The Java HotSpot VM

Under the Hood

Tobias Hartmann

Compiler Group – Java HotSpot Virtual Machine
Oracle Corporation

May 2018

ORACLE

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

ORACLE

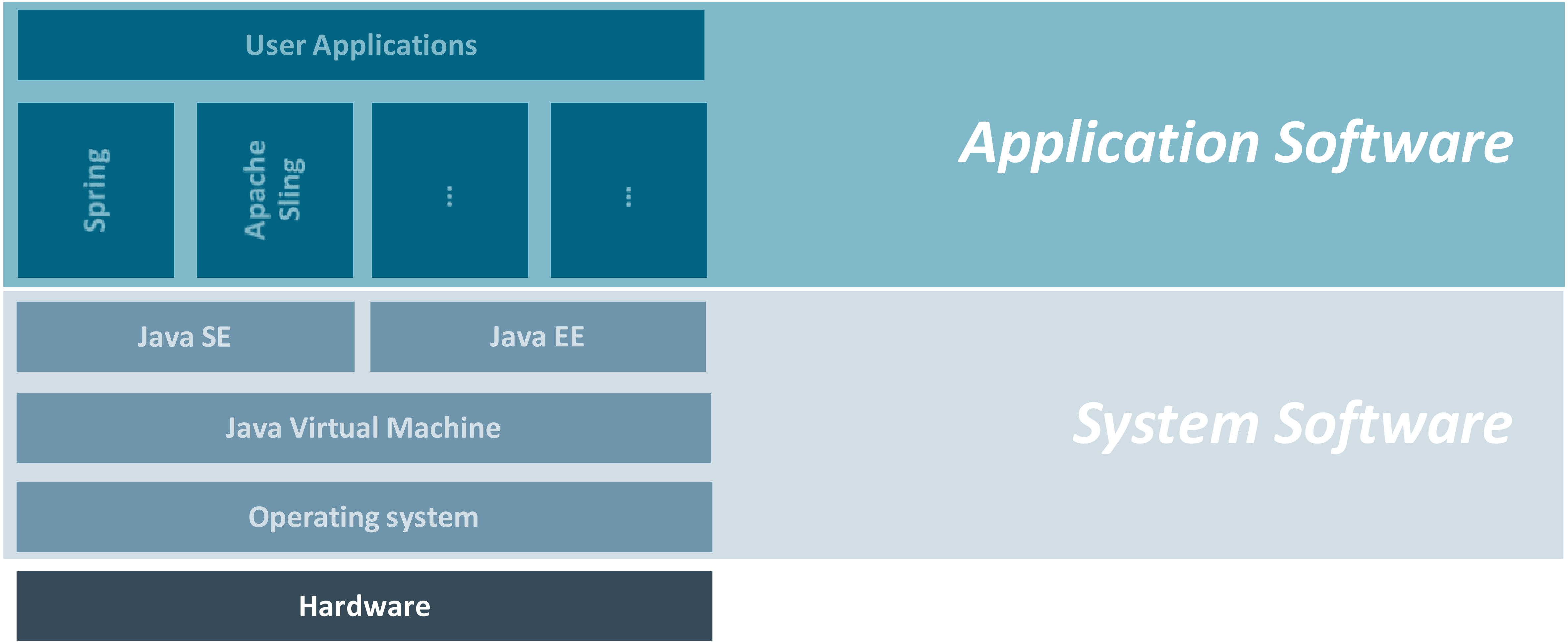
Safe Harbor Statement

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



ORACLE

A typical computing platform

User Applications

Spring

Apache Sling

...

...

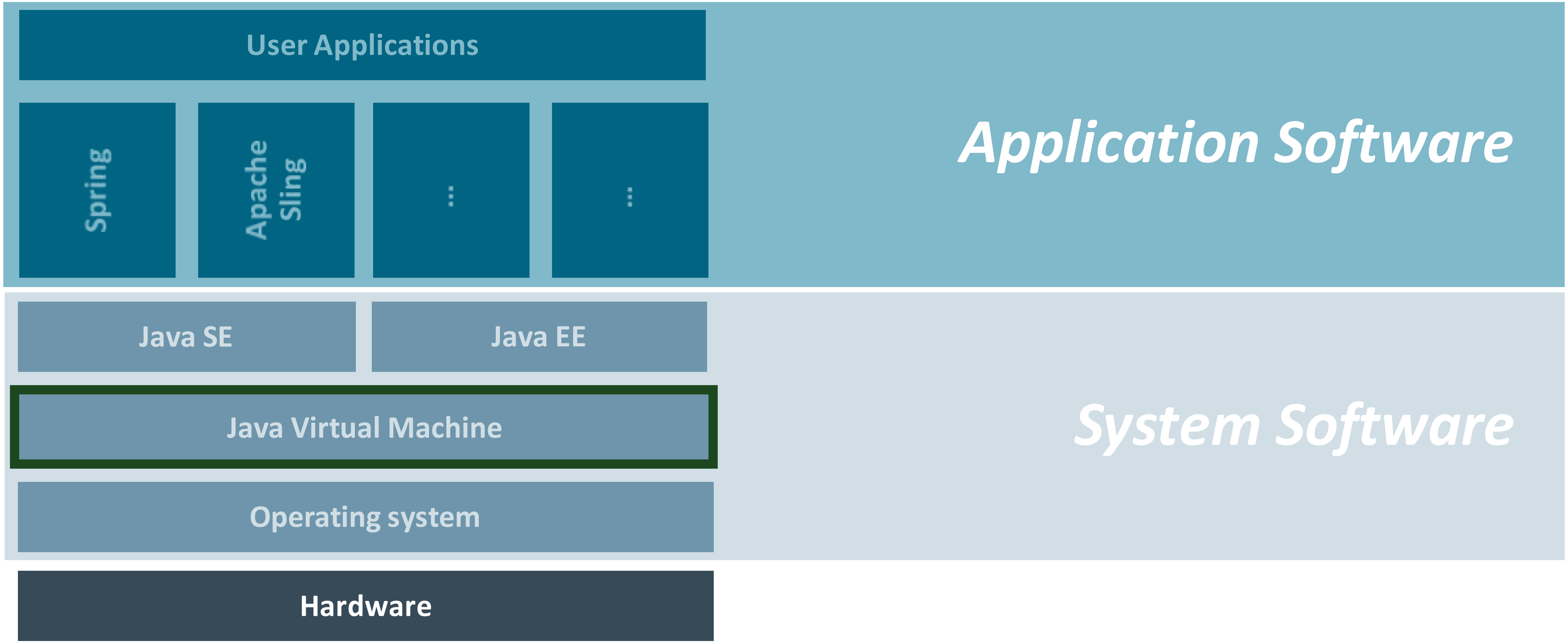
Application Software

System Software

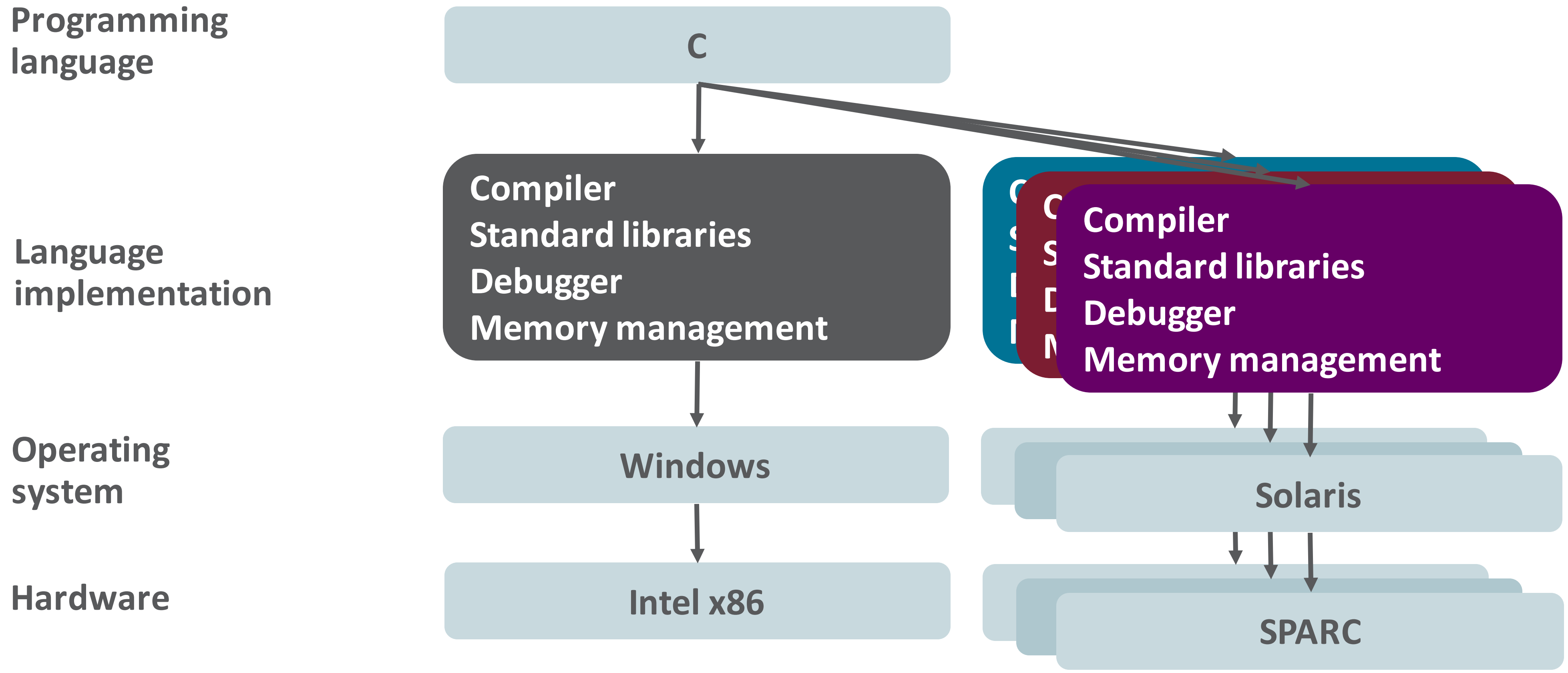
Hardware

ORACLE

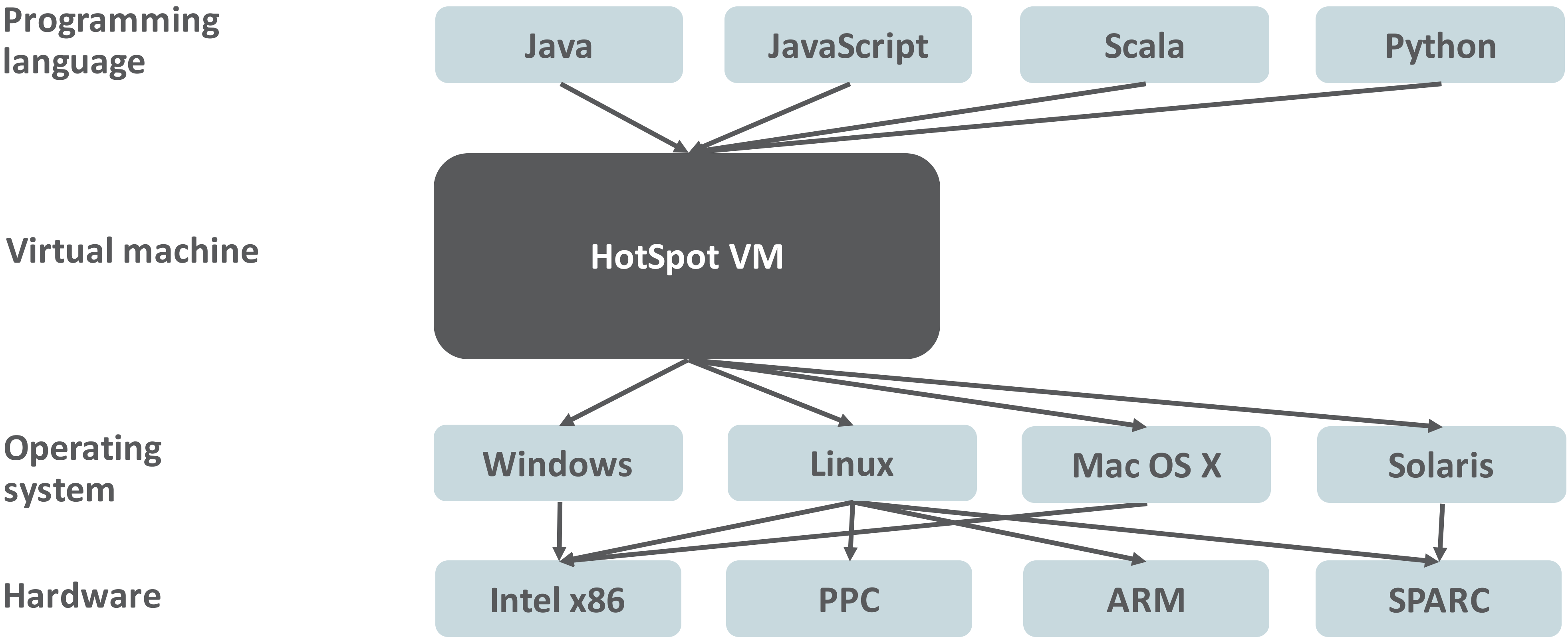
A typical computing platform



Programming language implementation



(Language) virtual machine



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

ORACLE

The JVM: An application developer's view

Java source code

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

compile

Bytecodes

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

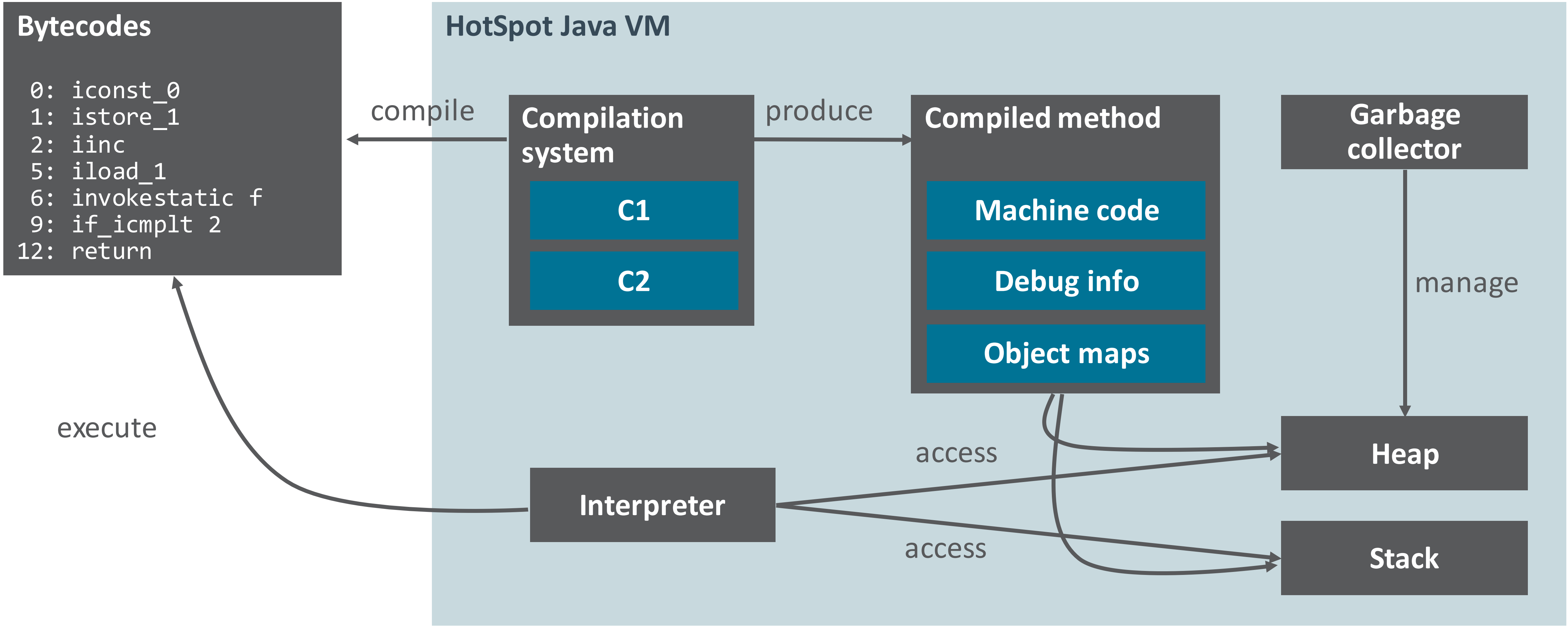
execute

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



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

ORACLE

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: invokestatic f
 9: if_icmplt 2
12: return
```

Machine code

```
mov     -0x8(%r14), %eax
movzbl   0x1(%r13), %ebx
inc      %r13
mov      $0xff40,%r10
jpmq     *(%r10, %rbx, 8)
```

Load local variable 1

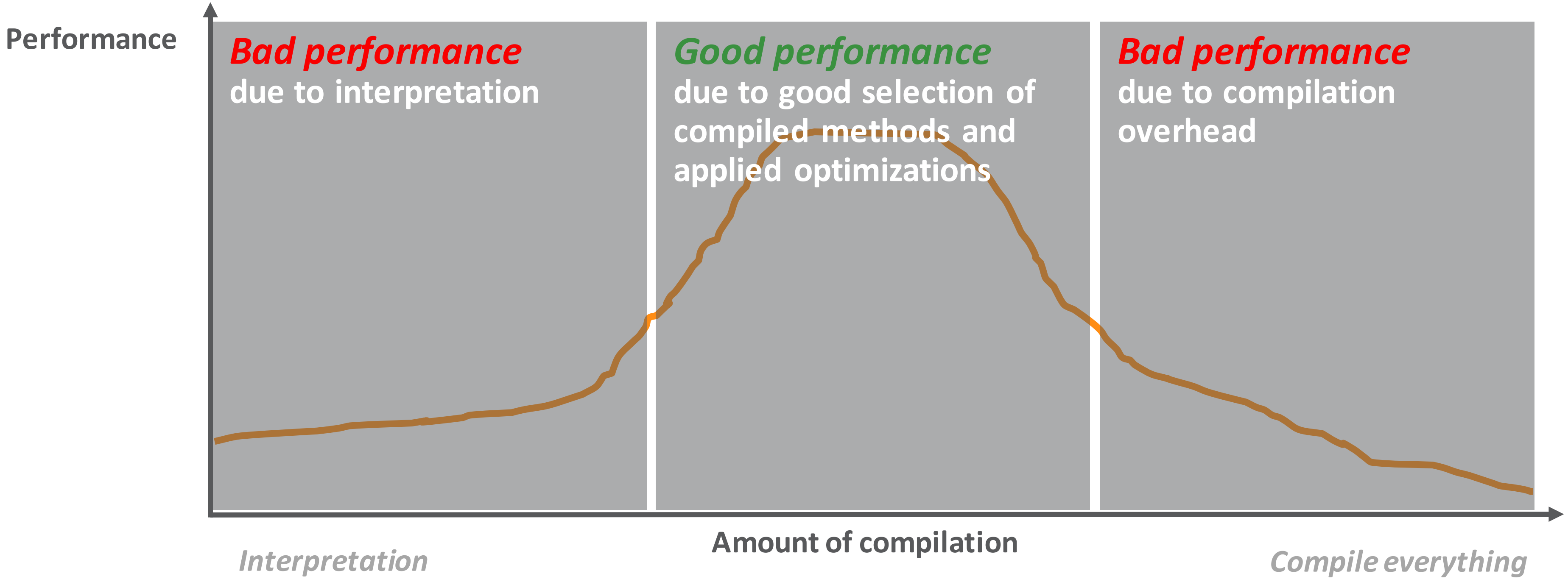
Dispatch next instruction

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

ORACLE

Ahead-of-time vs. just-in-time compilation

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



ORACLE

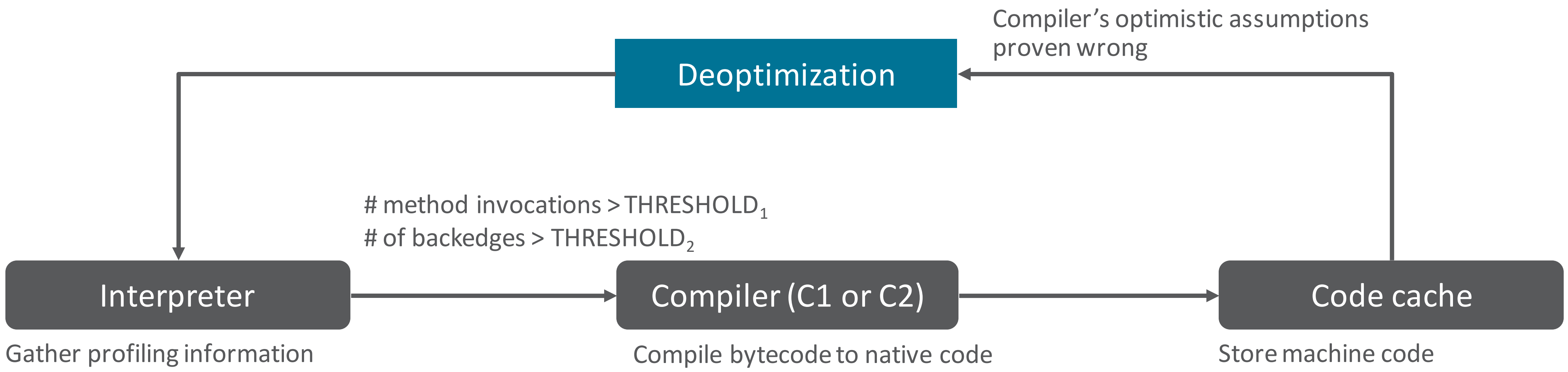
JIT compilation in HotSpot

- **Resource usage** vs. **performance**
  - Getting to the “sweet spot”

1. Selecting methods to compile
2. Selecting compiler optimizations

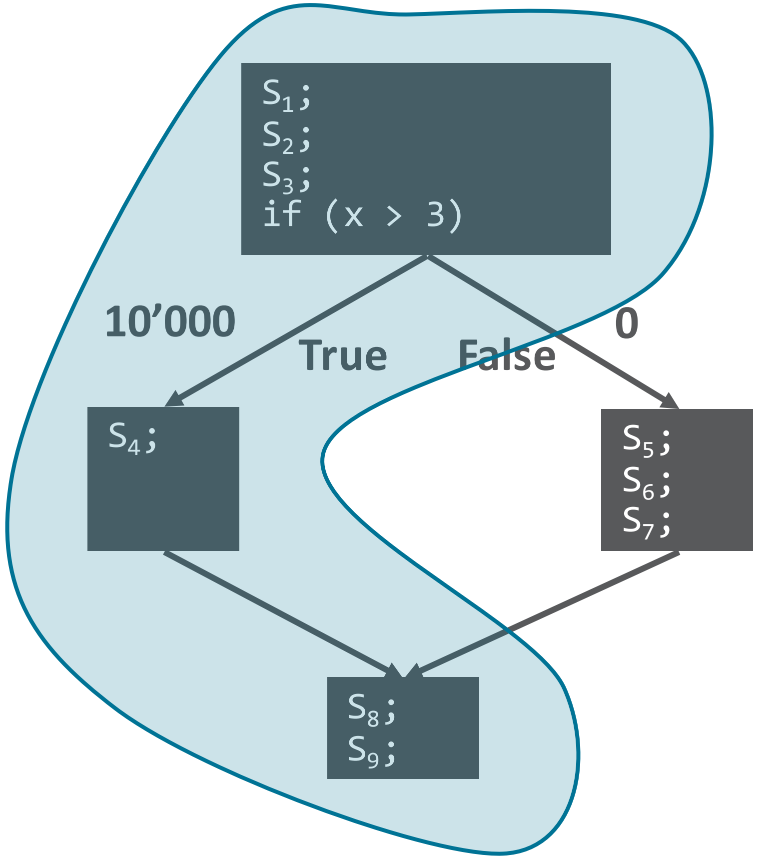
1. Selecting methods to compile

- **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

Example optimization: Virtual call inlining

Class hierarchy

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

loaded

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

not loaded

Method to be compiled

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

Compiler:
Inline call?
Yes.

Example optimization: Virtual call inlining

Class hierarchy

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

loaded

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

not loaded

Method to be compiled

```
void foo() {
  A a = create(); // return A or B
  S1;
}
```

Compiler:
Inline call?
Yes.

- **Benefits of inlining**
  - Virtual call avoided
  - Code locality
- **Optimistic assumption: only A is loaded**
  - Note dependence on class hierarchy
  - Deoptimize if hierarchy changes

Example optimization: Virtual call inlining

Class hierarchy

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

loaded

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

loaded

Method to be compiled

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

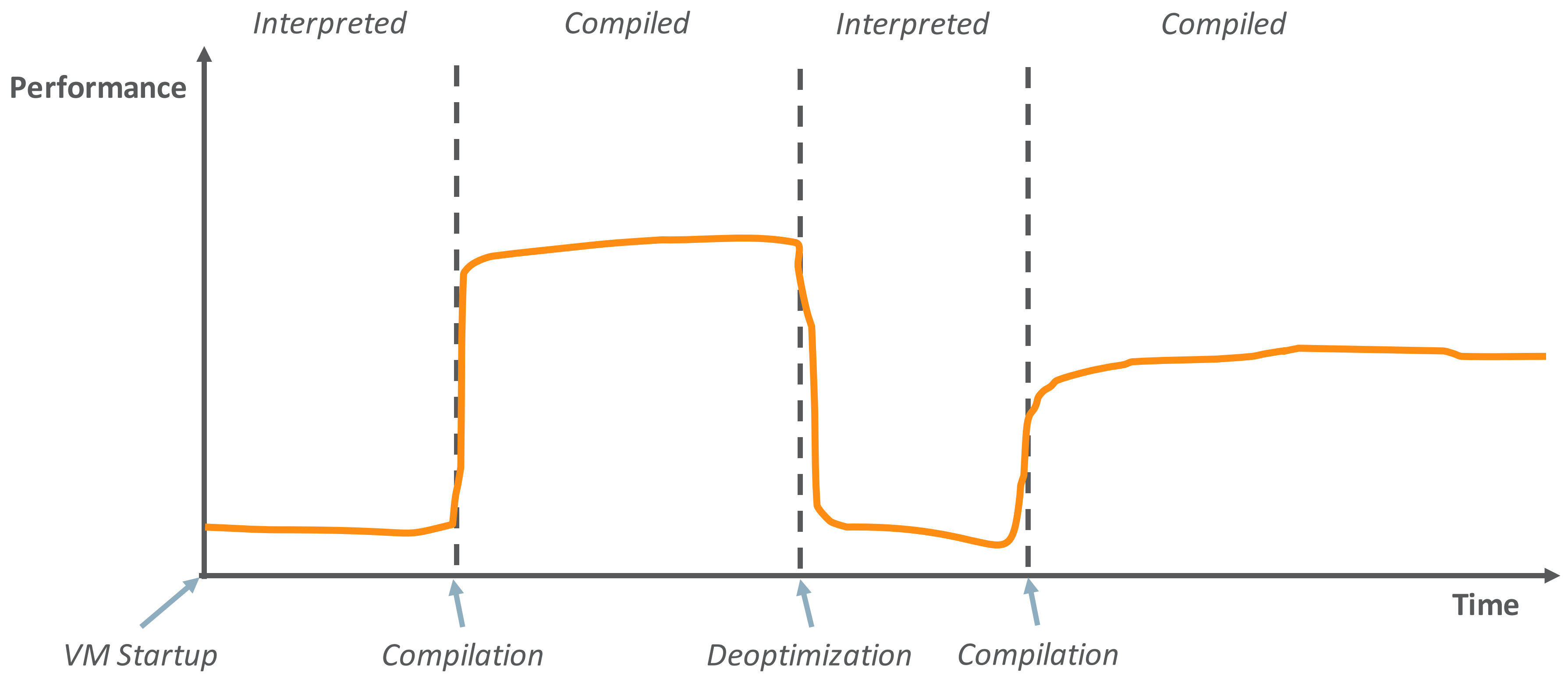
Compiler:
Inline call?
No.

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

ORACLE

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)

ORACLE

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

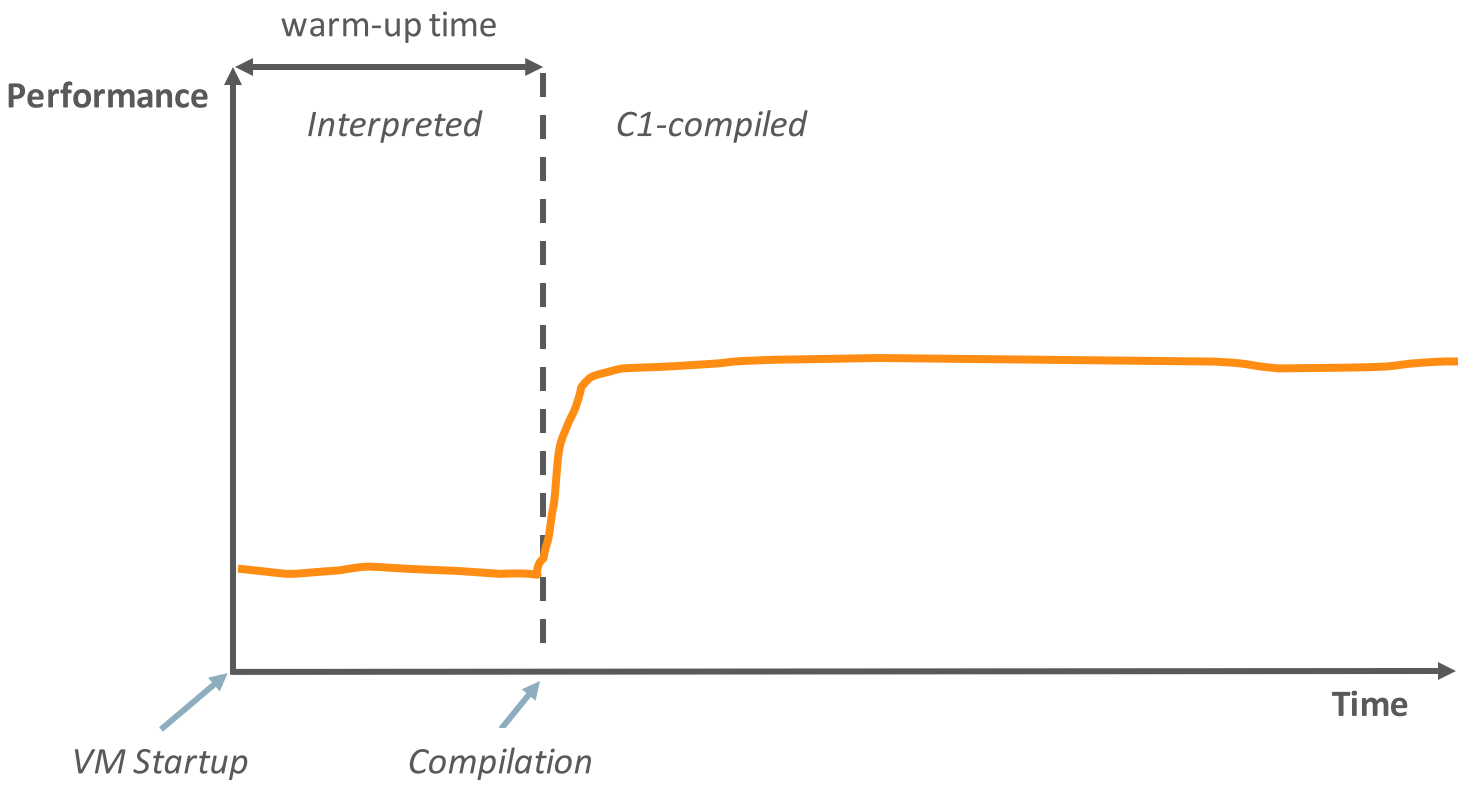
ORACLE

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

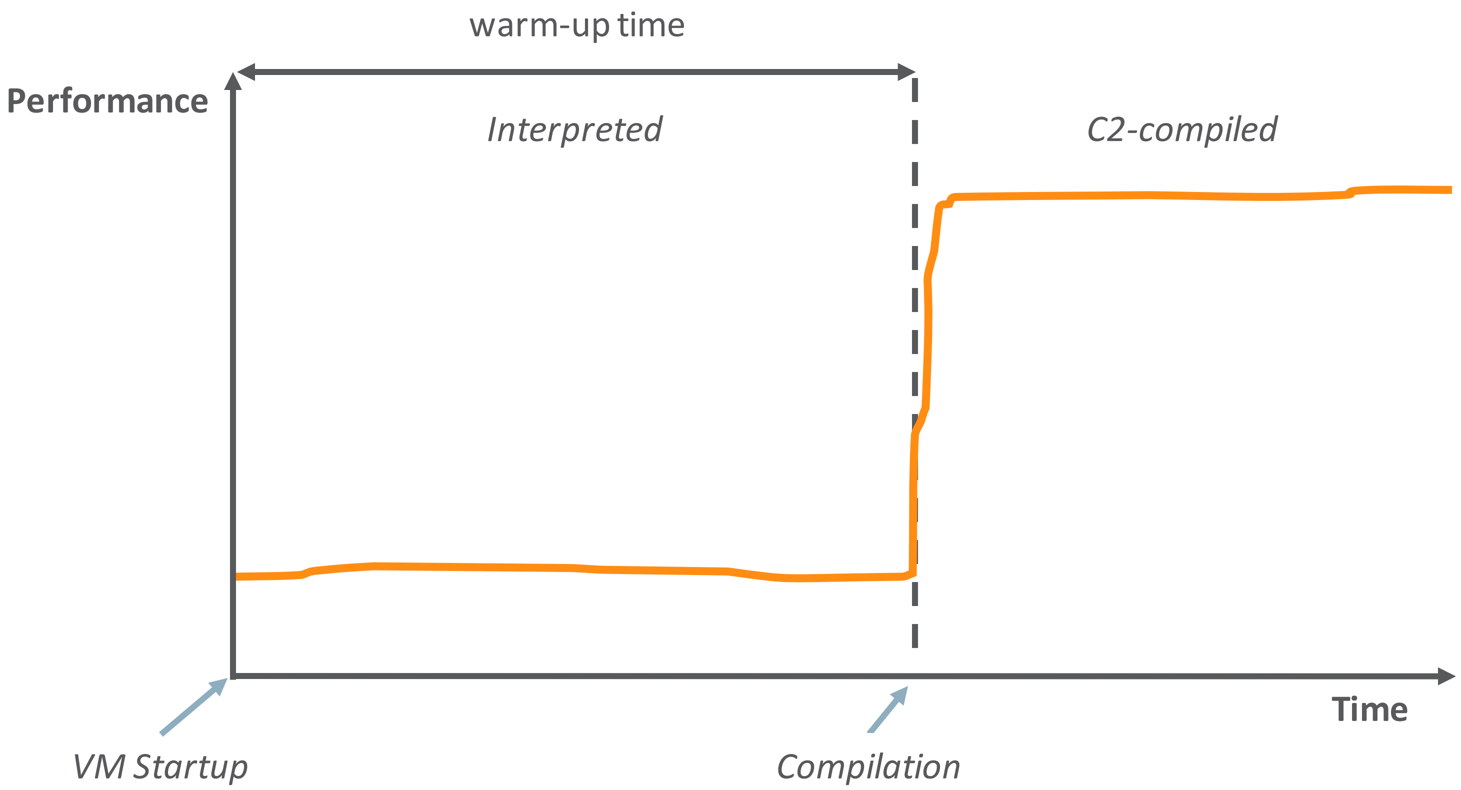
Benefits of Tiered Compilation

Client VM (C1 only)



Benefits of Tiered Compilation

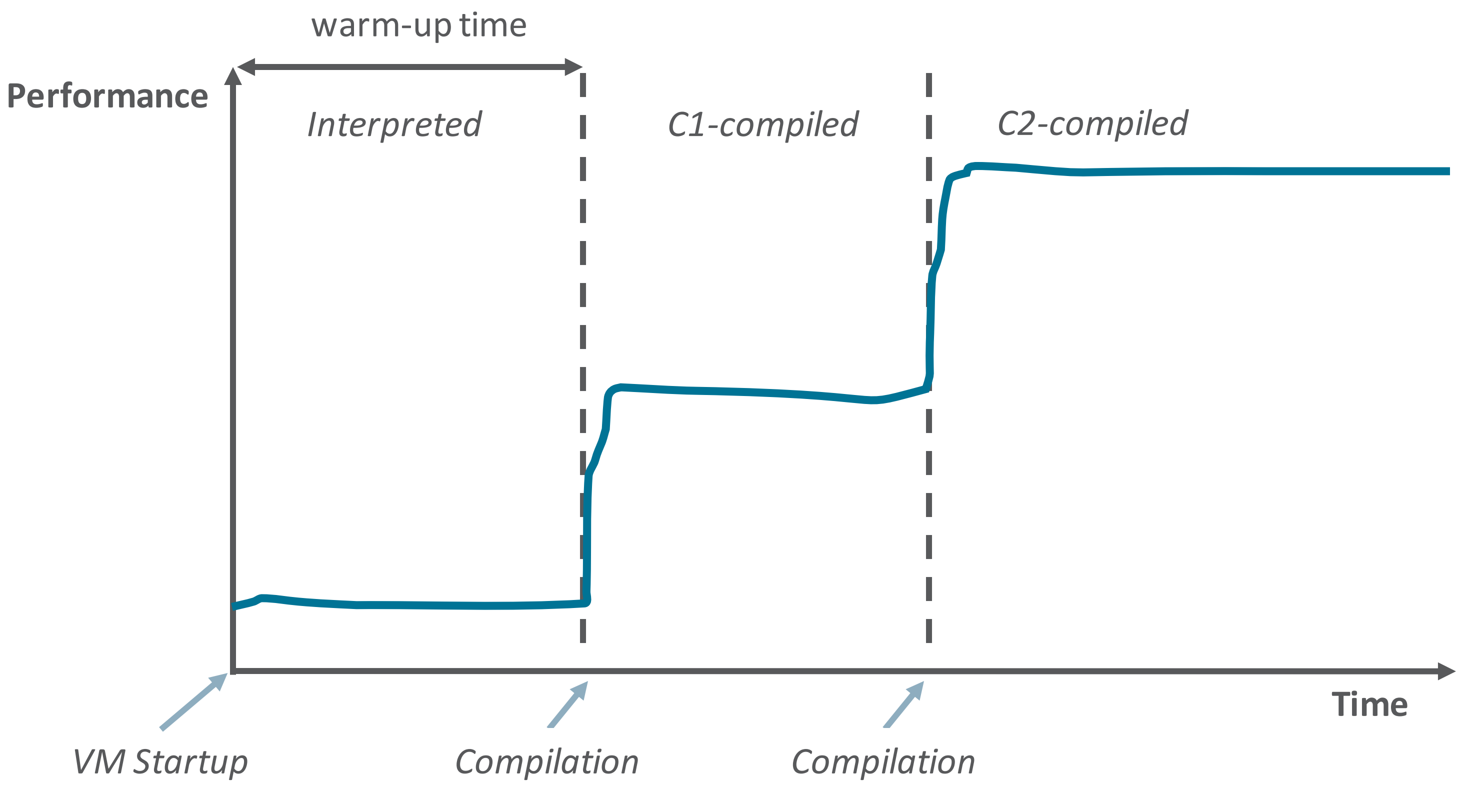
Server VM (C2 only)



ORACLE

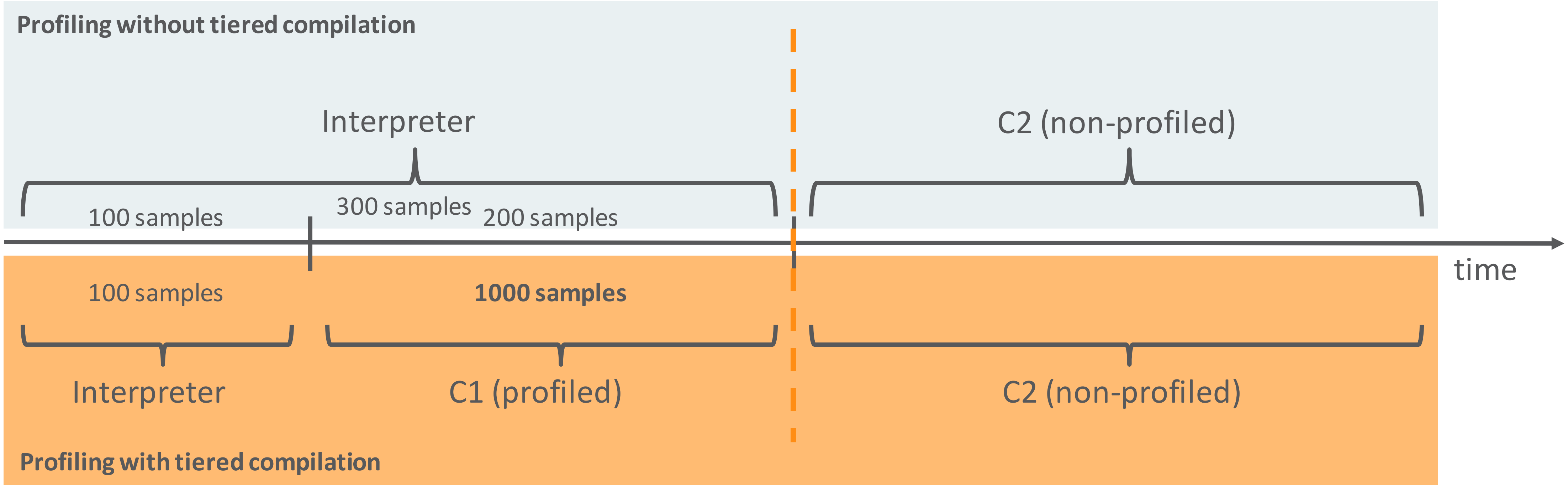
Benefits of Tiered Compilation

Tiered compilation



Additional benefit: More accurate profiling

w/o tiered compilation: 300 samples gathered
w/ tiered compilation: 1’100 samples gathered

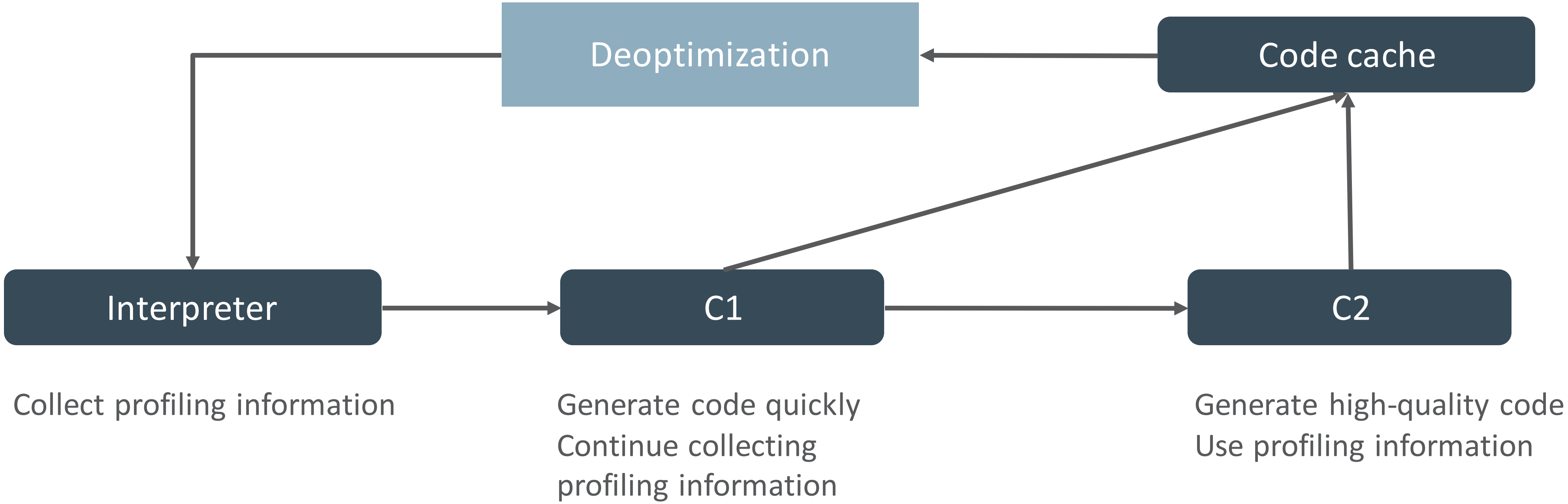


Tiered Compilation

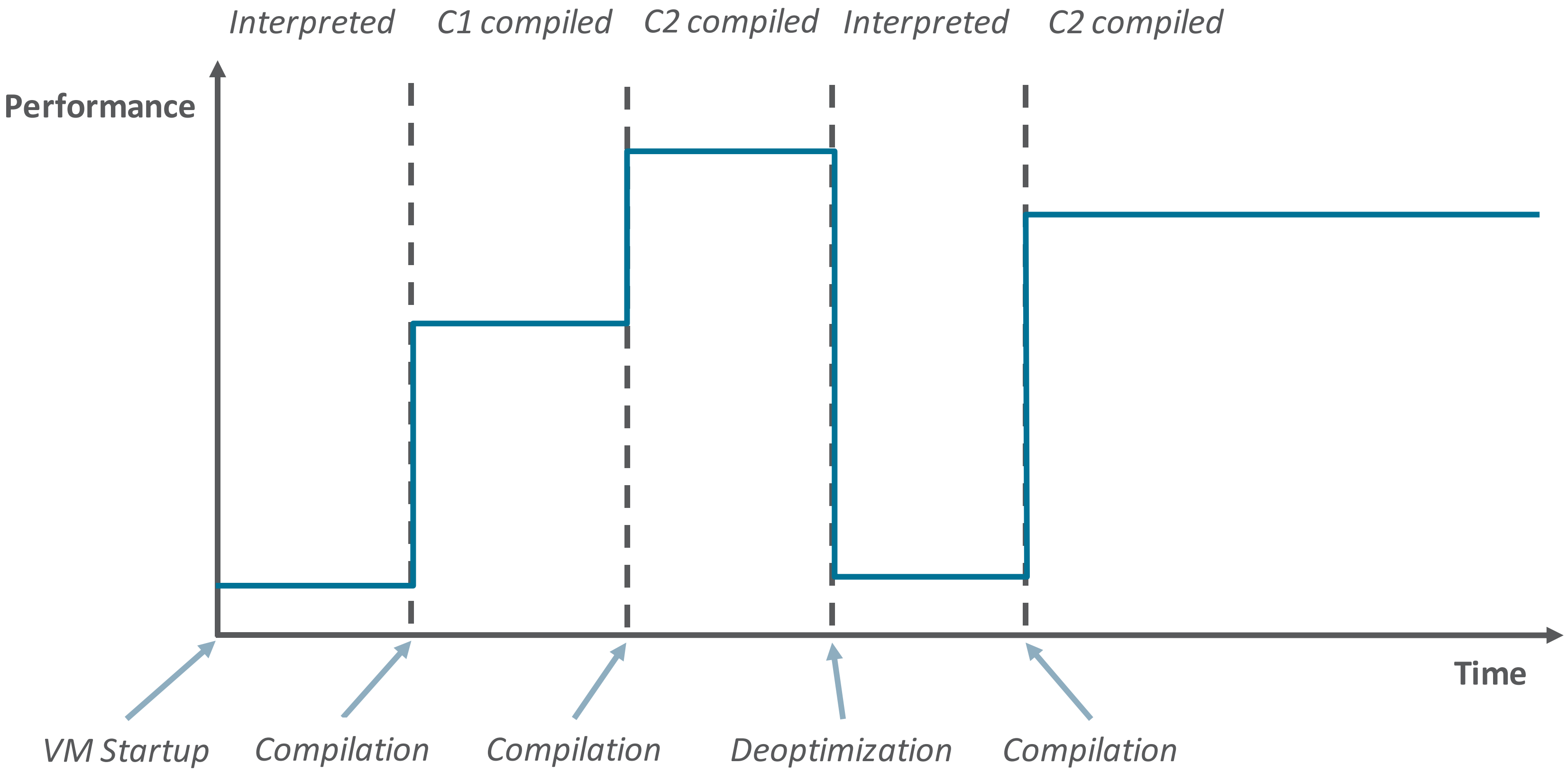
- **Combined benefits of interpreter, C1, and C2**
- **Additional benefits**
  - More accurate profiling information
- **Drawbacks**
  - Complex implementation
  - Careful tuning of compilation thresholds needed
  - More pressure on code cache

ORACLE

A method’s lifetime (Tiered Compilation)

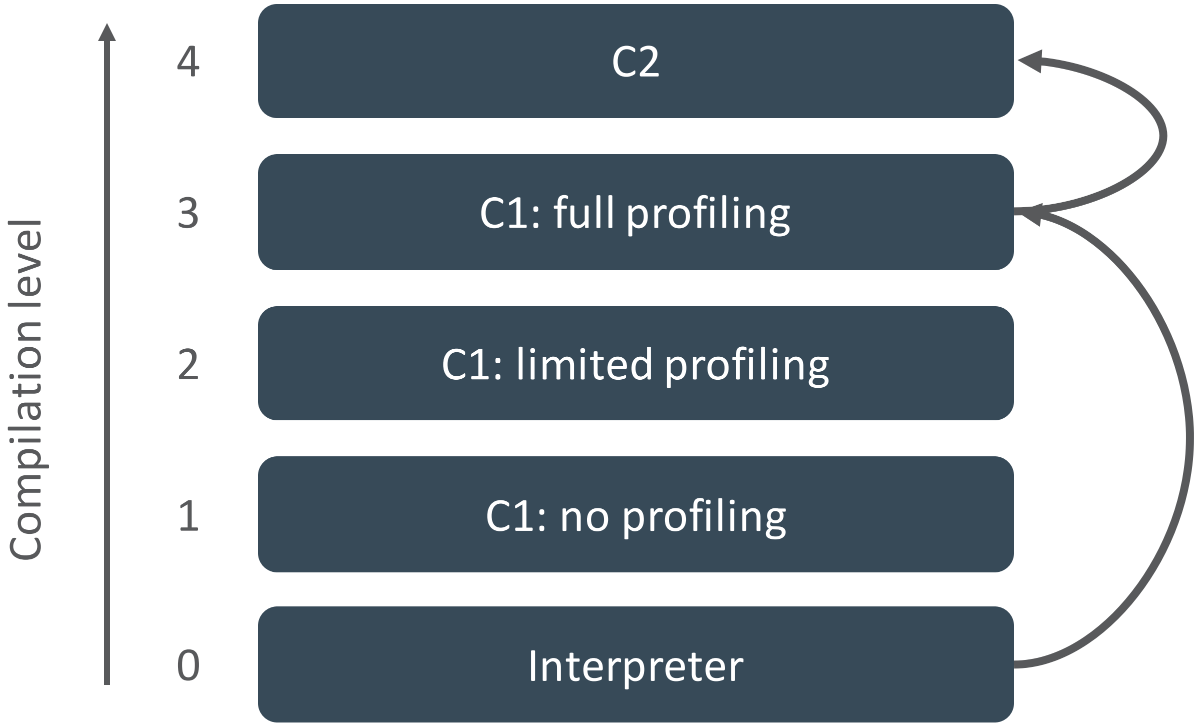


Performance of a method (Tiered Compilation)



Compilation levels (detailed view)

Typical compilation sequence



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

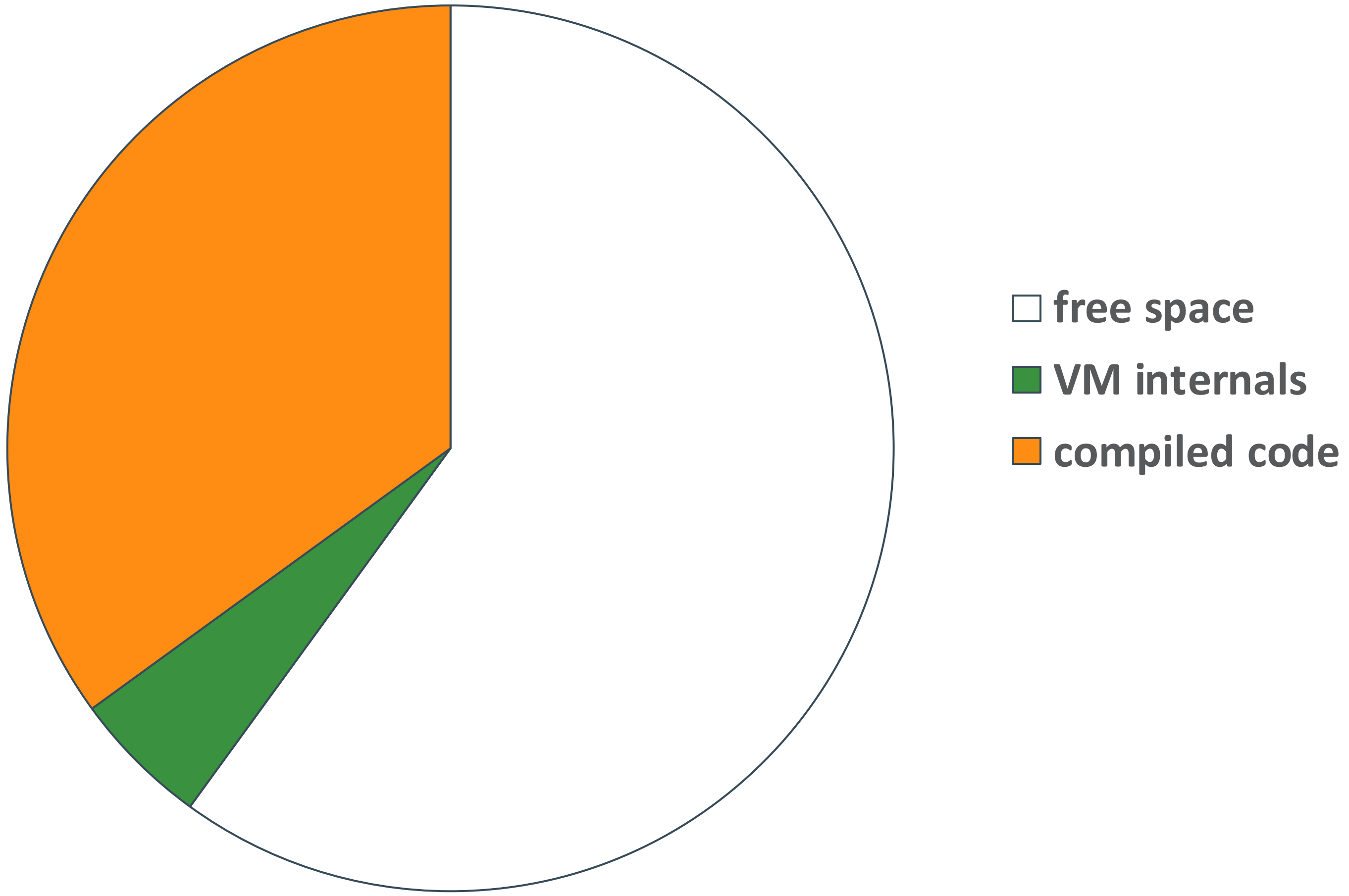
ORACLE

What is the code cache?

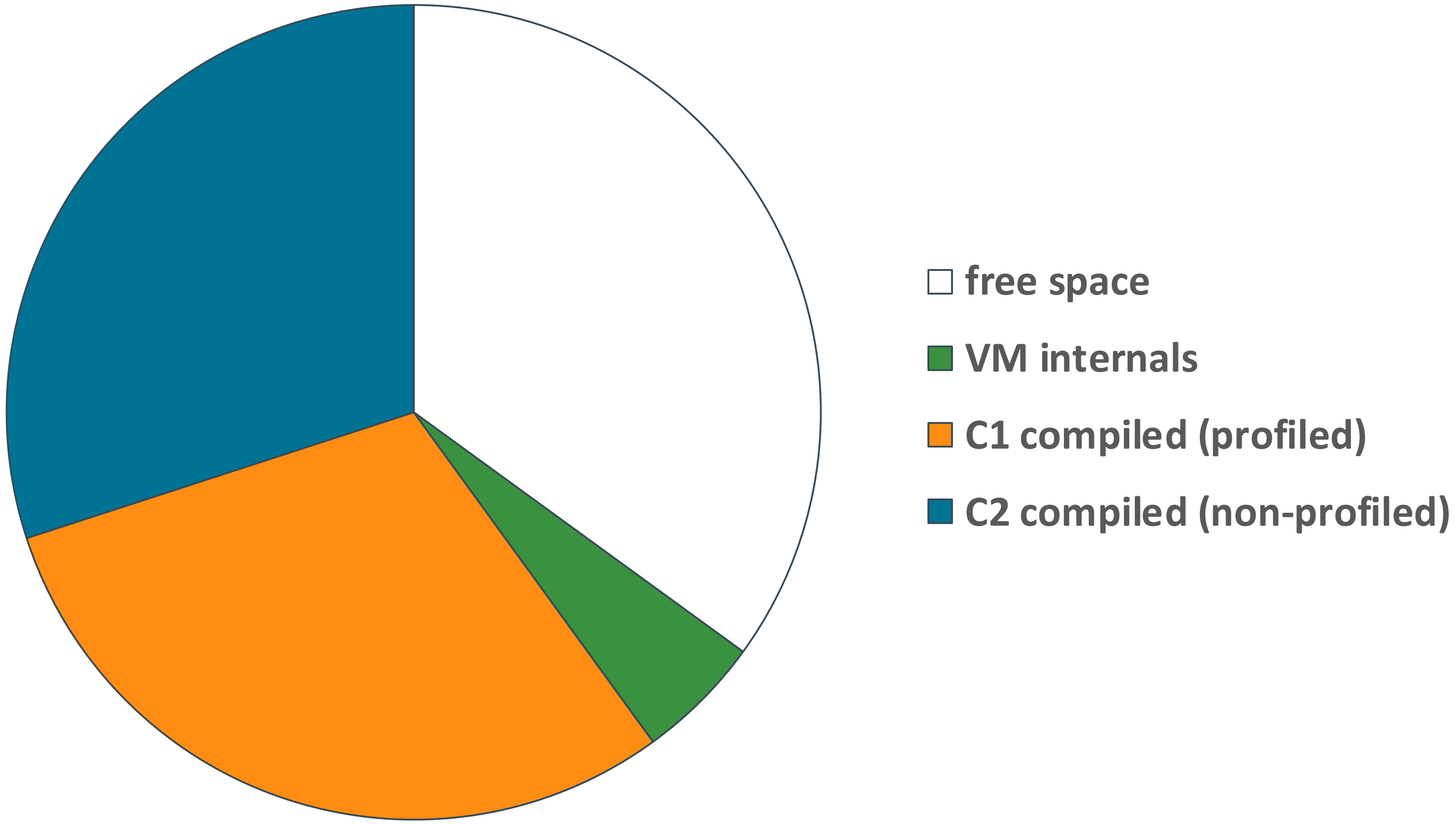
- Stores code generated by JIT compilers
- Continuous chunk of memory
  - Managed (similar to the Java heap)
  - Fixed size
- Essential for performance

ORACLE

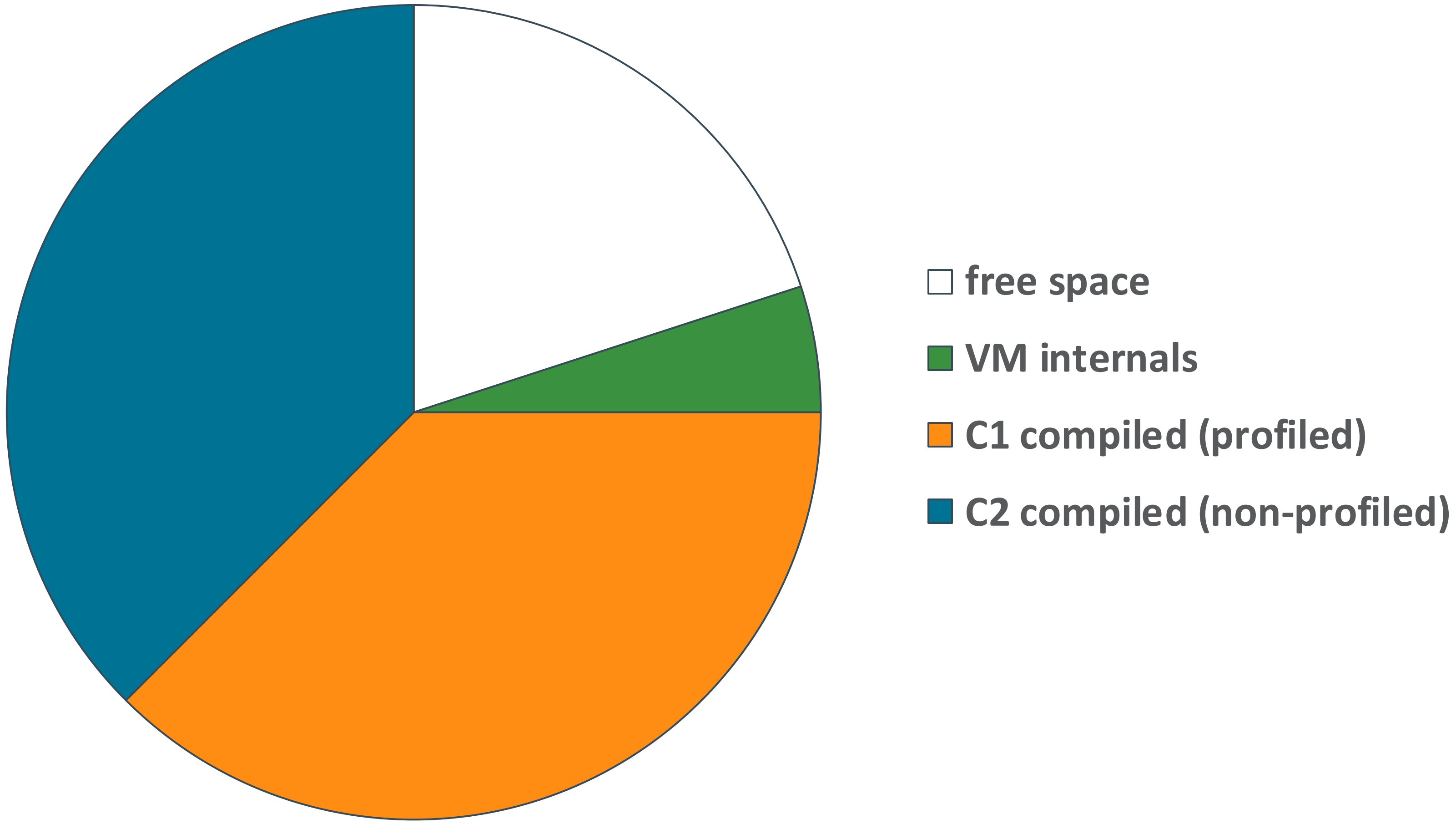
Code cache usage: JDK 6 and 7



Code cache usage: JDK 8 (Tiered Compilation)



Code cache usage: JDK 9

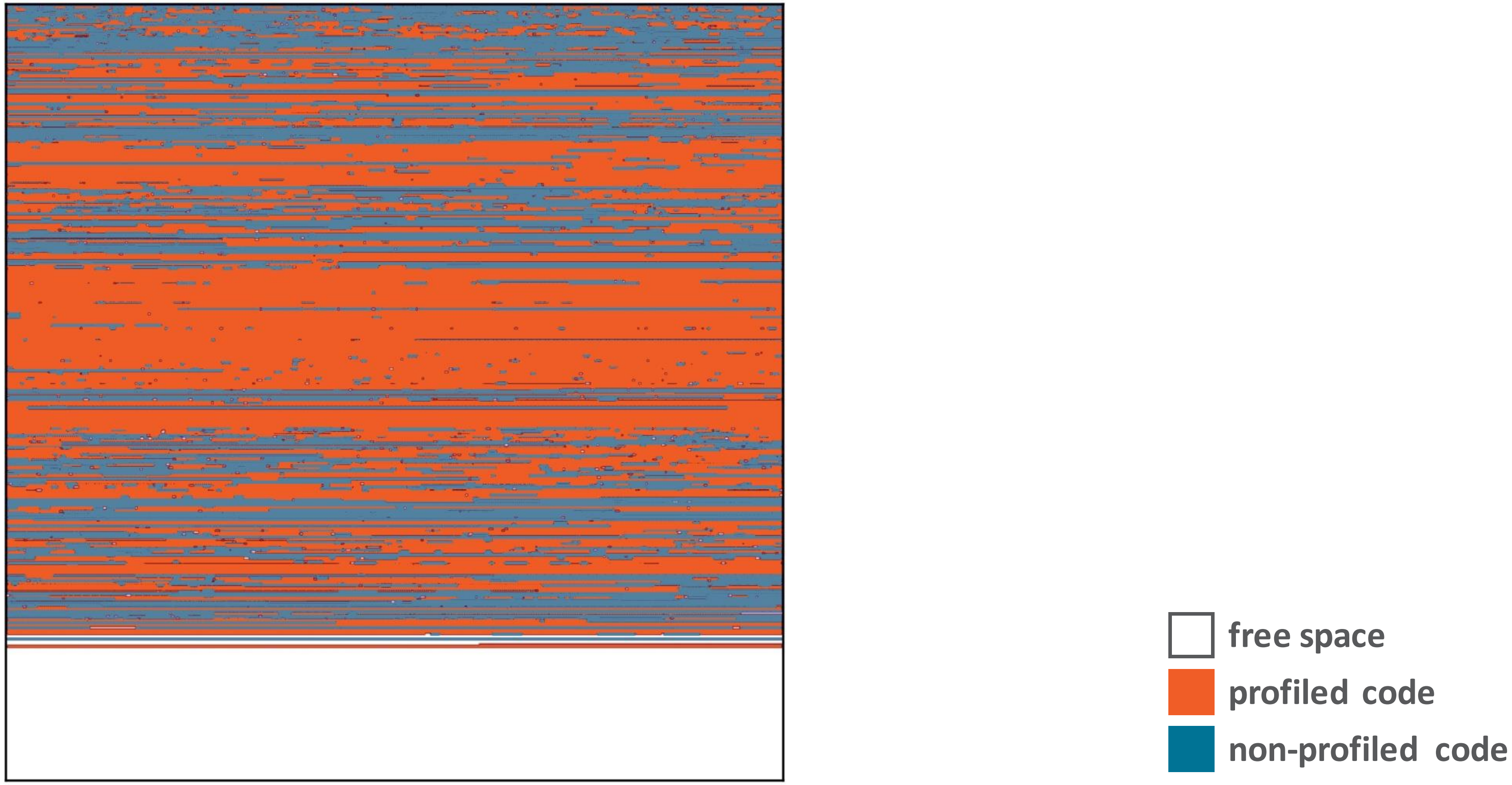


Challenges

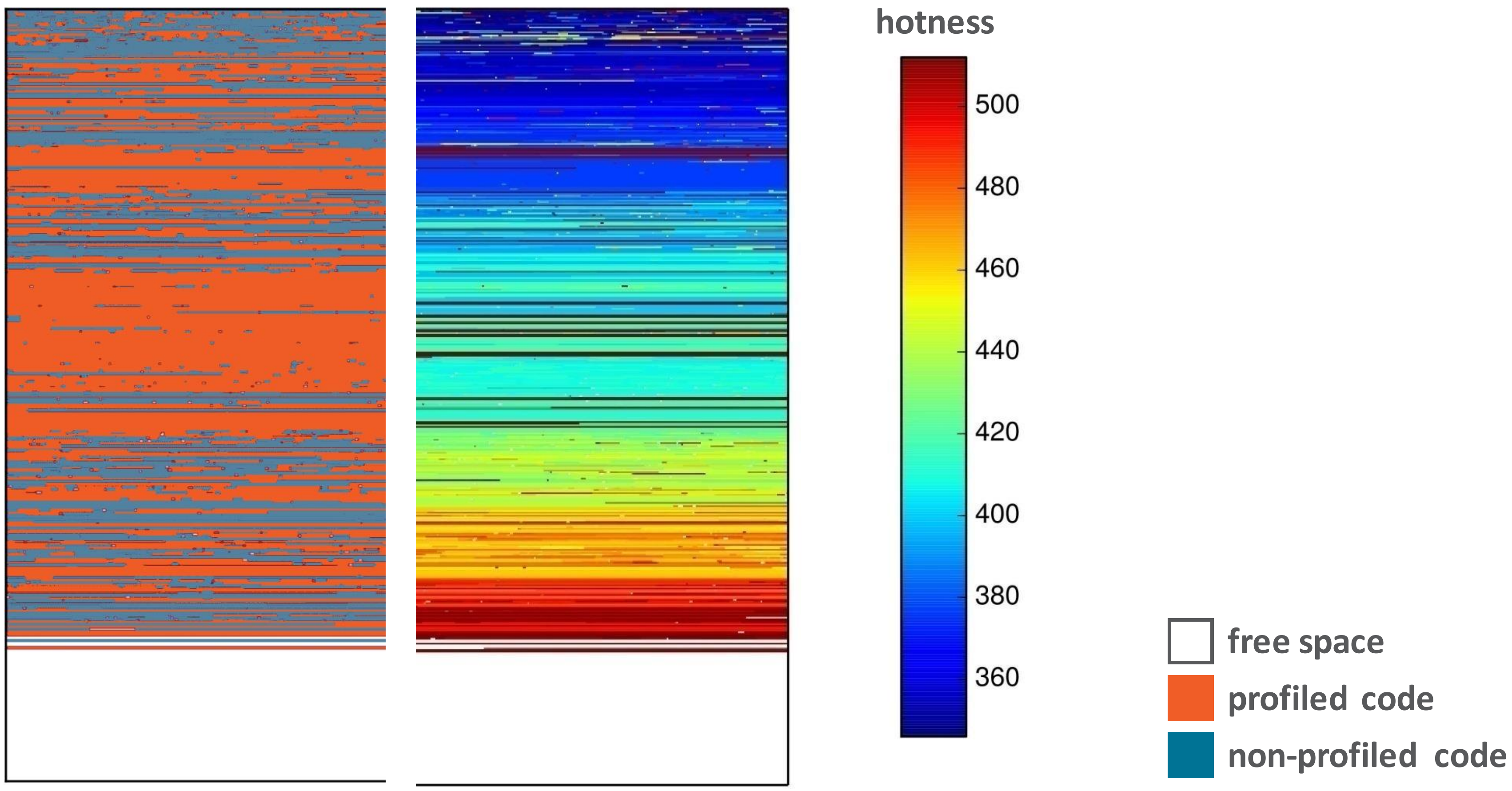
- Tiered compilation increases amount of code by up to **4X**
- All code is stored in a **single code cache**
- High **fragmentation** and **bad locality**

- But is this a problem in **real life**?

Code cache usage: Reality



Code cache usage: Reality



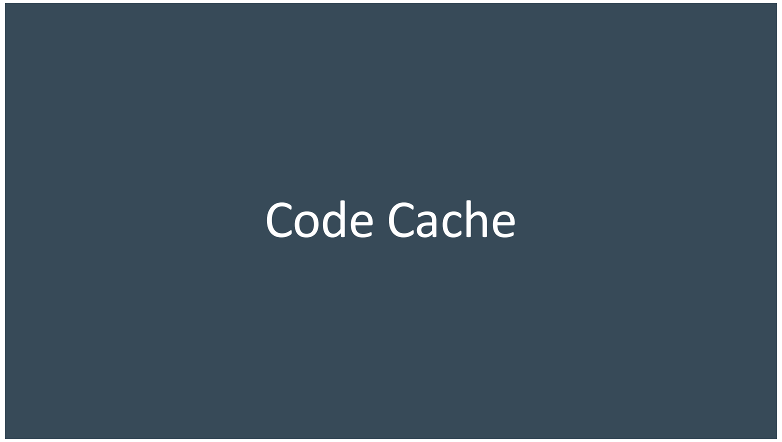
Design: Types of compiled code

	Optimization level	Size	Cost	Lifetime
Non-method code	optimized	small	cheap	immortal
Profiled code (C1)	instrumented	medium	cheap	limited
Non-profiled code (C2)	highly optimized	large	expensive	long

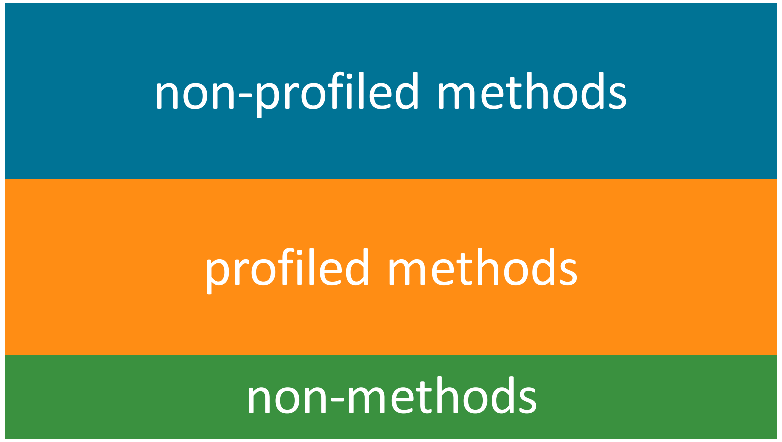
ORACLE

Design

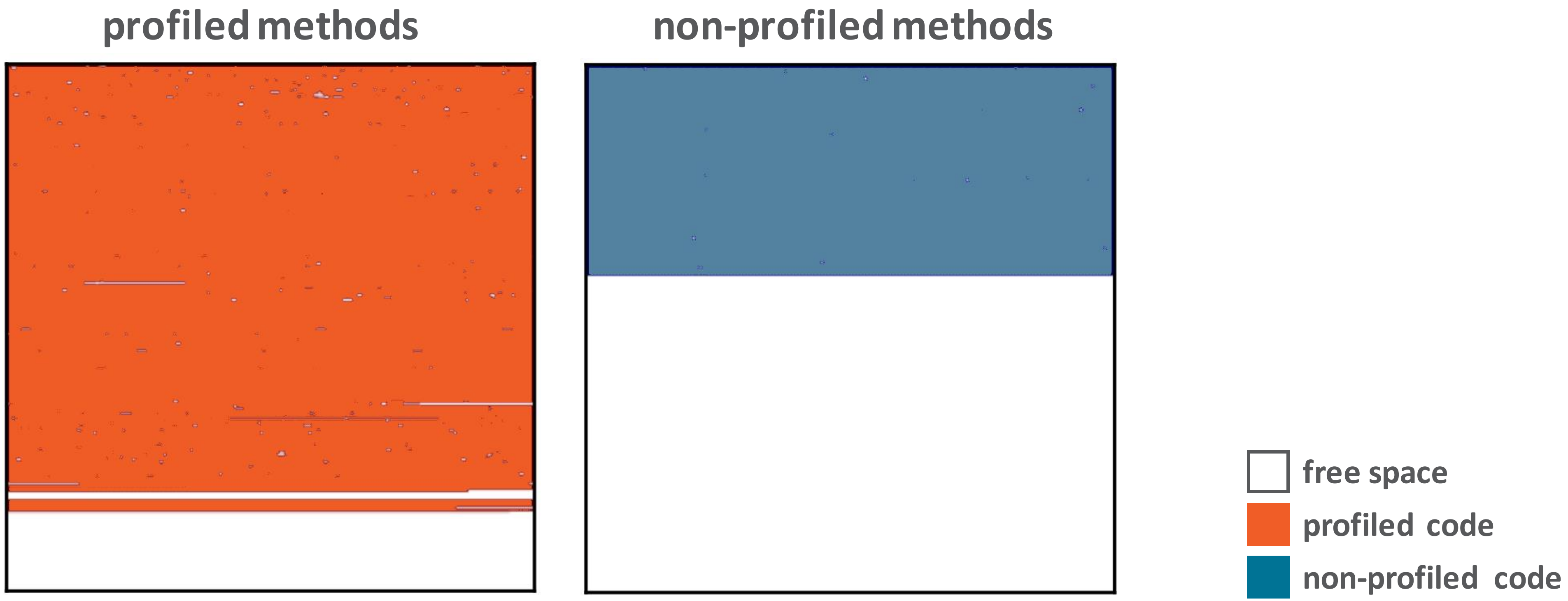
- **Without Segmented Code Cache**



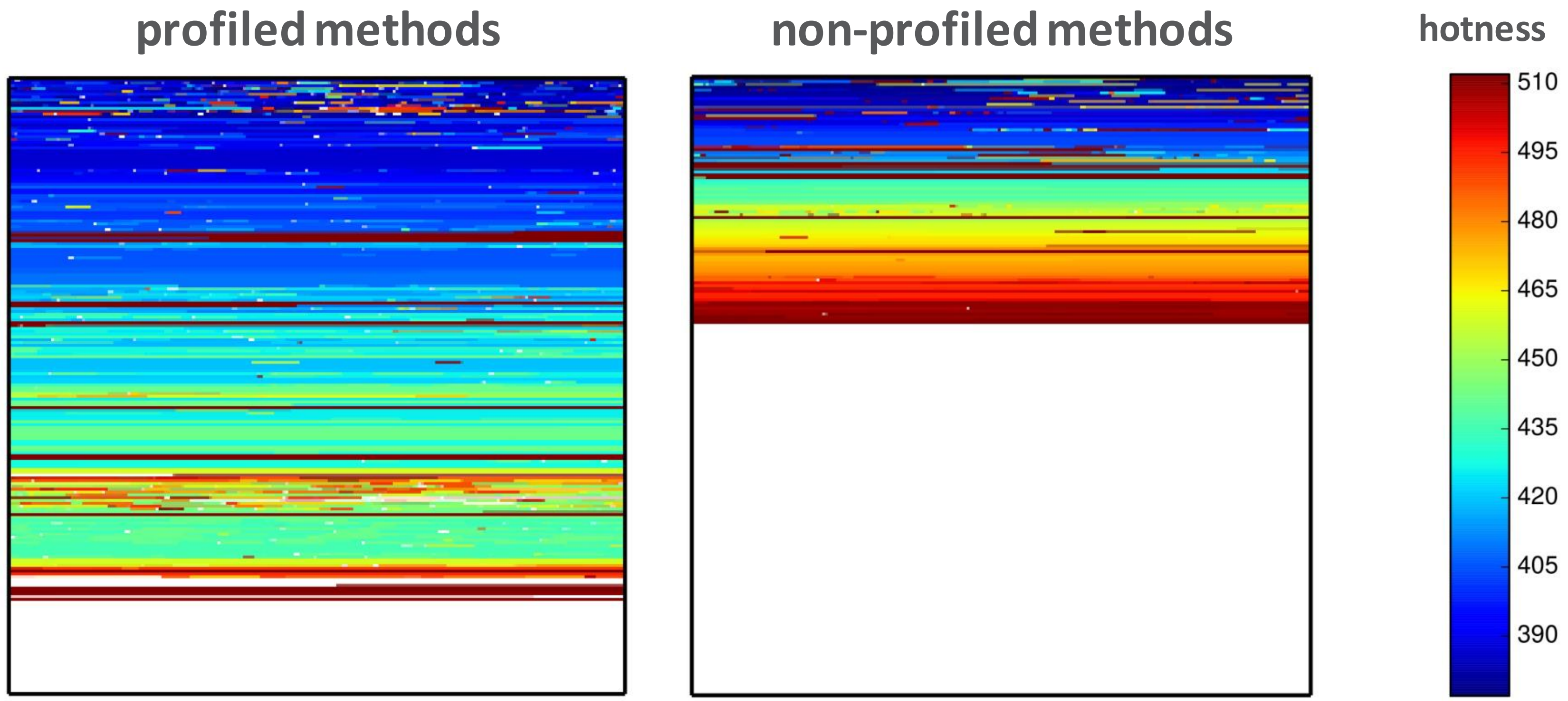
- **With Segmented Code Cache**



Segmented Code Cache: Reality



Segmented Code Cache: Reality



Evaluation: Code locality

```
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

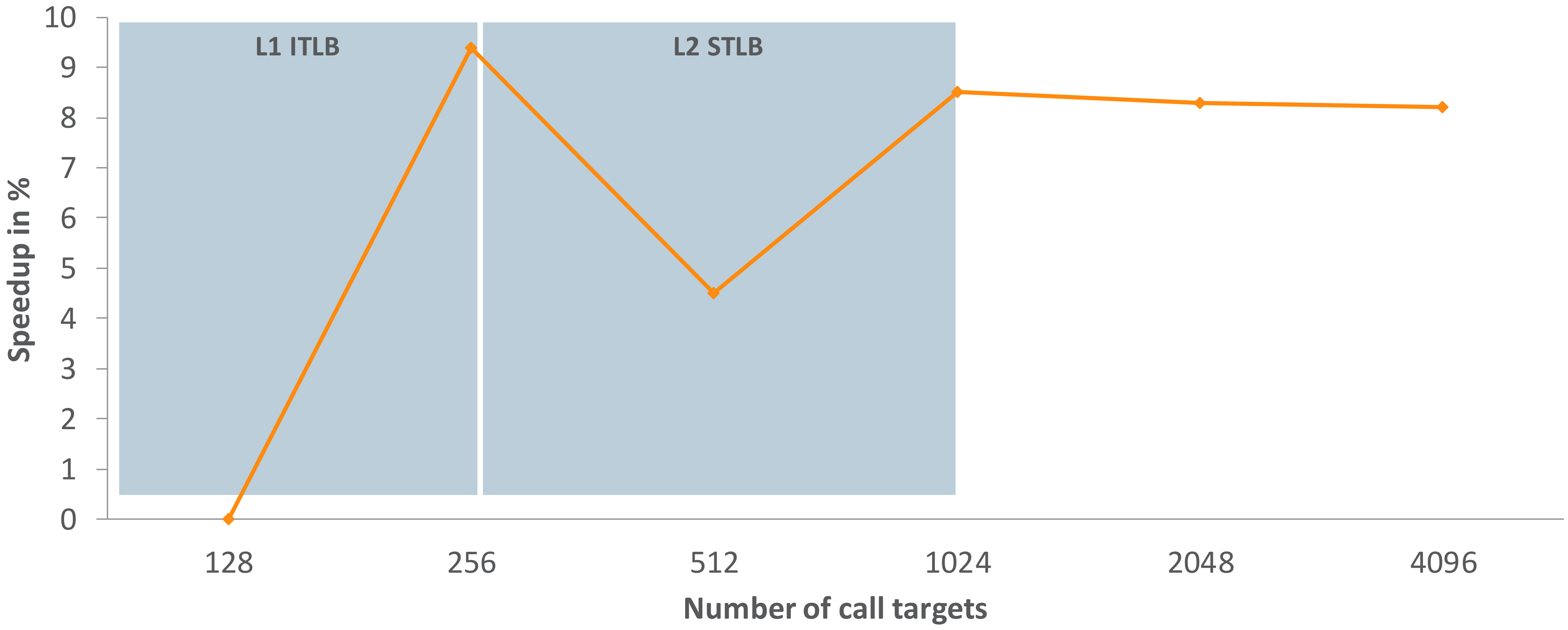
ORACLE

Evaluation: Code locality

```
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

Evaluation: Code locality



ORACLE

Evaluation: Code locality

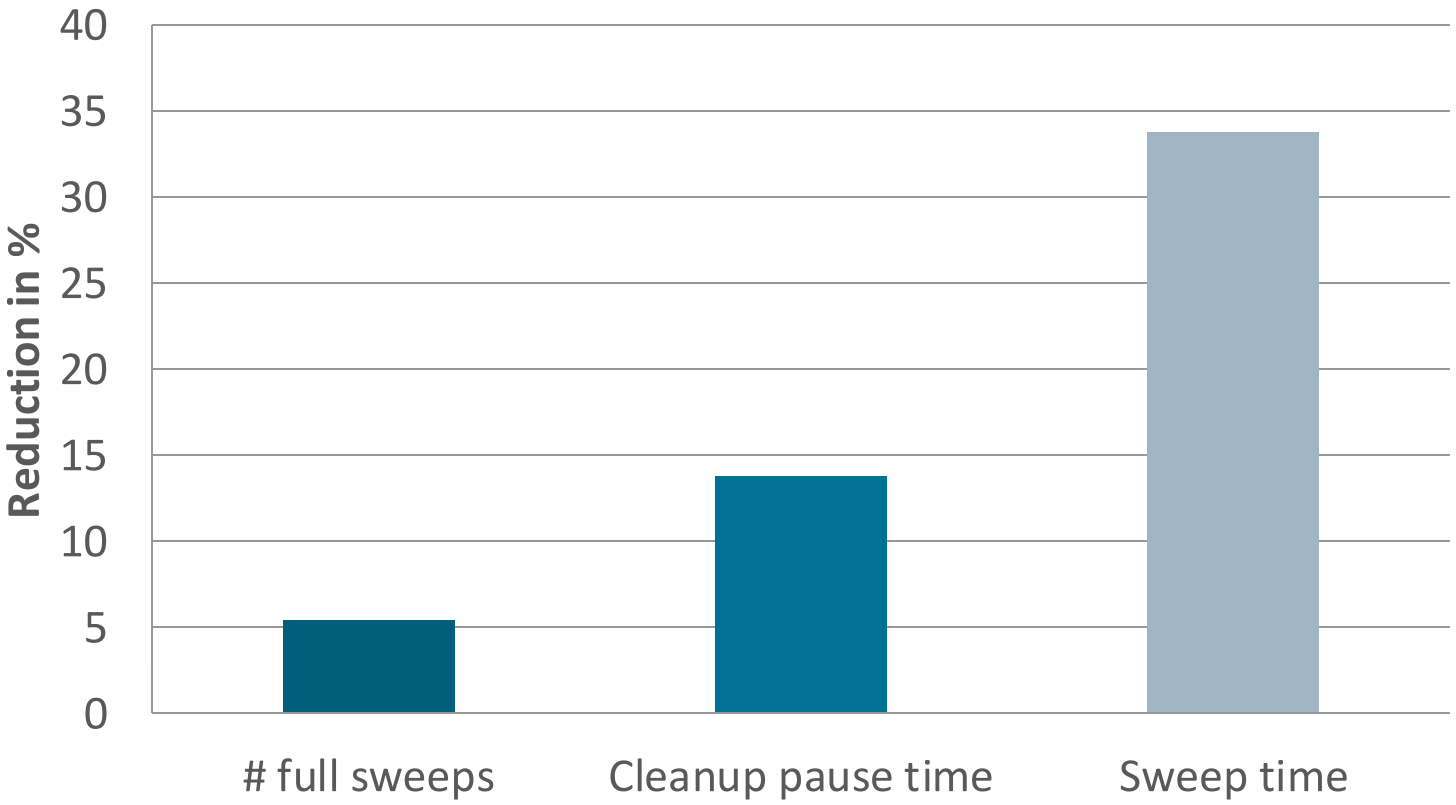
- **Instruction Cache (ICache)**
  - **14% less** ICache misses
- **Instruction Translation Lookaside Buffer (ITLB[1])**
  - **44% less** ITLB misses
- **Overall performance**
  - **9% speedup** with microbenchmark

[1] caches virtual to physical address mappings to avoid slow page walks

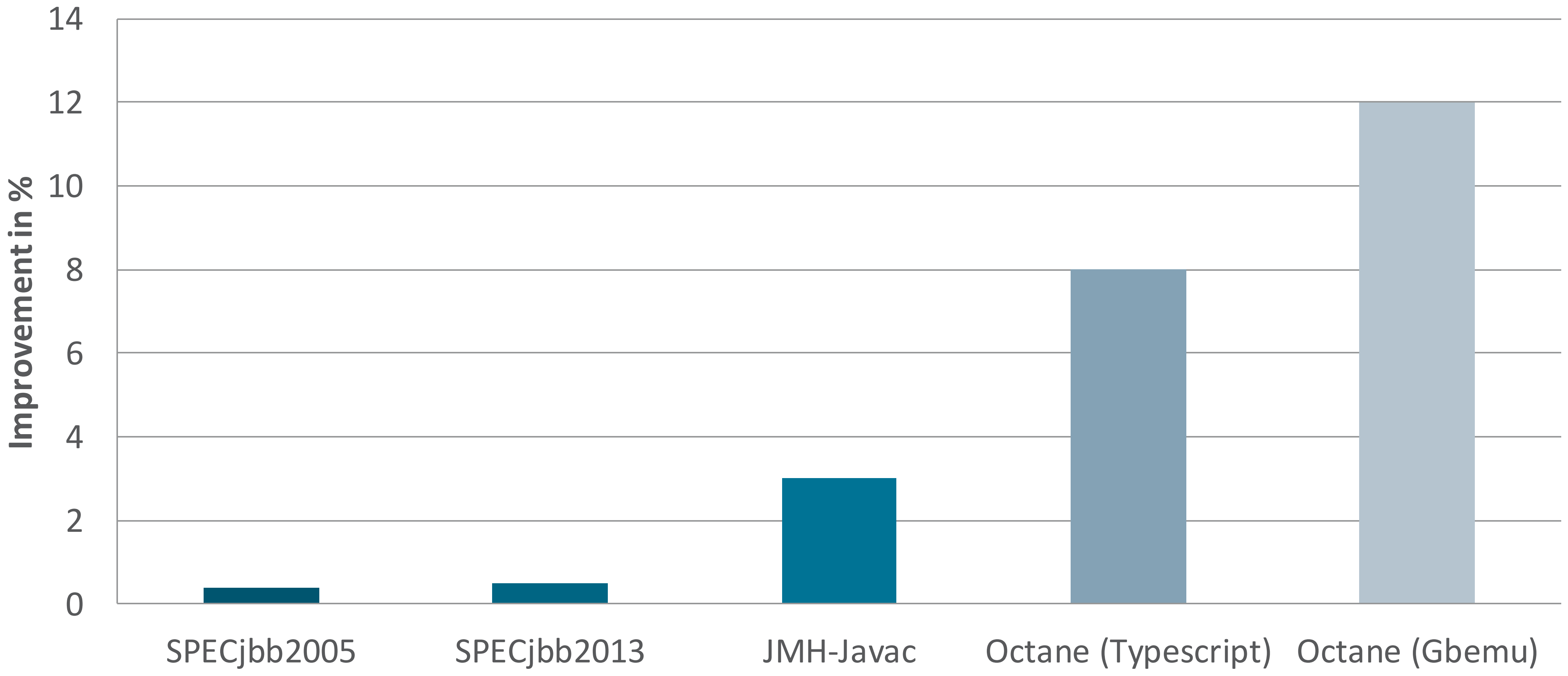
ORACLE

Evaluation: Responsiveness

- **Sweeper (GC for compiled code)**



Evaluation: Performance



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**

ORACLE

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

ORACLE

Java Strings

```java
public class HelloWorld {
    public static void main(String[] args) {
        String myString = "HELLO";
        System.out.println(myString);
    }
}
```

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

`char value[] =`

H	E	L	L	O
0x0048	0x0045	0x004C	0x004C	0x004F

UTF-16 encoded

2 bytes

ORACLE

"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:

	H	E	L	L	O
`char value[] =`	0x**00**48	0x**00**45	0x**00**4C	0x**00**4C	0x**00**4F
	2 bytes				

- But is this a problem in **real life**?

ORACLE

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

ORACLE

Design

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

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

- Additional 'coder' field indicates which encoding is used

	H		E		L		L		O		
`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**
	H	E	L	L	O	

ORACLE

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 0x00 0x4C 0x00 0x4F
```

UTF-16 encoded

~~Compression~~ ↓ ↑ **Inflation**

```
byte value[] = 0x48 0x45 0x4C 0x4C 0x4F
```

Latin-1 encoded

ORACLE

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

```
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

	Performance ns/op			Allocated b/op		
	1	10	100	1	10	100
Baseline	149	153	231	888	904	1680
CS disabled	152	150	230	888	904	1680
CS enabled	142	139	169	504	512	904

- **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-jOps improvement
- **Weblogic (startup)**
  - 10% footprint reduction
  - 5% startup time improvement
- **Released with JDK 9**

ORACLE

Outline

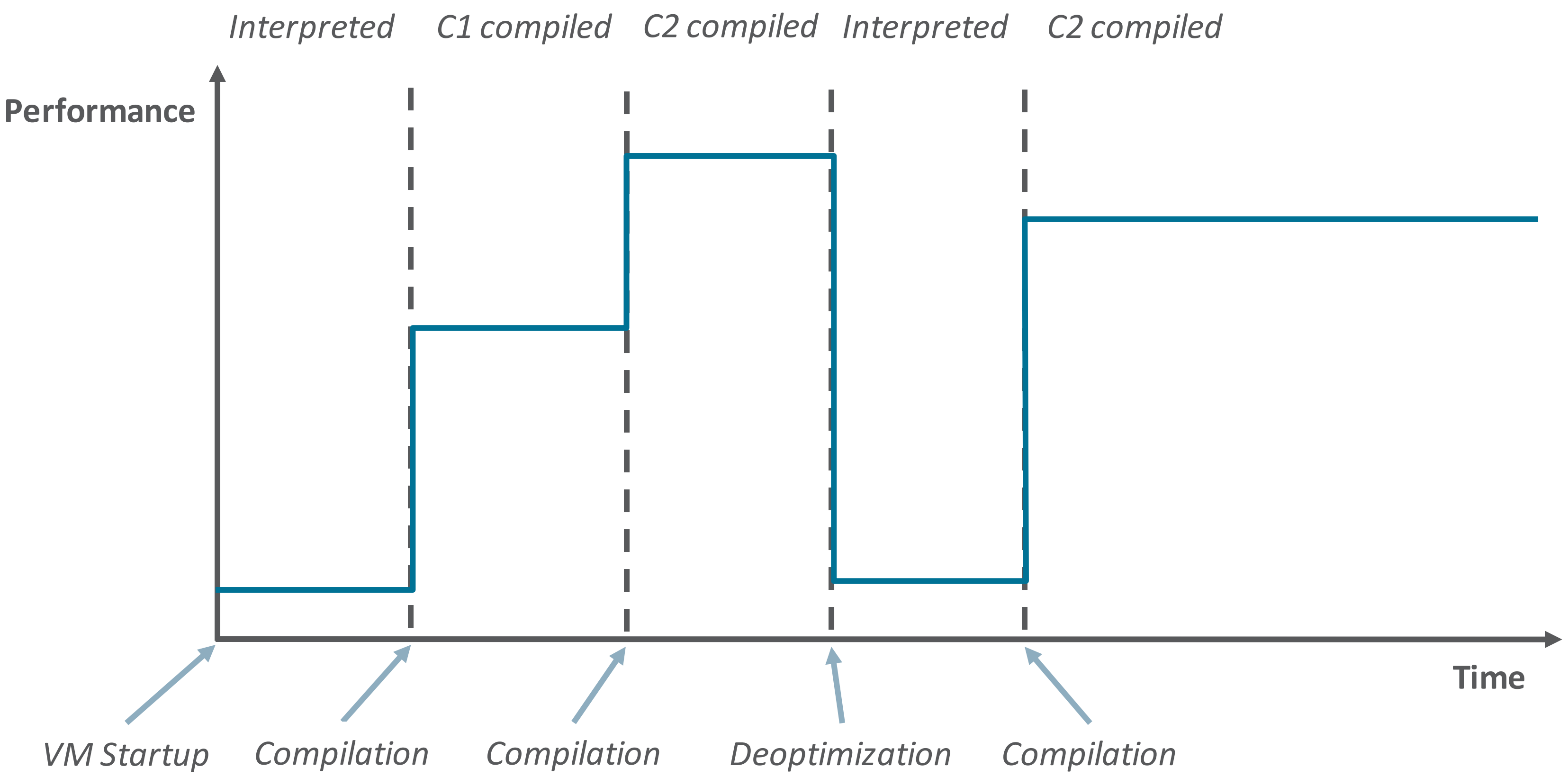
- **Intro: Why virtual machines?**
- **Part 1: The Java HotSpot VM**
  – JIT compilation in HotSpot
  – Tiered Compilation
- **Part 2: What's new in Java**
  – Segmented Code Cache
  – Compact Strings
  – Ahead-of-time Compilation
  – Value Types

ORACLE

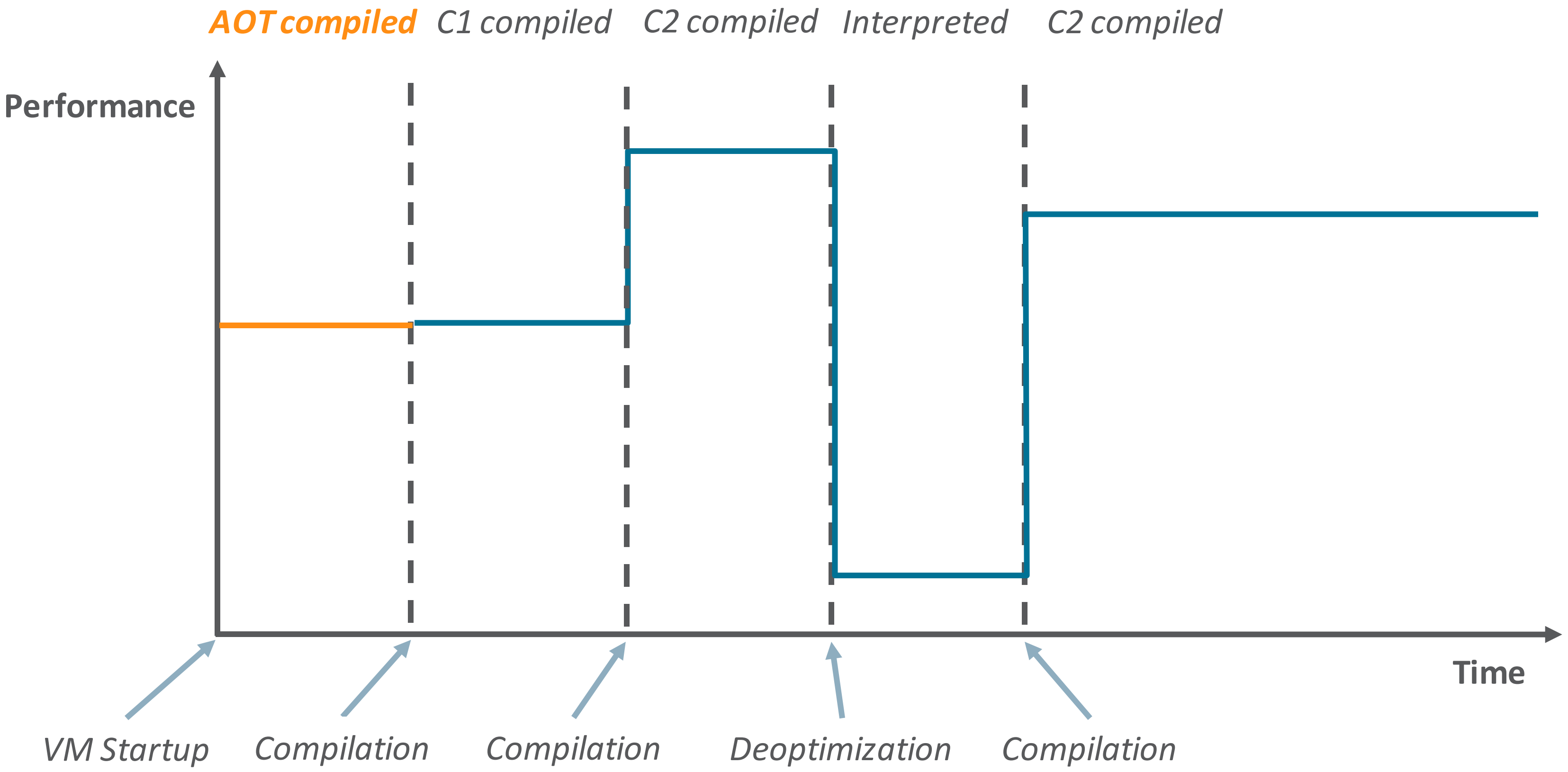
Ahead-of-Time Compilation

- **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)



ORACLE

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

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

ORACLE

Value Types

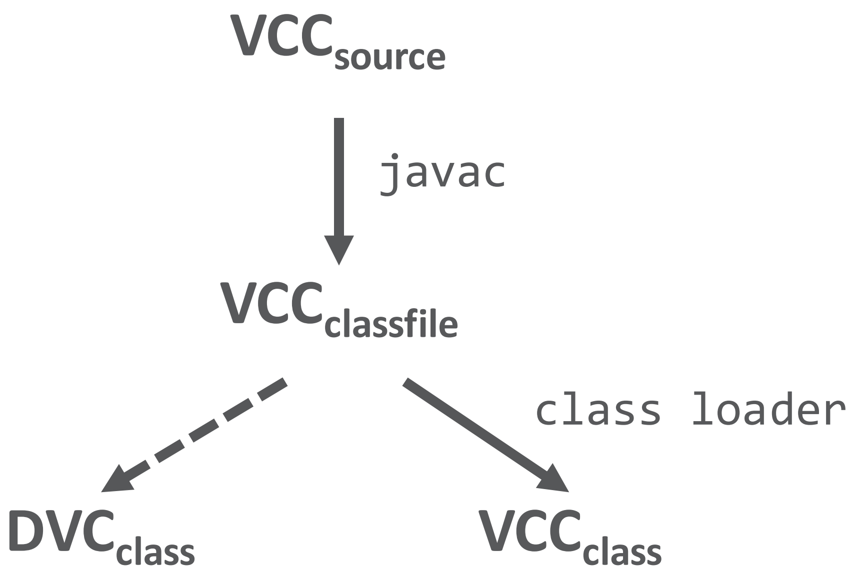
- **Value types are immutable, identityless aggregates**
  - User defined primitives
  - Non-synchronizable, non-nullable
  - *"Codes like a class, works like an int!"*
- **Introduced for performance**
  - Better spatial locality (no indirection, no header)
  - Avoid heap allocations to reduce GC pressure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

ORACLE

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

ORACLE

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
  }
  ...
}
```

ORACLE

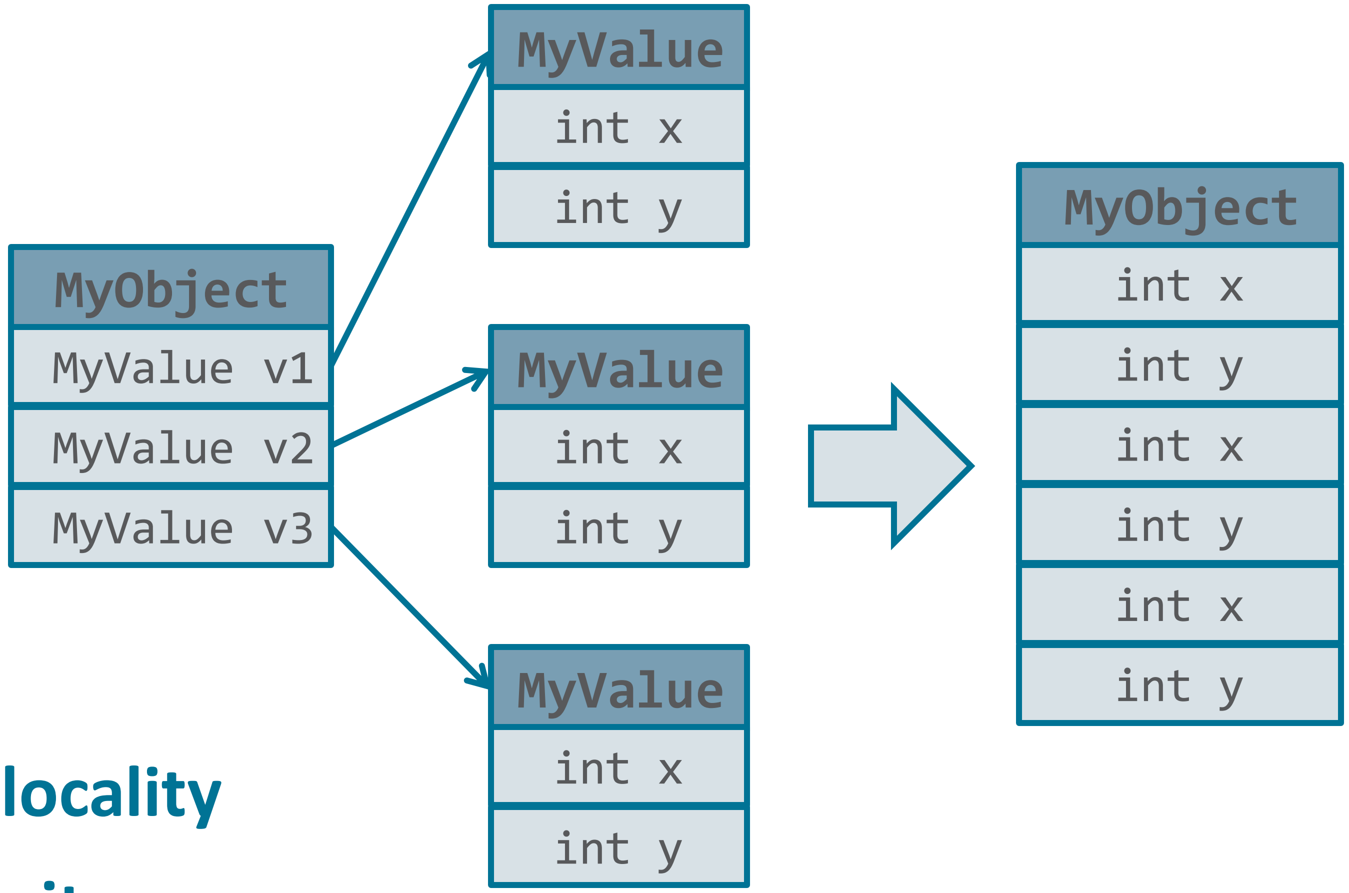
Storage formats

- 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

ORACLE

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

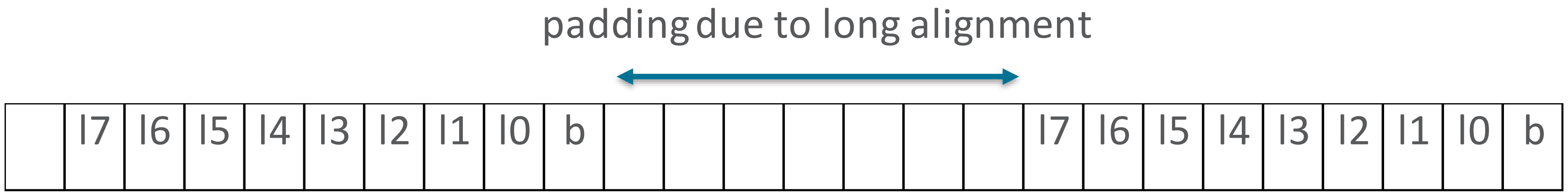
- Only for non-static fields
- Works both for object and value type holders
- Requires pre-loading of value types to determine field size
- Flattened fields keep their layout (no intermixing)
- Optional via `-XX:ValueFieldMaxFlatSize`

ORACLE

Value Type Array Flattening

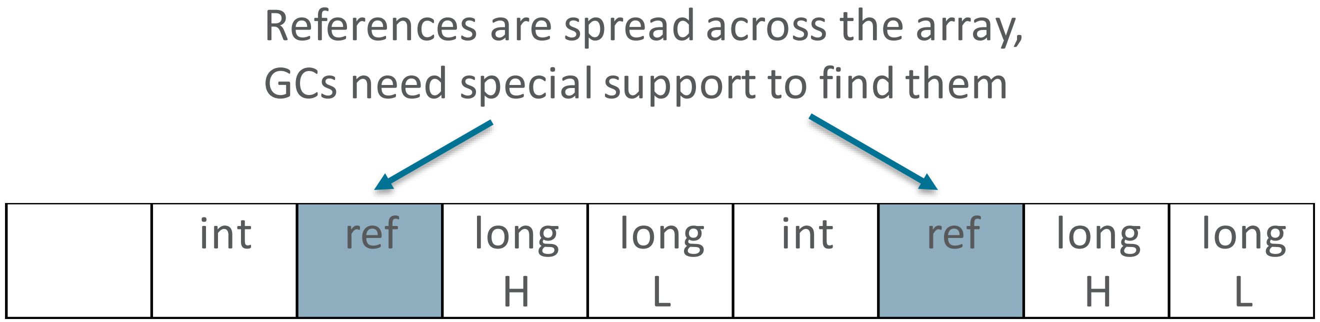
```
__ByValue final class MyValue1 {
  final long l;
  final byte b;
}
```

MyValue1[] array =




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

MyValue2[] array =



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

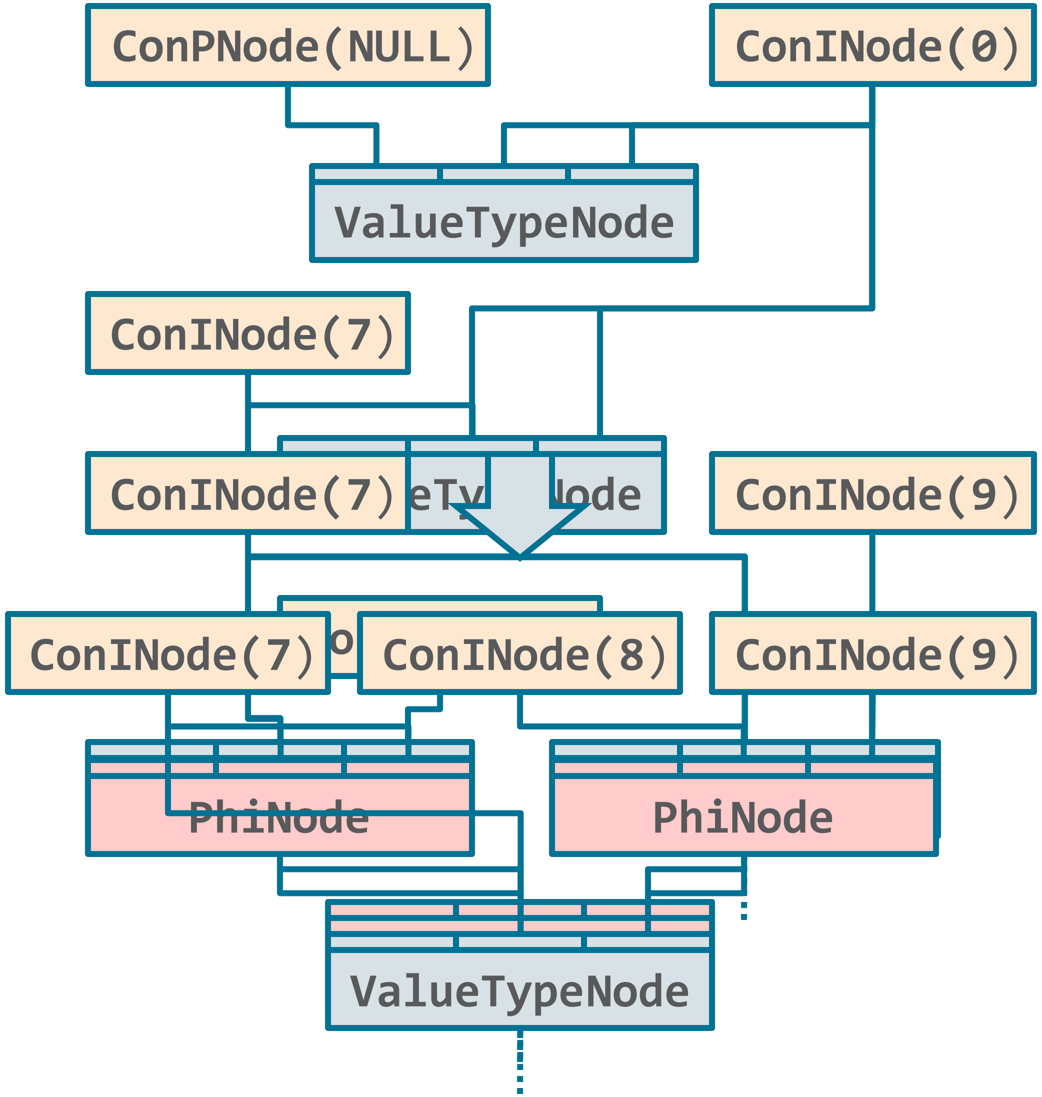
Avoiding value type allocations

- **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!**

ORACLE

IR representation

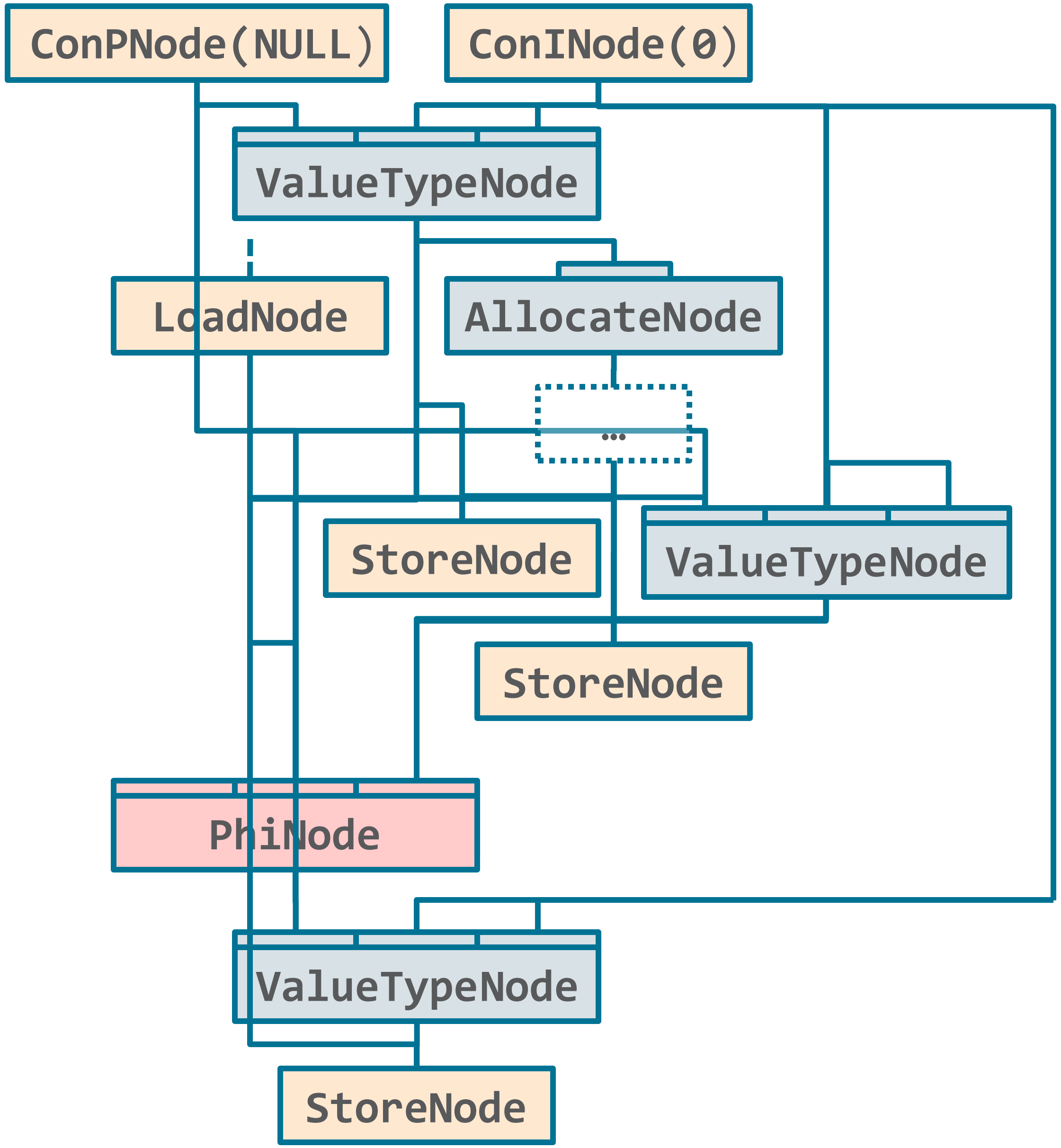
```
__ByValue final class MyValue {
  final int x, y;
  ...
}

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



ORACLE

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
```

ORACLE

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
```

ORACLE

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

ORACLE

Calling convention

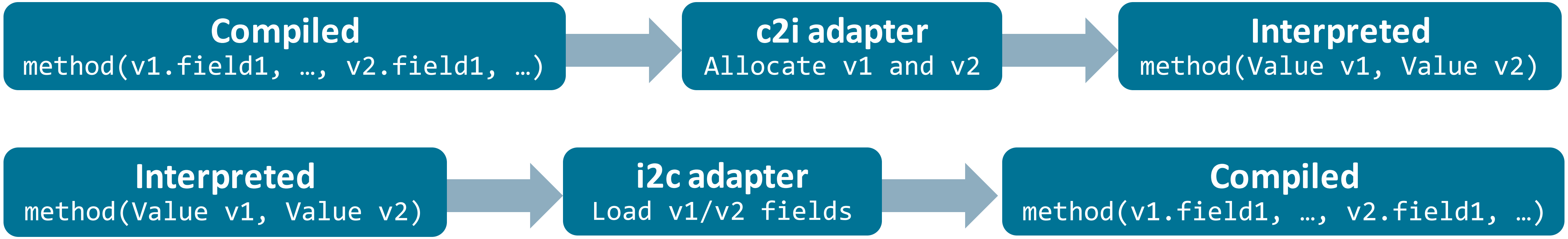
1) Calling a method with a value type argument

- **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

ORACLE

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**

ORACLE

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

ORACLE

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