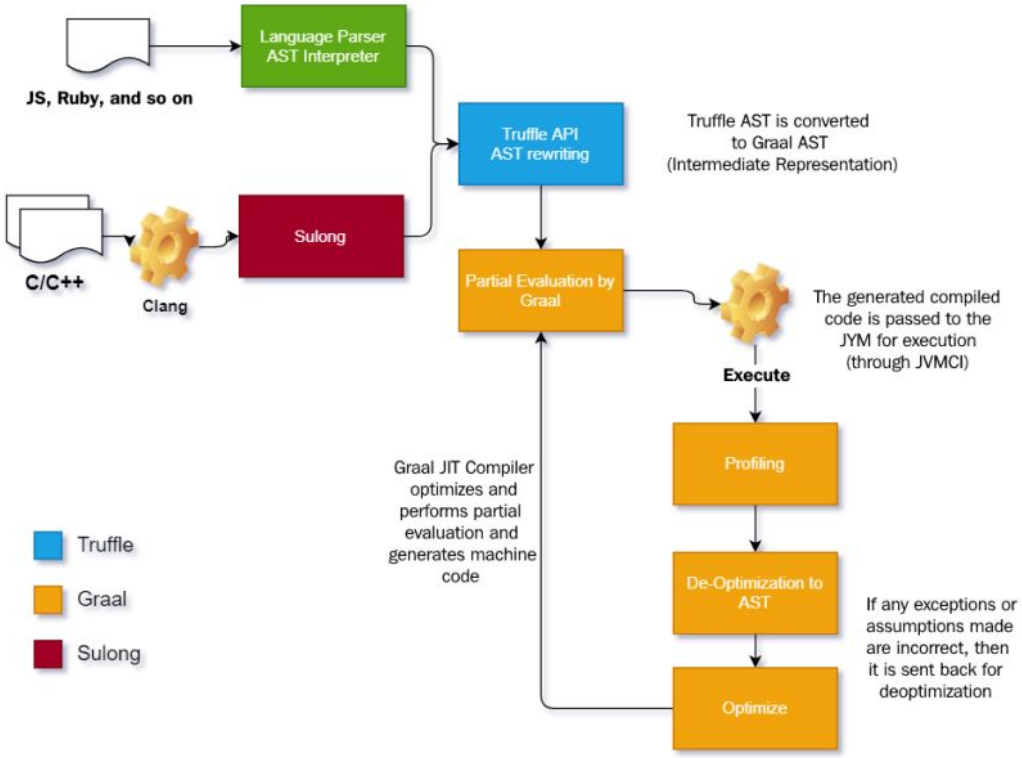




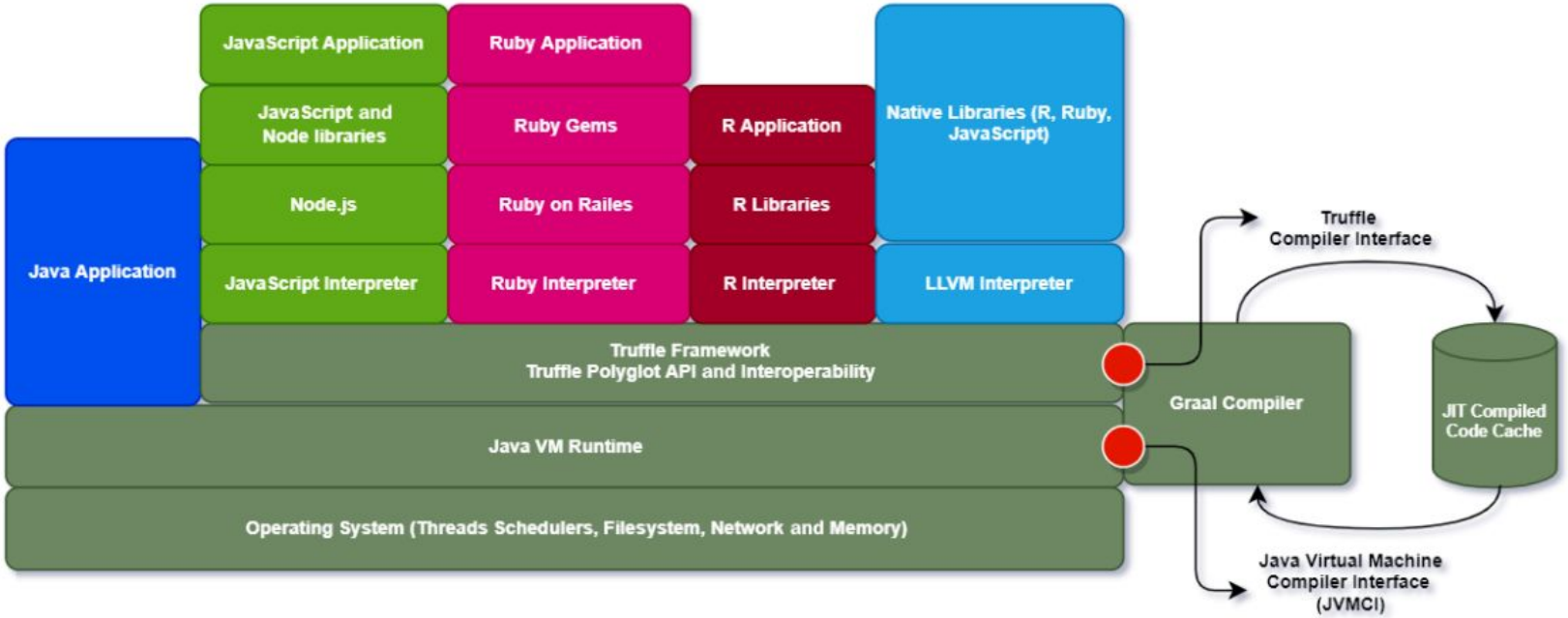
Graal JIT

- Graal uses AST as intermediate representation
- This abstracts language-specific syntax and semantics from the logic of optimizing the code
- This approach makes GraalVM capable of optimizing and running code written in any language

Graal JIT


Language Parser
AST Interpreter
JS, Ruby, and so on
Truffle API
AST rewriting
Truffle AST is converted
to Graal AST
(Intermediate Representation)
Sulong
C/C++
Clang
Partial Evaluation by
Graal
The generated compiled
code is passed to the
JYM for execution
(through JVMCI)
Execute
Profiling
Graal JIT Compiler
optimizes and
performs partial
evaluation and
generates machine
code
De-Optimization to
AST
If any exceptions or
assumptions made
are incorrect, then
it is sent back for
deoptimization
Optimize
Truffle
Graal
Sulong

Graal JIT


JavaScript Application
Ruby Application
JavaScript and Node libraries
Ruby Gems
R Application
Native Libraries (R, Ruby, JavaScript)
Java Application
Node.js
Ruby on Rails
R Libraries
JavaScript Interpreter
Ruby Interpreter
R Interpreter
LLVM Interpreter
Truffle Framework
Truffle Polyglot API and Interoperability
Graal Compiler
Truffle Compiler Interface
JIT Compiled Code Cache
Java VM Runtime
Operating System (Threads Schedulers, Filesystem, Network and Memory)
Java Virtual Machine Compiler Interface (JVMCI)

Let’s test Graal JIT

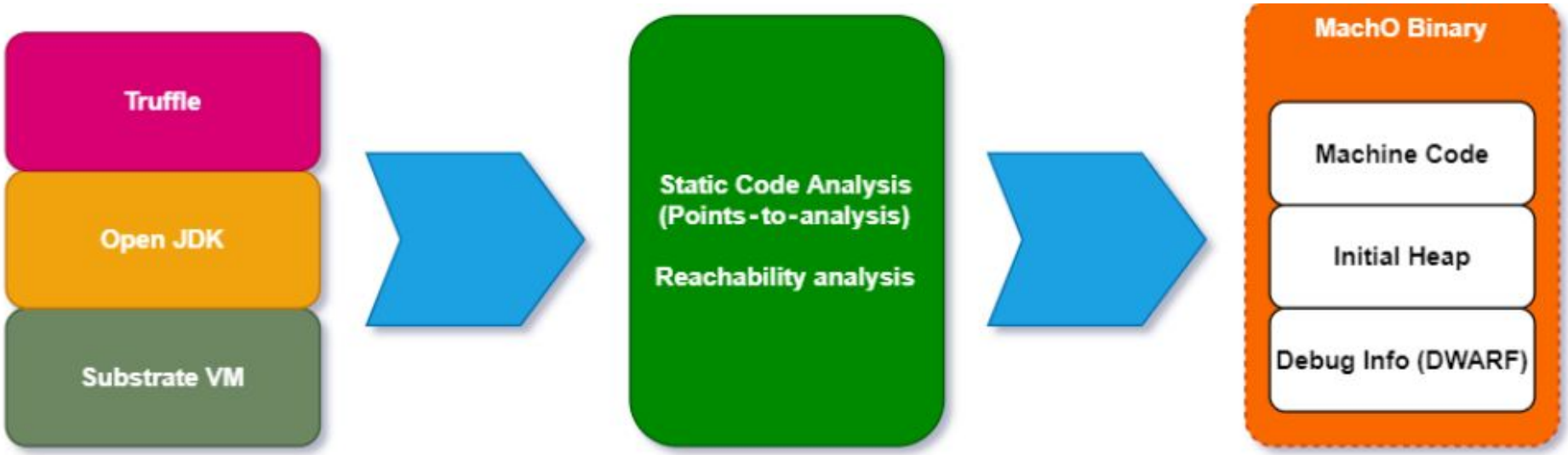
Graal JIT- partial escape analysis

```
public void method(boolean flag) {
    Class1 object1 = new Class1();

    Class2 object2 = new Class2();
    //some processing
    object1.parameter = value;
    //some more logic
    if(flag) {
        return object1;
    }
    return object2;
}
```

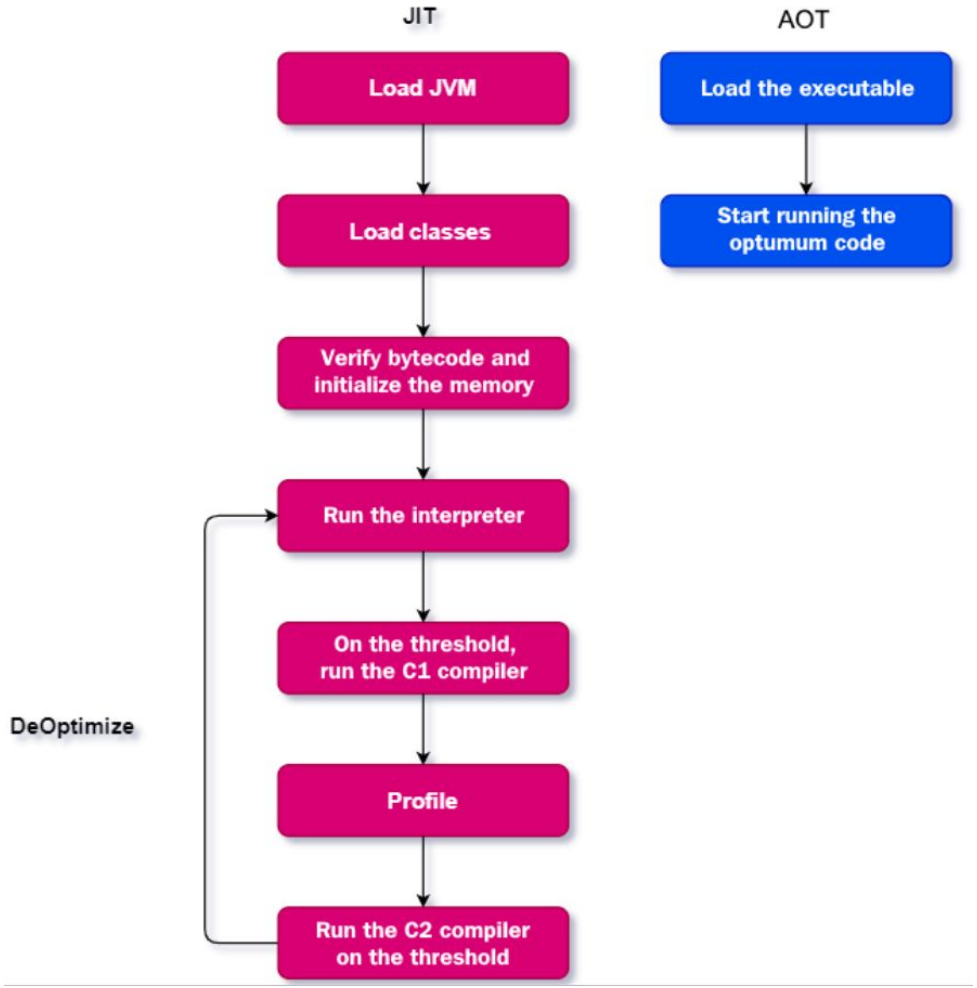
```
public void method(boolean flag) {

    Class2 object2 = new Class2();

    tempValue = value;

    if(flag) {
        Class1 object1 = new Class1();
        object1.parameter = tempValue;
            return object1;
    }
    return object2;
}
```

Graal AOT

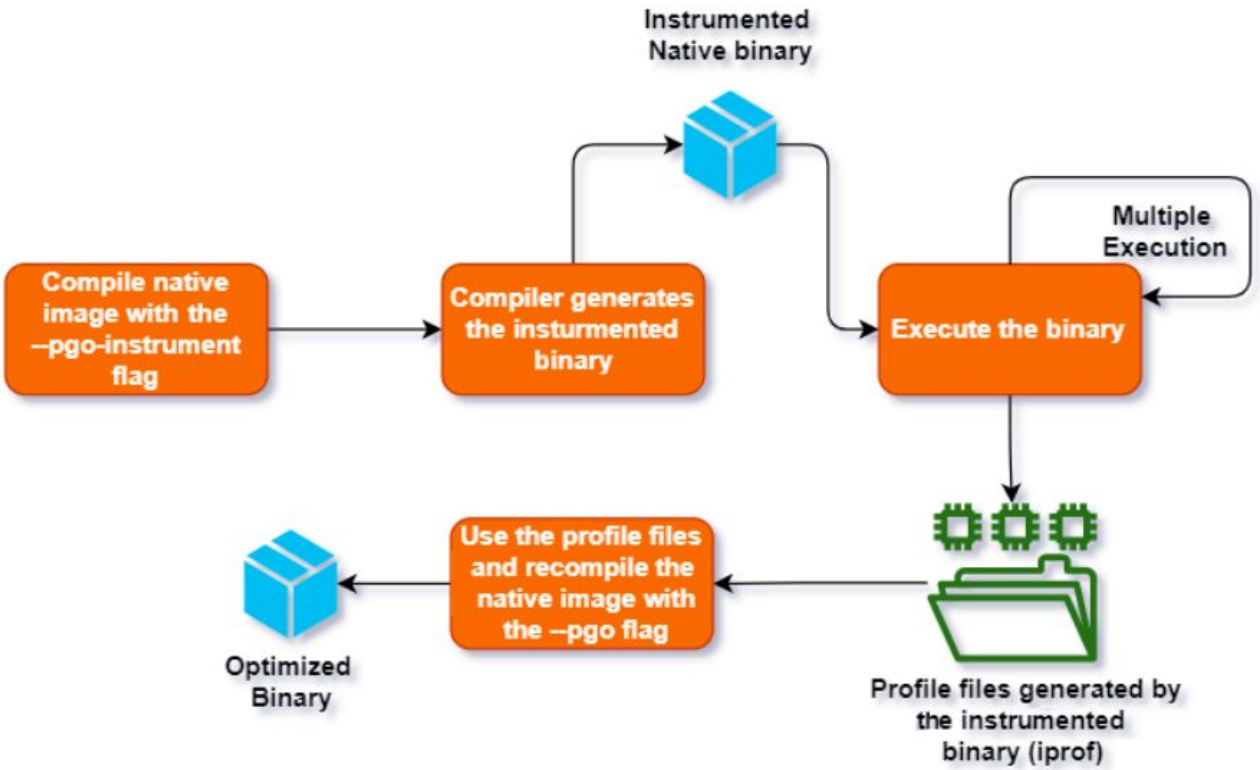
- In AOT compilation, the code is compiled directly to the machine code and executed. There is no runtime profiling or optimization/deoptimization.



Graal AOT


JIT
AOT
Load JVM
Load the executable
Load classes
Start running the optumum code
Verify bytecode and initialize the memory
Run the interpreter
On the threshold, run the C1 compiler
DeOptimize
Profile
Run the C2 compiler on the threshold

Graal PGO


Instrumented
Native binary
Multiple
Execution
Compile native
image with the
--pgo-instrument
flag
Compiler generates
the insturmented
binary
Execute the binary
Use the profile files
and recompile the
native image with
the --pgo flag
Optimized
Binary
Profile files generated by
the instrumented
binary (iprof)

Let’s test Graal AOT

Graal AOT Limitations

- **Loading classes dynamically**: Classes that are loaded at runtime, which will not be visible to the AOT compiler at build time, need to be specific in the configuration file. These configuration files are typically saved under META-INF/nativeimage/, and should be in CLASSPATH. If the class is not found during the compilation of the configuration file, it will throw a ClassNotFoundException.
- **Reflection**: Any call to the java.lang.reflect API to list the methods and fields or invoke them using the reflection API has to be configured in the reflect-config.json file under META-INF/native-image/. The compiler tries to identify these reflective elements through static analysis.

Graal AOT Limitations

- **Dynamic Proxy**: Dynamic proxy classes that are generated instances of java.lang.reflect.Proxy need to be defined during build time. The interfaces need to be configured in proxy-config.json.
- **Java Native Interface (JNI)**: Like reflection, JNI also accesses a lot of class information dynamically. Even these calls need to be configured in jni-config.json.
- **Serialization**: Java serialization also accesses a lot of class metadata dynamically. Even these accesses need to be configured ahead of time.

Graal AOT Limitations