The Java HotSpot VM
Under the Hood

Tobias Hartmann

Compiler Group – Java HotSpot Virtual Machine
Oracle Corporation

May 2018
About me

• Software engineer in the HotSpot JVM Compiler Team at Oracle
  – Based in Baden, Switzerland

• Master’s degree in Computer Science from ETH Zurich

• Worked on various compiler-related projects
  – Currently working on future Value Type support for Java
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, and timing of any features or functionality described for Oracle’s products remains at the sole discretion of Oracle.
Outline

• Intro: Why virtual machines?

• Part 1: The Java HotSpot VM
  – JIT compilation in HotSpot
  – Tiered Compilation

• Part 2: What's new in Java
  – Segmented Code Cache
  – Compact Strings
  – Ahead-of-time Compilation
  – Value Types
A typical computing platform

User Applications

Application Software

- Spring
- Apache Sling
- ...
- ...

System Software

- Java SE
- Java EE
- Java Virtual Machine
- Operating system
- Hardware
A typical computing platform

- **Hardware**
- **User Applications**
  - Spring
  - Apache Sling
  - ...
  - ...
- **Application Software**
- **System Software**
A typical computing platform

User Applications

Application Software

Spring
Apache Sling
...
...

Java SE
Java EE

Java Virtual Machine

System Software

Operating system

Hardware
Programming language implementation

Programming language

Language implementation

Operating system

Hardware

C

Compiler
Standard libraries
Debugger
Memory management

Windows

Intel x86

Compiler
Standard libraries
Debugger
Memory management

Solaris

SPARC
(Language) virtual machine

Programming language
- Java
- JavaScript
- Scala
- Python

Virtual machine
- HotSpot VM

Operating system
- Windows
- Linux
- Mac OS X
- Solaris

Hardware
- Intel x86
- PPC
- ARM
- SPARC
Outline

• **Intro: Why virtual machines?**

• **Part 1: The Java HotSpot VM**
  – JIT compilation in HotSpot
  – Tiered Compilation

• **Part 2: What's new in Java**
  – Segmented Code Cache
  – Compact Strings
  – Ahead-of-time Compilation
  – Value Types
The JVM: An application developer’s view

Java source code

```java
int i = 0;
do {
   i++;
} while (i < f());
```

Bytecodes

0: iconst_0
1: istore_1
2: iinc
5: iload_1
6: invokevirtual f
9: if_icmplt 2
12: return

HotSpot Java VM

• Ahead-of-time
  • Using javac

• Instructions for an abstract machine
  • Stack-based machine (no registers)
The JVM: A VM engineer’s view

Bytecodes
0: iconst_0
1: istore_1
2: iinc
5: iload_1
6: invokestatic f
9: if_icmplt 2
12: return

HotSpot Java VM

Compilation system
C1
C2

Compiled method
Machine code
Debug info
Object maps

Garbage collector

Heap
Stack

Interpreter

compile
produce
manage
execute
access
access
Outline

• Intro: Why virtual machines?

• Part 1: The Java HotSpot VM
  – JIT compilation in HotSpot
  – Tiered Compilation

• Part 2: What's new in Java
  – Segmented Code Cache
  – Compact Strings
  – Ahead-of-time Compilation
  – Value Types
Interpretation vs. compilation in HotSpot

**Template-based interpreter**
- Generated at VM startup (before program execution)
- Maps a well-defined machine code sequence to every bytecode instruction

```
Bytecodes
0: iconst_0
1: istore_1
2: iinc
5: iload_1
6: invokevirtual +
9: if_icmplt 2
12: return
```

```
Machine code
mov  -0x8(%r14), %eax
movzb1 0x1(%r13), %ebx
inc  %r13
mov  $0xff40,%r10
jmpq  *(%r10, %rbx, 8)
```

**Compilation system**
- Speedup relative to interpretation: ~100X
- Two *just-in-time compilers* (C1, C2)
- Aggressive optimistic optimizations
Ahead-of-time vs. just-in-time compilation

- **AOT**: Before program execution
- **JIT**: During program execution
  - Tradeoff: Resource usage vs. performance of generated code
JIT compilation in HotSpot

• Resource usage vs. performance
  – Getting to the “sweet spot”

1. Selecting methods to compile
2. Selecting compiler optimizations
1. Selecting methods to compile

- **Hot methods** (frequently executed methods)
- **Profile** method execution
  - # of method invocations, # of backedges
- **A method’s lifetime in the VM**
Example optimization: Hot path compilation

Control flow graph

Generated code

guard(x > 3)
S_1;
S_2;
S_3;
S_4;
S_8;
S_9;
S_10’000

Deoptimize
Example optimization: Virtual call inlining

Class hierarchy

```java
class A {
    void bar() {
        S1;
    }
}

class B extends A {
    void bar() {
        S2;
    }
}
```

Method to be compiled

```java
void foo() {
    A a = create(); // return A or B
    a.bar();
}
```

Compiler: Inline call? Yes.
Example optimization: Virtual call inlining

- **Benefits of inlining**
  - Virtual call avoided
  - Code locality

- **Optimistic assumption: only A is loaded**
  - Note dependence on class hierarchy
  - Deoptimize if hierarchy changes

```java
class A {
    void bar() {
        S1;
    }
}
class B extends A {
    void bar() {
        S2;
    }
}

void foo() {
    A a = create(); // return A or B
    S1;
}
```
Example optimization: Virtual call inlining

Class hierarchy:

```java
class A {
    void bar() {
        S1;
    }
}
class B extends A {
    void bar() {
        S2;
    }
}
```

Method to be compiled:

```java
void foo() {
    A a = create(); // return A or B
    a.bar();
}
```

Compiler: Inline call? No.
Deoptimization

• Compiler’s **optimistic assumption** proven wrong
  – Assumptions about class hierarchy
  – Profile information does not match method behavior

• **Switch execution from compiled code to interpretation**
  – **Reconstruct state of the interpreter** at runtime
  – Complex implementation

• **Compiled code**
  – Possibly **thrown away**
  – Possibly reprofiled and recompiled
Performance effect of deoptimization

• Follow the variation of a method’s performance
JIT compilation in HotSpot

• Resource usage vs. performance
  – Getting to the “sweet spot”

1. Selecting methods to compile
2. Selecting compiler optimizations
2. Selecting compiler optimizations

- **C1 compiler**
  - Limited set of optimizations
  - Fast compilation
  - Small footprint

- **C2 compiler**
  - Aggressive optimistic optimizations
  - High resource demands
  - High-performance code

- **Graal**
  - Part of HotSpot for AOT since JDK 9
  - Available as experimental C2 replacement in JDK 11

Client VM

Server VM

Tiered Compilation (enabled since JDK 8)
Outline

• Why virtual machines?

• Part 1: The Java HotSpot VM
  – JIT compilation in HotSpot
  – Tiered Compilation

• Part 2: What's new in Java
  – Segmented Code
  – Compact Strings
  – Ahead-of-Time Compilation
  – Value Types
Tiered Compilation

• Introduced in JDK 7, enabled by default in JDK 8

• Combines the benefits of
  – Interpreter: Fast startup
  – C1: Fast compilation
  – C2: High peak performance

• Within the sweet spot
  – Faster startup
  – More profile information
Benefits of Tiered Compilation

Client VM (C1 only)

- **Interpreted**
  - VM Startup
  - Compilation
- **C1-compiled**
  - Performance
  - Warm-up time

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.
Benefits of Tiered Compilation

Server VM (C2 only)

Performance

VM Startup

Compilation

Time

Interpreted

C2-compiled

warm-up time
Benefits of Tiered Compilation

**Tiered compilation**

- **Interpreted**
- **C1-compiled**
- **C2-compiled**

![Graph showing performance and time with VM Startup, Compilation, and Compilation stages with warm-up time indicated.](image)
Additional benefit: More accurate profiling

Profiling without tiered compilation

Interpreter

100 samples

300 samples

200 samples

Interpreter

100 samples

1000 samples

C1 (profiled)

C2 (non-profiled)

Profiling with tiered compilation

w/o tiered compilation: 300 samples gathered

w/ tiered compilation: 1’100 samples gathered
Tiered Compilation

• Combined benefits of interpreter, C1, and C2

• Additional benefits
  – More accurate profiling information

• Drawbacks
  – Complex implementation
  – Careful tuning of compilation thresholds needed
  – More pressure on code cache
A method’s lifetime (Tiered Compilation)

- **Interpreter**
  - Collect profiling information

- **C1**
  - Generate code quickly
  - Continue collecting profiling information

- **C2**
  - Generate high-quality code
  - Use profiling information

- **Code cache**
- **Deoptimization**
Performance of a method (Tiered Compilation)

![Graph showing performance over time with stages: VM Startup, Compilation, C1 compiled, C2 compiled, Deoptimization, C2 compiled.](image)
Compilation levels (detailed view)

Typical compilation sequence

Compilation level

4
C2

3
C1: full profiling

2
C1: limited profiling

1
C1: no profiling

0
Interpreter

Associated thresholds:
- Tier4InvocationThreshold
- Tier4MinInvocationThreshold
- Tier4CompileThreshold
- Tier4BackEdgeThreshold

Associated thresholds:
- Tier3InvokeNotifyFreqLog
- Tier3BackedgeNotifyFreqLog
- Tier3InvocationThreshold
- Tier3MinInvocationThreshold
- Tier3BackEdgeThreshold
- Tier3CompileThreshold
Outline

• Intro: Why virtual machines?

• Part 1: The Java HotSpot VM
  – JIT compilation in HotSpot
  – Tiered Compilation

• Part 2: What's new in Java
  – Segmented Code Cache
  – Compact Strings
  – Ahead-of-time Compilation
  – Value Types
What is the code cache?

- **Stores code** generated by JIT compilers
- Continuous chunk of memory
  - Managed (similar to the Java heap)
  - Fixed size
- **Essential for performance**
Code cache usage: JDK 6 and 7
Code cache usage: JDK 8 (Tiered Compilation)
Code cache usage: JDK 9

- Free space
- VM internals
- C1 compiled (profiled)
- C2 compiled (non-profiled)
Challenges

• Tiered compilation increases amount of code by up to 4X
• All code is stored in a single code cache
• High fragmentation and bad locality

• But is this a problem in real life?
Code cache usage: Reality
Code cache usage: Reality

- Free space
- Profiled code
- Non-profiled code

Hotness scale:
- 500
- 480
- 460
- 440
- 420
- 400
- 380
- 360
### Design: Types of compiled code

<table>
<thead>
<tr>
<th></th>
<th>Optimization level</th>
<th>Size</th>
<th>Cost</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-method code</td>
<td>optimized</td>
<td>small</td>
<td>cheap</td>
<td>immortal</td>
</tr>
<tr>
<td>Profiled code (C1)</td>
<td>instrumented</td>
<td>medium</td>
<td>cheap</td>
<td>limited</td>
</tr>
<tr>
<td>Non-profiled code (C2)</td>
<td>highly optimized</td>
<td>large</td>
<td>expensive</td>
<td>long</td>
</tr>
</tbody>
</table>
Design

• Without Segmented Code Cache
  
  | Code Cache |

• With Segmented Code Cache
  
  | non-profiled methods |
  | profiled methods |
  | non-methods |
Segmented Code Cache: Reality

profiled methods

non-profiled methods

- free space
- profiled code
- non-profiled code
Segmented Code Cache: Reality

Profiled methods vs. non-profiled methods.
Evaluation: Code locality

```java
public abstract class A {
    abstract public int amount();
}

private final A[] targets = new A[SIZE];

@Benchmark
@OperationsPerInvocation(SIZE)
public int sum() {
    int s = 0;
    for (A i : targets) {
        s += i.amount();
    }
    return s;
}
```

**Code Cache**

- Profiled code
  - targets[0].amount()
  - targets[1].amount()
  - targets[2].amount()

- Non-profiled code
  - targets[0].amount()
Evaluation: Code locality

```java
public abstract class A {
    abstract public int amount();
}

private final A[] targets = new A[SIZE];

@Benchmark
@OperationsPerInvocation(SIZE)
public int sum() {
    int s = 0;
    for (A i : targets) {
        s += i.amount();
    }
    return s;
}
```

**Code Cache**

- targets[0].amount()
- targets[1].amount()
- targets[2].amount()

- **profiled code**
- **non-profiled code**
Evaluation: Code locality

![Graph showing speedup in % against number of call targets. The graph distinguishes between L1 ITLB and L2 STLB.]
Evaluation: Code locality

• Instruction Cache (ICache)
  – **14% less** ICache misses

• Instruction Translation Lookaside Buffer (ITLB\(^1\))
  – **44% less** ITLB misses

• Overall performance
  – **9% speedup** with microbenchmark

\(^1\) caches virtual to physical address mappings to avoid slow page walks
Evaluation: Responsiveness

• Sweeper (GC for compiled code)
Evaluation: Performance

![Performance Improvement Chart]

- SPECjbb2005
- SPECjbb2013
- JMH-Javac
- Octane (Typescript)
- Octane (Gbemu)
What we have learned

• **Segmented Code Cache helps**
  – To reduce the sweeper overhead and improve responsiveness
  – To reduce memory fragmentation
  – To improve code locality

• **And thus improves overall performance**

• Released with JDK 9
Outline

• Intro: Why virtual machines?
• Part 1: What's cool in Java 8
  – Background: JIT compilation in HotSpot
  – Tiered Compilation
• Part 2: What's new in Java
  – Segmented Code Cache
  – Compact Strings
  – Ahead-of-Time Compilation
  – Value Types
public class HelloWorld {
    public static void main(String[] args) {
        String myString = "HELLO";
        System.out.println(myString);
    }
}

public final class String {
    private final char value[];
    ...
}

color-box
char value[] = 0x0048 0x0045 0x004C 0x004C 0x004F
2 bytes

UTF-16 encoded
“Perfection is achieved, not when there is nothing more to add, but when there is nothing more to take away.”

— Antoine de Saint Exupéry
There is a lot to take away here..

• UTF-16 encoded Strings always occupy two bytes per char

• Wasted memory if only Latin-1 (one-byte) characters used:

```c
char value[] = {0x0048, 0x0045, 0x004C, 0x004C, 0x004F};
```

• But is this a problem in real life?
Real life analysis: char[] footprint

• 950 heap dumps from a variety of applications
  – char[] footprint makes up **10% - 45% of live data**
  – Majority of characters are **single byte**

• Predicted footprint reduction of **5% - 10%**
Project Goals

• Memory footprint reduction by improving space efficiency of Strings
• Meet or beat performance of JDK 9
• Full compatibility with related Java and native interfaces
• Full platform support
  – x86/x64, SPARC, ARM
  – Linux, Solaris, Windows, Mac OS X
Design

• String class now uses a byte[] instead of a char[]

```java
public final class String {
    private final byte value[];
    private final byte coder;
    ...
}
```

• Additional 'coder' field indicates which encoding is used

```java
byte value[] = 0x00 0x48 0x00 0x45 0x00 0x4C 0x00 0x4C 0x00 0x4F  // UTF-16 encoded

byte value[] = 0x48 0x45 0x4C 0x4C 0x4F  // Latin-1 encoded
```
Design

• If all characters have a zero upper byte
  → String is compressed to Latin-1 by stripping off high order bytes

• If a character has a non-zero upper byte
  → String cannot be compressed and is stored UTF-16 encoded

byte value[] = \[0x47, 0x48, 0x00, 0x45, 0x00, 0x4C, 0x4C, 0x4F\] \# UTF-16 encoded

byte value[] = \[0x48, 0x45, 0x4C, 0x4C, 0x4F\] \# Latin-1 encoded
Design

• Compression / inflation needs to fast

• Requires HotSpot support in addition to Java class library changes
  – JIT compilers: Intrinsics and String concatenation optimizations
  – Runtime: String object constructors, JNI, JVMTI
  – GC: String deduplication

• Kill switch to enforce UTF-16 encoding (-XX:-CompactStrings)
  – For applications that extensively use UTF-16 characters
Microbenchmark: LogLineBench

```java
public class LogLineBench {
    int size;

    String method = generateString(size);

    public String work() throws Exceptions {
        return "\[" + System.nanoTime() + "] " + Thread.currentThread().getName() + "Calling an application method \"" + method + "\" without fear and prejudice."
    }
}
```
LogLineBench results

<table>
<thead>
<tr>
<th></th>
<th>Performance ns/op</th>
<th>Allocated b/op</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Baseline</td>
<td>149</td>
<td>153</td>
</tr>
<tr>
<td>CS disabled</td>
<td>152</td>
<td>150</td>
</tr>
<tr>
<td>CS enabled</td>
<td>142</td>
<td>139</td>
</tr>
</tbody>
</table>

- Kill switch works (no regression)
- 27% performance improvement and 46% footprint reduction
Evaluation: Performance

• SPECjbb2005
  – 21% footprint reduction
  – 27% less GCs
  – 5% throughput improvement

• SPECjbb2015
  – 7% footprint reduction
  – 11% critical-jOeps improvement

• Weblogic (startup)
  – 10% footprint reduction
  – 5% startup time improvement

• Released with JDK 9
Outline

• Intro: Why virtual machines?
• Part 1: The Java HotSpot VM
  – JIT compilation in HotSpot
  – Tiered Compilation
• Part 2: What's new in Java
  – Segmented Code Cache
  – Compact Strings
  – Ahead-of-time Compilation
  – Value Types
Ahead-of-Time Compilation

- Compile Java classes to native code prior to launching the VM
- AOT compilation is done by new jaotc tool
  - Uses Java based Graal compiler as backend
  - Stores code and metadata in shared object file
- Improves start-up time
  - Limited impact on peak performance
- Sharing of compiled code between VM instances
Revisit: Performance of a method (Tiered Compilation)
Performance of a method (Tiered AOT)

- **AOT compiled**
- **C1 compiled**
- **C2 compiled**
- **Interpreted**
- **C2 compiled**

- **Time**

- **Performance**

- **VM Startup**
- **Compilation**
- **Compilation**
- **Deoptimization**
- **Compilation**
Ahead-of-Time Compilation

• Experimental feature
  – Supported on Linux x64
  – Limited to the java.base module

• Try with your own code - feedback is welcome!

• Released with JDK 9
  – More to come in future releases
Outline

• Intro: Why virtual machines?

• Part 1: The Java HotSpot VM
  – JIT compilation in HotSpot
  – Tiered Compilation

• Part 2: What's new in Java
  – Segmented Code Cache
  – Compact Strings
  – Ahead-of-time Compilation
  – Value Types
Value Types

• Value types are immutable, identityless aggregates
  – User defined primitives
  – Non-synchronizable, non-nullable
  – “Codes like a class, works like an int!”

• Introduced for performance
  – Better spatial locality (no indirection, no header)
  – Avoid heap allocations to reduce GC pressure
  – Properties enable JIT optimizations (for example, scalarization)
Minimal Value Types (MVT)

• Language changes are difficult
  – Provide early access to a subset of value type features
  – Without language support
  – EA build is out http://jdk.java.net/valhalla/

• Still affects many JVM components
  – GCs, compilers, JNI, JVMTI, reflection, serviceability, class loading, ...
  – ... and we should not break existing code/optimizations
Minimal Value Types

- User defines **Value Capable Class (VCC)** with annotation
  - Value type (DVC) is then derived by JVM at runtime

```
VCC_{source} -> javac -> VCC_{classfile} -> class loader
      |            |                          |
      |            |                          |
      |            |                          |
      |            |
DVC_{class}  VCC_{class}
```
Working with derived value classes

• Use new **value type bytecodes**
  – Without javac support
  – For example, through ASM
  – vload, vstore, vreturn, ...

• Error prone but good for experts

• Use **Java method handle API**
  – MethodHandles::arrayElementSetter,
    ValueType::defaultValueConstant,
    ValueType::findWither, ...

• **Difficult to write complex code**
Value Type Bytecodes

<table>
<thead>
<tr>
<th>Bytecode</th>
<th>Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>vload</td>
<td>Load value from local</td>
</tr>
<tr>
<td>vstore</td>
<td>Store value to local</td>
</tr>
<tr>
<td>vreturn</td>
<td>Return value from method</td>
</tr>
<tr>
<td>vaload</td>
<td>Load value from value array (flattened or not)</td>
</tr>
<tr>
<td>vastore</td>
<td>Store value to value array (flattened or not)</td>
</tr>
<tr>
<td>vbox</td>
<td>Convert a value to a reference</td>
</tr>
<tr>
<td>vunbox</td>
<td>Convert a reference to a value</td>
</tr>
<tr>
<td>vdefault</td>
<td>Create a default value (all-zero)</td>
</tr>
<tr>
<td>vwithfield</td>
<td>Create a new value from an existing value, with an updated field</td>
</tr>
</tbody>
</table>
## Method Handles

<table>
<thead>
<tr>
<th>Bytecode</th>
<th>Corresponding MethodHandle</th>
</tr>
</thead>
<tbody>
<tr>
<td>vaload</td>
<td>MethodHandles::arrayElementGetter</td>
</tr>
<tr>
<td>vastore</td>
<td>MethodHandles::arrayElementSetter</td>
</tr>
<tr>
<td>vbox</td>
<td>ValueType::box</td>
</tr>
<tr>
<td>vunbox</td>
<td>ValueType::unbox</td>
</tr>
<tr>
<td>vdefault</td>
<td>ValueType::defaultValueConstant</td>
</tr>
<tr>
<td>vwithfield</td>
<td>ValueType::findWither</td>
</tr>
<tr>
<td>anewarray</td>
<td>MethodHandles::arrayConstructor</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Beyond MVT: Experimental javac support

```java
__ByValue final class MyValue {
    final int x, y;

    __ValueFactory static MyValue createDefault() {
        return __MakeDefault MyValue1(); // vdefault
    }

    __ValueFactory static MyValue setX(MyValue v, int x) {
        v.x = x; // vwithfield
        return v; // vreturn
    }
    ...
}
```
Storage formats

• Buffered on **Java heap**
  – With header, not a L-type box but a Q-type

• Stored in **Thread Local Value Buffer** (TLVB)
  – With header, used by the interpreter

• **Scalarized** by JIT code
  – No header, on stack or in registers

• **Flattened** array or field
  – No header, type information stored in container’s metadata
Value Type Field Flattening

```
__ByValue final class MyValue {
    final int x, y;
    ...
}

class MyObject {
    MyValue v1, v2, v3;
}
```

- No indirections: better spatial **locality**
- No pointer/header: better **density**
Value Type Field Flattening

• Only for non-static fields
• Works both for object and value type holders
• Requires pre-loading of value types to determine field size
• Flattened fields keep their layout (no intermixing)
• Optional via -XX:ValueFieldMaxFlatSize
Value Type Array Flattening

__ByValue final class MyValue1 {
    final long l;
    final byte b;
}

MyValue1[] array =

__ByValue final class MyValue2 {
    final int i;
    final String s;
    final long l;
}

MyValue2[] array =

References are spread across the array, GCs need special support to find them

padding due to long alignment
Value Type Array Flattening

• Improves spatial **locality** and **density**
• Uses **multiple memory slices** for flattened fields
• Optional via
  – XX:ValueArrayFlatten/*ElemMaxFlatSize/*ElemMaxFlatOops
  – Non flattened arrays contain oops
JIT support: Goals

• Full feature support
  – New bytecodes, optional flattening, buffering, deoptimization, OSR, incremental inlining, method handles, ...

• Pass and return value types in registers or on the stack
  – No need to retain identity

• Avoid heap allocations through aggressive scalarization

• Avoid regressions in code that does not use value types
Avoiding value type allocations

• Rely on relaxed guarantees for value types
  – No identity, all fields final, no subclassing
  – Cannot be mixed with other types

• Value type specific IR representation and optimizations
  – Takes advantage of value type properties
  – Treats value types as identityless aggregates and passes fields individually
  – **Does not rely on escape analysis!**
__ByValue final class MyValue {
    final int x, y;
    ...
}

MyValue v = __MakeDefault MyValue1();
v.x = 7;
if (b) {
    v.y = 8;
} else {
    v.y = 9;
}int i = v.y;
IR optimizations

MyValue v = __MakeDefault MyValue();
if (b) {
    staticField1 = v; // allocate
    staticField2 = v; // allocate?
}
staticField3 = v; // allocate?

• **Re-use allocations** by propagating oop
• **Use pre-allocated instance** instead of allocating default value type
// Copy detection
public method1(MyValue v1) {
    MyValue v2 = __MakeDefault MyValue();
    v2.x = v1.x;
    v2.y = v1.y;
    staticField1 = v2; // allocate
}

// Re-use dominating allocations
public method2() {
    MyValue v = __MakeDefault MyValue();
    v.x = 42;
    method1(v); // late inlined
    staticField2 = v; // allocate
}

public method1(MyValue v1) {
    staticField1 = v1;
}

public method2() {
    MyValue v = __MakeDefault MyValue();
    v.x = 42;
    staticField1 = v; // allocate
    staticField2 = v; // allocate
}
Example: Complex number using POJOs

```java
class Complex {
    public final int x, y;

    public Complex(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public double abs() {
        return Math.sqrt(x*x + y*y);
    }
}

double computePOJO(int x, int y) {
    Complex c;
    if (y > THRESHOLD) {
        c = new Complex(x, THRESHOLD);
    } else {
        c = new Complex(x, y);
    }
    return c.abs();
}
```

Assembly for computePOJO:

```
2ff8: cmp $0x2a, %ecx ; y > THRESHOLD?
2ffe: jg 306f
...
300b: cmp 0x88(%r15), %r11 ; Fast alloc?
3012: jae 30b2 ; -> slow (RT call)
3016: mov %r10d, 0xc(%rax) ; c.x = x
3020: mov %r11d, %r10d ; c.y = y
3024: mov %r10d, %r10d ; load x
3028: imul %r11d, %r10d ; c.x*c.x + c.y*c.y
302c: vcvtsi2sd %r10d, %xmm0, %xmm0
3030: vsqrtsd %xmm0, %xmm0, %xmm0 ; sqrt
...
306f: mov 0x78(%r15), %rax ; Fast alloc?
307a: cmp 0x88(%r15), %r11 ; -> slow (RT call)
3081: jae 30c9 ; -> fast (TLB)
3085: mov %r11d, 0xc(%rax) ; c.x = x
3090: movq $0x2a, 0x10(%rax) ; c.y = THRESHOLD
309c: jmp 3044
```
Example: Complex number using Value Types

```java
__ByValue final class ComplexV {
    public final int x, y;

    static ComplexV create(int x, int y) {
        ...
    }

    public double abs() {
        return Math.sqrt(x*x + y*y);
    }
}

double computeValueType(int x, int y) {
    ComplexV c;
    if (y > THRESHOLD) {
        c = ComplexV.create(x, THRESHOLD);
    } else {
        c = ComplexV.create(x, y);
    }
    return c.abs();
}
```

Assembly for `computeValueType`:

```
0x6c: cmp $0x2a,%ecx ; y > THRESHOLD?
0x6f: jg 0x90
0x71: imul %ecx,%ecx
0x74: imul %edx,%edx
0x77: add %ecx,%edx ; c.x*c.x + c.y*c.y
0x79: vcvtsi2sd %edx,%xmm0,%xmm0
0x7d: vsqrtsd %xmm0,%xmm0,%xmm0 ; sqrt ...
0x90: mov $0x6e4,%ecx ; y = THRESHOLD^2
0x95: jmp 0x74
```
When do we (still) need to allocate?

1) **Calling** a method with a value type argument
   - Solved by calling convention changes

2) **Returning** a value type
   - Solved by calling convention changes

3) **Deoptimizing** with a live value type
   - Let the interpreter take care of re-allocating
   - Similar to scalar replacement for POJOs

4) **Writing to a non-flattened** field or array element
   - Cannot avoid allocation but try to re-use existing allocations
Calling convention

1) Calling a method with a value type argument

• **Problem:** Interpreter uses buffered values, passes references at calls, expects references when called

• No need to pass value type arguments as buffer references: no identity
  – Avoid allocation/store/load at non inlined call boundaries

• **Solution:** Each field can be passed as an argument
  – method(Value v1, Value v2) compiled as method(v1.field1, v1.field2, …, v2.field1, v2.field2, …)
  – Most fields are then passed in registers
Calling convention

1) Calling a method with a value type argument

- HotSpot already uses signature specific adapters for calls
  - Handle the compiler/interpreter calling convention mismatch
  - **Extend adapters** to handle value types that are passed as fields

• No allocation/loading for c2c and i2i transitions!
Calling convention

2) Returning a value type

• **Problem 1**: Interpreter returns references, expects references from a call
• No need to return a value type as a buffer reference: no identity
  – Avoid allocations at return sides

• **Solution 1**: Value type v can be returned as v.field1, v.field2, ...
  – **No adapter available**: c2i and i2c returns are frameless
  – Interpreter now always returns fields for a value type
  – On return to interpreter: runtime call to allocate/initialize value type
  – Only if all fields fit in available registers
Calling convention

2) Returning a value type

• **Problem 2**: How do we know the return type for a value?
  – From the signature of the callee? Signature is erased for method handle linkers

• **Solution 2**: When returning a value type \( v \):
  – from compiled code, return \((v \cdot \text{class}, v.\text{field1}, v.\text{field2}, ...\))
  – from the interpreter, return \((v, v.\text{field1}, v.\text{field2}, ...\))

• Caller can then either use \( v \) or allocate a new value from \( v.\text{class} \)
Method handles/lambda forms

- **Challenging** but core part of MVT
- Lambda Forms (LF) use the value type super type: `__Value`
  - Allows sharing
  - `__Value` is a pointer, **need some translation at LF boundaries**
- **Straightforward implementation**
  - *Allocate + store* to memory when entering inlined LFs
  - Load from memory when entering inlined Java methods
  - Relies on EA to remove allocation: **limited**
Method handles/lambda forms

• Instead, when exact type of value is known, new node: **ValueTypePtr**
• Similar to **ValueTypeNode**: list of fields
• Entering LF: create **ValueTypePtrNode** from **ValueTypeNode**
• Entering Java method: create **ValueTypeNode** from **ValueTypePtrNode**
• Similar to **ValueTypeNode**: push Phi through **ValueTypePtrNode**
• First edge, pointer to buffer is mandatory: possible allocation
• If all goes well, pointers to memory are optimized out
• If not, fall back to buffered value
Challenges

• Difficult to evaluate prototype implementation
  – Limited use cases
  – Limited code/tests that uses value types (we wrote 120 compiler tests)
  – Limited benchmarks

• Method handle chains are hard to optimize
  – Limited type information due to erasure of value type signature in lambdas

• Lots of complex changes are necessary
  – C2’s type system, calling convention, ...
Limitations

• Only x86 64-bit supported
• No C1 support
  – Tiered Compilation is disabled with `-XX:+EnableMVT/EnableValhalla`
• Not all C2 intrinsics are supported yet
• Most compiler tests rely on internal javac support
Next steps/future explorations

- **Current direction:** L-world (LWVT)
  - `java.lang.Object` as super type of values
  - Values implement interfaces

- **Facilitates migration**
  - Support for type mismatches L-Type -> Q-Type
  - How to optimize calling convention?

- **Fewer new bytecodes:** `vdefault/vwithfield`

- **But several existing bytecodes have modified behavior**
  - Some are illegal for values: `monitorenter`
  - What’s the result of `acmp` on values?
Next steps/future explorations

- Extensive use of **buffered values**
  - Including compiled code
  - Must not store a reference to a buffer on the heap

- More **runtime checks**
  - Evaluate how much they cost (with legacy and value type code)

- Can **profiling** help?
  - Value/not value
  - Buffer/not buffered?
More information

- **Early access**: http://jdk.java.net/valhalla/
- **Proposal for MVT (John Rose, Brian Goetz)**
  - http://cr.openjdk.java.net/~jrose/values/shady-values.html
- **Minimal Value Types – Origins and Programming Model (Maurizio Cimadamore)**
  - https://youtu.be/xyOLHcEuhHY
- **Minimal Values Under the Hood (Bjørn Vårdal and Frédéric Parain)**
  - https://youtu.be/7eDftOYjV-k
- **Proposal for L-World (Dan Smith)**
  - http://cr.openjdk.java.net/~dlsmith/values-notes.html
- **Proposal for Template Classes (John Rose)**
  - http://cr.openjdk.java.net/~jrose/values/template-classes.html
Summary

• Many cool features to come with Java
  – Segmented Code Cache, Compact Strings, Ahead-of-Time compilation, Value Types

• Java – A vibrant platform
  – Early access releases are available: jdk.java.net/11/

• The future of the Java platform
  "Our SaaS products are built on top of Java and the Oracle DB—that’s the platform.”
  Larry Ellison, Oracle CTO

• Questions?
  – tobias.hartmann@oracle.com