GraalVM – A compiler developer’s perspective

David Leopoldseder
Senior Researcher
Oracle Labs Austria, GraalVM Compiler Team
November 06, 2019
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 Early Adopter Status

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.
Agenda

1. What is GraalVM?
2. What is the GraalVM Compiler?
3. Graal IR – Diving into the Sea of Nodes
4. Scheduling the Sea of Nodes
5. Optimizing GraalIR
6. Try it out yourself!
Agenda

1. What is GraalVM?
2. What is the GraalVM Compiler?
3. Graal IR – Diving into the Sea of Nodes
4. Scheduling the Sea of Nodes
5. Optimizing GraalIR
6. Try it out yourself!
GraalVM magic in one tweet

Javac is a Java compiler written in Java. GraalVM is a full-blown Java compiler and VM written in Java. You can use a Java compiler written in Java, to compile another Java compiler written in Java, to native code, boosting its performance. Tell me your mind is not blown.

2:22 PM - 29 Apr 2019

26 Retweets 114 Likes
Why GraalVM?

- **High Performance**
  Optimize application performance with GraalVM compiler

- **Fast Startup**
  Compile your application AOT and start instantly

- **Polyglot**
  Mix & match languages with seamless interop

- **Open Source**
  See what’s inside, track features progress, contribute
Production-Ready

Pinned Tweet

GraalVM @graalvm · May 9

First production release - we are stoked to introduce GraalVM 19.0! 🚀🏆
Here's the announcement: medium.com/graalvm/announ…. Check out the release notes: graalvm.org/docs/release-n… and get the binaries:
Architecture

- JS implementation
- Python implementation
- R implementation
- ...

Language Implementation API

GraalVM Compiler

HotSpot VM
Architecture

- JVM Languages
  - Groovy
  - Clojure
  - Java
  - Kotlin
  - Scala

- HotSpot VM

- Language Implementation API (Truffle)
  - Truffle Languages
    - JavaScript
    - Red
    - R
    - Python
    - Sulong

- GraalVM Compiler
Community Edition (CE)

GraalVM CE is available for free for development and production use. It is built from the GraalVM sources available on GitHub. We provide pre-built binaries for GraalVM CE for Linux on x86 64-bit systems.

DOWNLOAD FROM GITHUB

Enterprise Edition (EE)

GraalVM EE provides additional performance, security, and scalability relevant for running critical applications in production. It is free for evaluation uses and available for download from the Oracle Technology Network. We provide binaries for GraalVM EE for Linux or Mac OS X on x86 64-bit systems.

DOWNLOAD FROM OTN
Agenda

1. What is GraalVM?
2. What is the GraalVM Compiler?
3. Graal IR – Diving into the Sea of Nodes
4. Scheduling the Sea of Nodes
5. Optimizing GraalIR
6. Try it out yourself!
GraalVM compiler

- Java JIT Compiler implemented itself in Java
- Highly Modular Design

- Novel Optimizations
- Foundation for more than 60 publications at premier venues (PLDI, CGO, OOPSLA, ...)
- Strong open source development and community
GraalVM compiler

- Highly Optimizing
- Based on Deoptimization & Speculation
- Various Optimizations
  - Peephole (Several Levels)
  - Constant Folding
  - Inlining
  - Partial Escape Analysis
  - Lock Elimination
  - Duplication
  - Loop Unrolling/Unswitching/Peeing/LICM/

- Null Check Elimination
- Constant InstanceOf Checks
- Safepoint Elimination
- Compressed Pointers
- Biased Locking
- Speculative Optimizations
  - Only compile executed code
  - Only compile executed exception paths
  - Class Hierarchy Analysis
Java dynamic execution
Tiered compilation

Deoptimize

Interpreter

C1 Compiler

Graal

Optimize
Agenda

1. What is GraalVM?
2. What is the GraalVM Compiler?
3. Graal IR – Diving into the Sea of Nodes
4. Scheduling the Sea of Nodes
5. Optimizing GraalIR
6. Try it out yourself!
Compiler IRs

- **Single-Pass Compilers**
- **Multi-Pass Compilers**
  
  Requires IR that supports

  - Optimizations in the Front End
  - Allows for straightforward code generation in the backend

  Vastly different goals
A choice of intermediate representations

- **CFG-Based**
  - LLVM
  - GCC
  - V8 Crankshaft
  - HotSpot C1
  - ...

- **Graph-Based**
  - HotSpot C2
  - V8 Turbofan
  - GraalVM Compiler
Graph-Based IRs

- What is a program?
  What do to? → Control → Flow of Control → Control-flow Graph
  What to use as input? → Data → Flow of Data → Data-Flow Graph

- Sea-Of-Nodes Idea
  Proposed by Cliff Click for the HotSpot Server Compiler
  Represent program as a graph
  Nodes are instructions/operations
  Edges are dependencies (control/data)
  No basic blocks
    Easier GVN
    Easier DCE
    ....

Basically a set of dependencies
Graal IR

- Based on sea of nodes idea
  - Tightened semantics
  - No floating control

- SSA-Form Intermediate Representation

- Superposition of 3 Graphs
  - Control Flow Graph
    - Red Edges
  - Data Flow Graph
    - Blue Edges
  - Memory Graph
    - Green Edges (not in picture)

```java
int addMin(int a, int b, int sum) {
    int min;
    if (a > b) {
        min = b;
    } else {
        min = a;
    }
    return min + sum;
}
```
Fixed VS Floating

- 2 Types of Nodes
  - Fixed Nodes
    - Fixed nodes in the CFG
  - Floating nodes
    - “Floating” – Only inputs determine final location
    - Side-effect free instructions
    - Can be re-executed every time
    - topological sorting

```c
int addMin(int a, int b, int sum) {
    int min;
    if (a > b) {
        min = b;
    } else {
        min = a;
    }
    return min + sum;
}
```
Loops

- Explicit IR Nodes
- Reducible Loops only (simplifies dominance computations)
GraalVM IR

static int S;
static int foo(int a) {
    for (int i = 0; i < a; i++) {
        S += i * a;
    }
    return S;
}
GraalVM compiler

Bytecode → HIR Generation → Lowering → IR High Tier → Lowering → IR Mid Tier → Lowering → IR Low Tier → LIR Generation → LIR → Code Generation → Machine Code → Execution

Frontend

- Bytecode
- Constant Folding
- Memory Optimization
- Dead Code Elim
- CFG Optimizations
- Bytecode independent machine level
- Platform independent machine level

IR High Tier

- GVN
- Guard Optimizations
- Scheduling
- Null Check Removal
- Platform dependent machine level

IR Mid Tier

- Inlining
- Lock Optimizations

IR Low Tier

- Partial Escape Analysis
- Guard Optimizations
- Partial Redundancy Elim

Platform

- Read Elimination
- Conditional Elimination
- Duplication
- Loop Optimizations
- Dead Code Elim
- CFG Optimizations
- Null Check Removal
- Platform dependent machine level

Backend
GraalVM compiler development tooling

- **Bytecode**
  - HIR Generation
  - Lowering

- **IR High Tier**
  - Constant Folding
  - GVN
  - Inlining
  - Partial Escape Analysis
  - Read Elimination
  - Conditional Elimination
  - Duplication
  - Loop Optimizations
  - ... (more optimizations)

- **IR Mid Tier**
  - Memory Optimization
  - Guard Optimizations
  - Lock Optimizations
  - ... (more optimizations)

- **IR Low Tier**
  - Dead Code Elim
  - Scheduling
  - Partial Redundancy Elim
  - ... (more optimizations)

- **LIR**
  - Code Generation
  - CFG Optimizations
  - Null Check Removal
  - ... (more optimizations)

- **Machine Code**
  - Execution

---

**Frontend**
- Bytecode Level
- Platform independent machine level

**Backend**
- Platform dependent machine level

---

**Ideal Graph Visualizer**

**C1 Visualizer**
Agenda

1. What is GraalVM?
2. What is the GraalVM Compiler?
3. Graal IR – Diving into the Sea of Nodes
4. Scheduling the Sea of Nodes
5. Optimizing GraalIR
6. Try it out yourself!
Remember?

- SSA-Form Intermediate Representation
- Superposition of 3 Graphs
  - Control Flow Graph
  - Data Flow Graph
  - Memory Graph

```c
int addMin(int a, int b, int sum) {
    int min;
    if (a > b) {
        min = b;
    } else {
        min = a;
    }
    return min + sum;
}
```
Problem

- How to get a sequential program out of this?
Topologically sort all floating nodes
Agenda

1. What is GraalVM?
2. What is the GraalVM Compiler?
3. Graal IR – Diving into the Sea of Nodes
4. Scheduling the Sea of Nodes
5. Optimizing GraalIR
6. Try it out yourself!
GraalVM Compiler

- Highly Optimizing
- Based on Deoptimization & Speculation
- Various Optimizations
  - Peephole (Several Levels)
  - Constant Folding
  - Inlining
  - Partial Escape Analysis
  - Lock Elimination
  - Duplication
  - Loop Unrolling/Unswitching/Peeling/LICM/…

  Null Check Elimination
  Constant Instanceof Checks
  Safepoint Elimination
  Compressed Pointers
  Biased Locking
  Speculative Optimizations
  - Only compile executed code
  - Only compile executed exception paths
  - Class Hierarchy Analysis
  ….
Escape Analysis

- **Compiler analysis used in optimizations**
  
  Objects are always allocated

  Objects “escape” the current scope if they leave the compiler’s field of view (i.e., the current compilation unit)

  Escaping means

  - Used as a parameter
  - Written to static field
  - Return value
  - Thrown
  - .....

Escape Analysis Example

```java
static class Example {
    int x;
    public Example(int x) { this.x = x; }
}
static Object SideEffect;
static final Example Cached = new Example(12);
public Example foo(int x) {
    Example e = new Example(x);
    if (e.x % 2 == 0) {
        System.out.println("Foo");
    } else {
        System.out.println(e);
    }
    if (e.x % 3 == 0) {
        SideEffect = e;
    }
    if (e.x == 17) {
        return e;
    }
    return Cached;
}
```
Escape Analysis

- What to do with this knowledge

  Object Allocations are expensive
  Want to avoid them if possible

  **Scalar Replacement**
  Replace the usage of an object with its fields
Scalar Replacement

static class Point {
    int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    Point add(Point p) {
        return new Point(this.x + p.x, this.y + p.y);
    }
    int toInt() {
        return x * y;
    }
}

static int pointUsage(int x1, int y1, int x2, int y2) {
    Point p1 = new Point(x1, y1);
    Point p2 = new Point(x2, y2);
    return p1.add(p2).toInt();
}

static int pointUsageInlined(int x1, int y1, int x2, int y2) {
    Point p1 = new Point(x1, y1);
    Point p2 = new Point(x2, y2);
    Point tmp = new Point(p1.x + p2.x, p1.y + p2.y);
    return tmp.x * tmp.y;
}

static int pointUsageScalar(int x1, int y1, int x2, int y2) {
    int x = x1 + x2;
    int y = y1 + y2;
    return x * y;
}
Escape Analysis & Scalar Replacement

- **Multiple Possible Levels**
  - Object does not escape compilation unit
    - Object allocations are removed & fields are replaced with their *stack allocated* values (typically reside in registers in the compiled code)
  
  Object does not escape thread
  - **Lock Removal**: If no other thread can see the object, we can remove locking

  Object escapes
  - No Optimization possible (but optimized allocation → TLABs)
Traditional Escape Analysis

- Traditional escape analysis is done on an entire compilation unit
  If an object escapes anywhere in the scope, scalar replacement is prohibited

```java
public Example fooSimple(int x) {
    Example e = new Example(x);
    if (e.x % 2 == 0) {
        System.out.println("Foo");
        return Cached;
    } else {
        return e;
    }
}
```

But escaping `e` in false branch disables EA and scalar replacement on the entire method.
What if?

- What if the code at runtime hardly requires the allocation?

```java
public Example fooSimple(int x) {
    Example e = new Example(x);
    if (e.x % 2 == 0) {
        System.out.println("Foo");
        return Cached;
    } else {
        return e;
    }
}
```

What if the probability for the true successor is 99%? → We would allocate although we only need the new e in 1% of the cases.

- e escapes
- e does not escape
Partial Escape Analysis

- What if the code at runtime hardly requires the allocation?

```java
class Example {
    int x;
    public Example(int x) {
        this.x = x;
    }
    public String toString() {
        return "Example: \n\t\t\tx = " + x;
    }
}

public Example fooSimple(int x) {
    Example e = new Example(x);
    if (e.x % 2 == 0) {
        System.out.println("Foo");
        return Cached;
    } else {
        return e;
    }
}

public Example fooSimple_PEA(int x) {
    if (x % 2 == 0) {
        System.out.println("Foo");
        return Cached;
    } else {
        return new Example(x);
    }
}
```

e escapes

e does not escape

e still escapes but the allocation is only done in 1% of the cases
Agenda

1. What is GraalVM?
2. What is the GraalVM Compiler?
3. Graal IR – Diving into the Sea of Nodes
4. Scheduling the Sea of Nodes
5. Optimizing GraalIR
6. Try it out yourself!
Further Reading

GraalVM

- Twitter [https://twitter.com/graalvm](https://twitter.com/graalvm)
- GraalVM CE Github [https://github.com/oracle/graal](https://github.com/oracle/graal)
- GraalVM Website [https://www.graalvm.org/](https://www.graalvm.org/)
- Publications [https://www.graalvm.org/community/publications/](https://www.graalvm.org/community/publications/)
- Tutorials [https://www.graalvm.org/community/videos/](https://www.graalvm.org/community/videos/)
GraalVM Internship Program

- https://www.graalvm.org/community/internship/
- Apply before November 30th
GraalVM™
Thank you

David Leopoldseder
Senior Researcher
Oracle Labs Austria, GraalVM Compiler Team