Compilation in the HotSpot VM

Zoltán Majó

HotSpot Compiler Team
Oracle Corporation

December 2015

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.

References

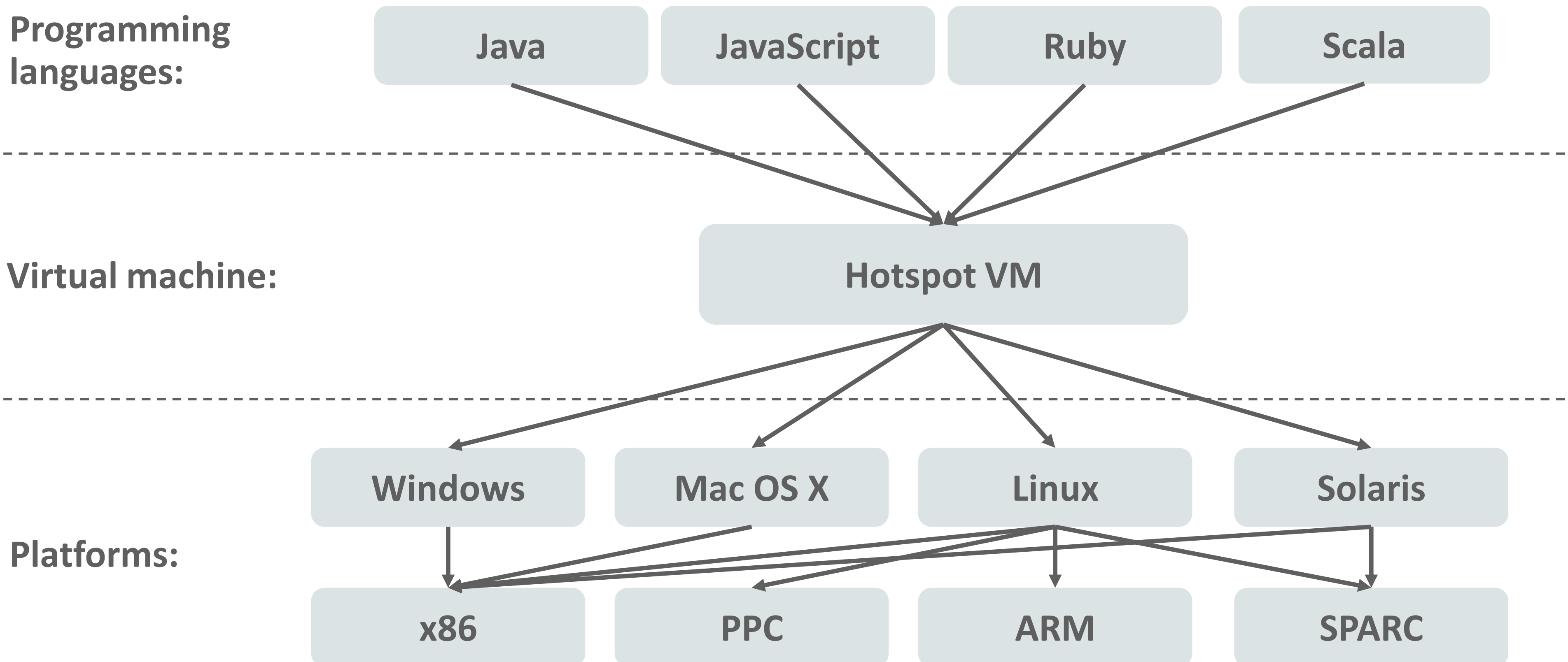
Some of the material presented here is based on

Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, David Cox:

Design of the Java HotSpot™ client compiler for Java 6.

[*TACO 5(1) (2008)*]

HotSpot: Multi-language virtual machine



Outline

- **Overview of the HotSpot *Java* VM**
- **Compilation in HotSpot**
  - Just-in-time compilation
  - Optimizations
  - Tiered compilation
  - C1 compiler
  - C2 compiler
- **OpenJDK project**
- **Future of HotSpot**

Stages of a Java method's lifetime

Java source code

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

compile →

Java bytecodes

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

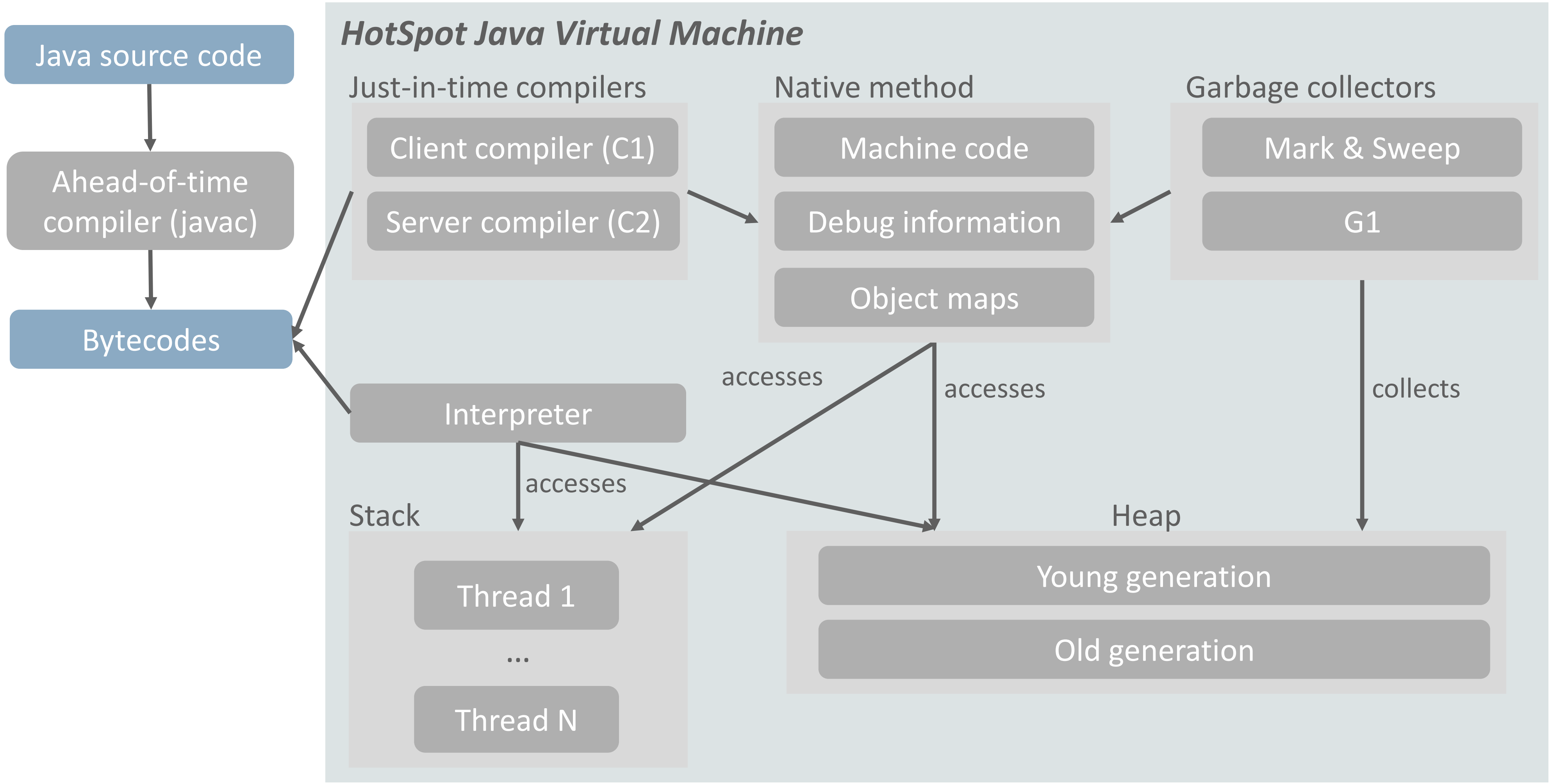
execute →

Java VM

Compilation:
- Ahead-of-time
- Tool: javac

Bytecodes:
- Instructions…
- …for an abstract machine

HotSpot's components



Major components of HotSpot

- **Runtime**
  - Interpreter(s)
  - Thread management
  - Synchronization
  - Class loading
  - and many others…
- **Heap management**
  - Garbage collectors
- **Just-in-time compilation system**

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

AOT compilation

- ***Before*** program execution
- ***Time-consuming*** optimizations
- ***Good startup/warmup*** behavior
- ***Offline*** profiling
- ***Conservative*** optimizations

JIT compilation

- ***During*** program execution
- ***Limited time*** budget
- Time is needed to ***compile "hot" methods***
- Profiling ***at runtime***
- ***Optimistic*** optimizations

Compilers in HotSpot

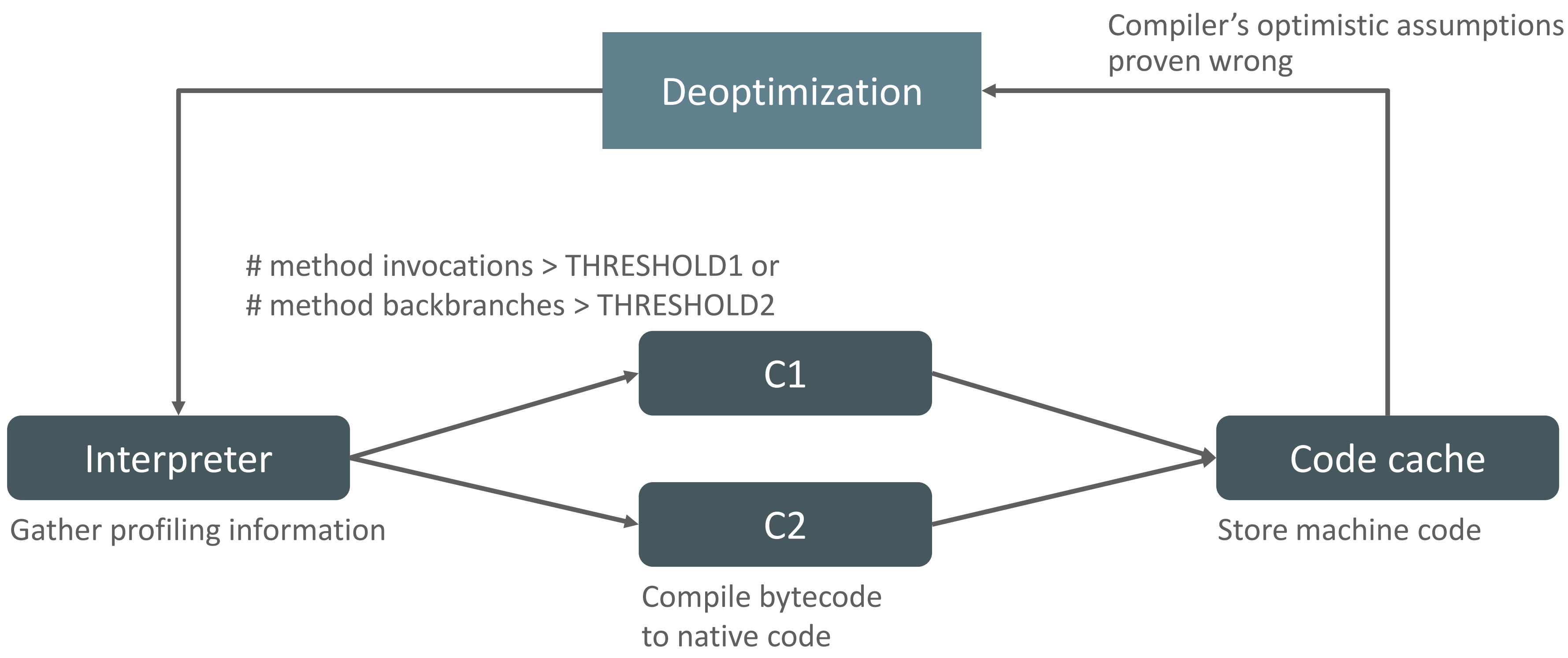
- **Tradeoff:** ***resource usage*** **vs.** ***performance of generated code***
- **C1 compiler**
  - Fast compilation
  - Small footprint
  - Code could be better
- **C2 compiler**
  - High resource demands
  - High-performance code
- **Graal**
  - Experimental compiler
  - Not part of HotSpot

Client VM (C1 compiler)

Server VM (C2 compiler)

Tiered compilation (C1 and C2 compilers)

Stages of a method's lifetime (cont'd)



Virtual call inlining

```
class A {
  void bar() { … }
}
```

```
class B extends A {
  void bar() { … }
}
```

```
A create() {
  if (…) {
    return new A()
  } else {
    return new B();
  }
}
```

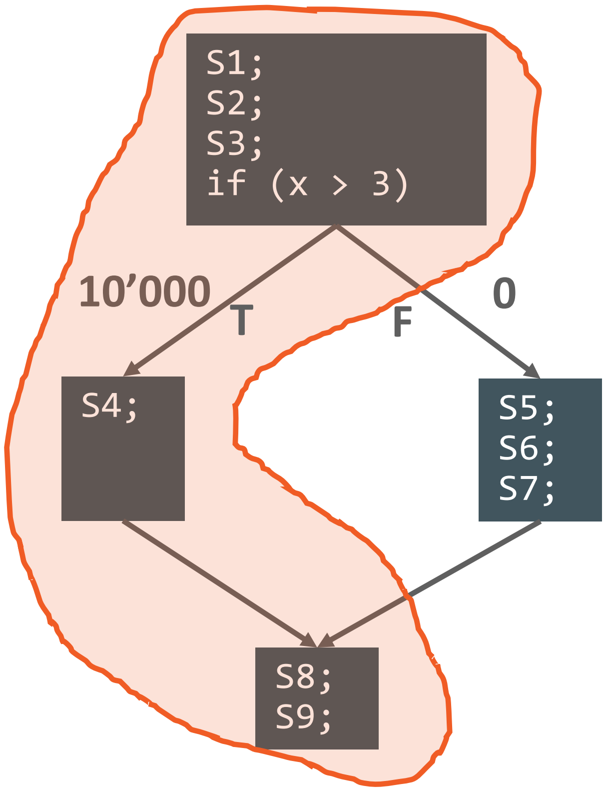
```
void foo() {
  A a = create();
  a.bar();  ← inline?
}
```

Inline if only A is loaded

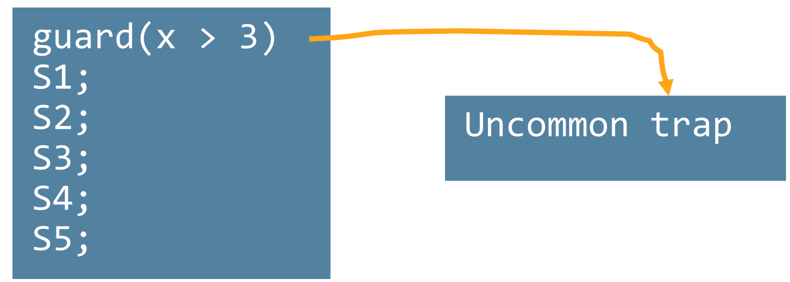
- Record foo's dependence on class hierarchy
- Check dependence when new class is loaded
- Deoptimize if assumed target is wrong

Hot path compilation

Control flow graph



Generated code



Deoptimization

- **Compiler's optimistic assumption proven wrong**
- **Switch execution from compiled code to interpreter**
  - Reconstruct state of interpreter
  - Complex implementation
- **Compiled code**
  - Possibly thrown away
  - Possibly recompiled

Outline

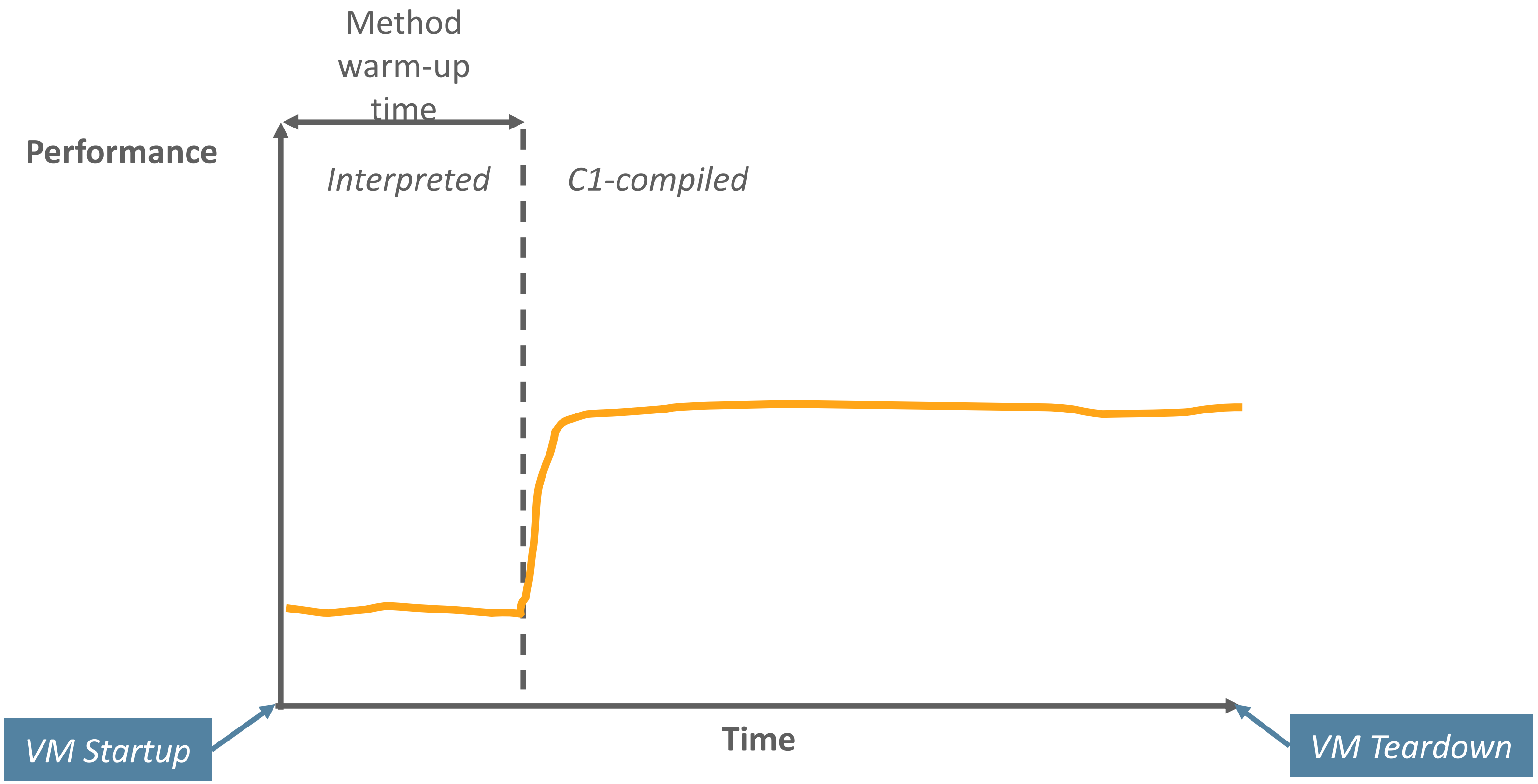
- **Overview of the HotSpot Java VM**
- **Compilation in HotSpot**
  - Just-in-time compilation
  - Optimizations
  - Tiered compilation
  - C1 compiler
  - C2 compiler
- **OpenJDK project**
- **Future of HotSpot**

Tiered compilation

- **Combine the benefits of**
  - Interpreter: Fast startup
  - C1: Fast warmup
  - C2: High peak performance

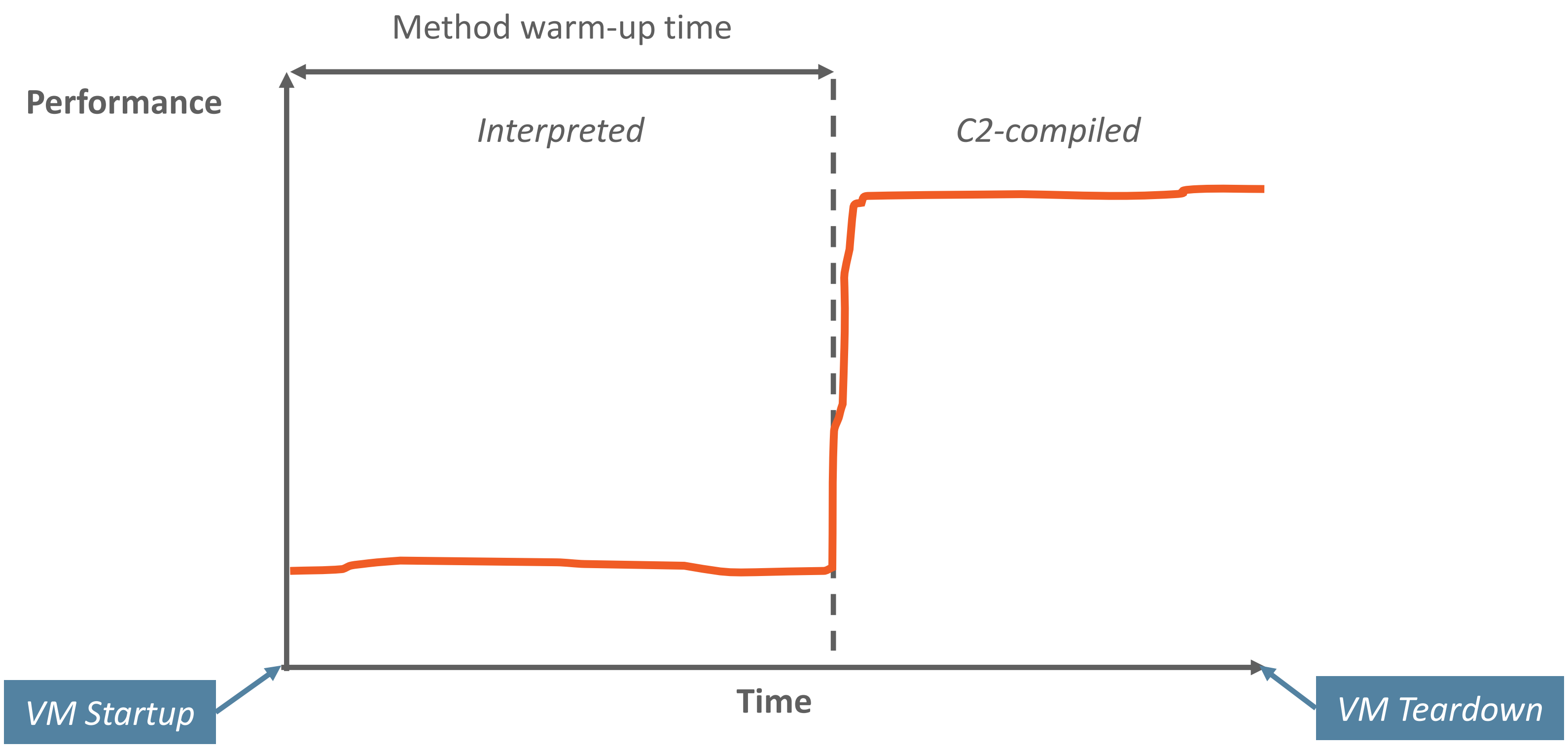
Benefits of tiered compilation (artist's concept)

Client VM (C1 only)



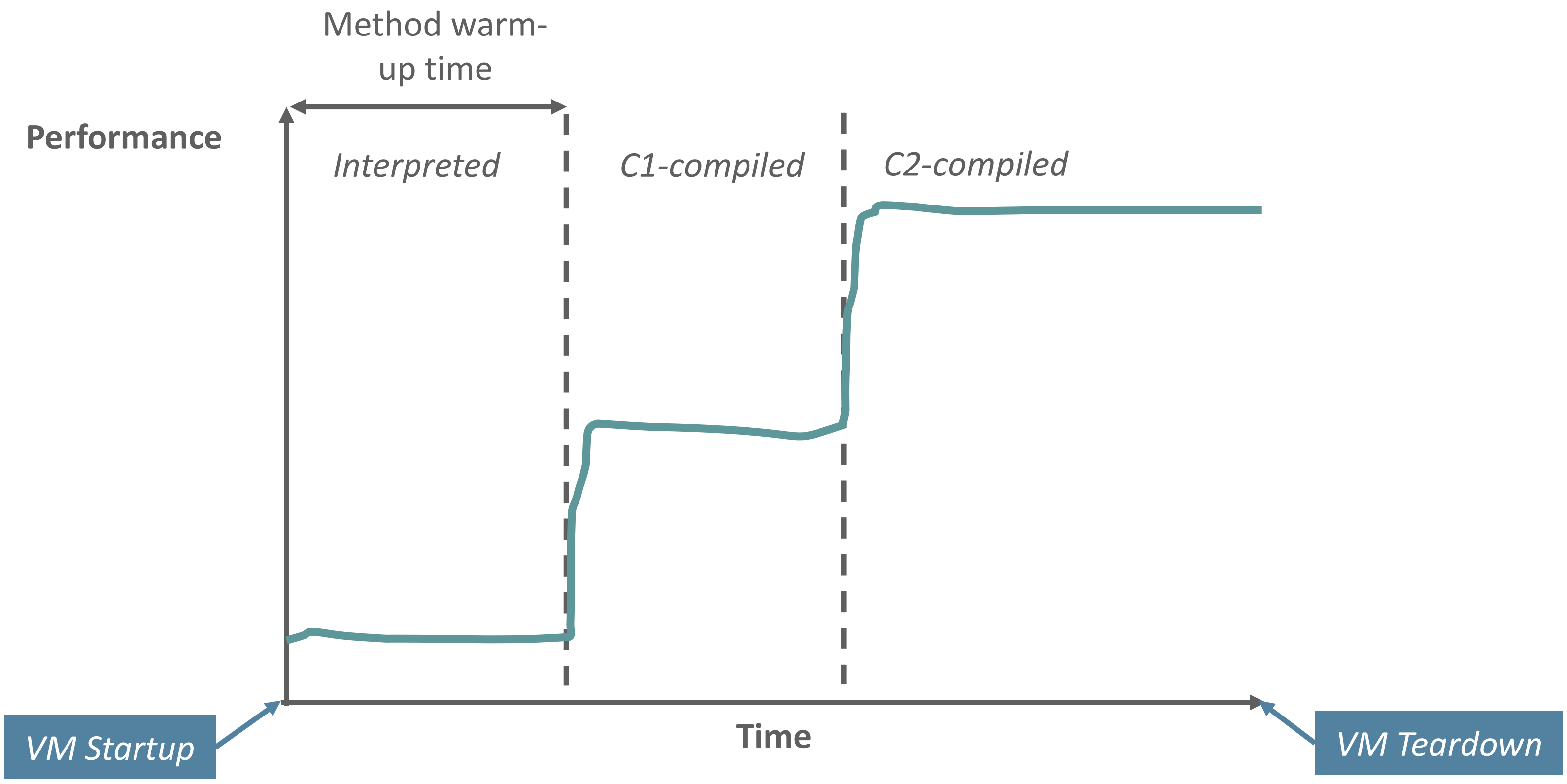
Benefits of tiered compilation (artist's concept)

Server VM (C2 only)



Benefits of tiered compilation (artist's concept)

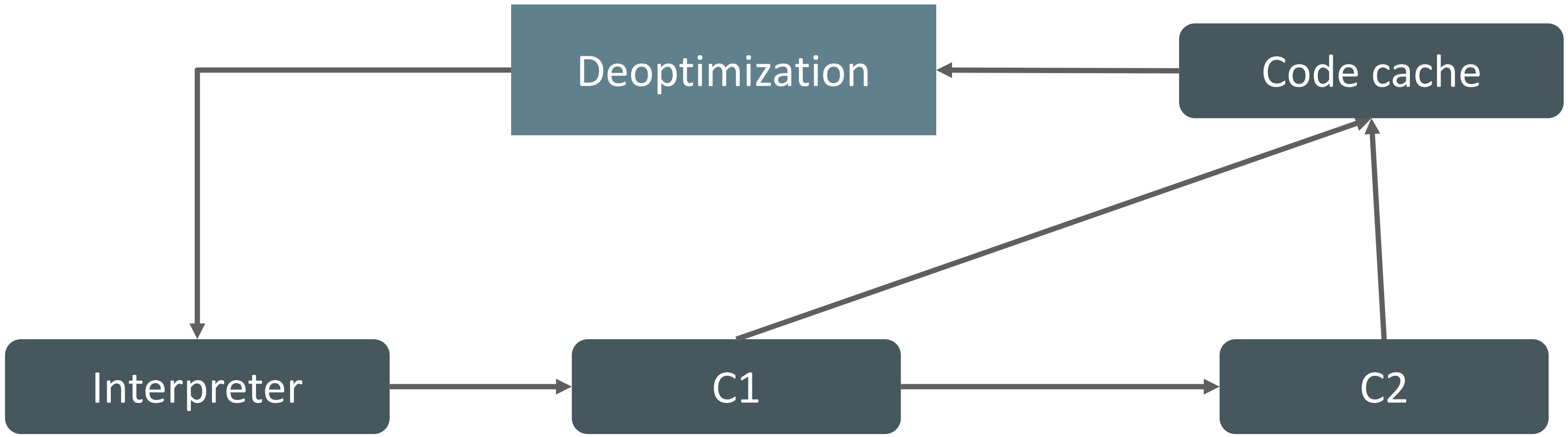
Tiered compilation



Tiered compilation

- **Combine the benefits of**
  - Interpreter: Fast startup
  - C1: Fast warmup
  - C2: High peak performance
- **Additional benefits**
  - More accurate profiling information
- **Drawbacks**
  - Complex implementation
  - Careful tuning of compilation thresholds needed
  - More pressure on code cache – Tobias will tell you more about that

A method's lifetime (w/ tiered compilation)

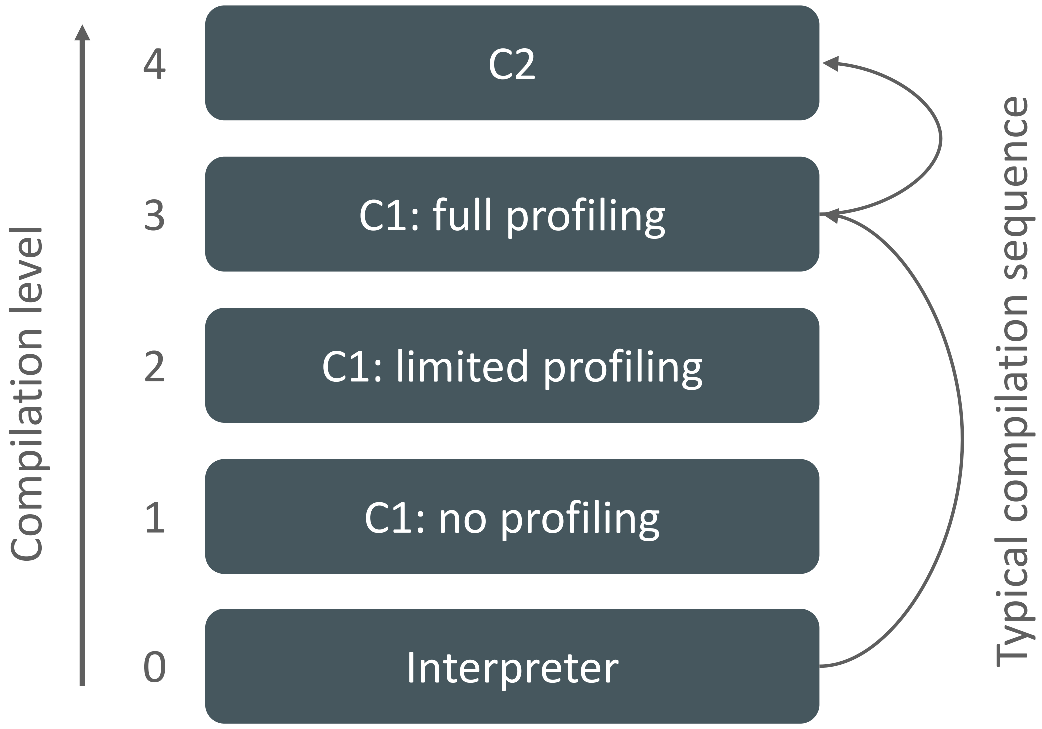


Collect profiling information

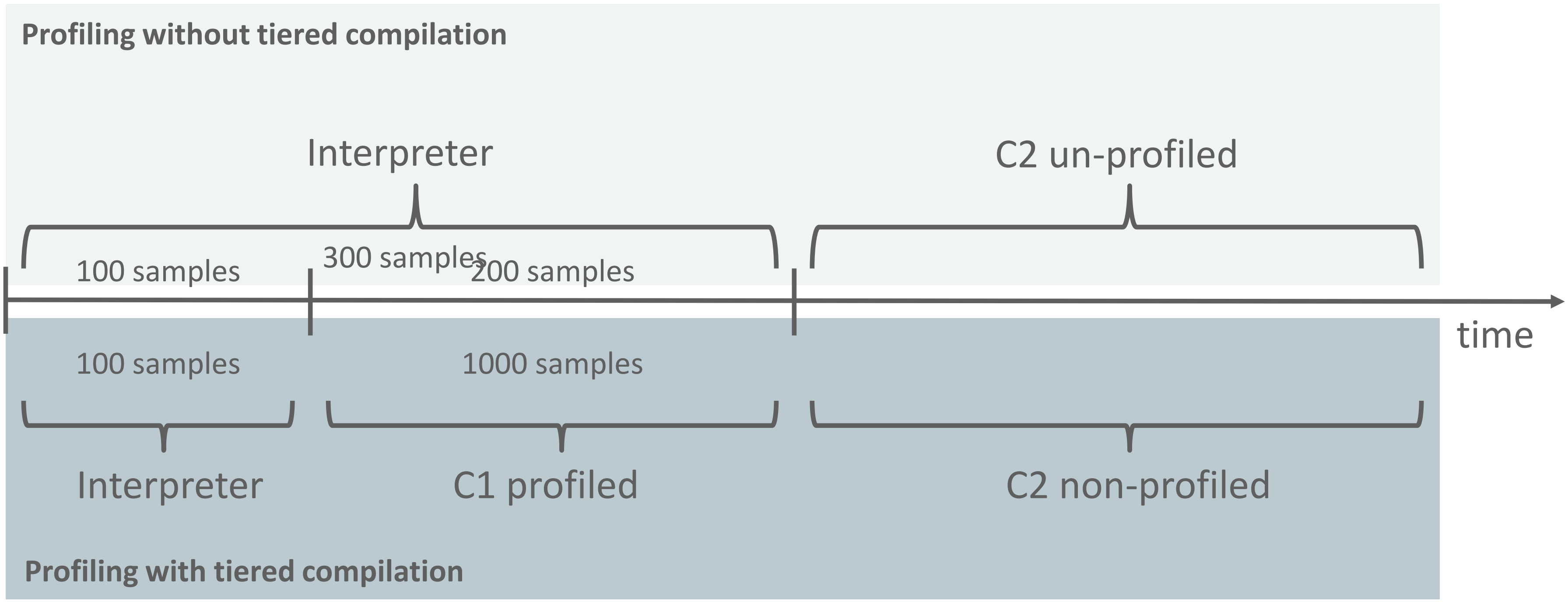
Generate code quickly
Continue collecting profiling information

Generate high-quality code
Use profiling information

Tiered compilation in detail



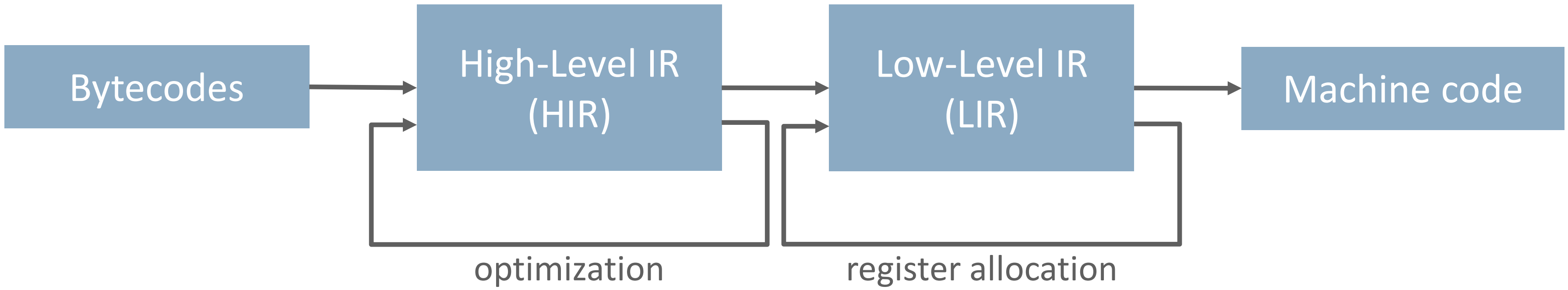
More accurate profiling



Outline

- **Overview of the HotSpot Java VM**
- **Compilation in HotSpot**
  - Just-in-time compilation
  - Optimizations
  - Tiered compilation
  - C1 compiler
  - C2 compiler
- **OpenJDK project**
- **Future of HotSpot**

Design of the C1 compiler



High-Level Intermediate Representation

- **Platform independent**
- **SSA form**
  - One assignment for every variable

Static Single Assignment Form (SSA)

Java code

```
a = b + c
a = a + 1
```

SSA form

$$a_1 = b_1 + c_1$$
$$a_2 = a_1 + 1$$

Static Single Assignment Form (SSA)

Java code

```
if (x == 1) {
  a = 1
} else {
  a = 2
}
b = a + 1
```

SSA form

$$
\begin{array}{l}
\texttt{if } (x_1 \texttt{ == } 1) \ \{ \\
\quad a_1 = 1 \\
\} \texttt{ else } \{ \\
\quad a_2 = 2 \\
\} \\
a_3 = \mathrm{phi}(a_1, a_2) \\
b_1 = a_3 + 1
\end{array}
$$

- **More about SSA in the Advanced Compiler Design lecture**

High-Level Intermediate Representation

- **Platform independent**
- **SSA form**
  - One assignment for every variable
- **Requires two passes over the bytecodes**
  - ***Pass 1***: Detect boundaries of basic blocks
    Simple loop analysis
  - ***Pass 2***: Create instructions by abstract interpretation of bytecodes
    Link basic blocks to control flow graph
- **HIR instruction: represents an operation and its result**

HIR Example

- **Time for a demo…**
- **Command line to obtain C1 graph**
```
java -XX:+PrintCompilation
-XX:CompileCommand=compileonly,AClass::main
-Xcomp
-XX:TieredStopAtLevel=1
-XX:+PrintCFGToFile AClass # The method of interest
is AClass::main
```
- **Remember: you need a *fastdebug* build**

Low-Level Intermediate Representation (LIR)

- **Similar to machine code**
- **Does not use SSA forms**
  - Phi functions of HIR are resolved by register moves
- **Use explicit operands**
  - Virtual registers, physical registers, memory addresses, constants
- **Input to Linear Scan Register Allocator (LSRA)**
  - Maps virtual registers to physical registers

Machine code generation

- Emit appropriate machine instruction(s) for every LIR instruction
- Generate object maps
- Generate debugging information

GC support

- **GC can only happen at *safepoints***
  - Loop back branches
  - Before method return
- **Object maps**
  - Information which registers contain references to objects
- **Implementation**

```
test   %eax,0x163eae66(%rip)   # 0x00007f2c07760000
```

  - Access a specific page
  - Access successful: no safepoint request
  - Access throws an exception: enter safepoint routine

Exception handling

- **Instructions that throw an exception do not end a basic block**
- **Exception in machine code**
  – Runtime searches for exception handler
- **Example: Null check**

Implicit null check

```
int foo(Dummy d) {
  return d.x;
}
```

```
# {method} {0x00007f2bed4e8330} 'foo' '(LDummy;)I' in 'Test'
 # parm0:    rsi:rsi   = 'Dummy'
 #           [sp+0x40]  (sp of caller)
 ;;  block B1 [0, 0]

 0x00007f2bf1375180: mov    %eax,-0x16000(%rsp)
 0x00007f2bf1375187: push   %rbp
 0x00007f2bf1375188: sub    $0x30,%rsp        ;*aload_0
                                             ; - Test::foo@0 (line 12)

 ;;  block B0 [0, 4]

 0x00007f2bf137518c: mov    0xc(%rsi),%eax    ;*getfield x
                                             ; - Test::foo@1 (line 12)
                                             ; implicit exception: dispatches to 0x00007f2bf137519b
 0x00007f2bf137518f: add    $0x30,%rsp
 0x00007f2bf1375193: pop    %rbp
 0x00007f2bf1375194: test   %eax,0x163eae66(%rip)       # 0x00007f2c07760000
                                             ;   {poll_return}
 0x00007f2bf137519a: retq
 ;; ImplicitNullCheckStub slow case
 0x00007f2bf137519b: callq  0x00007f2bf0fd8420  ; OopMap{off=32}
                                             ;*getfield x
                                             ; - Test::foo@1 (line 12)
                                             ;   {runtime_call}
 0x00007f2bf13751a0: mov    %rsp,-0x28(%rsp)
```


HIR Optimizations

- **Constant folding**
  - Simplify arithmetic instructions with constant operands
- **Local value numbering**
  - Eliminate common sub-expressions within a basic block
- **Method inlining**
  - Replace method call by a copy of the method body
- **Global value numbering**
  - Two instructions are equivalent if they perform the same operation on the same operands
- **Null-check elimination**

Outline

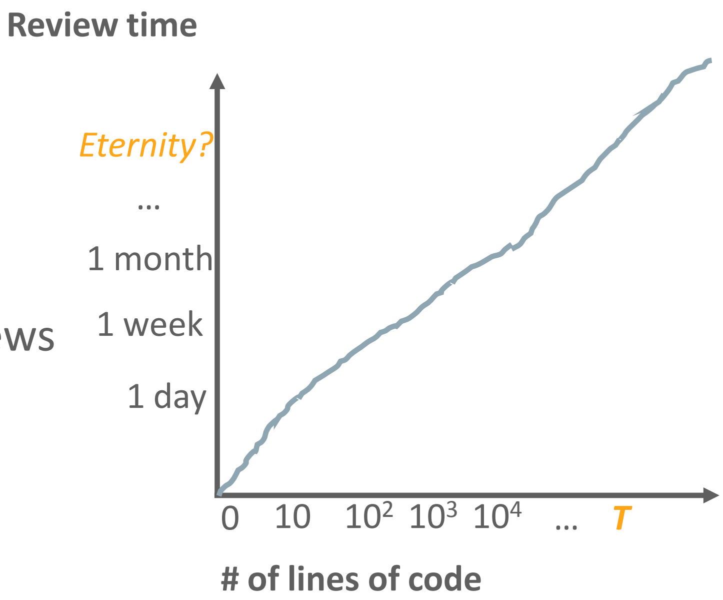
- **Overview of the HotSpot Java VM**
- **Compilation in HotSpot**
  – Just-in-time compilation
  – Optimizations
  – Tiered compilation
  – C1 compiler
  – C2 compiler
- **OpenJDK project**
- **Future of HotSpot**

C2 server compiler overview

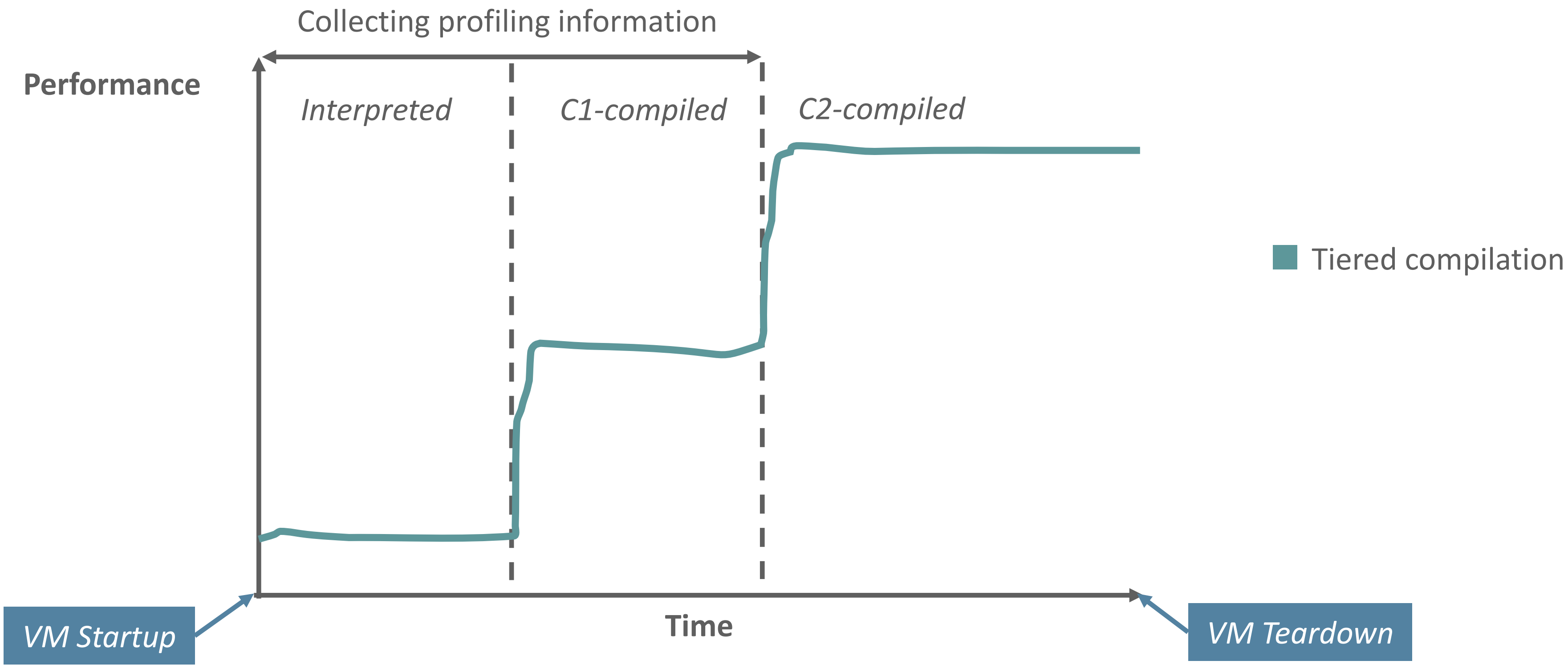
- **Highly optimizing compiler**
- **SSA form**
- **IR: Program dependence graph "Sea of nodes"**
  - No basic blocks, instructions can "float" in the graph
  - Explicit control/data dependency
  - Allows many optimizations with little effort
  - Hard to understand and debug
- **Many optimizations during parsing**
- **Graph coloring register allocator**

OpenJDK

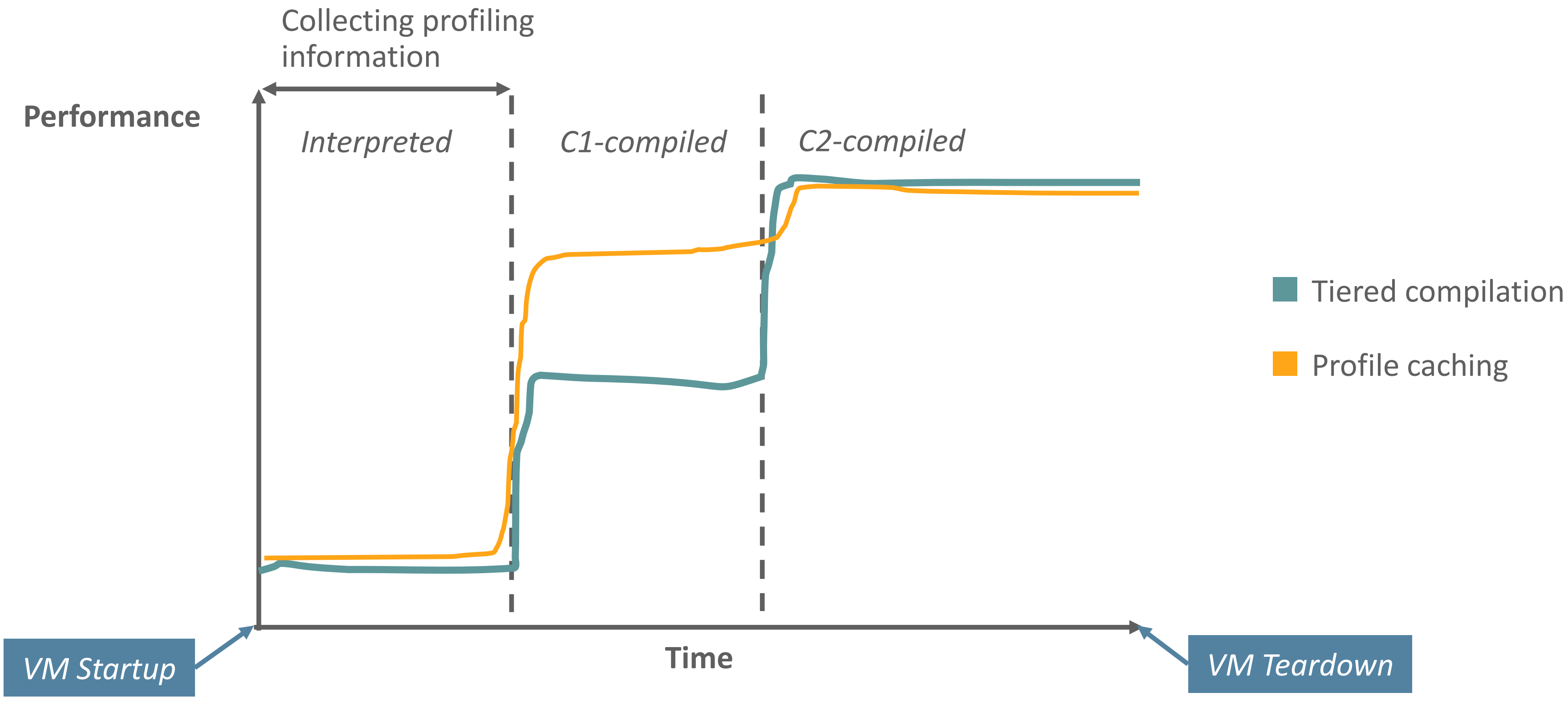
- **HotSpot is part of OpenJDK**
- **Open-source project**
- **Well-defined reviewing process**
  - Statuses: Author, Committer, Reviewer
  - Each change requires least two Reviewer's reviews
  - Advantage: Feedback, changes are traceable
  - Disadvantage: No moderation
- **OpenJDK is a good research vehicle**
  - Example: profile caching Bachelor's thesis by M Mohler



Tiered compilation

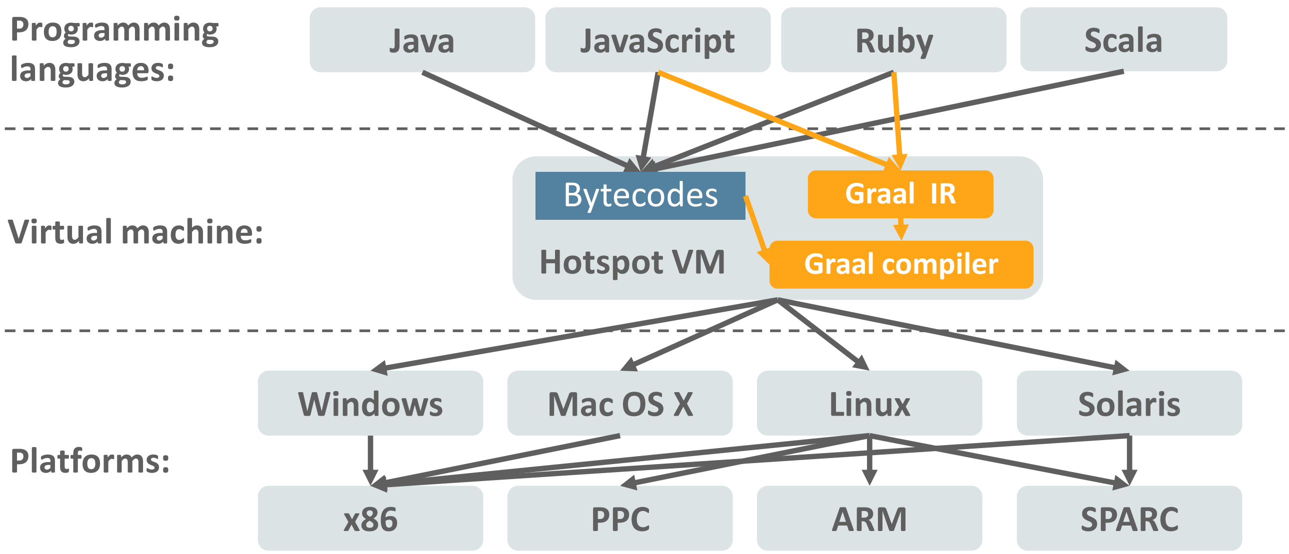


Profile caching



Future

- **Multi-language VM**



- **AOT compilation to native code (not to bytecodes)**

Thank you for your attention!

Backup slides

On-Stack Replacement

```
void foo() {
  while (condition) {
    // Do work in this block
  }
}
```

- foo() executes for a long time
- Compile hot code in foo()
- Execute compiled code instead of using the interpreter

JDK 9 Projects

Oracle
HotSpot Compiler Team
Tobias Hartmann

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

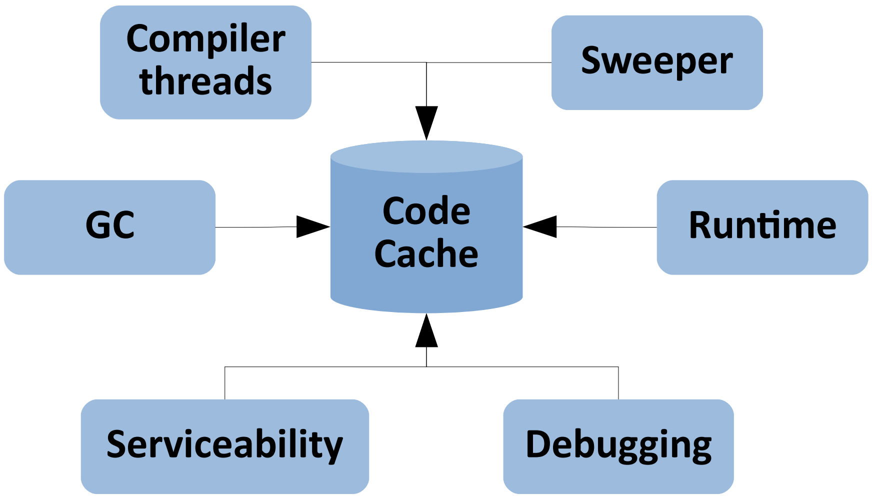
- **Segmented Code Cache**
  - Background and history
  - Challenges
  - Design and Implementation
  - Evaluation
- **Compact Strings**
  - Java String encoding
  - Analysis of Strings
  - Design and Implementation
  - Evaluation

Segmented Code Cache

Oracle
HotSpot Compiler Team
Tobias Hartmann

Code cache

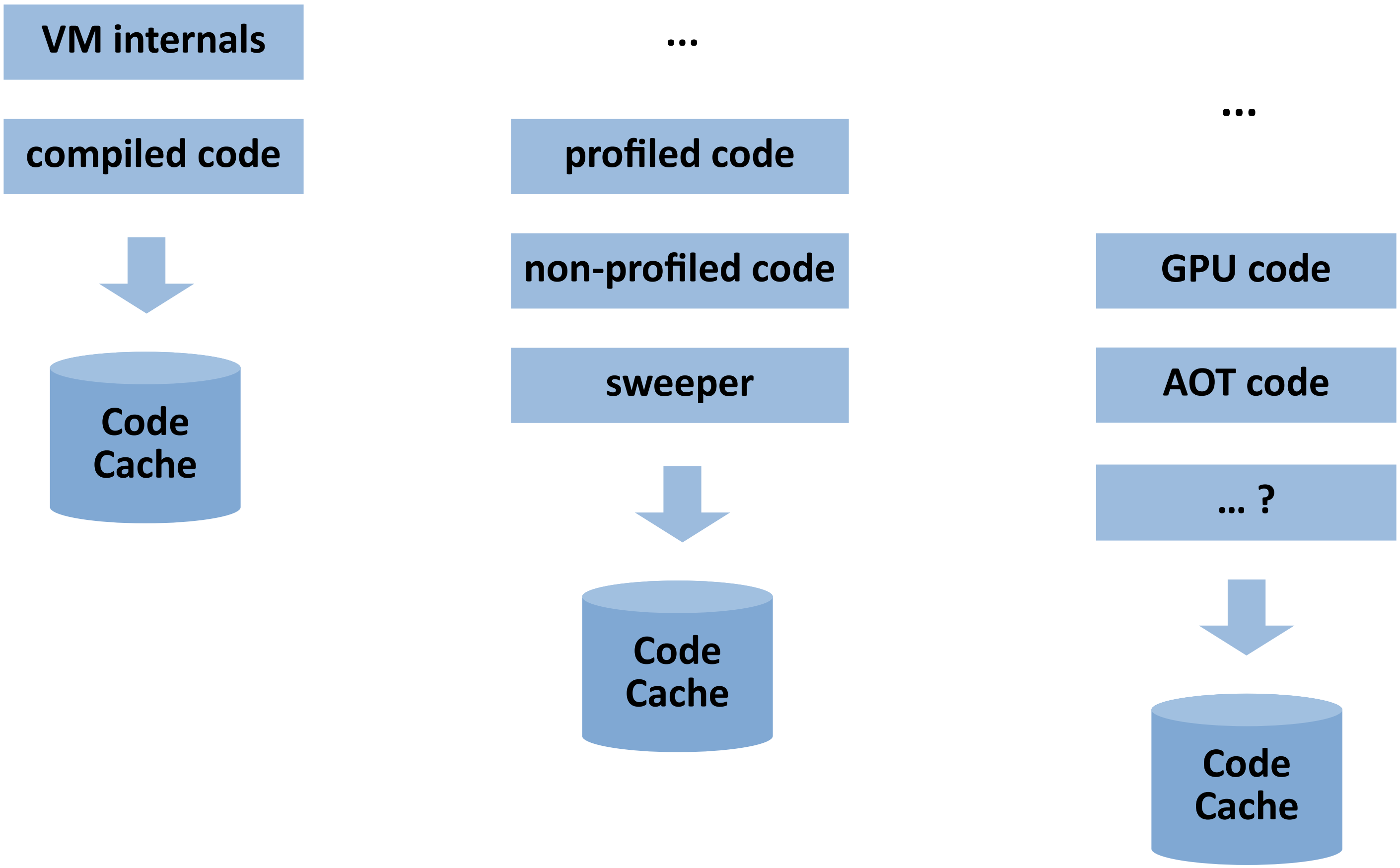
- Central component



- Continuous chunk of memory
  - Fixed size
  - Bump pointer allocation with free list

History

- JDK 6
- JDK 7/8
- JDK 9



Challenges

- **Tiered compilation increases amount of code**
  - 2 - 4 X
- **All code in one cache**
  - Different types with different characteristics
  - Access to specific code requires full iteration
- **Code cache fragmentation**

Challenges

- **Tiered compilation increases amount of code**
  - 2 - 4 X
- **All code in one cache**
  - Different types with different characteristics
  - Access to specific code requires full iteration
- **Code cache fragmentation**

- **Solution: Segmented Code Cache**

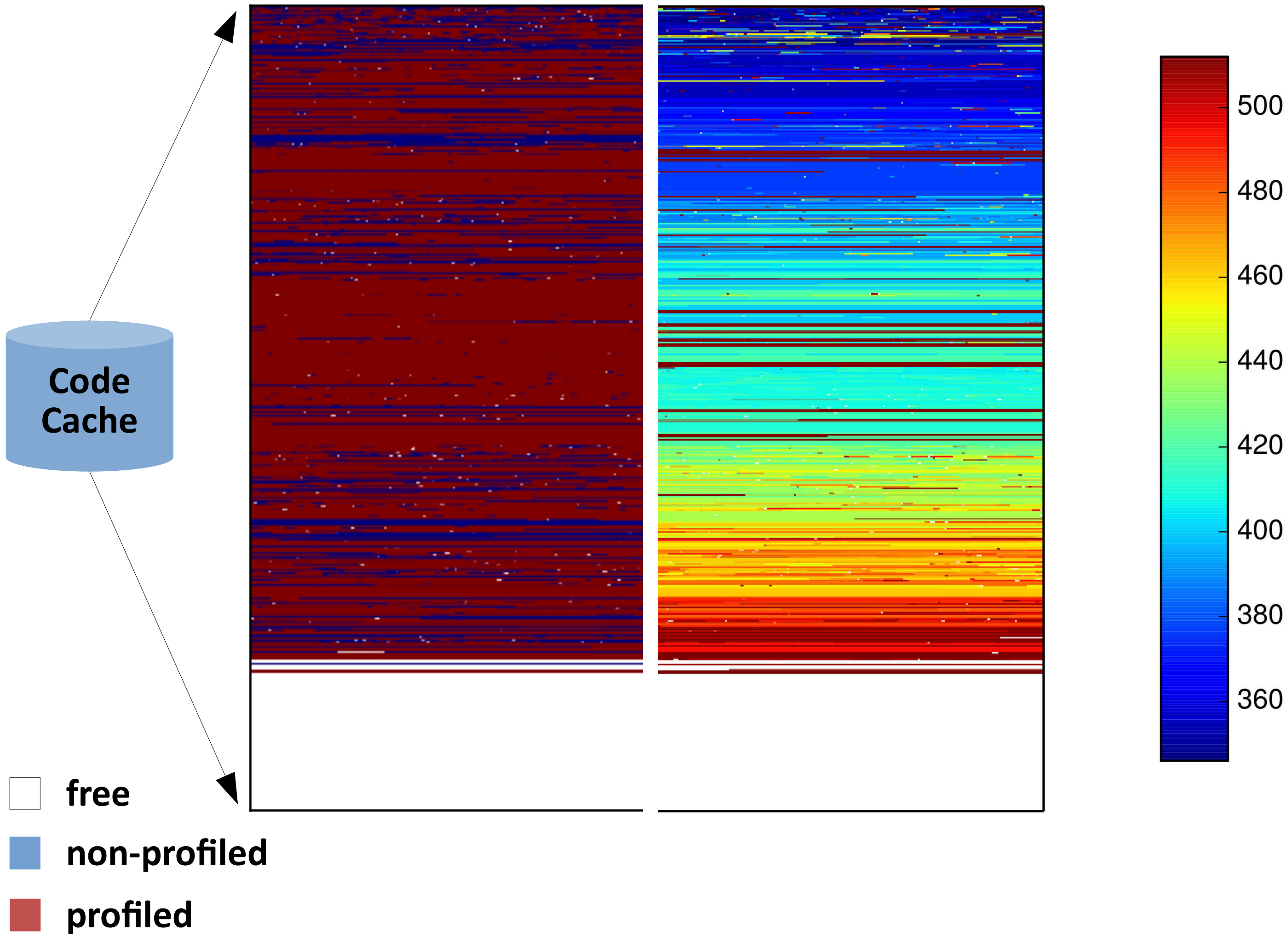
Properties of compiled code

- Lifetime
- Size
- Cost of generation
- Level of optimization

Types of compiled code

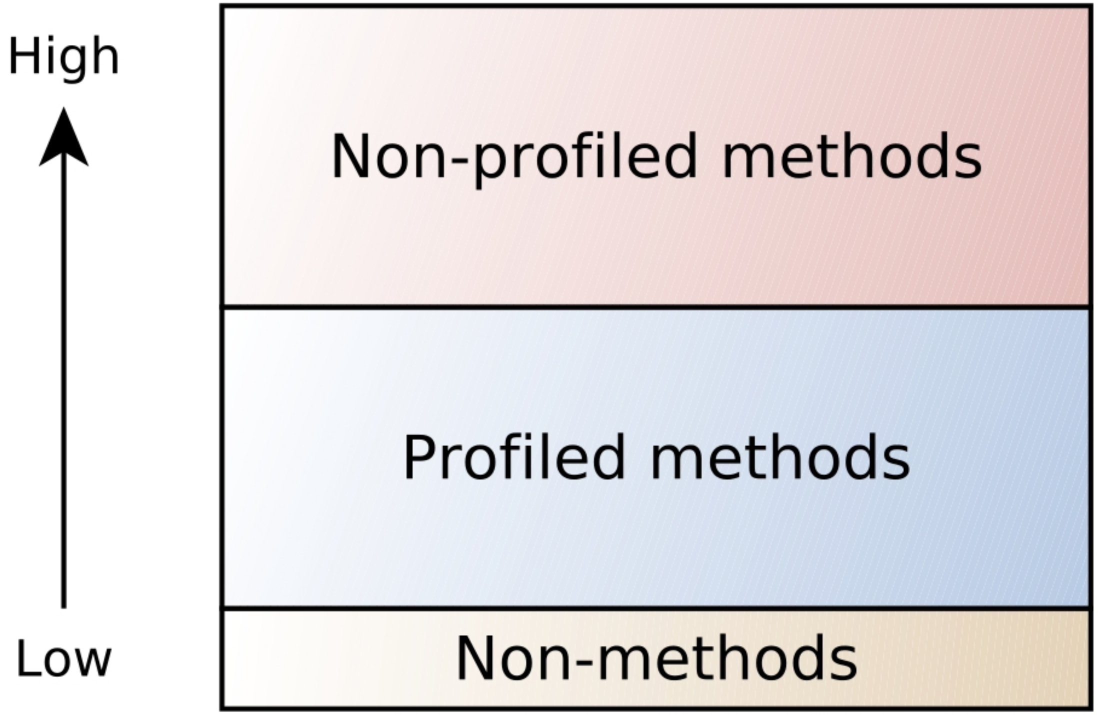
- **Non-method code**
- **Profiled method code**
  - Instrumented (C1)
  - Limited lifetime
- **Non-profiled method code**
  - Highly optimized code (C2)
  - Long lifetime

Code cache fragmentation

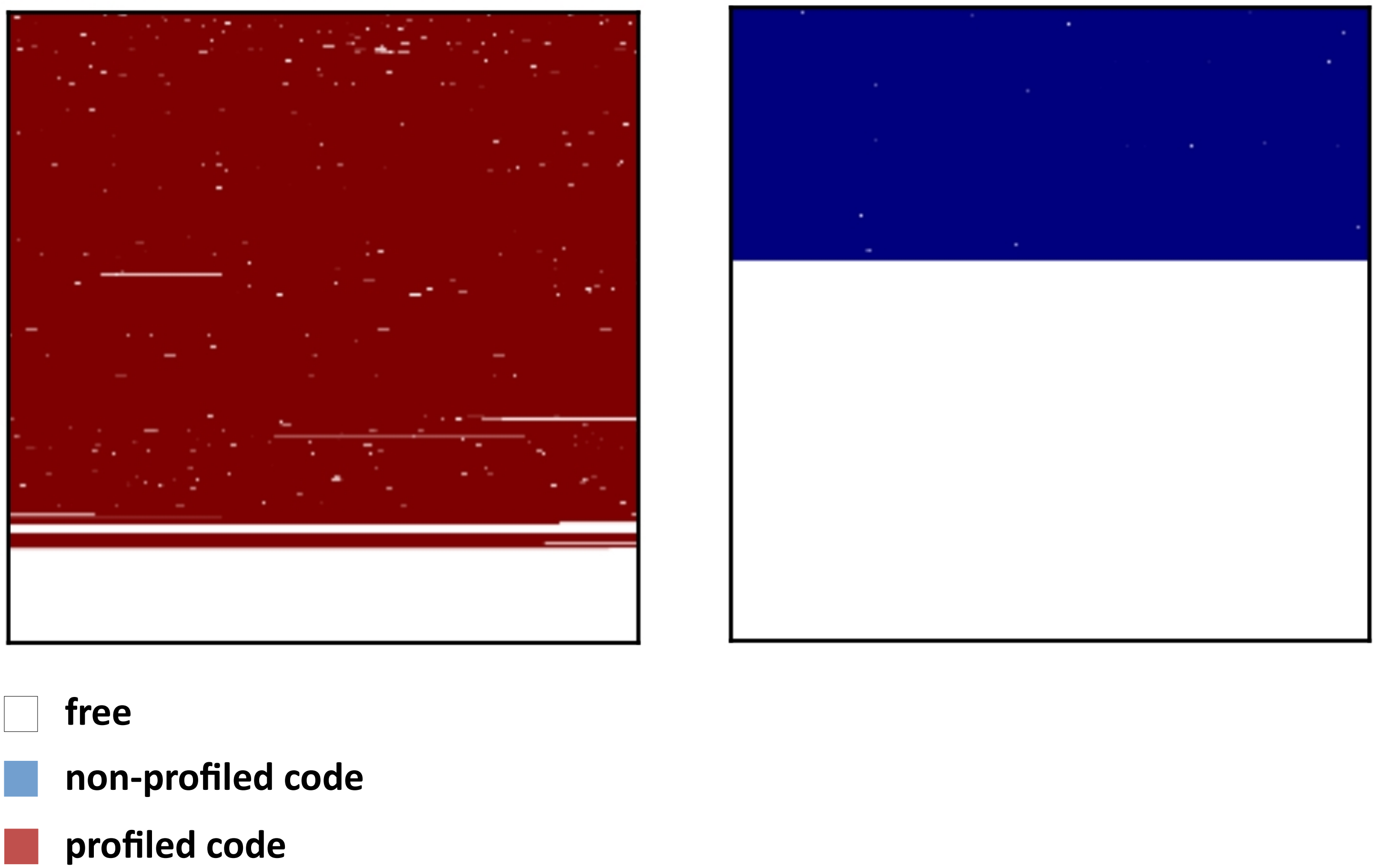


Design

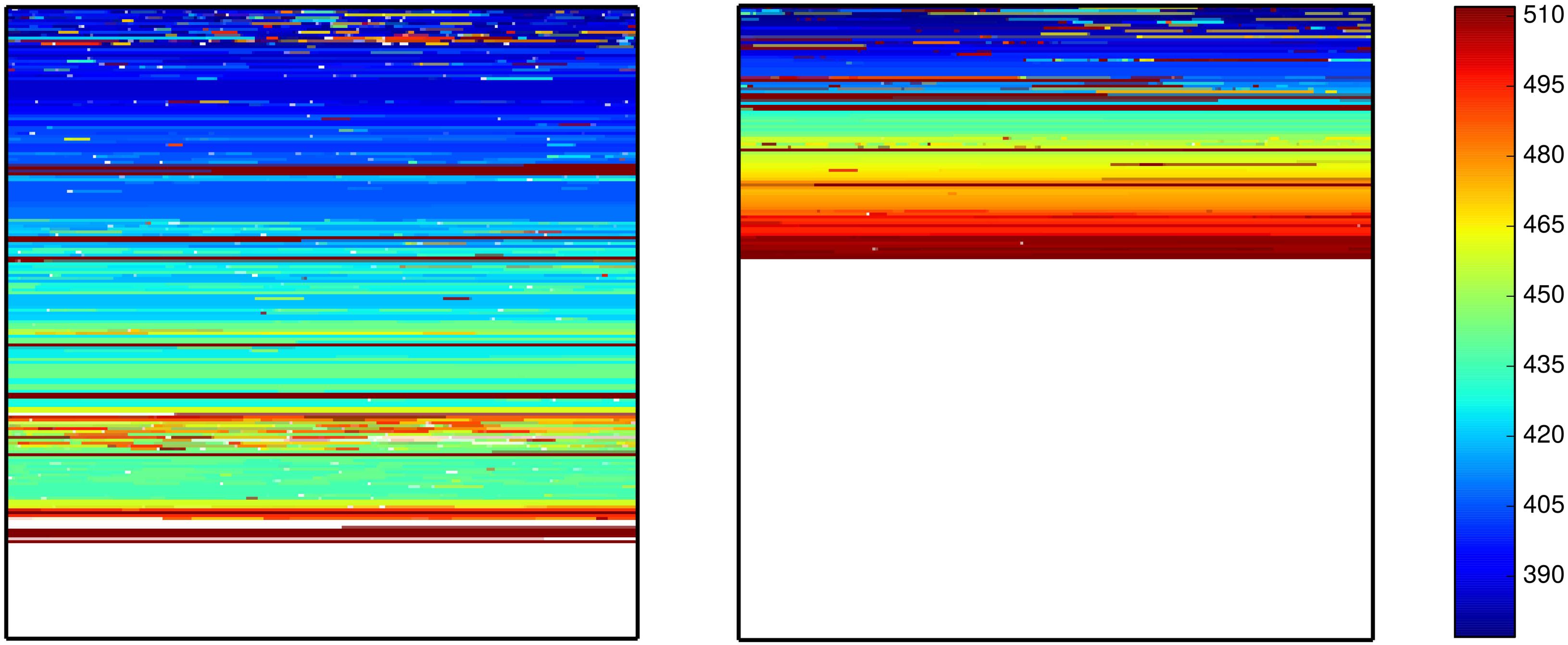
- **Split code cache into segments**



Fragmentation



Hotness



profiled code

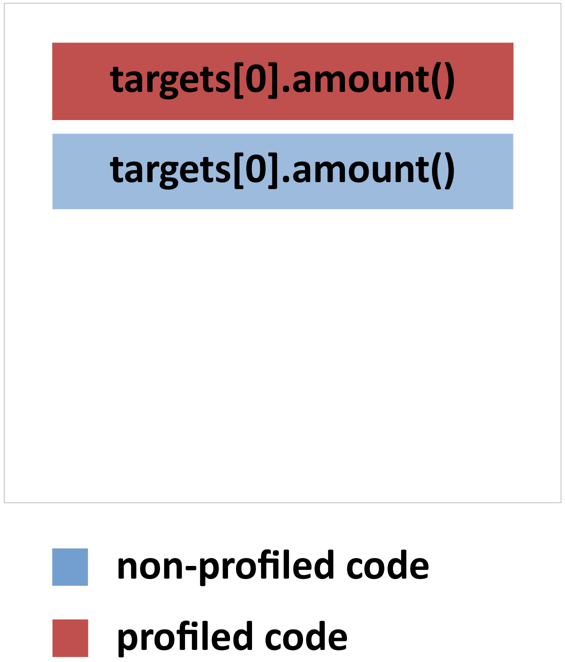
non-profiled code

iTLB

```
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;
}
```

iTLB

```
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;
}
```



iTLB

```
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;
}
```

targets[0].amount()

targets[0].amount()

targets[1].amount()

targets[1].amount()

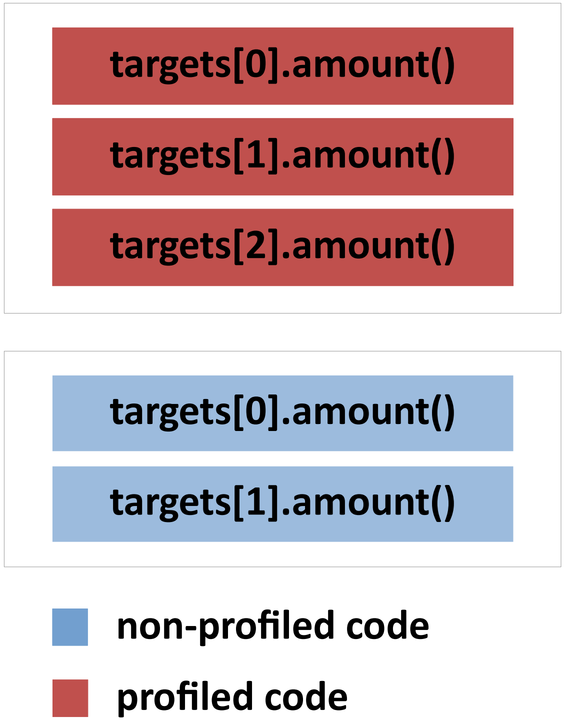
targets[2].amount()

non-profiled code

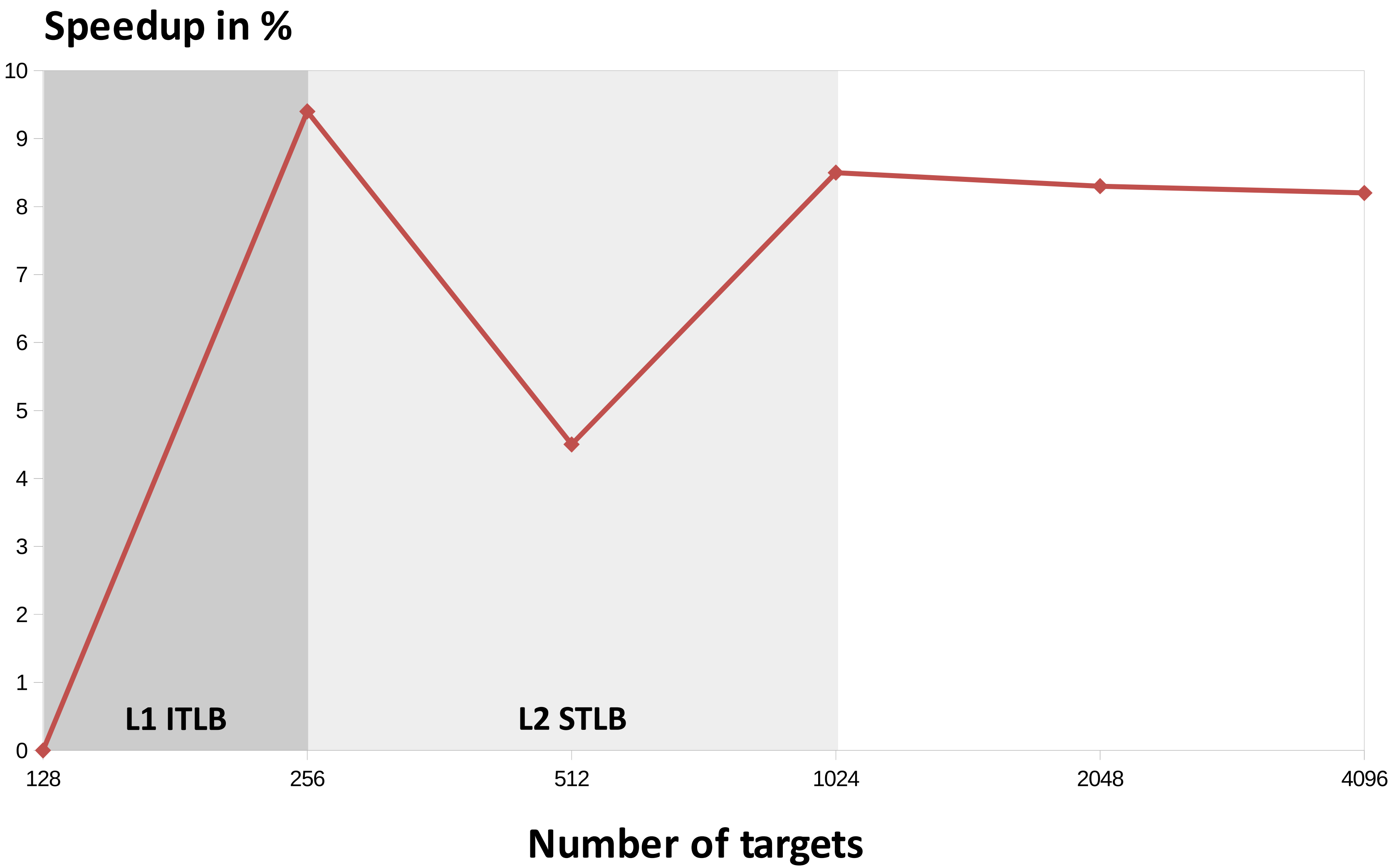
profiled code

iTLB

```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;
}
```

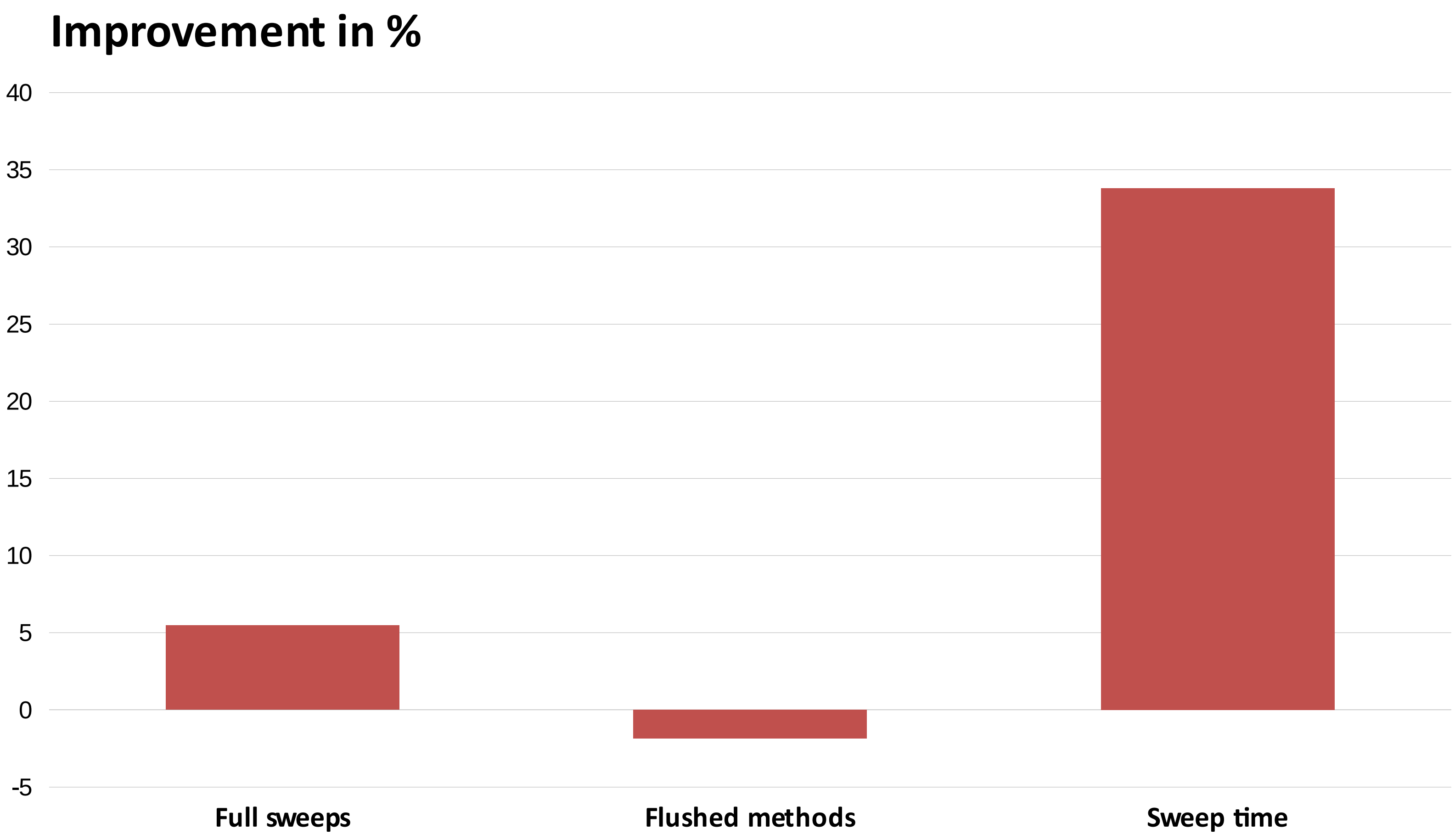


iTLB

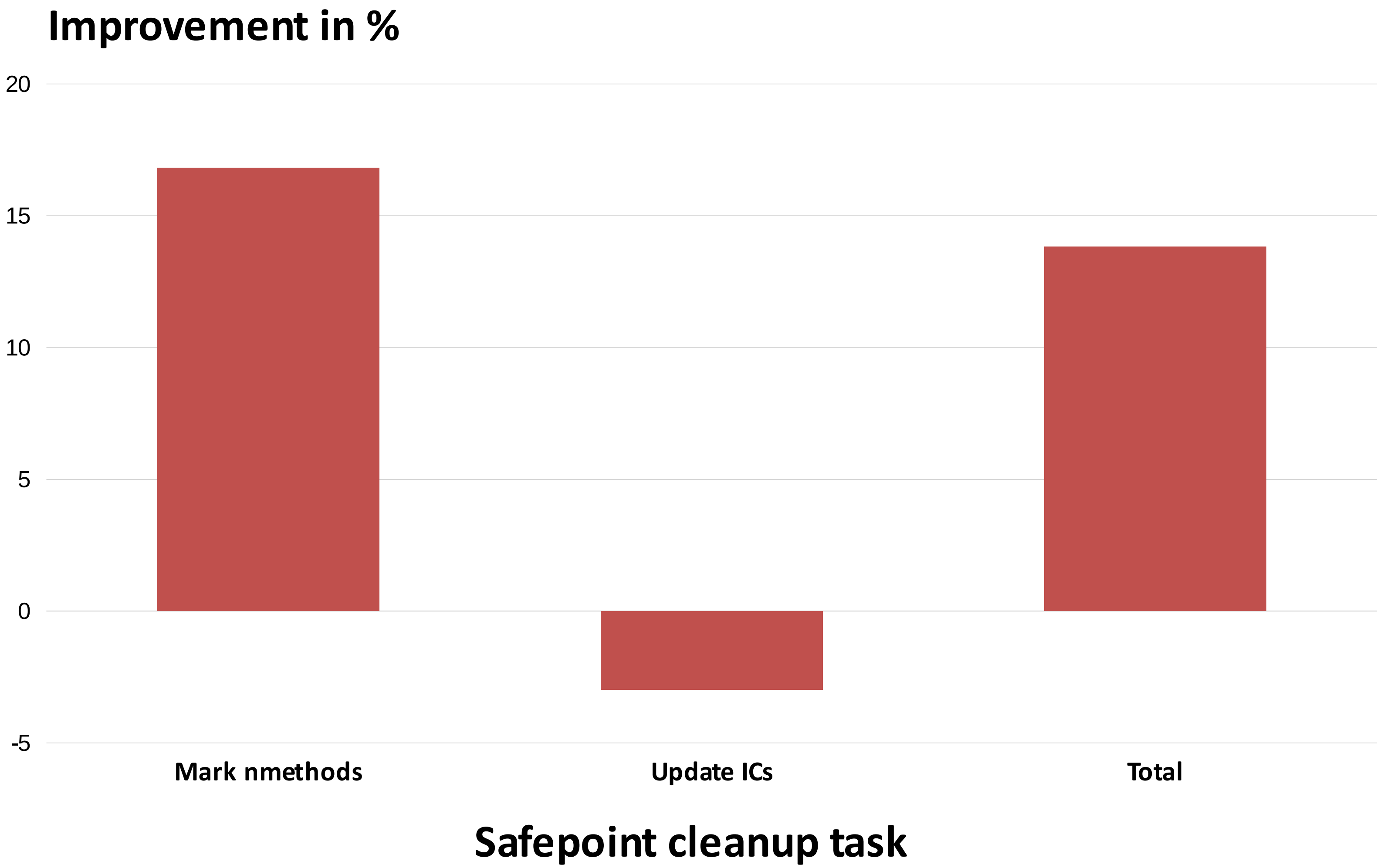


Speedup in %
10
9
8
7
6
5
4
3
2
1
0
L1 ITLB
L2 STLB
128
256
512
1024
2048
4096
Number of targets

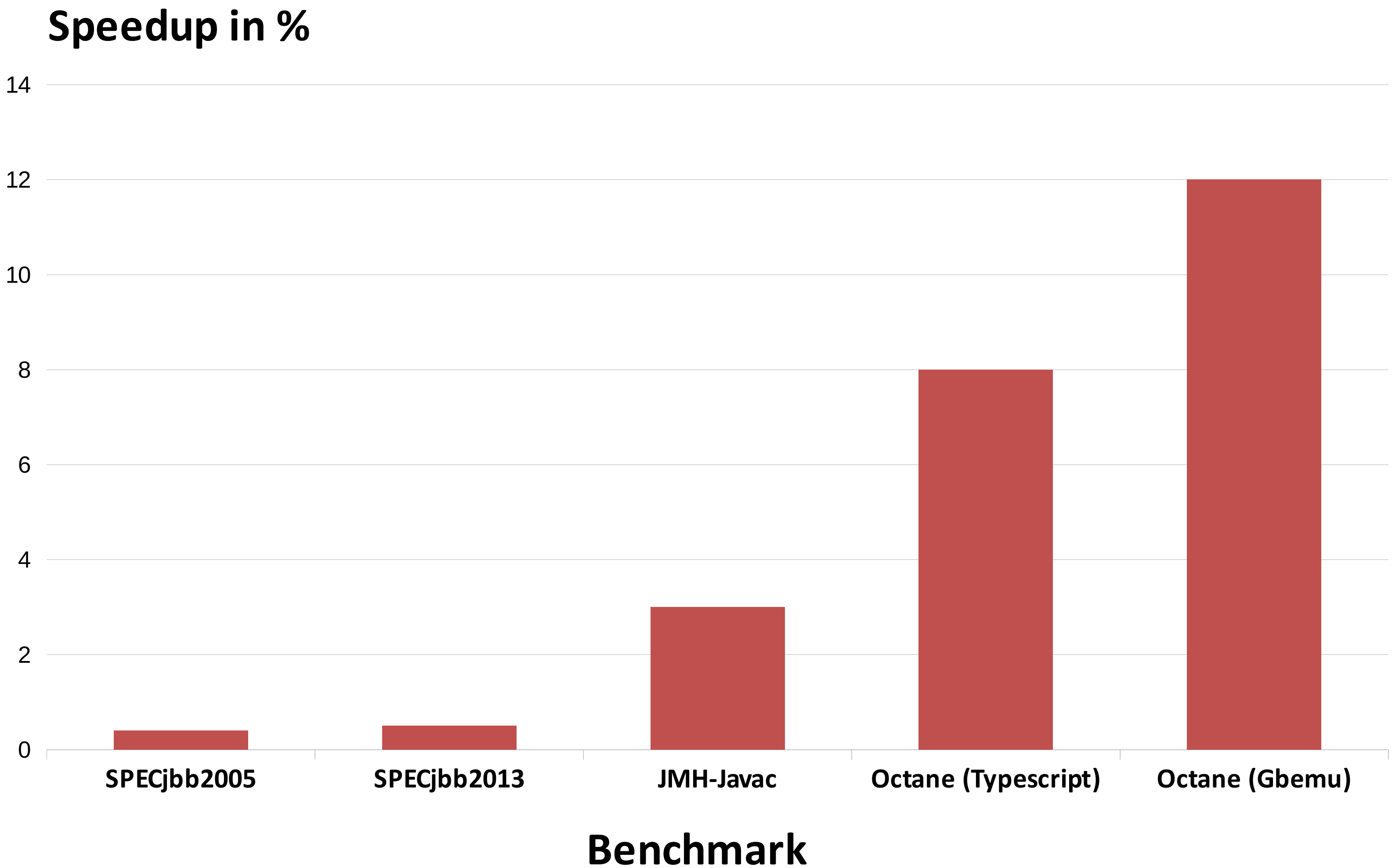
Code cache sweeper



Safepoint pause time



Runtime


Speedup in %
14
12
10
8
6
4
2
0
SPECjbb2005
SPECjbb2013
JMH-Javac
Octane (Typescript)
Octane (Gbemu)
Benchmark

Conclusion

- **Code layout has significant impact on performance**
  - code locality reduces iTLB misses
  - less iteration overhead

- **Will be released with JDK 9**
  - openjdk.java.net/jeps/197

Compact Strings

Oracle
HotSpot Compiler Team
Tobias Hartmann

Goals

- **Memory footprint reduction**
  - Improve space efficiency of Strings
- **Meet or beat throughput performance of baseline JDK 9**
- **Full compatibility with related Java and native interfaces**
- **Full platform support**
  - x86/x64, SPARC, ARM 32/64
  - Linux, Solaris, Windows, Mac OS X

Java String encoding

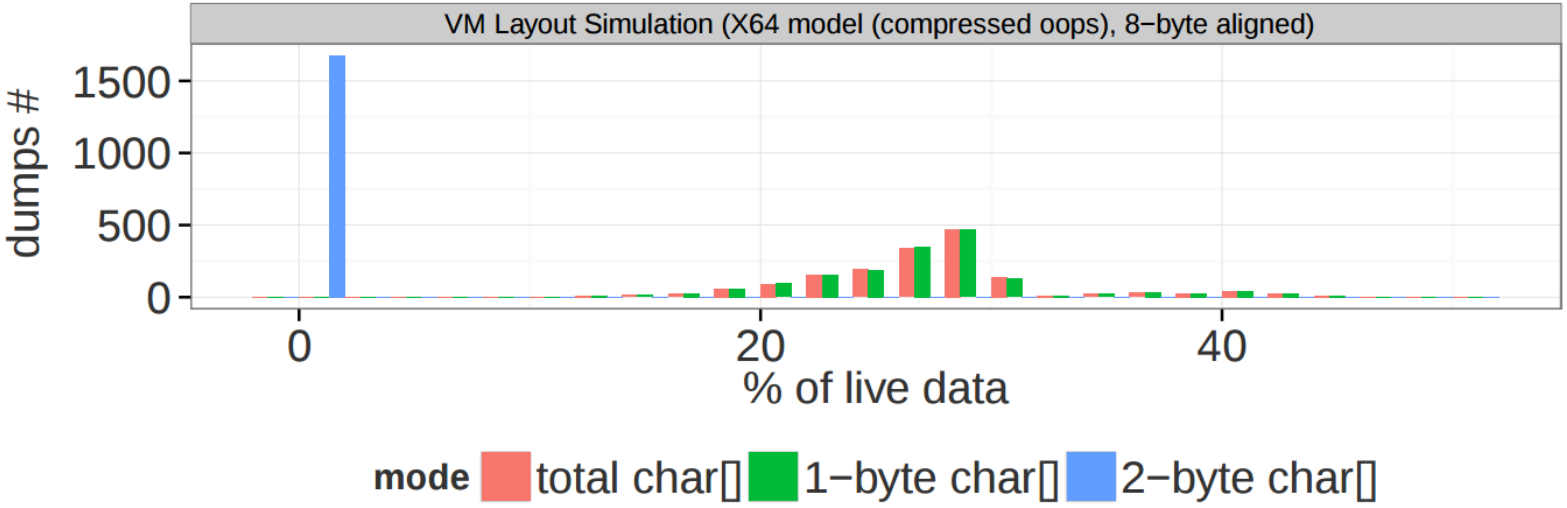
- String.value is a char array
- Uses UTF-16 encoding: 2 byte per character

	H	E	L	L	O
char value[] =	0x0048	0x0045	0x004C	0x004C	0x004F

2 byte

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**



Design

- **UTF-16 characters always occupy two bytes**
  - Lots of wasted memory
- **Changed String class to use byte array**

	H	E	L	L	O
char value[] =	0x0048	0x0045	0x004C	0x004C	0x004F

byte value[] =	0x00	0x48	0x00	0x45	0x00	0x4C	0x00	0x4C	0x00	0x4F

1 byte

Design

- String either encoded as UTF-16 or Latin-1
- Encoding field indicates which encoding is used

	H		E		L		L		O	
UTF-16	0x00	0x48	0x00	0x45	0x00	0x4C	0x00	0x4C	0x00	0x4F

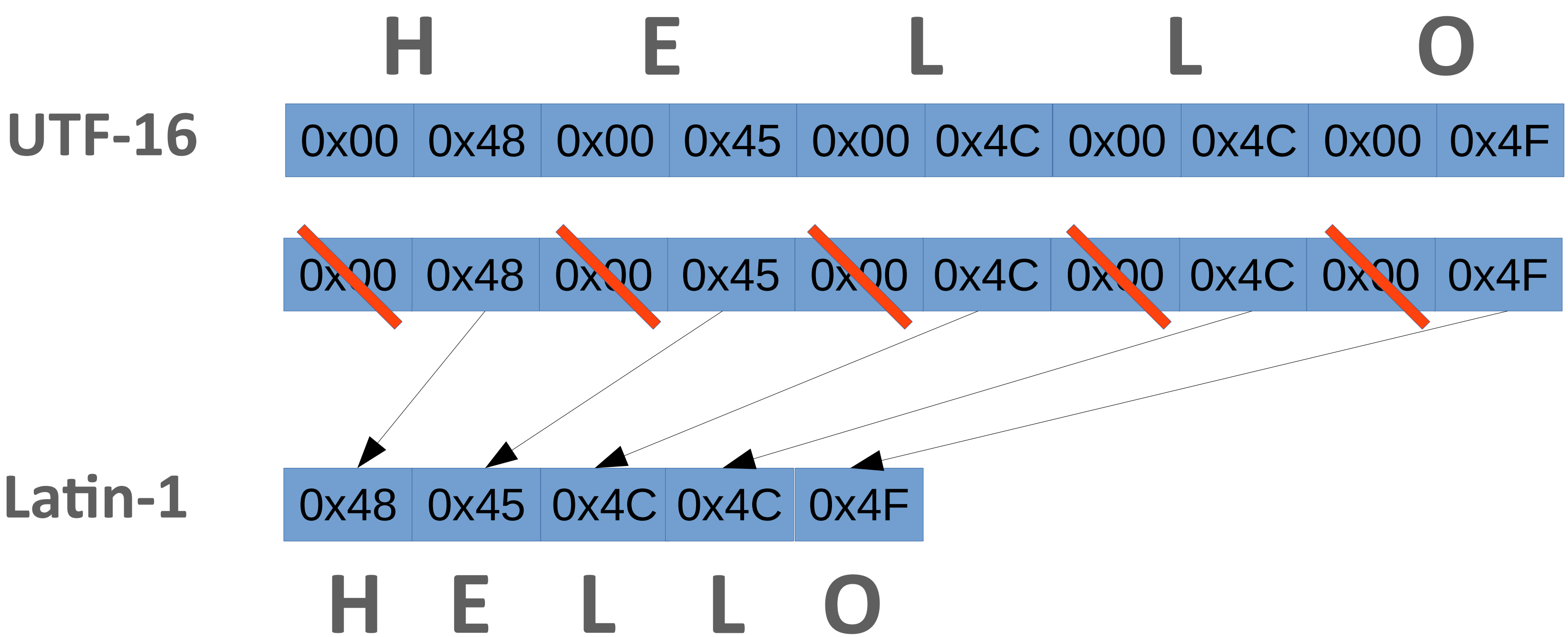
Design

- String either encoded as UTF-16 or Latin-1
- Encoding field indicates which encoding is used

	H		E		L		L		O	
UTF-16	0x00	0x48	0x00	0x45	0x00	0x4C	0x00	0x4C	0x00	0x4F
	~~0x00~~	0x48	~~0x00~~	0x45	~~0x00~~	0x4C	~~0x00~~	0x4C	~~0x00~~	0x4F

Design

- String either encoded as UTF-16 or Latin-1
- Encoding field indicates which encoding is used



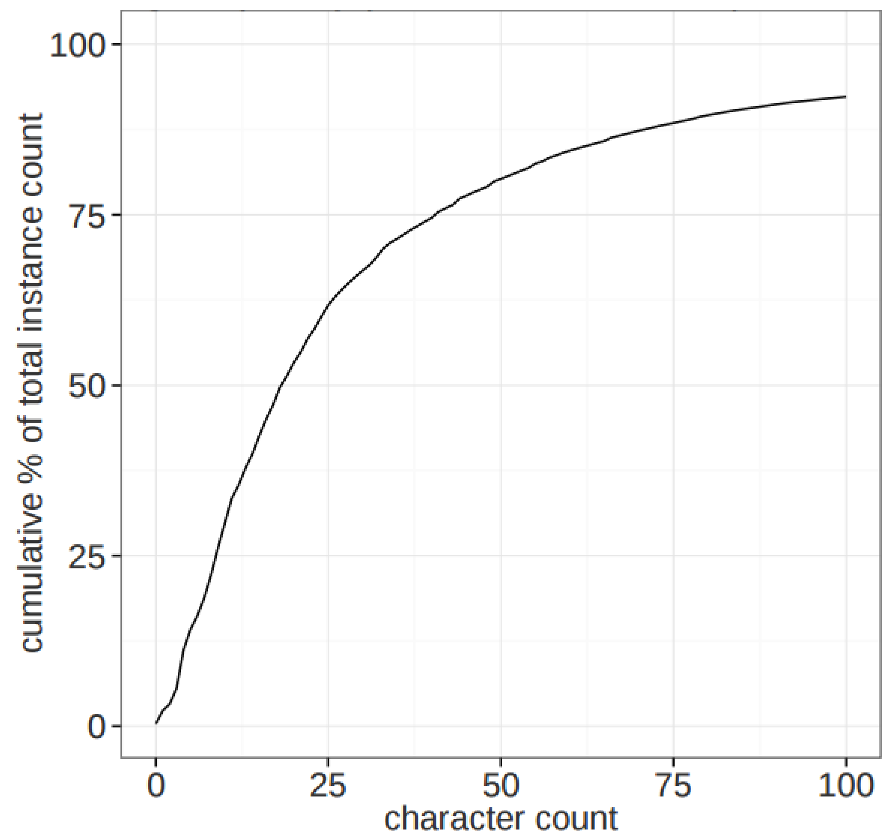
Design

- **Strings containing a character with non-zero upper byte**
  - Cannot be compressed
  - Stored as 2 byte characters using UTF-16 encoding
- **Strings containing only characters with zero upper byte**
  - Can be compressed to Latin-1
  - High-order zero bytes are stripped off
- **Invariant**
  - A UTF-16 String has at least one non-compressible character
  - Allows O(1) fastpath for String.equals() and String.indexOf()

Analysis: String size distribution

- 75% of Strings are smaller than 35 characters

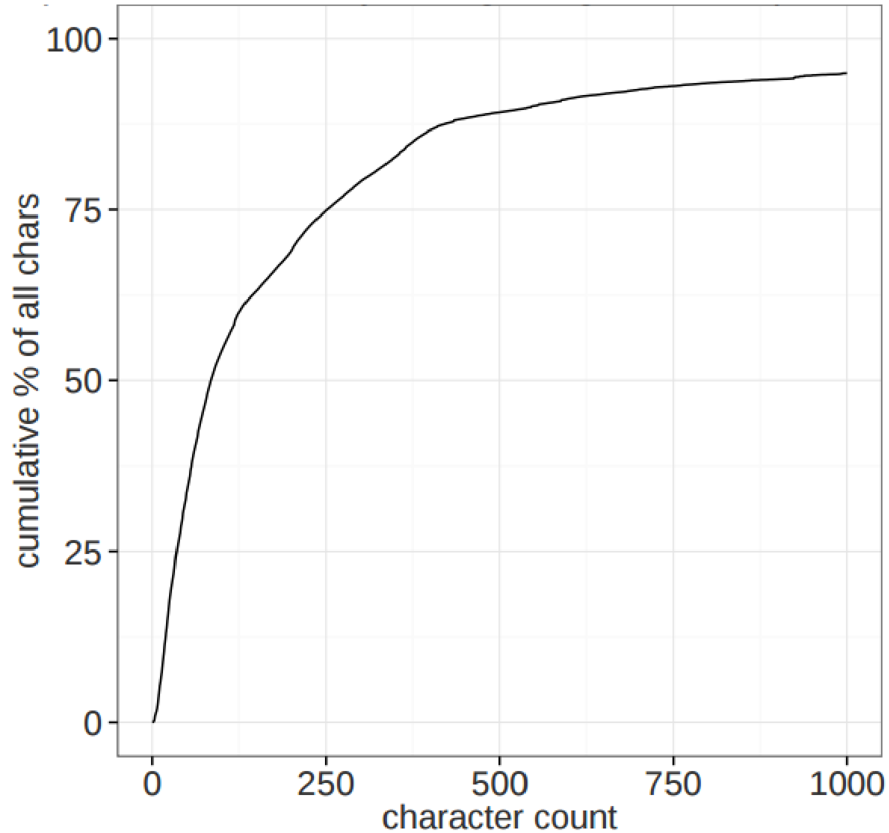
String frequency per size



Analysis: String size distribution

- 75% of Strings are smaller than 35 characters
- 75% of charactes are in Strings of length < 250
- **Predicted footprint reduction of 5% - 15%**

Space consumed by Strings of given size



Implementation

- **Hotspot support in addition to library changes**
  - JIT compilers: Intrinsics and String concatenation optimization
  - Runtime: String object constructors, JNI, JVMTI
  - GC: String deduplication
- **Compression:** char[] → byte[]
  - On String construction
- **Inflation:** byte[] → char[]
  - Whenever we need a char[] representation

Implementation

- **String construction**
  - Allocate byte[], try to compress input char[], bailout if it fails
  - Alternative: look at first character(s) and then decide (JDK-8139814)
- **New compiler intrinsics for most important methods**
- **Adapted existing intrinsics and C2 optimizations**
  - String.equals, String.compareTo, String.indexOf
- **Enable or disable via -XX:CompactStrings flag**
  - Enabled by default on x86 and SPARC

Evaluation

- **Micro-benchmarks* at the String API level**
  - Compare throughput performance to baseline JDK 9
- **Larger workloads / benchmarks**
  - For evaluating footprint, throughput and latency

Performance on x86 (Haswell)

- **SpecJbb2005**
  - 21% footprint reduction
  - 27% less GCs
  - 5% throughput improvement
- **SpecJbb2015**
  - 7% footprint reduction
  - 11% critical-jOps improvement

Performance on SPARC (T5)

- **SpecJbb2005**
  - 19% footprint reduction
  - 21% less GCs
  - 2% throughput improvement
- **SpecJbb2015**
  - 4% critical-jOps improvement
- **WLS startup**
  - 10% footprint reduction
  - 5% cold startup improvement
  - 3% warm startup improvement

Conclusion

- **String density matters**
  - Footprint reduction of up to 21%
  - Performance improvements due to less GC pressure

- **Will be released with JDK 9**
  - openjdk.java.net/jeps/254

Questions?

tobias.hartmann@oracle.com

Java™

ORACLE®