State of the Values

From a compiler engineer's perspective

Tobias Hartmann, Oracle
Roland Westrelin, Red Hat
January 2018

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.

Agenda

- **Background: (Minimal) Value Types**
- **Storage formats and optimizations**
- **JIT support**
  - IR representation and optimizations
  - Real world examples
  - Calling convention changes
  - Method handles / lambda forms
- **Challenges and limitations**
- **Next steps (L-World explorations)**

Thanks to Bjørn Vårdal and Frédéric Parain for the RT slides!

Value Types

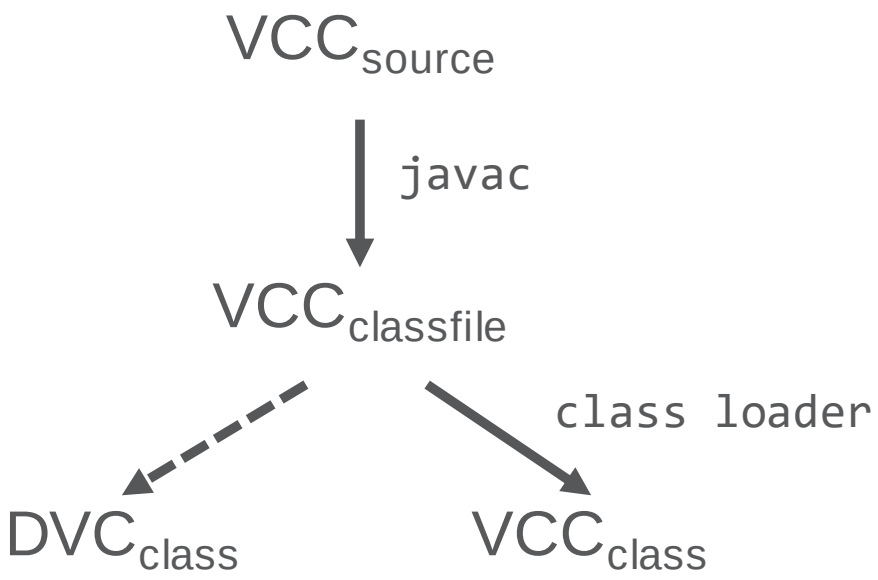
- **Value types are immutable, identityless aggregates**
  - User defined primitives
  - Non-synchronizable, non-nullable
  - *„Codes like a class, works like an int!“*
- **Introduced for performance**
  - Better spatial locality (no indirection, no header)
  - Avoid heap allocations to reduce GC pressure
  - 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



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

Bytecode	Behaviour
vload	Load value from local
vstore	Store value to local
vreturn	Return value from method
vaload	Load value from value array (flattened or not)
vastore	Store value to value array (flattened or not)
vbox	Convert a value to a reference
vunbox	Convert a reference to a value
vdefault	Create a default value (all-zero)
vwithfield	Create a new value from an existing value, with an updated field

Method Handles

Bytecode	Corresponding MethodHandle
vaload	MethodHandles::arrayElementGetter
vastore	MethodHandles::arrayElementSetter
vbox	ValueType::box
vunbox	ValueType::unbox
vdefault	ValueType::defaultValueConstant
vwithfield	ValueType::findWither
anewarray	MethodHandles::arrayConstructor
…	

Beyond MVT: Experimental javac support

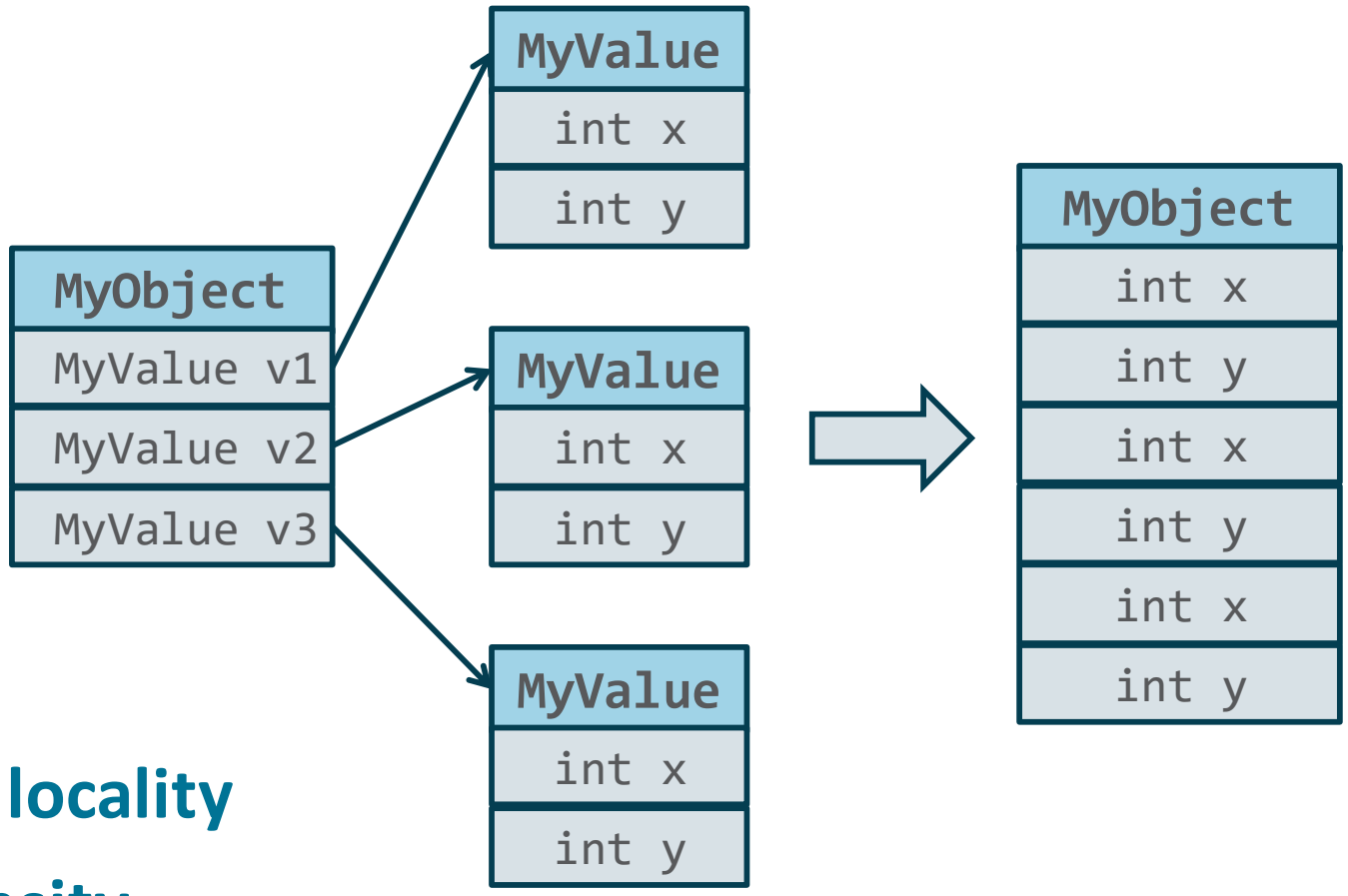
```
__ByValue final class MyValue {
  final int x, y;

  __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;
}
```



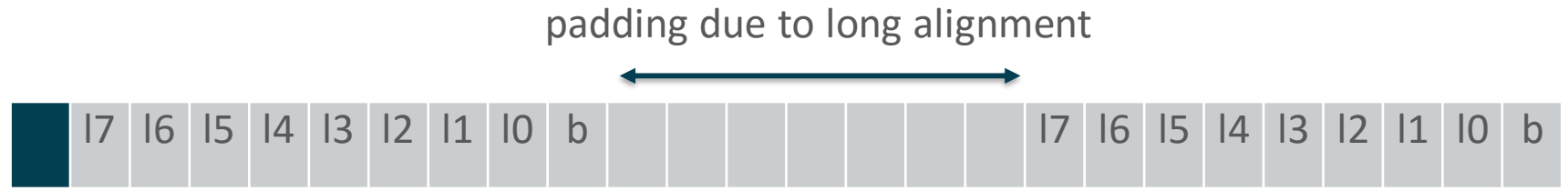
- No indirections: better spatial **locality**
- No pointer/header: better **density**

Value Type Field Flattening

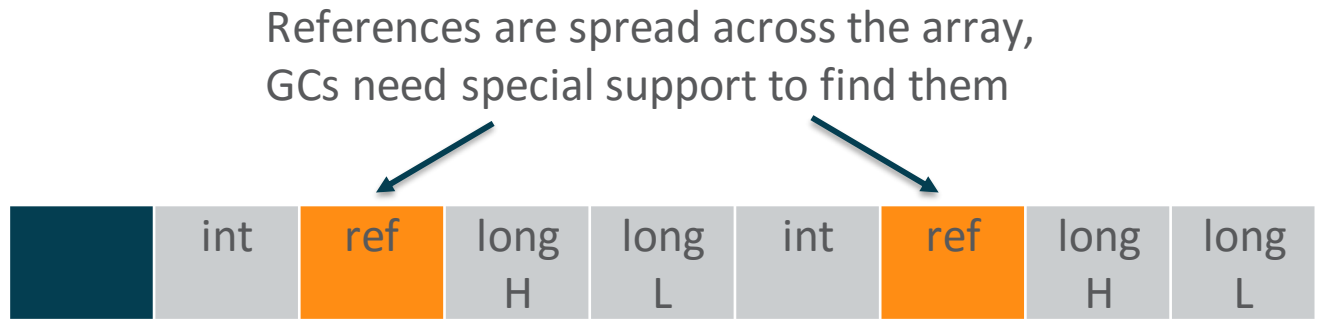
- Only for non-static fields
- Works both for object and value type holders
- Requires pre-loading of value types to determine field size
- Flattened fields keep their layout (no intermixing)
- Optional via `-XX:ValueFieldMaxFlatSize`

Value Type Array Flattening

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




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



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

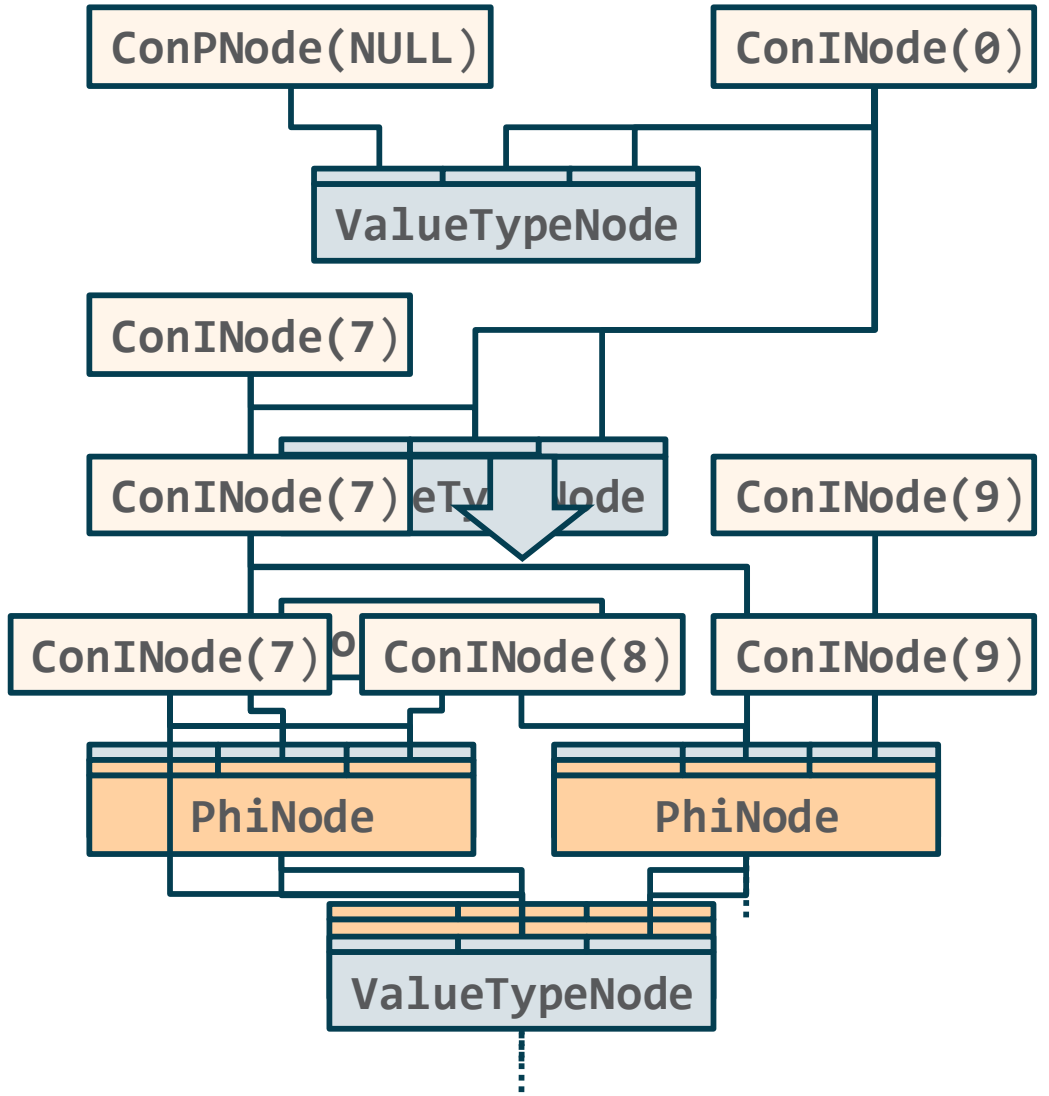
- Full feature support
  - New bytecodes, optional flattening, (interpreter-)buffering, deoptimization, OSR, incremental inlining, method handles, …
- **Pass and return value types in registers** or on the stack
  - No need to retain identity
- **Avoid heap allocations** through aggressive scalarization
- Avoid boxing from DVC to VCC
- 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!**

IR representation

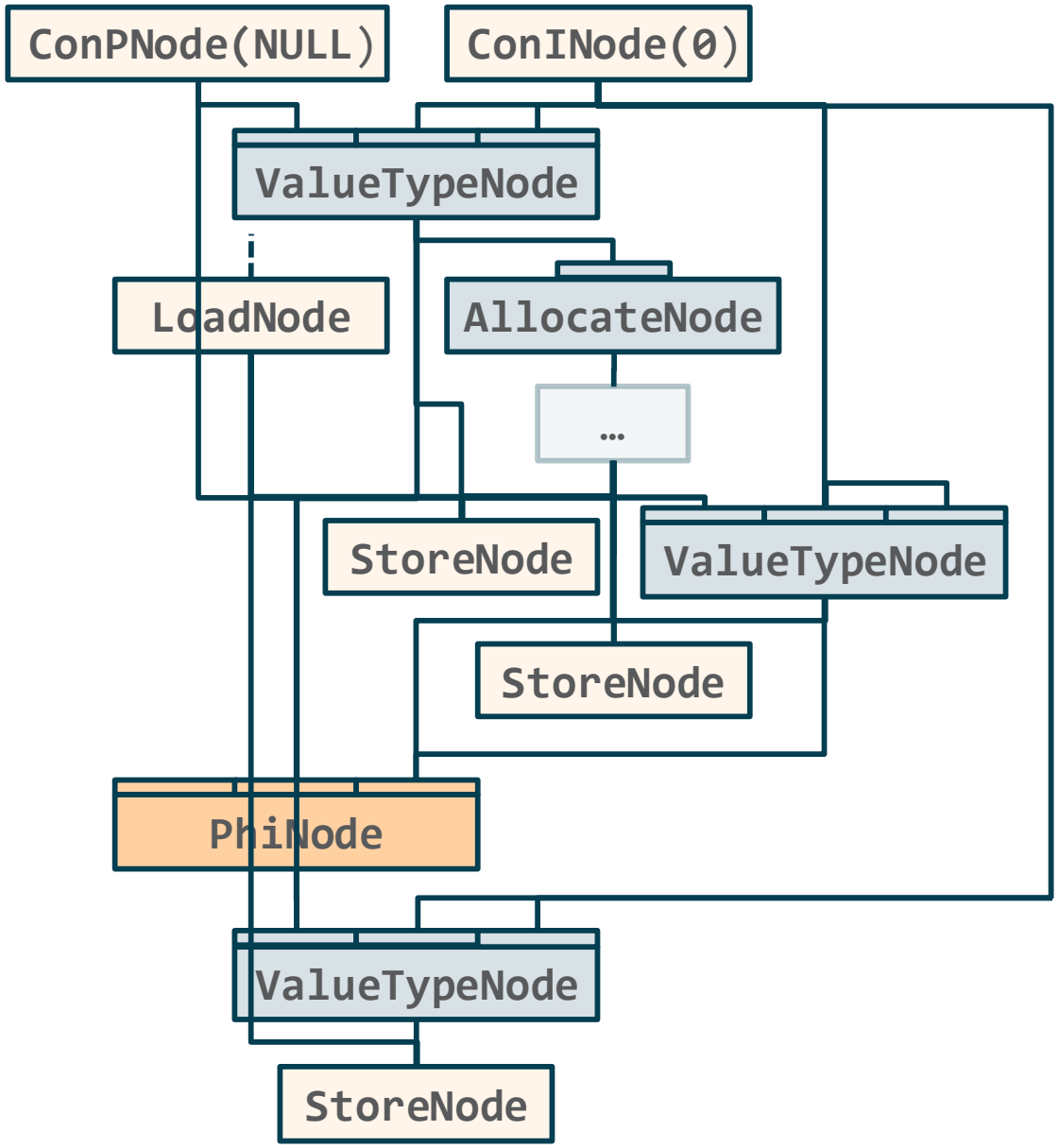
```
__ByValue final class MyValue {
  final int x, y;
  ...
}

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



Advanced IR optimizations

```
// Copy detection
public method1(MyValue v1) {
  MyValue v2 = __MakeDefault MyValue();
  v2.x = v1.x;
  v2.y = v1.y;
  staticField1 = v2; // allocate
}
```

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

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

```
public method2() {
  MyValue v = __MakeDefault MyValue();
  v.x = 42;
  staticField1 = v  // allocate
  staticField2 = v; // allocate
}
```

Example: Complex number using POJOs

```
final 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:

```
2ffb: cmp    $0x2a,%ecx            ; y > THRESHOLD?
2ffe: jg     306f
...
300b: cmp    0x88(%r15),%r11       ; Fast alloc?
3012: jae    30b2                  ; -> slow (RT call)
...                                ; -> fast (TLB)
303d: mov    %r10d,0xc(%rax)       ; c.x = x
3041: mov    %ebp,0x10(%rax)       ; c.y = y
3044: mov    0x10(%rax),%r11d      ; load y
3048: mov    0xc(%rax),%r10d       ; load x
304c: imul   %r11d,%r11d
3050: imul   %r10d,%r10d
3054: add    %r11d,%r10d           ; c.x*c.x + c.y*c.y
3057: vcvtsi2sd %r10d,%xmm0,%xmm0
305c: vsqrtsd %xmm0,%xmm0,%xmm0    ; sqrt
...

306f: mov    0x78(%r15),%rax
...
307a: cmp    0x88(%r15),%r11       ; Fast alloc?
3081: jae    30c9                  ; -> slow (RT call)
...                                ; -> fast (TLB)
30a4: mov    %r11d,0xc(%rax)       ; c.x = x
30a8: movq   $0x2a,0x10(%rax)      ; c.y = THRESHOLD
30b0: jmp    3044
```

Example: Complex number using Value Types

```
__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

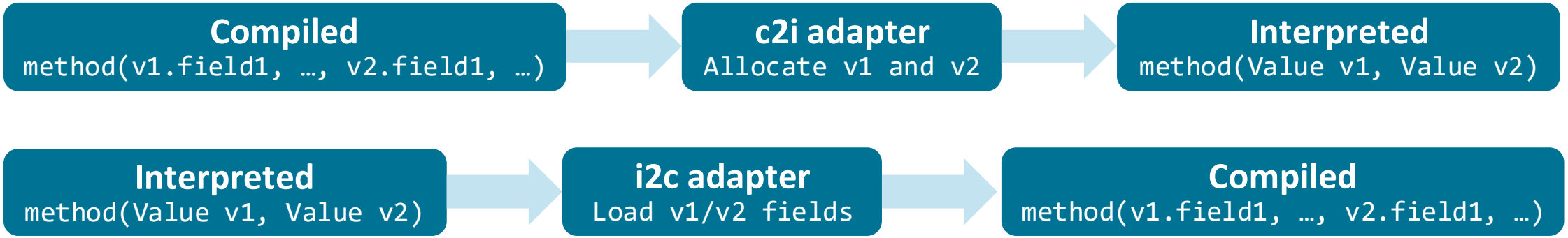
1) Calling a method with a value type argument

- **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.class, v.field1, v.field2, …`)
  - from the interpreter, return (`v , v.field1, v.field2, …`)
- Caller can then either use v or allocate a new value from `v.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


Java™
ORACLE®

	from JVMS		from implementation	
Bytecodes	**Object vs Value test**	**null test (VBC migration)**	**Flattening test**	**Buffering test**
aastore	yes	yes	yes	yes
aaload			yes	
areturn	yes	yes		yes
invokeinterface	yes (synchronized)			
invokespecial	yes (synchronized)			
invokestatic	yes (synchronized)			
invokevirtual	yes (synchronized)			
monitorenter	yes			
monitorexit	yes			
new	yes (test on class arg)			
getfield			yes	
putfield	yes	yes	yes	yes
putstatic	yes	yes		yes
vdefault	yes (test on class arg)			
vwithfield	yes	yes	yes	yes
acmp_*	yes			
anewarray			yes	
multianewarray			yes	