Graal as a native JIT in GraalVM

Tom Rodriguez
GraalVM Compiler Team

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, timing, and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

GraalVM Native Image technology (including SubstrateVM) is early adopter technology. It is available only under an early adopter license and remains subject to potentially significant further changes, compatibility testing and certification.

GraalVM™

Java™ Ruby python™ C

Scala Groovy Kotlin R JS C++

OpenJDK™

node JS

ORACLE® Database

Native Image

What is Graal VM?

Drop-in replacement for Oracle Java 8 and Java 11

- Run your Java application faster

Ahead-of-time compilation for Java

- Create standalone binaries with low footprint

High-performance JavaScript, Python, Ruby, R, ...

- The first VM for true polyglot programming
- Implement your own language or DSL

Community Edition

GraalVM Community is available for free for evaluation, development and production use. It is built from the GraalVM sources available on GitHub. We provide pre-built binaries for Linux, macOS X, and Windows platforms on x86 64-bit systems. Windows support is experimental.

DOWNLOAD FROM GITHUB

Enterprise Edition

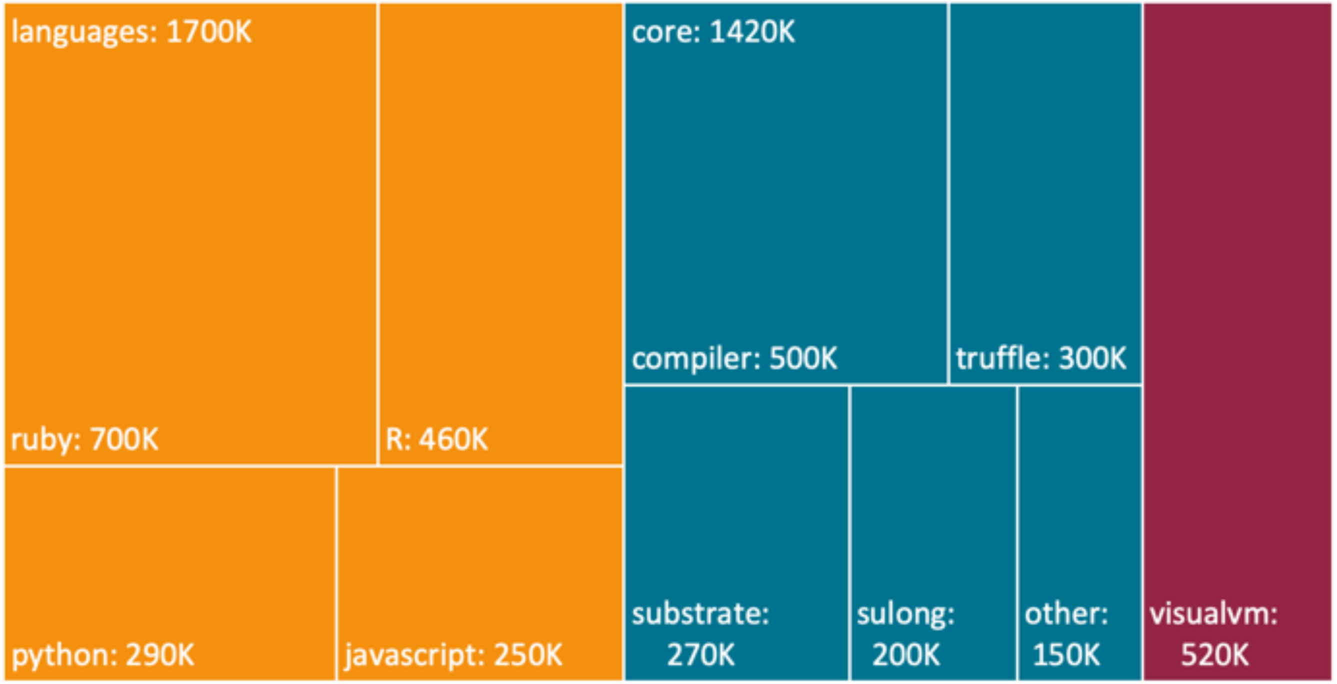
GraalVM Enterprise provides additional performance, security, and scalability relevant for running applications in production. It is free for evaluation uses and available for download from the Oracle Technology Network. We provide binaries for Linux, macOS X, and Windows platforms on x86 64-bit systems. Windows support is experimental.

DOWNLOAD FROM OTN

FREE on Oracle Cloud!

GraalVM Open Source

Open Source LOC actively maintained by GraalVM team



Total: 3,640,000 lines of code

Why libgraal

Warmup effects of pure Java (jargraal)

- Increased compilation with C1
- Increased heap allocation
- Gradual warmup of Graal itself

Visible as normal Java code

- Profile pollution
- Java debugging and profiling tools

Complicates JDK Testing

- -Xcomp, -Xbatch and -XX:-TieredCompilation

What is libgraal

Graal compiled with native image as a shared library

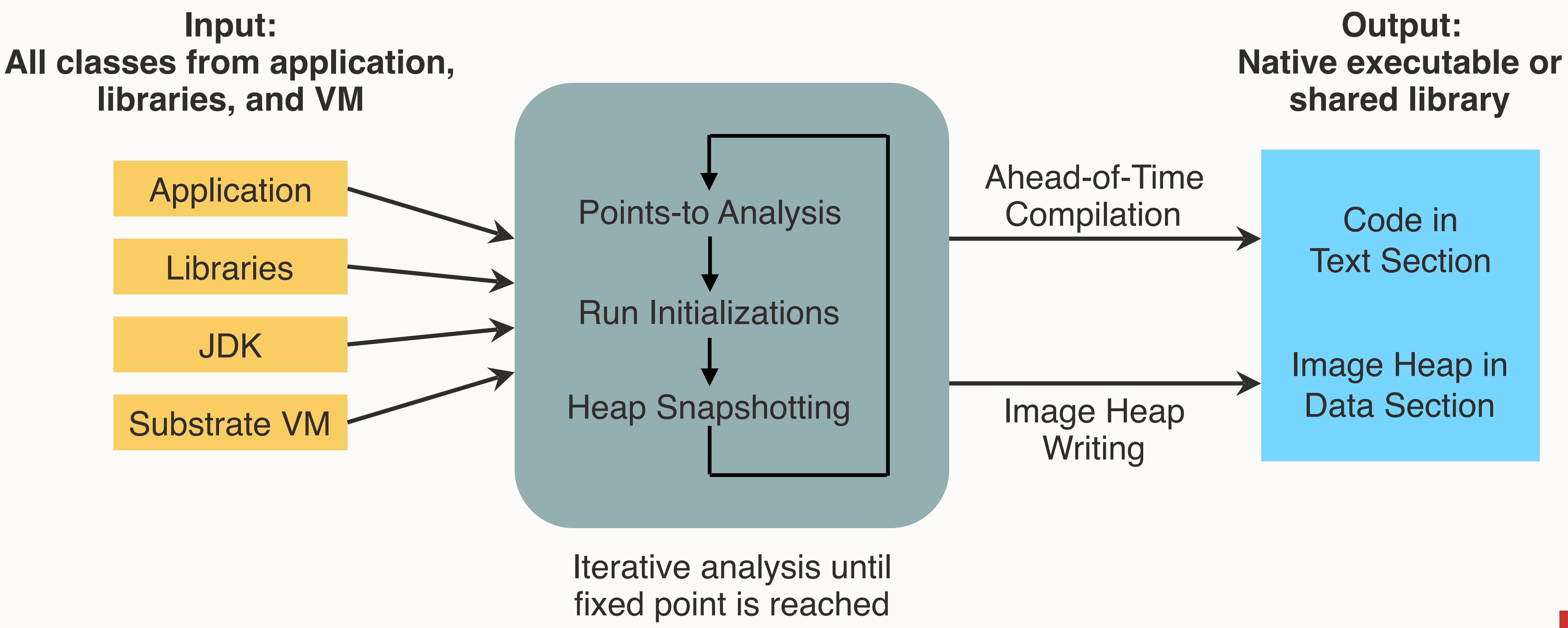
- It's actually libjvmcicompiler.so
- Interacts with HotSpot more like C1 or C2

Default way of using Graal in GraalVM since 19.0

- Required JVMCI support in Labs JDK 8, 11 and 13+
- Can be loaded by JDK8, JDK11 and JDK13+
  - Required JVMCI and Graal API and implementation changes

https://medium.com/graalvm/libgraal-graalvm-compiler-as-a-precompiled-graalvm-native-image-26e354bee5c

Native Image processing



Native image

Builds a standalone executable or library from a set of Java classes

Closed World Assumption

- The points-to analysis needs to see all bytecode
  - Removes unused classes, methods, and fields cannot be removed
  - Compiles all reachable code
- Dynamic parts of Java require configuration at build time
  - Reflection, JNI, Proxy, resources, …
- No loading of new classes at run time

Native Image Heap

Execution at run time starts with an initial heap: the “image heap”

- Objects are allocated in the Java VM that runs the image generator
- Heap snapshotting gathers all objects that are reachable at run time

Do things once at build time instead at every application startup

- Class initializers, initializers for static and static final fields
- Explicit code that is part of a so-called “Feature”

Examples for objects in the image heap

- java.lang.Class objects, Enum constants

Design goals of libgraal

Minimize differences between jargraal and libgraal paths

- Easier to maintain/debug

Use existing native image machinery as much as possible

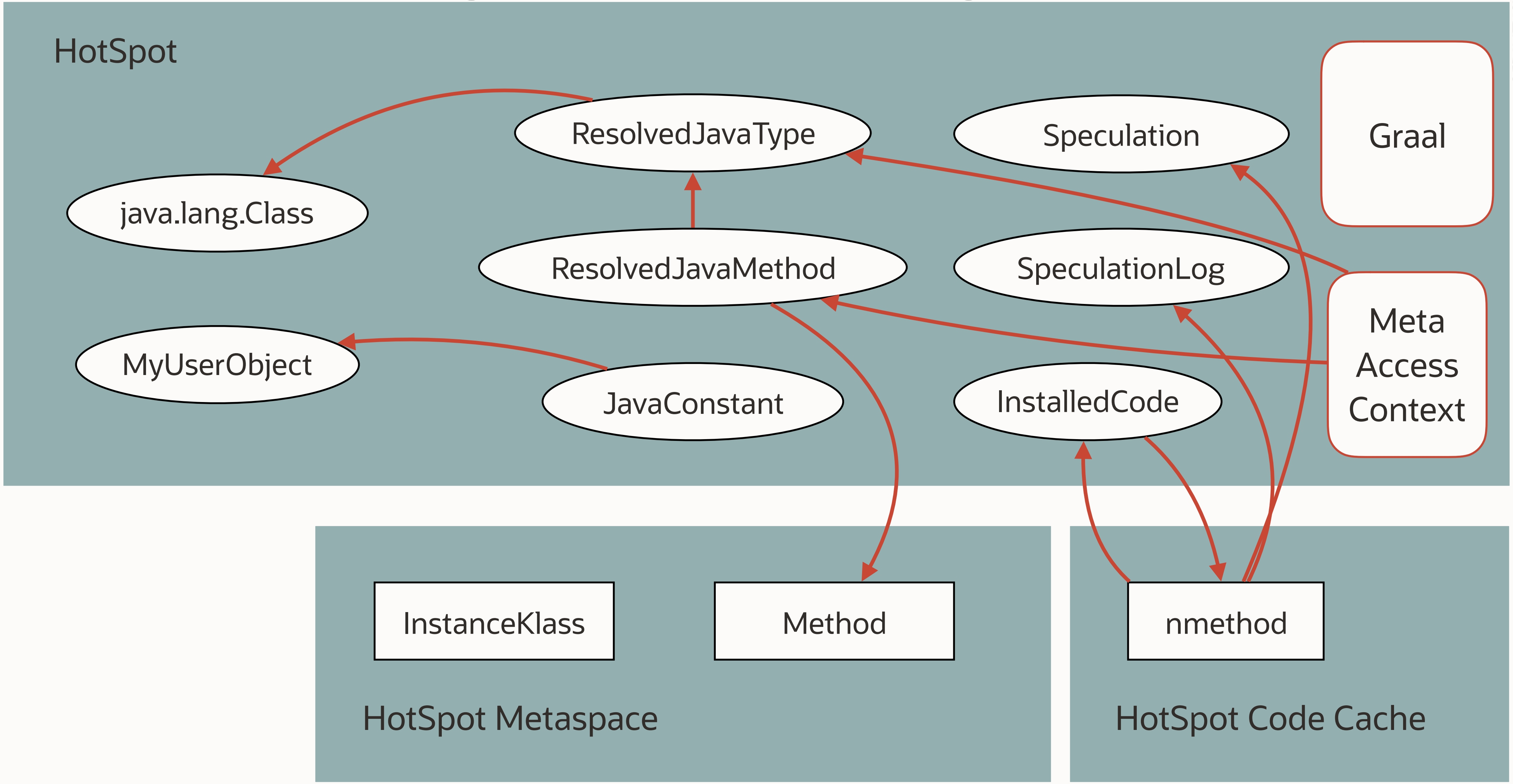
- JNI
- Encoded graphs

Rely on explicit logic instead of substitutions where possible

- Improves maintainability

Allow both jargraal and libgraal JVMCI runtimes

Original JVMCI design



JVMCI API changes

Create a stronger distinction between compiler objects and runtime objects

- Use JavaConstant instead of Object or subtypes in API
- Encoding of SpeculationReason as JavaConstant
- Eliminate exposed references to Class and other Objects

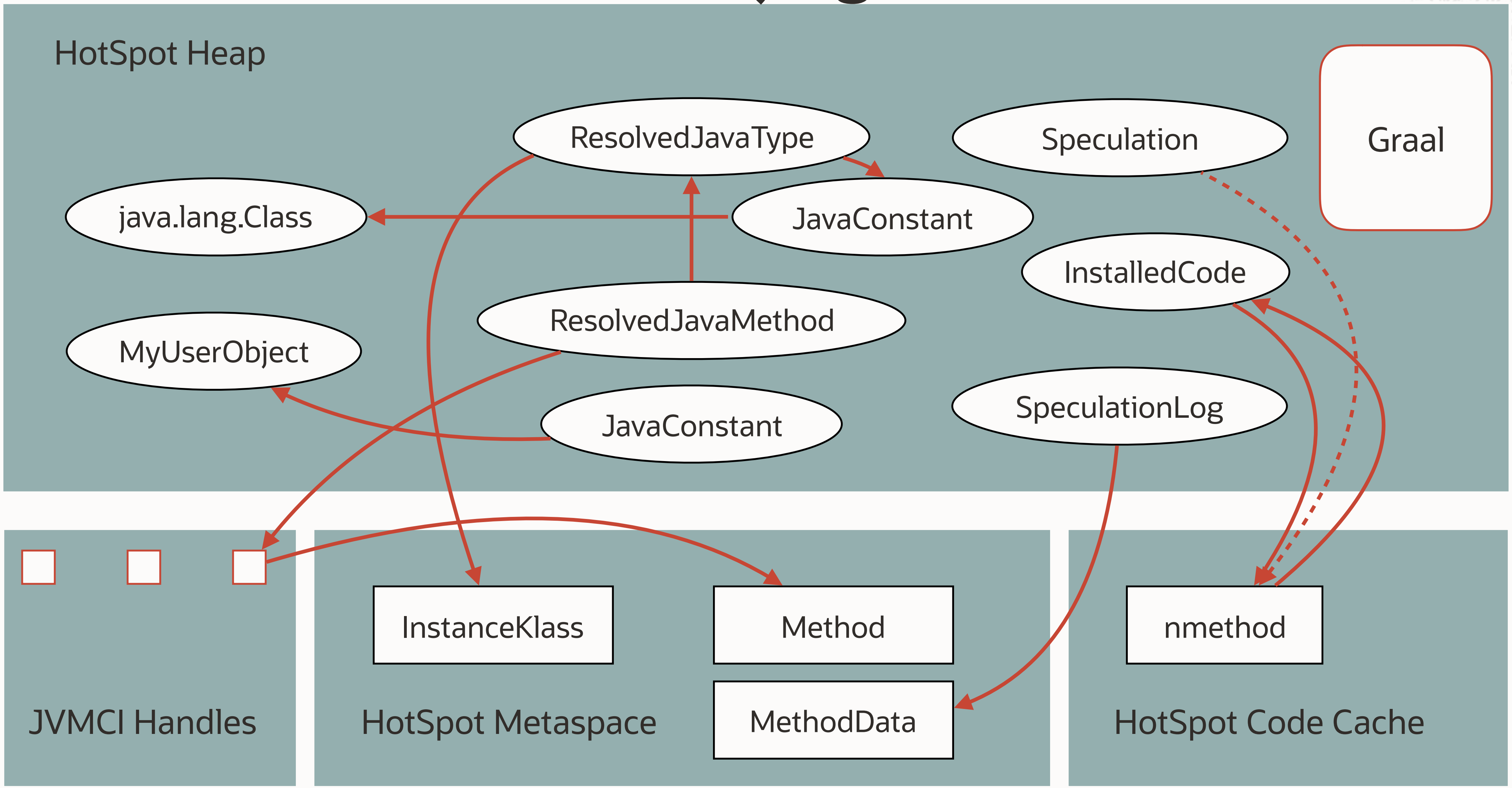
Compiler environment might not be the same as the execution environment

- Use of Unsafe to get offsets instead of querying JVMCI directly
- Use of Java reflection instead of JVMCI APIs

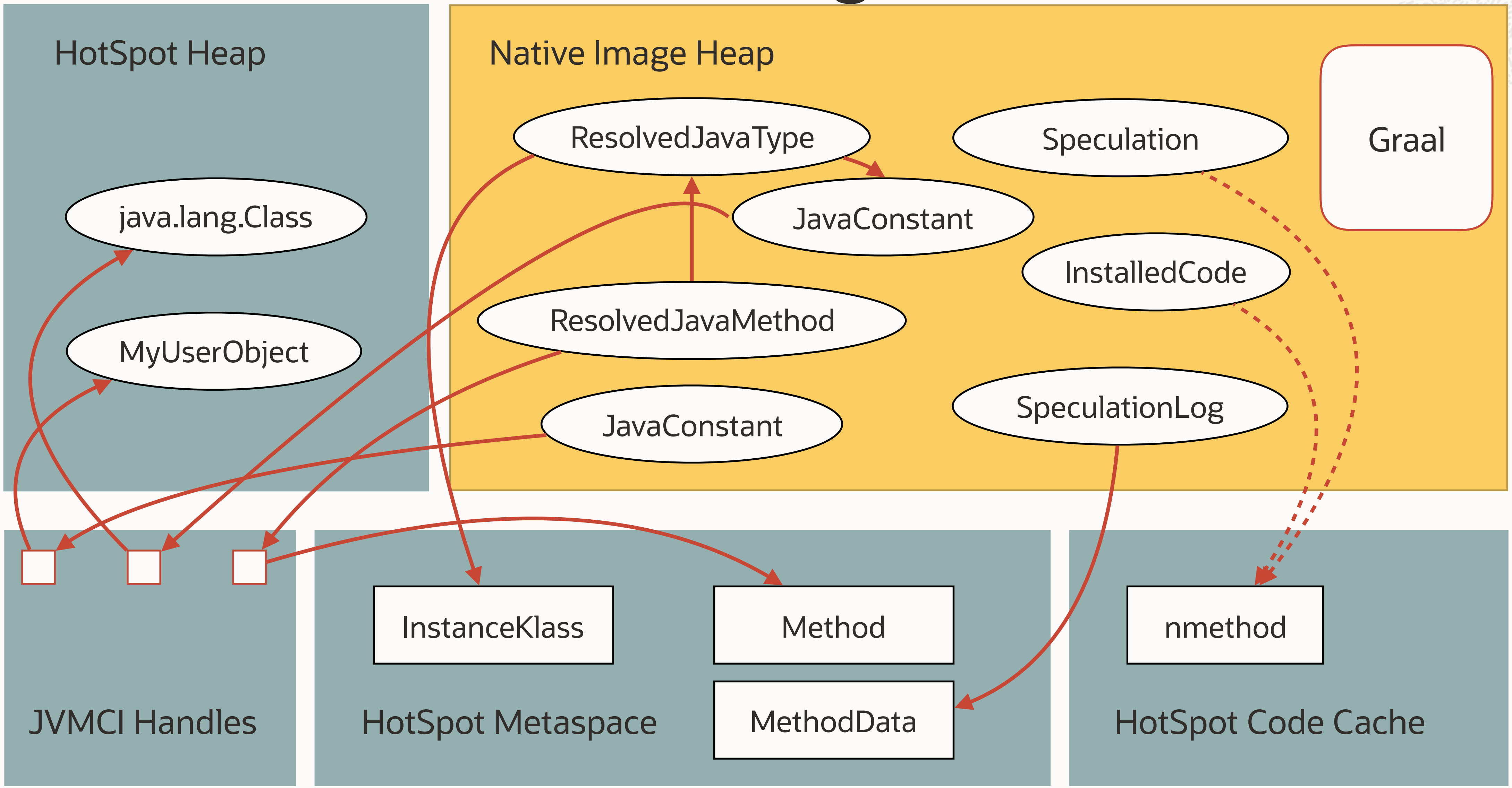
Integrated in JDK11 to ease later backporting

- JDK-8205824

JVMCI with jargraal



JVMCI with libgraal



JVMCI implementation changes to support libgraal

HotSpot assumes JVMCI objects belong to its Java heap

- C++ code must have indirect access to JVMCI objects

2 independent runtimes and GC

- reduce coupling between JVMCI and HotSpot

Native image Java code can't interact directly with HotSpot objects

- More JVMCI native methods

jdk.vm.ci.services.Services

- IS_BUILDING_NATIVE_IMAGE/IS_IN_NATIVE_IMAGE

Caching of various service lookups

Caching of Annotations required by the compiler

HotSpot C++ Changes

JVMCIEnv and JVMCIObject

- Abstracts away to interact with JVMCI objects
  - Creating constants
- Use JNI under the covers to talk to libgraal
  - Puts some restrictions on the HotSpot code
  - Increased overhead

Existing code translates and reads fairly naturally

Looser coupling between HotSpot and JVMCI objects where necessary

- InstalledCode and method
- Speculations become nmethod data

HotSpot JVMCI Example before

```
C2V_VMENTRY(jobject, getConstantPool, (JNIEnv *, jobject, jobject object_handle))
  constantPoolHandle cp;
  oop object = JNIHandles::resolve(object_handle);
  if (object == NULL) {
    THROW_0(vmSymbols::java_lang_NullPointerException());
  }
  if (object->is_a(SystemDictionary::HotSpotResolvedJavaMethodImpl_klass())) {
    cp = CompilerToVM::asMethod(object)->constMethod()->constants();
  } else if (object->is_a(SystemDictionary::HotSpotResolvedObjectTypeImpl_klass())) {
    cp = InstanceKlass::cast(CompilerToVM::asKlass(object))->constants();
  } else {
    THROW_MSG_0(vmSymbols::java_lang_IllegalArgumentException(),
                err_msg("Unexpected type: %s", object->klass()->external_name()));
  }
  assert(!cp.is_null(), "npe");
  JavaValue method_result(T_OBJECT);
  JavaCallArguments args;
  args.push_long((jlong) (address) cp());
  JavaCalls::call_static(&method_result, SystemDictionary::HotSpotConstantPool_klass(),
vmSymbols::fromMetaspace_name(), vmSymbols::constantPool_fromMetaspace_signature(), &args,
CHECK_NULL);
  return JNIHandles::make_local(THREAD, (oop)method_result.get_jobject());
}
```

HotSpot JVMCI Example after

```
C2V_VMENTRY_NULL(jobject, getConstantPool, (JNIEnv* env, jobject, jobject object_handle))
  constantPoolHandle cp;
  JVMCIObject object = JVMCIENV->wrap(object_handle);
  if (object.is_null()) {
    JVMCI_THROW_NULL(NullPointerException);
  }
  if (JVMCIENV->isa_HotSpotResolvedJavaMethodImpl(object)) {
    cp = JVMCIENV->asMethod(object)->constMethod()->constants();
  } else if (JVMCIENV->isa_HotSpotResolvedObjectTypeImpl(object)) {
    cp = InstanceKlass::cast(JVMCIENV->asKlass(object))->constants();
  } else {
    JVMCI_THROW_MSG_NULL(IllegalArgumentException,
                err_msg("Unexpected type: %s", JVMCIENV->klass_name(object)));
  }
  assert(!cp.is_null(), "npe");

  JVMCIObject result = JVMCIENV->get_jvmci_constant_pool(cp, JVMCI_CHECK_NULL);
  return JVMCIENV->get_jobject(result);
}
```

Dynamic Graal compilation in native image binaries

Uses serialized representation of parsed graphs

- No bytecodes available
- Faster than parsing

Fully initialized compiler stored in image heap

Single JVMCI namespace

- Only Substrate types

Dynamic Graal compilation in libgraal

Hybrid JVMCI environment

- BytecodeParser is used for Java code
- encoded graphs used for snippets and method substitutions
  - Refer to types that are actually part of libgraal

Compiler must be initialized at start of isolate

- Connections between JVMCI and HotSpot must be built dynamically

Snippets and method substitutions

Snippets are stylized pieces of Java code that implement low level features

- Fast/slow allocation paths or identityHashCode for example

Parsed into a graph and then inlined to replace other nodes

- @Fold is used to inject constant values from the environment
  - Field offsets or mark word values for instance
  - Boxes result into a JavaConstant
- @NodeIntrinsic is used to insert a particular IR node
  - Lets snippets perform low level operations
  - Often takes the result of @Fold as an input

Snippet Example: fast identityHashCode

```
@Snippet
static int identityHashCodeSnippet(Object x) {
  if (probability(NOT_FREQUENT_PROBABILITY, x == null)) {
    return 0;
  }

  Word mark = loadWordFromObject(x, markOffset());

  final Word biasedLock = mark.and(
                          biasedLockMaskInPlace());
  if (probability(FAST_PATH_PROBABILITY,
                          biasedLock.equal(WordFactory.unsigned(
                          unlockedMask())))) {
    int hash = (int) mark.unsignedShiftRight(
                         identityHashCodeShift()).rawValue();
    if (probability(FAST_PATH_PROBABILITY,
                  hash !=
uninitializedIdentityHashCodeValue())) {
      return hash;
    }
  }

  return identityHashCode(IDENTITY_HASHCODE, x);
}
```

The snippet is in class `HashCodeSnippets`

Node intrinsic

Constant folding during snippet parsing

Machine-word sized value

Snippet preparation and use in libgraal

Graph is parsed normally but all @Fold operations are deferred

- NodeIntrinsic might be deferred as well

Graph is encoded and stored in the libgraal heap

- The graph may contain constant references to HotSpot JVMCI types
  - Converted to unresolved types during image building

During dynamic compilation the snippet is decoded

- Fold and NodeIntrinsic are processed during decode
- Symbolic type references are resolved against HotSpot

Method substitutions

Like a snippet but inlined on the fly by the BytecodeParser

- Uses similar tricks to a Snippet but is often simpler
- Has to be careful about FrameStates

More problematic for libgraal because it's not a graph

- Alternate compilation mode for libgraal to encode the graph
- Alternate BytecodeParser path to inline the decoded graph

More likely to reference random JDK types

- May complicate decoding the graph

Method substitution example: SHA crypt

Avoid use of Unsafe

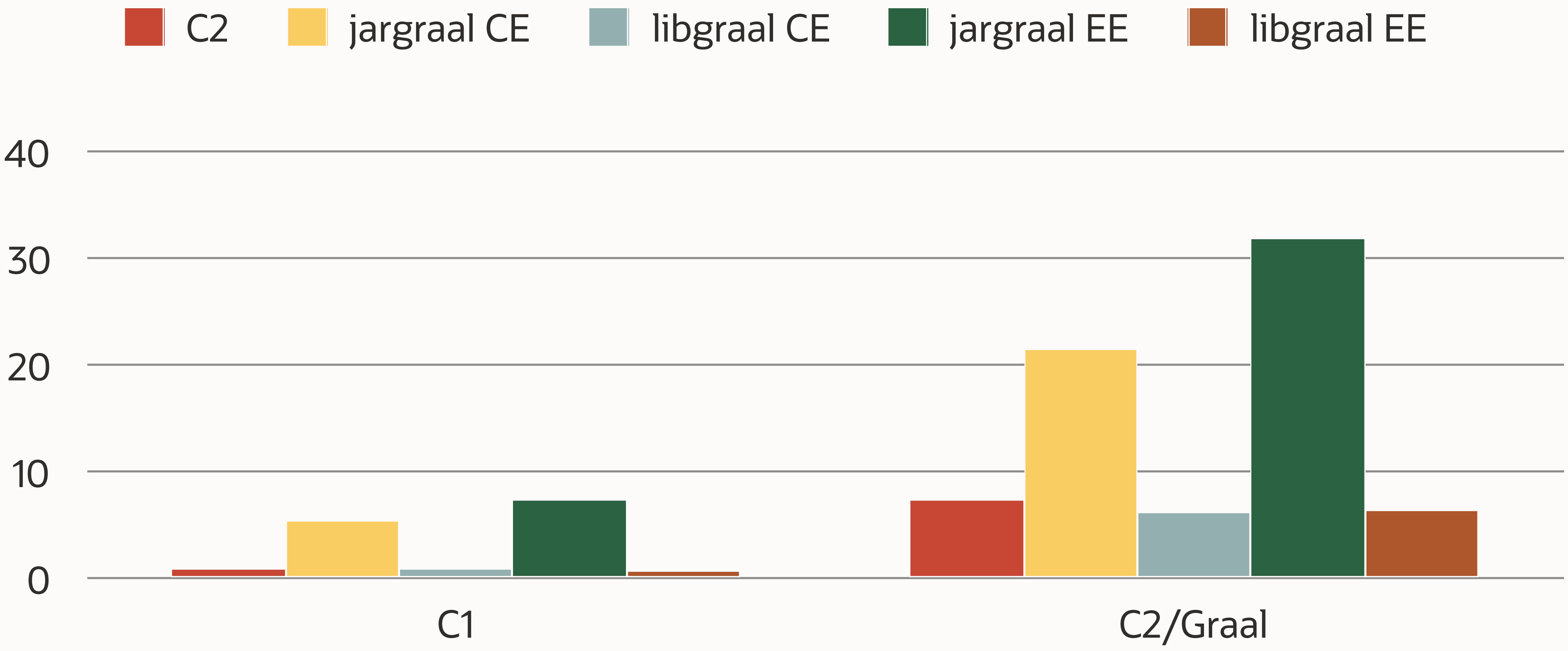
Injected HotSpot JVMCI type

Machine-word sized value

```
    @MethodSubstitution(isStatic = false)
    static void implCompress0(Object receiver, byte[] buf, int ofs) {
        Object realReceiver = PiNode.piCastNonNull(receiver,
                              HotSpotReplacementsUtil.methodHolderClass(INJECTED_INTRINSIC_CONTEXT));
        Object state = RawLoadNode.load(realReceiver, stateOffset(INJECTED_INTRINSIC_CONTEXT),
                                  JavaKind.Object, LocationIdentity.any());
        Word bufAddr = WordFactory.unsigned(ComputeObjectAddressNode.get(buf,
                       ReplacementsUtil.getArrayBaseOffset(INJECTED_METAACCESS, JavaKind.Byte) + ofs));
        Word stateAddr = WordFactory.unsigned(ComputeObjectAddressNode.get(state,
                         ReplacementsUtil.getArrayBaseOffset(INJECTED_METAACCESS, JavaKind.Int)));
        HotSpotBackend.sha5ImplCompressStub(bufAddr, stateAddr);
    }

    @Fold
    static long stateOffset(@InjectedParameter IntrinsicContext context) {
        return HotSpotReplacementsUtil.getFieldOffset(HotSpotReplacementsUtil.methodHolderClass(context),
"state");
    }
```

Total compile time for DaCapo lusearch



Tradeoffs

Native image GC is slower than HotSpot GC

- For most compiles it's not an issue
- Very large graphs
- Serializes multiple compilation threads

Increased JNI overhead from JNI

- Mainly affects the final code installation step

Native image compiled Graal may be slower than fully warmed up jargraal

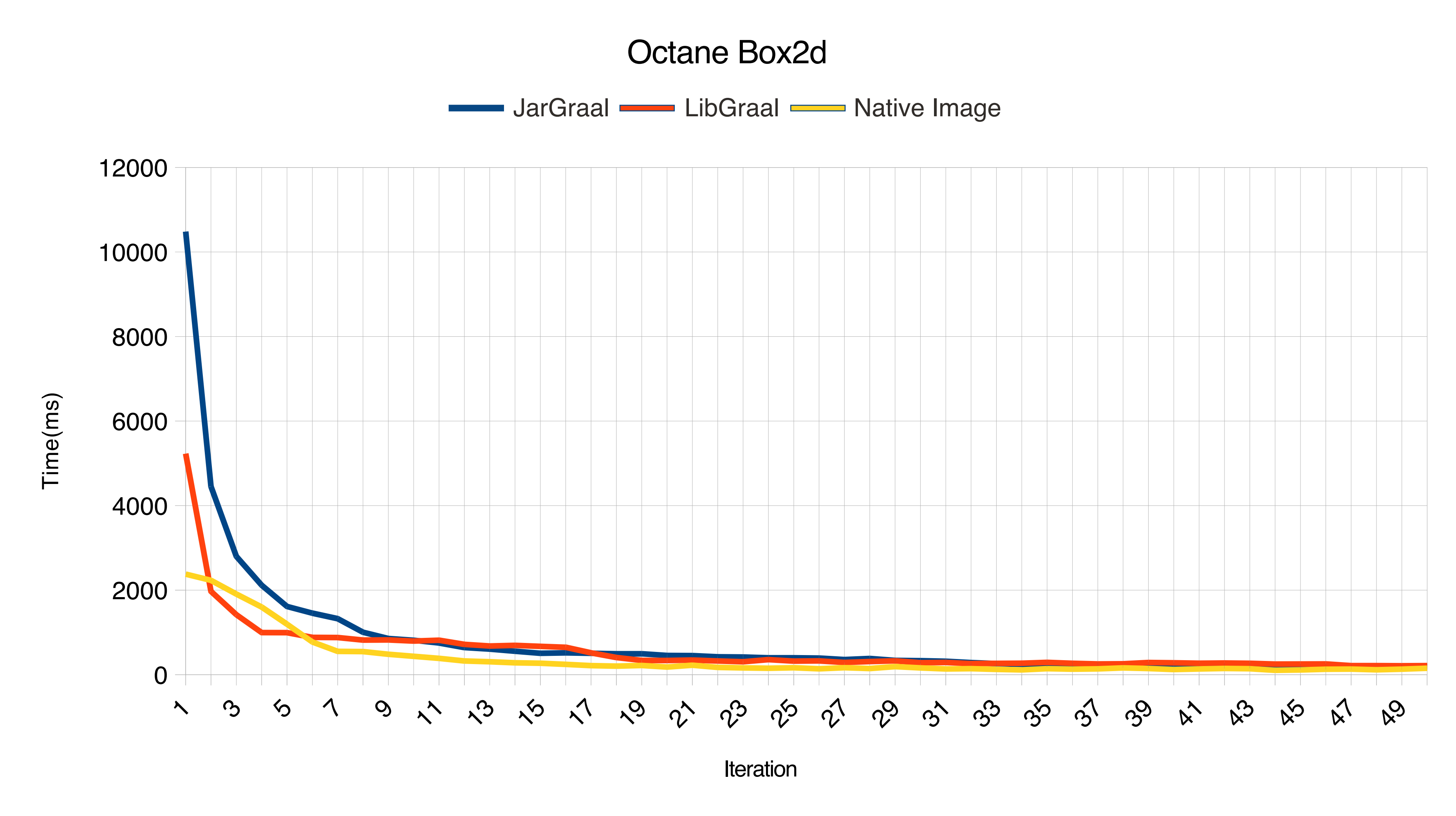
- Mitigated in GraalVM EE with PGO and compressed oops

Large shared library

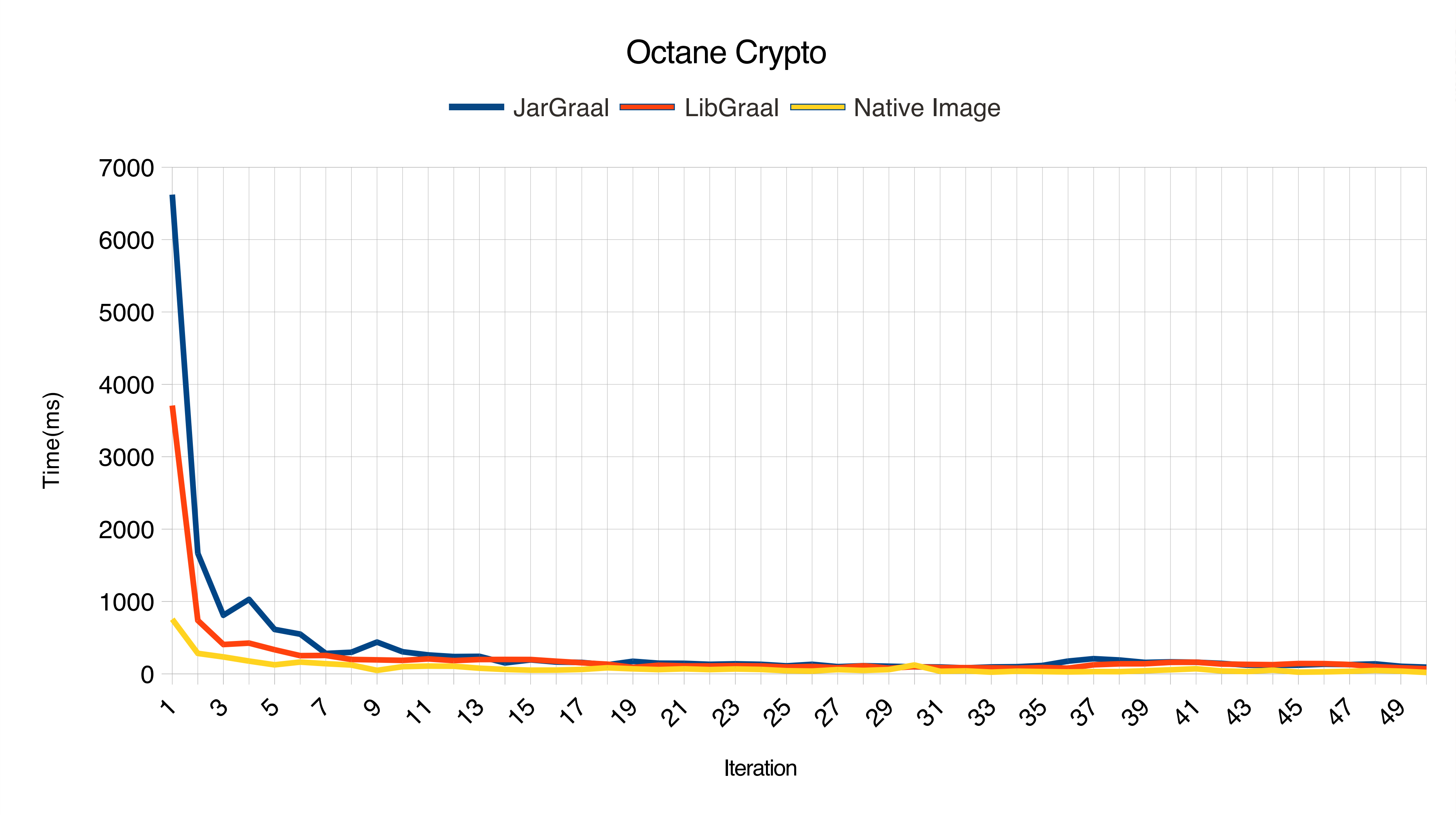
Truffle partial evaluation using libgraal

Truffle runtime can use libgraal to compile Truffle methods

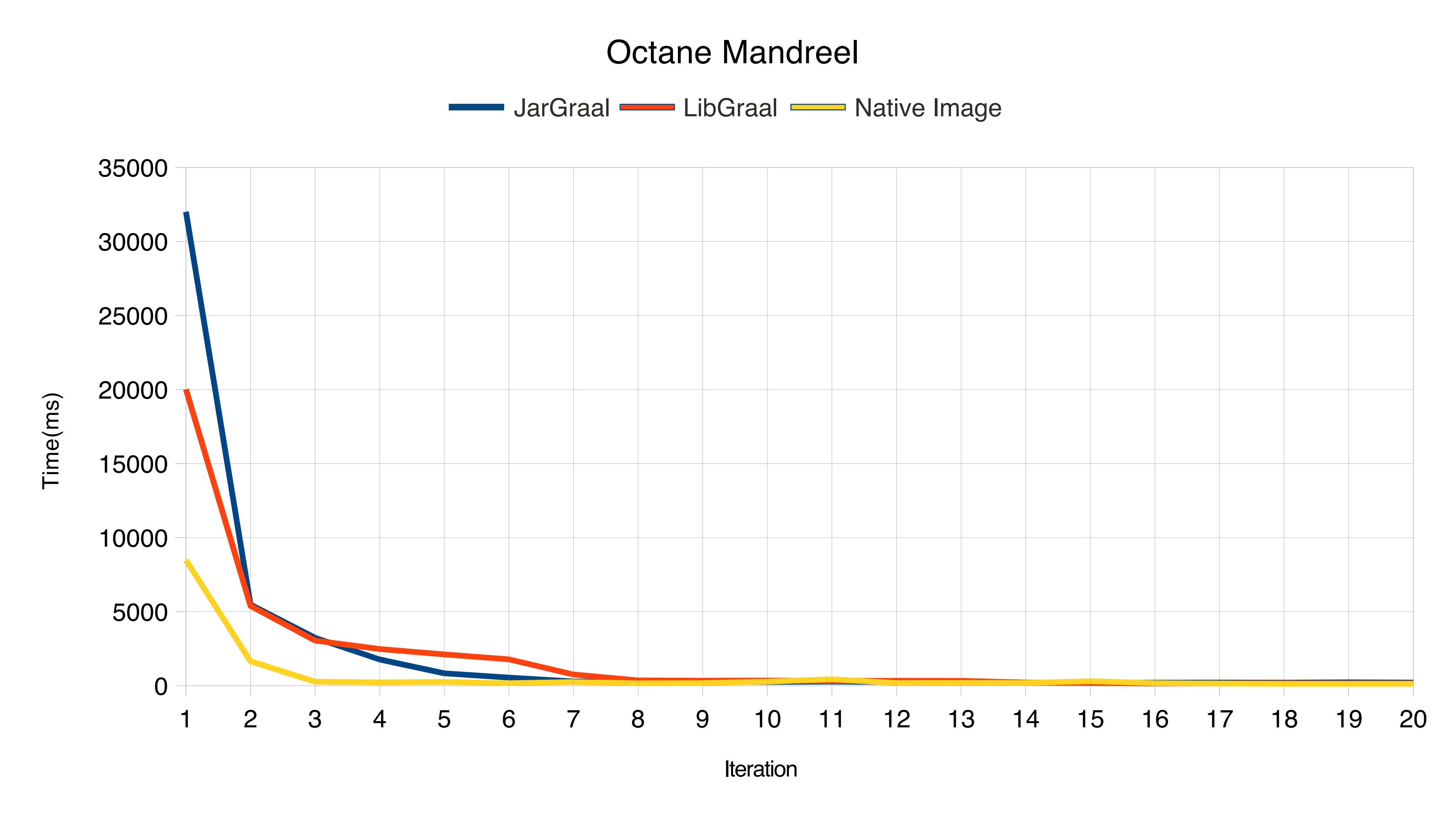
- Truffle runtime must always run as normal Java code
- libgraal exports a Truffle compilation entry point
  - Appears as a normal Java native method
  - See substratevm/ImplementingNativeMethodsInJavaWithSVM.md
- Invoked by passing JVMCI objects from jargraal to libgraal
  - All compilation is performed by libgraal
- Faster start up time



Time of each iteration running Octane Box2d
Times are average values of 10 executions
100 iterations time reduced from 58,58s to 35,83s



Time of each iteration running Octane Crypto
Times are average values of 10 executions
100 iterations time reduced from 18,39s to 10,17s



Time of each iteration running Octane Mandreel
Times are average values of 10 executions
100 iterations time reduced from 47,60s to 41.6s

Future work

Investigating multiple isolates

- Complicates monitoring and output

Completely isolate libgraal types from HotSpot runtime

- libgraal becomes completely standalone

GC tuning

Reduce library size

- Make some Graal options statically disabled

ORACLE