Graal: where it’s come from and where it’s going

Chris Seaton
Research Manager
Oracle Labs
February 2019
Safe Harbor Statement

The following is intended to provide some insight into a line of research in Oracle Labs. 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. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.
Automatic transformation of interpreters to compiler

GraalVM™

Embeddable in native or managed applications

OpenJDK™ node ORACLE®

MySQL™

standalone
Why is this one group working on all these diverse things?
Automatic transformation of interpreters to compiler

GraalVM™

Embeddable in native or managed applications

OpenJDK™ node ORACLE Database MySQL standalone

>
Automatic transformation of interpreters to compiler

Embeddable in native or managed applications

GraalVM™

OpenJDK™  node  ORACLE® Database  MySQL™  standalone
What *is* Graal?
What is Graal?

• A compiler from Java intermediate representations to native machine code*
  – Intermediate representation could be JVM bytecode or Graal’s own IR
  – ‘Java’ means any JVM language in general
  – Not a compiler from Java source code to Java bytecode

• It’s a compiler library rather than being an executable, like GCC

• Some things I didn’t tie Graal down to there:
  – I didn’t say it was specifically a just-in-time compiler
  – I didn’t say the bytecode or IR had to come directly from a program as written
Graal is graphical

Graal IR: An Extensible Declarative Intermediate Representation

Gilles Duboscq* Lukas Stadler* Thomas Würthinger†
Doug Simon† Christian Wimmer† Hanspeter Mössenböck*
Graal is *graphical*
Graal is *optimizing* and *speculative*.
Graal is a Java library

```java
public byte[] compile(byte[] bytecode) {
    ...
}
```
Where did Graal come from?
Maxine: An Approachable Virtual Machine For, and In, Java

CHRISTIAN WIMMER, MICHAEL HAUPT, MICHAEL L. VAN DE VANTER, MICK JORDAN, LAURENT DAYNÈS, and DOUGLAS SIMON, Oracle Labs

ACM Transactions on Architecture and Code Optimization, Vol. 9, No. 4, Article 30, Publication date: January 2013.
Fig. 1. Structure of the Maxine VM.
Equivalent to JVMCI

Became Graal
5. FUTURE WORK

The short-term plans for the Maxine VM focus on improving performance. We are working on a generational GC, which will reduce long GC pause times that currently occur with large heap sizes. Simultaneously, we are working on an improved optimizing compiler, which will work both in the Java HotSpot VM and the Maxine VM. It is developed in a separate OpenJDK project called Graal [Oracle 2012g]. The Graal Compiler-Runtime-Interface (CRI) is an improved version of the Maxine CRI, so the integration of Graal into Maxine will be straightforward. Finally, we will keep Maxine up to date with respect to improvements of the Java VM specification. We plan to implement method handles and the invokedynamic bytecode that were introduced for Java 7. Currently Maxine can work with a JDK 7 class library, but cannot execute applications needing VM features introduced for Java 7.

On the long term, we envision Maxine as a research platform for multiple languages. Exploiting the already modular structure and scheme abstractions, we want to make Maxine a truly modular VM. Benefits and a possible structure of a modular VM based on the Maxine VM are described in Wimmer et al. [2012].
C1 to Graal

• C1 (the ‘client’ compiler)
  – I believe it’s actually newer than C2 (the ‘server’, or ‘optimizing’ or ‘opto’ compiler)
  – Designed to produce reasonably good code reasonably quickly
  – More than a template compiler

• Became C1X when rewritten in Java for Maxine
  – “more or less literal Java port of the C++ code of C1”

• Became Graal when made a component usable outside of Maxine
  – Not really sure this is quite an accurate thing to say
  – Graal used the same LIR as C1X, but the IR in Graal is sea-of-nodes, while C1 is CFG
  – The LIR has evolved a lot since then as well
5. FUTURE WORK

The short-term plans for the Maxine VM focus on improving performance. We are working on a generational GC, which will reduce long GC pause times that currently occur with large heap sizes. Simultaneously, we are working on an improved optimizing compiler, which will work both in the Java HotSpot VM and the Maxine VM. It is developed in a separate OpenJDK project called Graal [Oracle 2012g]. The Graal Compiler-Runtime-Interface (CRI) is an improved version of the Maxine CRI, so the integration of Graal into Maxine will be straightforward. Finally, we will keep Maxine up to date with respect to improvements of the Java VM specification. We plan to implement method handles and the invokedynamic bytecode that were introduced for Java 7. Currently Maxine can work with a JDK 7 class library, but cannot execute applications needing VM features introduced for Java 7.

On the long term, we envision Maxine as a research platform for multiple languages. Exploiting the already modular structure and scheme abstractions, we want to make Maxine a truly modular VM. Benefits and a possible structure of a modular VM based on the Maxine VM are described in Wimmer et al. [2012].
What does Graal look like today?
Graal is a dynamic compiler written in Java that integrates with the HotSpot JVM. It has a focus on high performance and extensibility. In addition, it provides optimized performance for Truffle-based languages running on the JVM.

Setup
JEP 243: Java-Level JVM Compiler Interface

Owner   John Rose
Type    Feature
Scope   JDK
Status  Closed / Delivered
Release 9
Component hotspot/compiler
Discussion hotspot dash compiler dash dev at openjdk dot java dot net
Effort   M
Duration M
Reviewed by Douglas Simon, Mikael Vidstedt, Thomas Wuerthinger, Vladimir Kozlov
Endorsed by Mikael Vidstedt
Created 2014/10/29 20:43
Updated 2017/05/19 01:58
Issue 8062493

Summary
Develop a Java based JVM compiler interface (JVMCI) enabling a compiler written in Java to be used by the JVM as a dynamic compiler.

Goals
- Allow a Java component programmed against the JVMCI to be loaded at runtime and used by the JVM’s compile broker.
- Allow a Java component programmed against the JVMCI to be loaded at runtime and used by trusted Java code to install machine code in the JVM that can be called via a Java reference to the installed code.

Non-Goals
- Integration of a dynamic compiler (such as Graal) based on JVMCI.
public byte[] compile(byte[] bytecode) {
    ...
}

public byte[] compile(byte[] bytecode, 
        MetaData profiledData) {

    ...

}
public void compile(byte[] bytecode, MetaData profiledData, CodeCache codeCache) {
    codeCache.install(...);
}
public void compile(Method method,
            MetaData profiledData,
            CodeCache codeCache) {
    codeCache.install(... method.getBytecode() ...);
}
public interface JVMCICompiler {
    CompilationRequestResult compileMethod(CompilationRequest request);
}
JVMCI

• An interface from a JVM to a compiler implemented in Java
• A bit like JVM agents
• Equivalent to CRI in Maxine
• Graal implements JVMCI
• Other compilers could as well
• And other JVM’s could implement JVMCI
• Not really quite as simple as the interface suggests...
  – The compiler needs to know about the VM’s object layout, deoptimization, safepoint mechanism, GC barriers, and more
Run on OpenJDK 11 with Graal enabled

$ java -XX:+UnlockExperimentalVMOptions -XX:+EnableJVMCI -XX:+UseJVMCICompiler ...

This is using Java 11, not GraalVM!
Run on OpenJDK 11 with a custom Graal build

$ java -XX:+UnlockExperimentalVMOptions \
-XX:+EnableJVMCI \
-XX:+UseJVMCICompiler \
-Djvmci.class.path.append=graal.jar \
...

This is using Java 11, and using a custom Graal build
Is this metacircularity for the sake of it?
Maxine: An Approachable Virtual Machine For, and In, Java

CHRISTIAN WIMMER, MICHAEL HAUPT, MICHAEL L. VAN DE VANTER, MICK JORDAN, LAURENT DAYNÈS, and DOUGLAS SIMON, Oracle Labs

ACM Transactions on Architecture and Code Optimization, Vol. 9, No. 4, Article 30, Publication date: January 2013.
A highly productive platform accelerates the production of research results. The design of a Virtual Machine (VM) written in the Java™ programming language can be simplified through exploitation of interfaces, type and memory safety, automated memory management (garbage collection), exception handling, and reflection. Moreover, modern Java IDEs offer time-saving features such as refactoring, auto-completion, and code navigation. Finally, Java annotations enable compiler extensions for low-level “systems programming” while retaining IDE compatibility. These techniques collectively make complex system software more “approachable” than has been typical in the past.

The Maxine VM, a metacircular Java VM implementation, has aggressively used these features since its inception. A co-designed companion tool, the Maxine Inspector, offers integrated debugging and visualization of all aspects of the VM’s runtime state. The Inspector’s implementation exploits advanced Java language features, embodies intimate knowledge of the VM’s design, and even reuses a significant amount of VM code directly. These characteristics make Maxine a highly approachable VM research platform and a productive basis for research and teaching.
synchronized (foo) {
    ...
}

synchronized (foo) {
    ...
}
public class LockEliminationPhase extends Phase {

    @Override
    protected void run(StructuredGraph graph) {
        for (MonitorExitNode monitorExitNode : graph.getNodes(MonitorExitNode.TYPE)) {
            FixedNode next = monitorExitNode.next();
            if (next instanceof MonitorEnterNode) {
                if (next instanceof RawMonitorEnterNode) {
                    // should never happen, nor monitor enters are always direct successors of the graph
                    // start
                    AccessMonitorNode monitorEnterNode = AccessMonitorNode next;
                    if (isCompatibleLock(monitorEnterNode, monitorExitNode)) {
                        // We've coarsened the lock so use the same monitor id for the whole region,
                        // otherwise the monitor operations appear to be unrelated.
                        MonitorIdNode enterId = monitorEnterNode.getMonitorId();
                        MonitorIdNode exitId = monitorExitNode.getMonitorId();
                        if (exitId == enterId)
                            enterId.replaceAndDelete(exitId);
                        GraphUtil.removeFixedWithUnusedInputs(monitorEnterNode);
                        GraphUtil.removeFixedWithUnusedInputs(monitorExitNode);
                    }
                }
            }
        }
    }

    /**
     * Check that the paired operations operate on the same object at the same lock depth.
     */
    public static boolean isCompatibleLock(AccessMonitorNode lock1, AccessMonitorNode lock2) {
        // It is not always the case that sequential monitor operations on the same object have the
        // same lock depth: Escape analysis can have removed a lock operation that was in between,
        // leading to a mismatch in lock depth.
        ValueNode object1 = GraphUtil.unproxify(lock1.object());
        ValueNode object2 = GraphUtil.unproxify(lock2.object());
        return object1 == object2 && lock1.getMonitorId().getLockDepth() == lock2.getMonitorId().getLockDepth();
    }
}
@Override
protected void run(StructuredGraph graph) {
    for (MonitorExitNode monitorExitNode : graph.getNodes(MonitorExitNode.TYPE)) {
        FixedNode next = monitorExitNode.next();
        if (!(next instanceof MonitorEnterNode || next instanceof RawMonitorEnterNode)) {
            assert !(next instanceof OSRMonitorEnterNode);
            AccessMonitorNode monitorEnterNode = (AccessMonitorNode) next;
            if (isCompatibleLock(monitorEnterNode, monitorExitNode)) {
                /
                * We've coarsened the lock so use the same monitor id for the whole region,
                * otherwise the monitor operations appear to be unrelated.
                */
                MonitorIdNode enterId = monitorEnterNode.getMonitorId();
                MonitorIdNode exitId = monitorExitNode.getMonitorId();
                if (enterId != exitId) {
                    enterId.replaceAndDelete(exitId);
                }
                GraphUtil.removeFixedWithUnusedInputs(monitorEnterNode);
                GraphUtil.removeFixedWithUnusedInputs(monitorExitNode);
            }
        }
    }
}
Partial Escape Analysis and Scalar Replacement for Java

Lukas Stadler
Johannes Kepler University
Linz, Austria
lukas.stadler@jku.at

Thomas Würthinger
Oracle Labs
thomas.wuerthinger
@oracle.com

Hanspeter Mössenböck
Johannes Kepler University
Linz, Austria
moessenboeck@ssw.jku.at
static Object global;
void foo(int x) {
    Integer i = new Integer(x);
    global = null;
    ...
}

(a) After inlining.

(b) After Partial Escape Analysis.

Partial Escape Analysis and Scalar Replacement for Java

Lukas Stadler
Johannes Kepler University
Linz, Austria
lukas.stadler@jku.at

Thomas Würthinger
Oracle Labs
thomas.wuerthinger@oracle.com

Hanspeter Mössenböck
Johannes Kepler University
Linz, Austria
moessenboeck@ssw.jku.at
Trace-based Register Allocation in a JIT Compiler

Josef Eisl†
Institute for System Software
Johannes Kepler University
Linz, Austria

Matthias Grimmer†
Institute for System Software
Johannes Kepler University
Linz, Austria

Doug Simon†
Oracle Labs
Switzerland

Thomas Würthinger†
Oracle Labs
Switzerland

Hanspeter Mössenböck†
Institute for System Software
Johannes Kepler University
Linz, Austria
boolean equals(int[] a, int[] b) {
    /*B1*/ if (b.length != a.length)
    /*B2*/     return false;
    /*B3*/     int i = 0;
    /*B4*/     while (i < a.length) {
    /*B5*/         if (a[i] != b[i])
    /*B6*/             return false;
    /*B7*/             i++;
    /*B8*/         }
    /*B9*/     return true;
}
What are the applications of Graal?
What are the applications of Graal?

Graal as a JIT compiler for JVM languages
Automatic transformation of interpreters to compiler

GraalVM™

Embeddable in native or managed applications

OpenJDK™
public static void main(String[] args) {
    Arrays.stream(args)
        .flatMap(TopTen::fileLines)
        .flatMap(line -> Arrays.stream(line.split("\\b")))
        .map(word -> word.replaceAll("[^a-zA-Z]", ""))
        .filter(word -> word.length() > 0)
        .map(word -> word.toLowerCase())
        .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))
        .entrySet().stream()
        .sorted((a, b) -> a.getValue().compareTo(b.getValue()))
        .limit(10)
        .forEach(e -> System.out.format("%s = %d%n", e.getKey(), e.getValue()));
Running using the GraalVM EE distribution

$ javac TopTen.java
$ time java TopTen large.txt

... real 0m18.905s
Compare to standard OpenJDK

$ time java -XX:-UseJVMCICompiler TopTen large.txt
...
real 0m23.102s
What are the applications of Graal?

Graal as a specialized compiler for JVM applications
Graal as a specialized compiler for JVM applications

• Graal is modular, and the JVMCI has a plug-in architecture
• Can we write custom optimization passes for specific applications and plug them into Graal?
  – Optimizations that only make sense for your application?
  – Optimizations that break the normal rules of the JVM Spec but you’re happy they’re safe for your application?
• github.com/jruby/jruby-graal
public static class JRubyGraalCompilerConfiguration extends CoreCompilerConfiguration {
    @Override
    public PhaseSuite<HighTierContext> createHighTier(OptionValues options) {
        HighTier highTier = new HighTier(options);
        ListIterator iter = highTier.findPhase(PartialEscapePhase.class);
        iter.previous();
        iter.add(new JRubyVirtualizationPhase());
        return highTier;
    }
}
@Override
protected void run(StructuredGraph structuredGraph, PhaseContext phaseContext) {
    NodeIterable<Node> nodes = structuredGraph.getNodes();
    nodes.forEach(n -> {
        if (n.getClass() == NewInstanceNode.class) {
            NewInstanceNode newInstance = (NewInstanceNode) n;
            if (isVirtual(newInstance.instanceClass())) {
                System.out.println("virtualizing fixnum: " + newInstance);
                JRubyNewInstanceNode jnin = structuredGraph.add(new JRubyNewInstanceNode(
                    newInstance.instanceClass(), newInstance.fillContents(), newInstance.stateBefore()));
                structuredGraph.replaceFixedWithFixed(newInstance, jnin);
            }
        }
    });
}

private boolean isVirtual(ResolvedJavaType type) {
    String name = type.getName();

    if (name.contains("RubyFixnum") || name.contains("RubyFloat")) return true;
    return false;
}
public static class JRubyNewInstanceNode extends NewInstanceNode {

    public JRubyNewInstanceNode(ResolvedJavaType type, boolean fillContents, FrameState stateBefore) {
        super(type, fillContents, stateBefore);
    }

    @Override
    public void virtualize(VirtualizerTool tool) {
        /*
         * This is always for virtualizable JRuby objects, so always virtualize.
         */
        VirtualInstanceNode virtualObject = new VirtualInstanceNode(instanceClass(), false);
        ResolvedJavaField[] fields = virtualObject.getFields();
        ValueNode[] state = new ValueNode[fields.length];
        for (int i = 0; i < state.length; i++) {
            state[i] = defaultValueOfField(fields[i]);
        }
        tool.createVirtualObject(virtualObject, state, Collections.<MonitorIdNode> emptyList(), false);
        tool.replaceWithVirtual(virtualObject);
    }
}
What are the applications of Graal?

Graal as an AOT compiler for JVM languages
GraalVM™

Automatic transformation of interpreters to compiler

Embeddable in native or managed applications

OpenJDK™, node, ORACLE Database, MySQL™, standalone

Scala, Kotlin
Run as normal

$ time java TopTen small.txt
...
real 0m0.408s
Compile to native using native-image

$ native-image TopTen
...
$ time ./topten small.txt
...
real 0m0.112s
$ du -h topten
8.8M topten
$ otool -L topten

topten:
/usr/lib/libSystem.B.dylib
/usr/lib/libz.1.dylib
/System/Library/CoreFoundation
SubstrateVM using Graal

• The native-image tool is using SubstrateVM and Graal
• SubstrateVM does closed-world analysis on a Java application
• Produces a set of methods to compile
• Runs Graal to compile them
  – (literally calls a method called GraalCompiler.compile)
  – Replaces the in-memory code-cache with an on-disk code cache
• Details around configuring for static code, but the big idea translates pretty literally to the code
Maxine: An Approachable Virtual Machine For, and In, Java

CHRISTIAN WIMMER, MICHAEL HAUPT, MICHAEL L. VAN DE VANTER, MICK JORDAN, LAURENT DAYNÈS, and DOUGLAS SIMON, Oracle Labs

ACM Transactions on Architecture and Code Optimization, Vol. 9, No. 4, Article 30, Publication date: January 2013.
Fig. 4. Maxine VM boot image generation process.
Fig. 4. Maxine VM boot image generation process.
SVM provides much of this, much like Maxine
Doesn’t provide this
What are the applications of Graal?

Graal as a JIT compiler for dynamic languages
Automatic transformation of interpreters to compiler

GraalVM™

Embeddable in native or managed applications

OpenJDK™, node.js, Oracle Database, MySQL, standalone
**Current situation**

- Prototype a new language
- Parser and language work to build syntax tree (AST), AST Interpreter
- Write a “real” VM
  - In C/C++, still using AST interpreter, spend a lot of time implementing runtime system, GC, …
- People start using it
- People complain about performance
  - Define a bytecode format and write bytecode interpreter
- Performance is still bad
  - Write a JIT compiler
  - Improve the garbage collector

**How it should be**

- Prototype a new language in Java
- Parser and language work to build syntax tree (AST)
- Execute using AST interpreter
- People start using it
  - And it is already fast
One VM to Rule Them All

Thomas Würthinger*  Christian Wimmer*  Andreas Wöß†  Lukas Stadler†
Gilles Duboscq†  Christian Humer†  Gregor Richards§  Doug Simon*  Mario Wolczko*

Onward! 2013, October 29–31, 2013, Indianapolis, Indiana, USA.
Node Rewriting for Profiling Feedback

Node Transitions
- Uninitialized
- Integer
- String
- Double
- Generic

AST Interpreter Uninitialized Nodes

Compilation using Partial Evaluation

Compiled Code
Deoptimization to AST Interpreter

Node Rewriting to Update Profiling Feedback

Recompilation using Partial Evaluation
```java
@Specialization(rewriteOn=ArithmeticException.class)
int addInt(int a, int b) {
    return Math.addExact(a, b);
}

@Specialization
double addDouble(double a, double b) {
    return a + b;
}

@Generic
Object addGeneric(Frame f, Object a, Object b) {
    // Handling of String omitted for simplicity.
    Number aNum = Runtime.toNumber(f, a);
    Number bNum = Runtime.toNumber(f, b);
    return Double.valueOf(aNum.doubleValue() +
                          bNum.doubleValue());
}
```
boolean interpreterCall(OptimizedCallTarget callTarget) {
    int intCallCount = ++callCount;
    int intAndLoopCallCount = ++callAndLoopCount;
    if (!callTarget.isCompiling() && !compilationFailed) {
        // Check if call target is hot enough to compile, but took not too long to get hot.
        int callThreshold = compilationCallThreshold; // 0 if TruffleCompileImmediately
        int callAndLoopThreshold = compilationCallAndLoopThreshold;
        if ((intAndLoopCallCount >= callAndLoopThreshold && intCallCount >= callThreshold) || callThreshold == 0) {
            return callTarget.compile(!multiTierEnabled);
        }
    }
    return false;
}
protected PEGraphDecoder createGraphDecoder(...) {
    final GraphBuilderConfiguration newConfig = configForParsing.copy();
    ...
    Plugins plugins = newConfig.getPlugins();
    ...
    plugins.appendInlineInvokePlugin(new ParsingInlineInvokePlugin(replacements, parsingInvocationPlugins, loopExplosionPlugin));
    ...
    return new CachingPEGraphDecoder(..., newConfig, ...);
}
public InlineKind getInlineKind(ResolvedJavaMethod original, boolean duringPartialEvaluation) {
    TruffleBoundary truffleBoundary = getAnnotation(TruffleBoundary.class, original);
    if (truffleBoundary != null) {
        if (duringPartialEvaluation || !truffleBoundary.allowInlining()) {
            // Since this method is invoked by the bytecode parser plugins, which can be invoked
            // by the partial evaluator, we want to prevent inlining across the boundary during
            // partial evaluation,
            // even if the TruffleBoundary allows inlining after partial evaluation.
            if (!truffleBoundary.throwsControlFlowException() && truffleBoundary.transferToInterpreterOnException()) {
                return InlineKind.DO_NOT_INLINE_DEOPTIMIZE_ON_EXCEPTION;
            } else {
                return InlineKind.DO_NOT_INLINE_WITH_EXCEPTION;
            }
        } else if (getAnnotation(TruffleCallBoundary.class, original) != null) {
            return InlineKind.DO_NOT_INLINE_WITH_EXCEPTION;
        }
        return InlineKind.INLINE;
    }
}
What are the applications of Graal?

Graal as a JIT compiler for native languages
Automatic transformation of interpreters to compiler

GraalVM™

Embeddable in native or managed applications

OpenJDK™ node ORACLE® Database MySQL™

standalone

>
define i8* @psd_native_util_clamp(i8* %self,
    i8* %r_num, i8* %r_min, i8* %r_max) nounwind uwtable ssp {
    %1 = call i32 @FIX2INT(i8* %r_num)
    %2 = call i32 @FIX2INT(i8* %r_min)
    %3 = call i32 @FIX2INT(i8* %r_max)
    %4 = icmp sgt i32 %1, %3
    br i1 %4, label %5, label %6
; <label>:5
    br label %12
; <label>:6
    %7 = icmp slt i32 %1, %2
    br i1 %7, label %8, label %9
; <label>:8
    br label %10
; <label>:9
    br label %10
; <label>:10
    %11 = phi i8* [% %r_min, %8 ], [ %r_num, %9 ]
    br label %12
; <label>:12
    %13 = phi i8* [%r_max, %5 ], [ %11, %10 ]
    ret i8* %13
}
%4 = icmp sgt i32 %1, %3
br i1 %4, label %5, label %6
; <label>:5
  br label %12
; <label>:6
%7 = icmp slt i32 %1, %2
br i1 %7, label %8, label %9

<table>
<thead>
<tr>
<th>t4 = t1 &gt; t3</th>
</tr>
</thead>
<tbody>
<tr>
<td>if t4</td>
</tr>
<tr>
<td>goto l5</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>goto l6</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>l5: goto l12</td>
</tr>
<tr>
<td>l6: t7 = t1 &lt; t2</td>
</tr>
<tr>
<td>if t7</td>
</tr>
<tr>
<td>goto l8</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>goto l9</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>
What are the applications of Graal?

Graal as a tool for embedding languages
Automatic transformation of interpreters to compiler

GraalVM™

Embeddable in native or managed applications

OpenJDK™, node.js, Database, MySQL™
**validator.js**

A library of string validators and sanitizers.

**Strings only**

This library validates and sanitizes strings only.

If you're not sure if your input is a string, coerce it using `input + ''`. Passing anything other than a string is an error.

**Installation and Usage**

**Server-side usage**

Install the library with `npm install validator`

No ES6

```javascript
var validator = require('validator');

validator.isEmail('foo@bar.com'); //=> true
```

**ES6**

```javascript
import validator from 'validator';
```
Demo using the Oracle Database MLE

- Multi-lingual (polyglot) edition
- Available as a Docker image
- Subject to the Oracle Technology Network license agreement, so you need to accept that and download it yourself

https://oracle.github.io/oracle-db-mle/releases/0.2.7/docker/
$ npm install validator
$ npm install @types/validator

$ dbjs deploy -u ... -p ... -c localhost:1521/ORCLCDB validator

$ sqlplus .../...@localhost:1521/ORCLCDB

(Demo simplified - see the website to get specifics)
SQL> select validator.isEmail('chris.seaton@oracle.com');
0

SQL> select validator.isEmail('chris.seaton');
1

(Demo simplified - see the website to get specifics)
AOT + JIT
The Apache SIS™ library

Apache Spatial Information System (SIS) is a free software, Java language library for developing geospatial applications. SIS provides data structures for geographic features and associated metadata along with methods to manipulate those data structures. The library is an implementation of GeoAPI 3.0 interfaces and can be used for desktop or server applications.

The SIS metadata module forms the base of the library and enables the creation of metadata objects which comply with the model of international standards. The SIS referencing module enable the construction of geodetic data structures for geospatial referencing such as axis, projection and coordinate reference system definitions, along with the associated operations which enable the conversion or transformation of coordinates between different systems of reference. The SIS storage modules will provide a common approach to the reading and writing of metadata, features and coverages.

Some Apache SIS features are:

- Geographic metadata (ISO 19115-1:2014)
  - Read from or written to ISO 19139 compliant XML documents
  - Read from netCDF, GeoTIFF, Landsat, GPX and Moving Feature CSV encoding

- Referencing by coordinates (ISO 19111:2007)
  - Well Known Text (WKT) version 1 and 2 (ISO 19162:2015)
  - Geographic Markup Language (GML) version 3.2.1 (ISO 19136:2007)
  - EPSG geodetic dataset for geodetic definitions and for coordinate operations. See the list of supported coordinate reference systems.
  - Mercator, Transverse Mercator, Lambert Conic Conformal, stereographic and more map projections. See the list of supported operation methods.
  - Optional bridge to Proj.4 as a complement to Apache SIS own referencing engine.

- Referencing by Identifiers (ISO 19112:2003)
  - Geohashes (a simple encoding of geographic coordinates into short strings of letters and digits)
  - Military Grid Reference System (MGRS), also used for some civilian uses.

- Units of measurement
  - Implementation of JSR-363 with parsing, formatting and unit conversion functionalities.
import org.graalvm.nativeimage.IsolateThread;
import org.graalvm.nativeimage.c.function.CEntryPoint;

public class Distance {

    ... 

    @CEntryPoint(name = "distance")
    public static double distance(IsolateThread thread,
            double a_lat, double a_long,
            double b_lat, double b_long) {
        return DistanceUtils.getHaversineDistance(a_lat, a_long, b_lat, b_long);
    }

    ...
$ native-image -cp sis.jar:. -H:Kind=SHARED_LIBRARY \ 
   -H:Name=libdistance
#include <stdlib.h>
#include <stdio.h>

#include <libdistance.h>

int main(int argc, char **argv) {
    graal_isolate_t *isolate = NULL;
    graal_isolatethread_t *thread = NULL;

    if (graal_create_isolate(NULL, &isolate) != 0 || (thread = graal_current_thread(isolate)) == NULL) {
        fprintf(stderr, "initialization error\n");
        return 1;
    }

    double a_lat  = strtod(argv[1], NULL);
    double a_long = strtod(argv[2], NULL);
    double b_lat  = strtod(argv[3], NULL);
    double b_long = strtod(argv[4], NULL);

    printf("%f km\n", distance(thread, a_lat, a_long, b_lat, b_long));

    return 0;
}
$ clang -I. -L. -ldistance distance.c -o distance
$ otool -L distance

distance:
    libdistance.dylib
    /usr/lib/libSystem.B.dylib
$ ./distance 51.507222 -0.1275 40.7127 -74.0059
5570.25 km
How is Graal being used today?
How is Graal being used today?

- Graal as a Java JIT
  - JVM bytecode → Graal → machine code
- Graal as a custom Java JIT
  - JVM bytecode → Graal → custom phases → Graal → machine code
- Graal as a Java AOT
  - JVM classes → Graal → executable or shared library
- Graal as a dynamic language JIT
  - source → interpreter → Truffle PE → Graal → machine code
- Graal as a native language JIT
  - LLVM bitcode → Sulong → Truffle PE → Graal → machine code
- Graal as a language embedded
  - interpreter → Graal → executable or shared library
Why is there both Graal and GraalVM?
“We’re running on Graal / GraalVM”
Graal and GraalVM?

• Graal is a compiler
  – Runs in different configurations
    • Ahead-of-time
    • Just-in-time
  – Used in different applications
    • Java compiler
    • Dynamic language compiler
    • Native language compiler
  – Distributed in different ways
    • Inside OpenJDK
    • In GraalVM

• GraalVM is a distribution of the wider Graal ecosystem
  – Compilers, tools, language implementations
  – Includes Graal configured as
    • A JVM language JIT compiler
    • A JVM language AOT compiler
    • A JIT compiler for dynamic languages
    • A JIT compiler for native languages
  – Realizes the potential of Graal
  – Graal is the enabler of all this
Graal is a dynamic compiler written in Java that integrates with the HotSpot JVM. It has a focus on high performance and extensibility. In addition, it provides optimized performance for Truffle-based languages running on the JVM.

Setup
GraalVM™
Run Programs Faster Anywhere

WHY GRAALVM
GET STARTED

High-performance polyglot VM

GraalVM is a universal virtual machine for running applications written in JavaScript, Python, Ruby, R, JVM-based languages like Java, Scala, Kotlin, Clojure, and LLVM-based languages such as C and C++.

GraalVM removes the isolation between programming languages and enables interoperability.
Where is Graal going in the future?
From: John Rose
To: discuss@openjdk.java.net
Subject: Call for Discussion: New Project: Metropolis

I would like to invite discussion on a proposal for a new OpenJDK Project[1], to be titled “Project Metropolis”, an incubator for experimenting with advanced JVM implementation techniques. Specifically, we wish to re-implement significant parts of Hotspot’s C++ runtime in Java itself, a move we call Java-on-Java. The key experiments will center around investigating Graal[2] as a code generator for the JVM in two modes: as an online compiler replacing one or more of Hotspot’s existing JITs, and as an offline compiler for Java code intended to replace existing C++ code in Hotspot. In the latter role, we will experiment with static compilation techniques (such as the Substrate VM[3]) to compile Java into statically restricted formats that can easily integrate with C++ as used in Hotspot.

The Project will be an experimental technology incubator, similar to the Lambda, Panama, Valhalla, and Amber projects. Such incubator projects absorb changes from the current Java release, but do not directly push to Java releases. Instead, they accumulate prototype changes which are sometimes discarded and sometimes merged by hand (after appropriate review) into a Java release.

(In this model, prototype changes accumulate quickly, since they are not subject to the relatively stringent rules governing JDK change-sets. These rules involving review, bug tracking, regression tests, and pre-integration builds. The Metropolis project will have similar rules, of course, but they are likely to be more relaxed.)
Implementing the Java runtime in the Java-on-Java style has numerous advantages, including:

- **Self-optimization**: We obtain more complete control of optimization techniques used for compiling the JVM itself.
- **Self-determination**: We can decouple the JVM from changes (possibly destabilizing ones) in other implementation languages (C++/NN).
- **Simplification**: More consistent usage of the “native” language of the Java ecosystem, reducing costs to contributors and maintainers.
- **Speed**: More agile delivery of new JVM backends (future hardware), new JVM frontends (value type bytecodes), new bytecode shapes (stream optimizations), and application formats (static application assembly).
However, the Java-on-Java tactic has **significant risks** which must be investigated and reduced before we can think about deploying products. These risks are:

- **Startup**: Startup overheads for Java code must not harm overall JVM startup.
- **Isolation**: GC or JIT activity required by Java-on-Java execution must not interfere with application execution.
- **Density**: Java-based data structures may require enhancement (such as value types) to support dense data structures competitive with C++.
- **Succession**: Adoption of Java-on-Java implementations must not cause regressions for customers who rely on the quality and performance of existing modules.
If these experiments are successful, numerous additional experiments are possible within the overall goal of implementing Java-on-Java:

- Using Graal as a replacement for the client JIT (C1).
- Using Graal to code-generate a bytecode interpreter.
- Using Graal to spin adapters, such as native-to-Java bindings.
- Using Graal to dynamically customize other JVM hot paths.
- Prototyping new JVM features, such as value types, in Graal.
- Coding native methods in statically-compiled Java.
- Coding metadata access and processing in Java.
- Coding smaller JVM modules in statically-compiled Java, such as class file parsing or verification.
- Coding GC logic in statically-compiled Java.
Learning more and getting in touch
Learning more and getting in touch

• Website – graalvm.org
• Papers – graalvm.org/community/publications
• GitHub – github.com/oracle/graal
• Gitter – gitter.im/graalvm/graal-core
• Twitter – @GraalVM
• Mailing lists – graal-dev@openjdk.java.net, graalvm-users@oss.oracle.com
• Email – chris.seaton@oracle.com
Team

Oracle
Florian Angerer
Danilo Ansaloni
Stefan Anzinger
Martin Balin
Cosmin Basca
Daniele Bonetta
Dušan Bálek
Matthias Brantner
Lucas Braun
Petr Chalupa
Jürgen Christ
Laurent Daynès
Gilles Duboscq
Svatopluk Dědic
Martin Entlicher
Pit Fender
François Farquet
Brandon Fish
Matthias GINNER
Christian Häubl
Peter Hofer
Bastian Hossbach
Christian Humer
Tomáš Hůrka
Mick Jordan

Oracle (continued)
Vojin Jovanovic
Anantha Kandukuri
Harshad Kasture
Cansu Kaynak
Peter Kessler
Duncan MacGregor
Jiří Maršík
Kevin Menard
Miloslav Metelka
Tomáš Myšík
Petr Pišl
Oleg Piss
Jakub Podlešák
Aleksandar Prokopec
Tom Rodriguez
Roland Schatz
Benjamin Schlegel
Chris Seaton
Jiří Sedláček
Doug Simon
Štěpán Sindelář
Zbyněk Slajchrt
Boris Spasojevic
Lukas Stadler
Codrut Stancu

Oracle (continued)
Jan Štola
Tomáš Stupka
Farhan Taukheeda
Jaroslav Tulacha
Alexander Ulrich
Michael Van De Vantera
Aleksandar Vitorovic
Christian Wimmer
Christian Wirth
Paul Wögerer
Mario Wolczeko
Andreas Wöß
Thomas Würthinger
Tomáš Zezula
Yudi Zheng

Oracle Interns
Brian Belleville
Ondrej Douda
Juan Fumero
Miguel Garcia
Hugo Guiroux
Shams Imam
Berkin Ibeli
Hugo Kapp
Alexey Karyakin
Stephen Kell
Andreas Kunst
Volker Lanting
Gero Leinemann
Julian Lettner
Joe Nash
Tristan Overney
Aleksandar Pejovic
David Piorkowski
Philipp Riedmann
Gregor Richards
Robert Seilbeck
Rifat Shariyar

Intel
Michael Berg

Twitter
Chris Thalinger

Oracle Alumni
Erik Eckstein
Michael Haupt
Christos Kotselidis
David Leibs
Adam Welc
Till Westmann

JKU Linz
Hanspeter Mössenböck
Benno Daloze
Josef Eisl
Thomas Feichtinger
Josef Haider
Christian Huber
David Leopoldseder
Stefan Marr
Manuel Rigger
Stefan Rumzucker
Bernhard Urban

TU Berlin:
Volker Markl
Andreas Kunst
Jens Meiners
Tilmann RABL

University of Edinburgh
Christophe Dubach
Juan José Fumero Alfonso
Ranjeet Singh
Toomas Remmelg

LaBRI
Floréal Morandat

University of California, Irvine
Michael Franz
Yeoul Na
Mohaned Qunaibit
Gulsem Savrun Yeniceri
Wei Zhang

Purdue University
Jan Vitek
Tomas Kalibera
Petr Maj
Lei Zhao

T. U. Dortmund
Peter Marwedel
Helena Kotthaus
Ingo Korb

University of California, Davis
Duncan Temple Lang
Nicholas Ulle

University of Lugano, Switzerland
Walter Binder
Sun Haiyang

Red Hat
Andrew Dinn
Andrew Haley

University of California, Irvine
Michael Franz
Yeoul Na
Mohaned Qunaibit
Gulsem Savrun Yeniceri
Wei Zhang

Purdue University
Jan Vitek
Tomas Kalibera
Petr Maj
Lei Zhao

T. U. Dortmund
Peter Marwedel
Helena Kotthaus
Ingo Korb

University of California, Davis
Duncan Temple Lang
Nicholas Ulle

University of Lugano, Switzerland
Walter Binder
Sun Haiyang

Oracle Interns
Brian Belleville
Ondrej Douda
Juan Fumero
Miguel Garcia
Hugo Guiroux
Shams Imam
Berkin Ibeli
Hugo Kapp
Alexey Karyakin
Stephen Kell
Andreas Kunst
Volker Lanting
Gero Leinemann
Julian Lettner
Joe Nash
Tristan Overney
Aleksandar Pejovic
David Piorkowski
Philipp Riedmann
Gregor Richards
Robert Seilbeck
Rifat Shariyar

Intel
Michael Berg

Twitter
Chris Thalinger

Oracle Alumni
Erik Eckstein
Michael Haupt
Christos Kotselidis
David Leibs
Adam Welc
Till Westmann

JKU Linz
Hanspeter Mössenböck
Benno Daloze
Josef Eisl
Thomas Feichtinger
Josef Haider
Christian Huber
David Leopoldseder
Stefan Marr
Manuel Rigger
Stefan Rumzucker
Bernhard Urban

TU Berlin:
Volker Markl
Andreas Kunst
Jens Meiners
Tilmann RABL

University of Edinburgh
Christophe Dubach
Juan José Fumero Alfonso
Ranjeet Singh
Toomas Remmelg

LaBRI
Floréal Morandat
Safe Harbor Statement

The preceding is intended to provide some insight into a line of research in Oracle Labs. 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. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.
Integrated Cloud
Applications & Platform Services