

**ORACLE®**



**ORACLE®**

# **Graal - A Bytecode Agnostic Compiler for the JVM**

Thomas Wuerthinger  
Oracle Labs

JVM Language Summit, 20st July 2011

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.

# My Background

- Dynamic Code Evolution VM
  - Supports unlimited class redefinition.
  - Binaries are available from <http://ssw.jku.at/dcevm/>.
- Working on Crankshaft/V8
  - Optimizing compiler for JavaScript.
  - Uses an SSA form intermediate representation.
  - 1.5x speedup compared to previous V8 version.
- Since April 2011: Oracle Labs

# Program Agenda

- Graal Compiler
- Compiler Extensions
- DLR within the JVM
- Workshop...

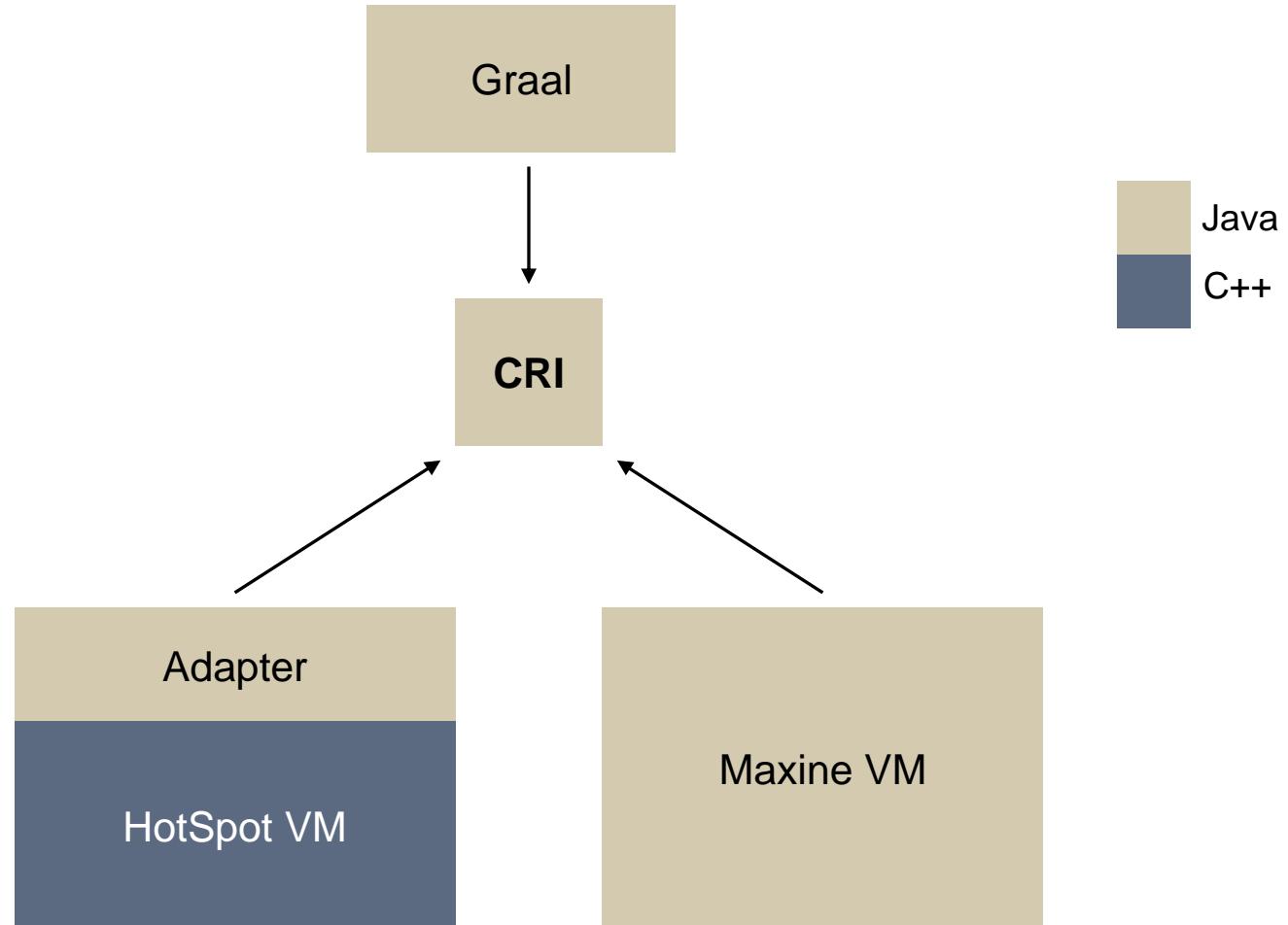


ORACLE®

# The Graal Compiler

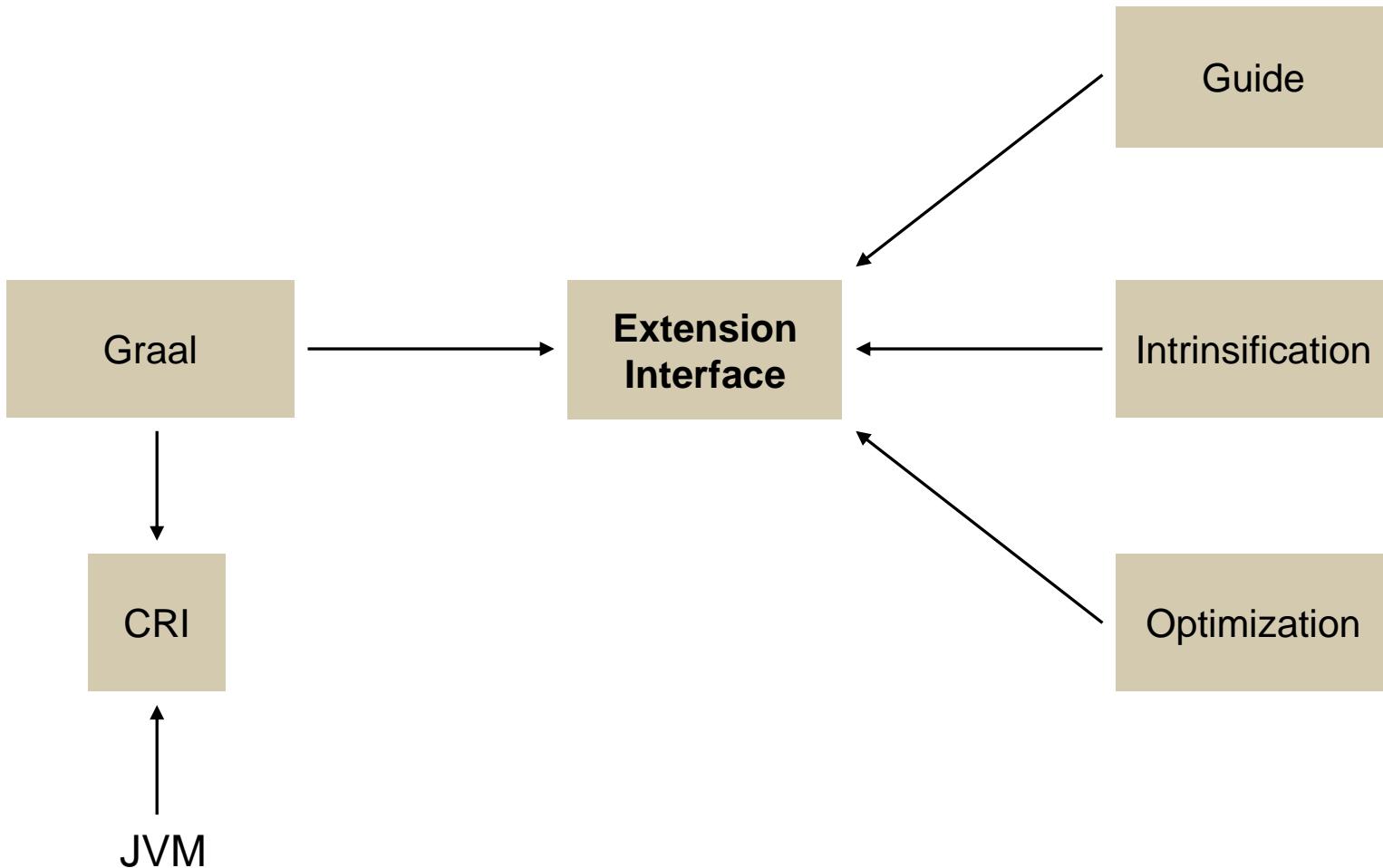
- Java JIT Compiler written in Java
  - Port of C1 from C++ to Java (C1X)
  - New high-level IR
- Basic Design
  - SSA form
  - Program Dependence Graph (“sea of nodes”)
  - Linear scan register allocator
- Extensibility
  - A compiler for multiple VMs
  - Customizable from Java application code

# Compiler-Runtime Separation



ORACLE®

# Compiler Extensions



# Guide Extensions (1)

## Example: Influencing Inlining Decisions

- Different parts of an application need different inlining settings (e.g. JRuby generated code versus Java code).

```
interface InliningGuide {  
    InliningHint getHint(int depth, RiMethod caller,  
                        int bci, RiMethod target);  
}
```

```
enum InliningHint {  
    NONE,  
    NEVER,  
    LESS,  
    MORE,  
    ALWAYS  
}
```

# Guide Extensions (2)

## Example: Influencing Inlining Decisions

```
class InliningGuideImpl implements InliningGuide {  
    public InliningHint getHint(int depth, RiMethod caller, int bci, RiMethod m) {  
        if (m.name().equals("neverInline")) return InliningHint.NEVER;  
        if (m.name().equals("alwaysInline") && depth < 50) return InliningHint.ALWAYS;  
        return InliningHint.NONE;  
    }  
}
```

Register as a service at [META-INF/services/com.oracle.max.graal.extensions.InliningGuide](#)

```
int test() {  
    return alwaysInline(30);  
}  
  
int alwaysInline(int value) {  
    if (value < 0) return neverInline(value);  
    return alwaysInline(value - 1);  
}  
  
int neverInline(int value) {  
    return value;  
}
```



# Intrinsics (1)

## Example: Efficient Detection of Integer Overflow

```
int safeAdd(int a, int b) {  
    int result = a + b;  
    if (b < 0 && result > a) {  
        throw new IllegalStateException("underflow");  
    } else if (b > 0 && result < a) {  
        throw new IllegalStateException("overflow");  
    }  
    return result;  
}
```

```
int test() {  
    int sum = 0;  
    for (int i = 0; i < N; i = safeAdd(i, 1)) {  
        sum = safeAdd(sum, i);  
    }  
    return sum;  
}
```

# Intrinsics (2)

## Example: Efficient Detection of Integer Overflow

- Special language constructs need an optimized machine code sequence.

```
interface Intrinsifier {
    Graph intrinsicGraph(RiRuntime runtime, RiMethod caller,
                         int bci, RiMethod method,
                         List<? extends Node> parameters);
}
```

# Intrinsification Extensions (3)

## Example: Efficient Detection of Integer Overflow

```
class IntrinsifierImpl implements Intrinsifier {
    public Graph intrinsicGraph(RiRuntime runtime, RiMethod caller,
                                int bci, RiMethod method, List<? extends Node> params) {
        if (method.name().equals("safeAdd")) {
            SafeAddNode node = new SafeAddNode(params.get(0), params.get(1));
            return CompilerGraph.create(node);
        }
        return null;
    }
}
```

```
class SafeAddNode extends IntegerArithmeticNode {

    // ...

    public void generate(LIRGenerator generator) {
        super.generate(generator);
        generator.deoptimizeOn(Condition.Overflow);
    }
}
```

# Optimization Extensions

## Example: Elimination of Overflow Checks

- Specific knowledge about the semantics of library methods enables custom optimization phases.

```
interface Optimizer {  
    void optimize(RiRuntime runtime, Graph graph);  
}
```

```
public class OptimizerImpl implements Optimizer {  
    public void optimize(RiRuntime runtime, Graph graph) {  
        for (SafeAddNode safeAdd : graph.getNodes(SafeAddNode.class)) {  
            if (!canOverflow(safeAdd)) {  
                IntegerAdd add = new IntegerAdd(safeAdd.x(), safeAdd.y());  
                safeAdd.replaceAndDelete(add);  
            }  
        }  
        // ...  
    }  
}
```

# DLR within the JVM (1)

## Direct Deoptimization into the DLR Interpreter

```
int safeAdd(int a, int b) {  
    int result = a + b;  
    if (b < 0 && result > a) {  
        throw new IllegalStateException("underflow");  
    } else if (b > 0 && result < a) {  
        throw new IllegalStateException("overflow");  
    }  
    return result;  
}
```

Dynamic Language  
Runtime Interpreter

```
int test() {  
    int sum = 0;  
    for (int i = 0; i < N; i = safeAdd(i, 1)) {  
        sum = safeAdd(sum, i);  
    }  
    return sum;  
}
```

# DLR within the JVM (2)

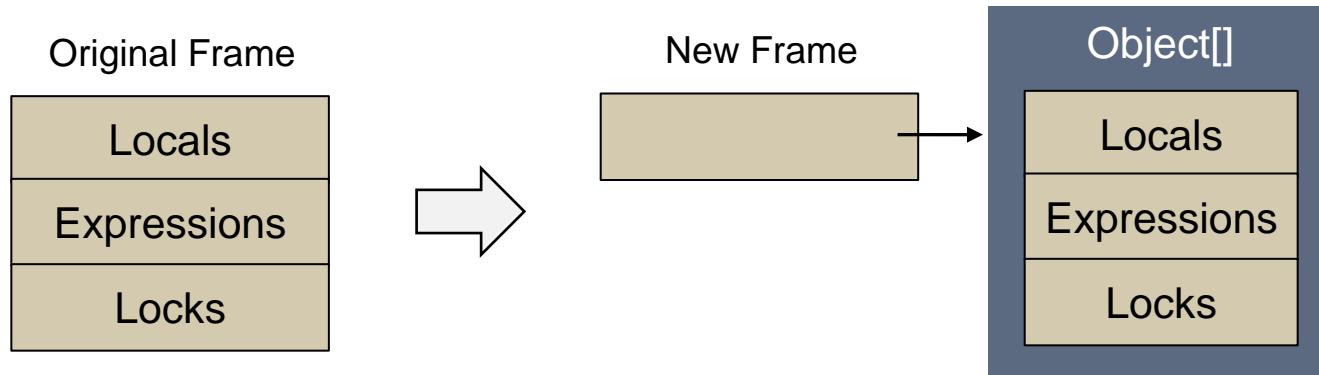
## Direct Deoptimization into the DLR Interpreter

- Graal enables changes to the deoptimization information using extensions.

```
interface FrameModifier {  
    CiFrame getFrame(RiRuntime runtime, CiFrame frame);  
}
```

# DLR within the JVM (3)

## Direct Deoptimization into the DLR Interpreter



```
public class FrameModifierImpl implements FrameModifier {
    public CiFrame getFrame(RiRuntime runtime, CiFrame frame) {
        if (frame.method.name().equals("test")) {
            RiMethod handlerMethod = runtime.getMethod("DeoptHandler.handle");
            CiValue[] values = new CiValue[frame.values.length];
            for (int i = 0; i < values.length; i++) {
                values[i] = CiVirtualObject.proxy(frame.values[i]);
            }
            CiVirtualObject local = CiVirtualObject.createObjectArray(values);
            return new CiFrame(method, new CiValue[]{local});
        }
        return frame;
    }
}
```

# DLR within the JVM (4)

## Direct Deoptimization into the DLR Interpreter

```
int test() {  
    int sum = 0;  
    for (int i = 0; i < N; i = safeAdd(i, 1)) {  
        sum = safeAdd(sum, i);  
    }  
    return sum;  
}
```

↓  
Jump on  
overflow / underflow  
↓

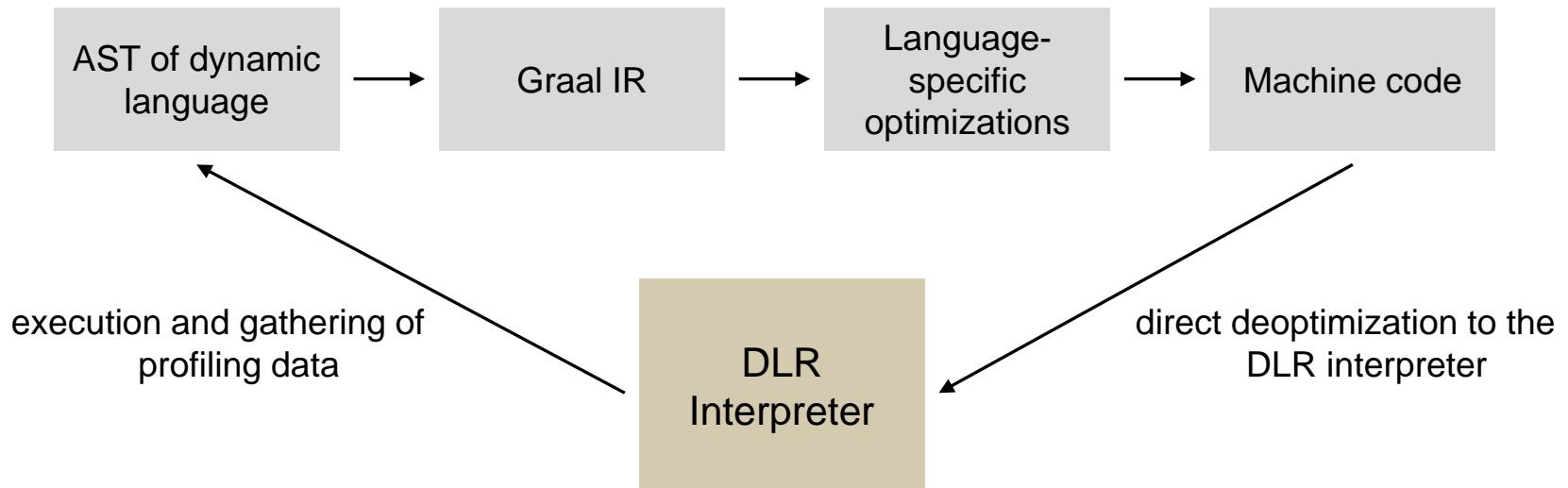
```
public class DeoptHandler {  
    public static int handle(Object[] values) {  
        // Call DLR interpreter with the state (values).  
        return 0;  
    }  
}
```

ORACLE®

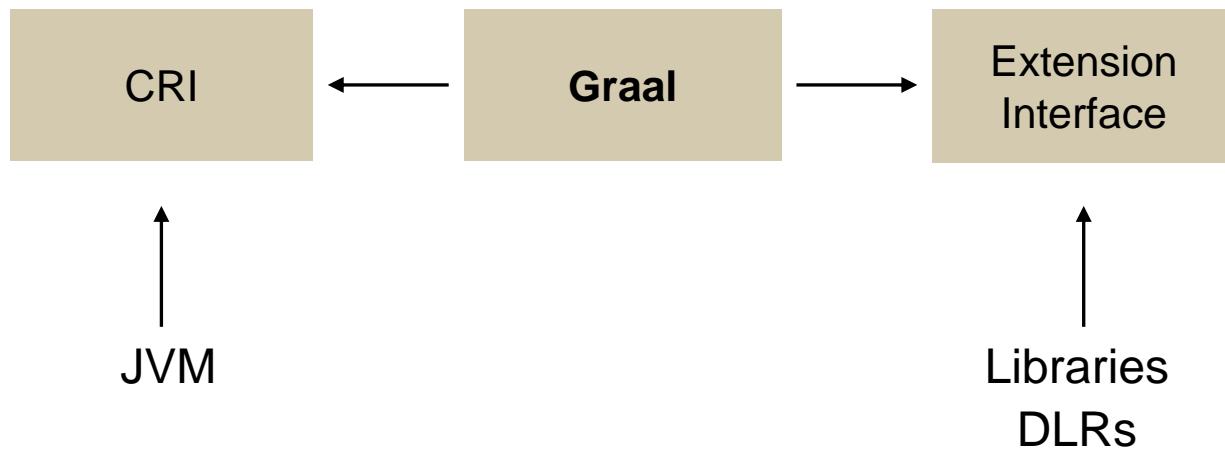
# DLR within the JVM (5)

## Direct Deoptimization into the DLR Interpreter

- The generated method is never executed in the interpreter => No need for bytecodes.



# Summary



# Acknowledgements

- Implementation
  - Lukas Stadler (JKU Linz)
  - Gilles Duboscq (Intern)
- Maxine Team
  - Mario Wolczko (manager)
  - Doug Simon
  - Laurent Daynès
  - Michael Van De Vanter
  - Mick Jordan
  - Michael Haupt
  - Christian Wimmer
- Discussions
  - John Rose
  - Tom Rodriguez
  - Peter Kessler
- Other
  - Ben Titzer

# Q&A

## Workshop...

ORACLE®

# **Hardware and Software**



## **Engineered to Work Together**



**ORACLE®**