JIT Compiler Design

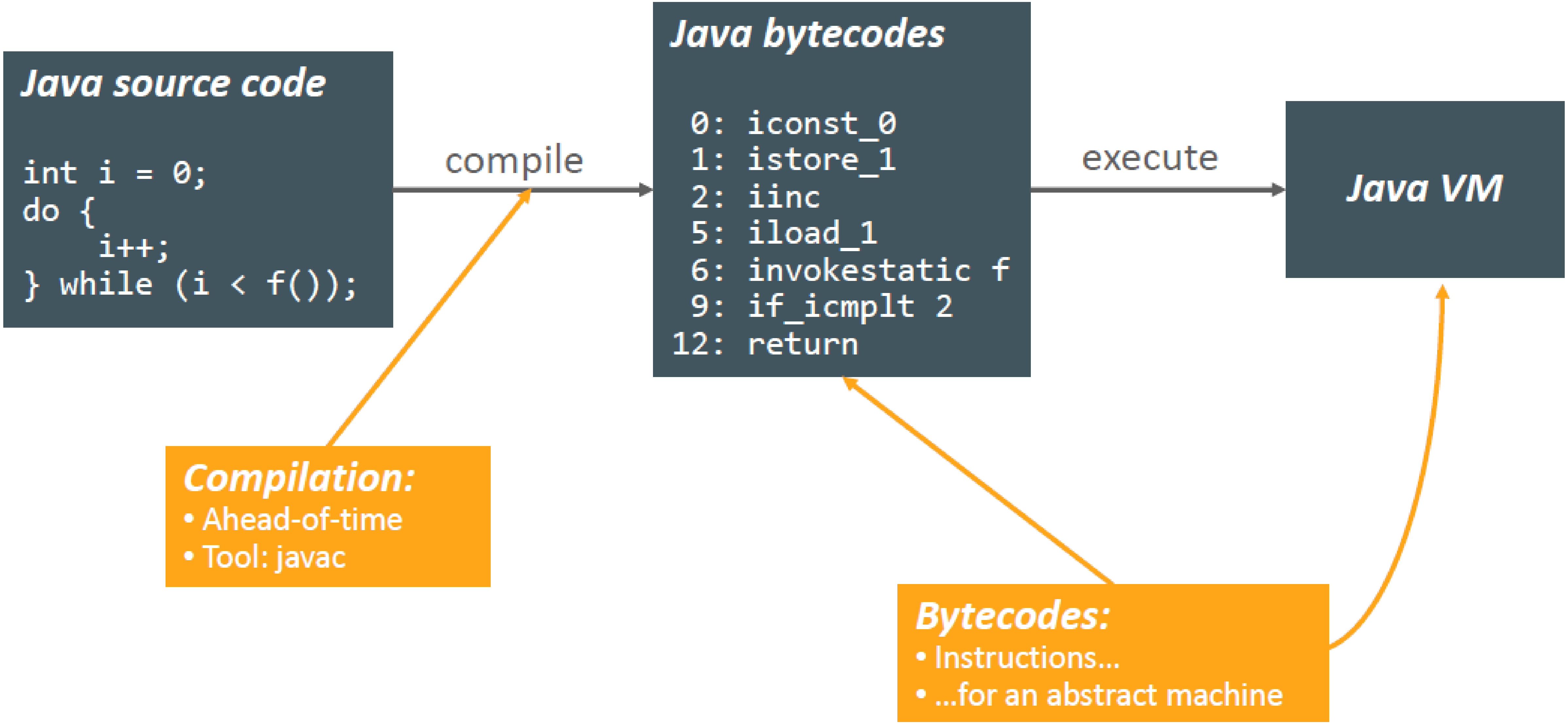
-Idhaya Elango

EECS 768

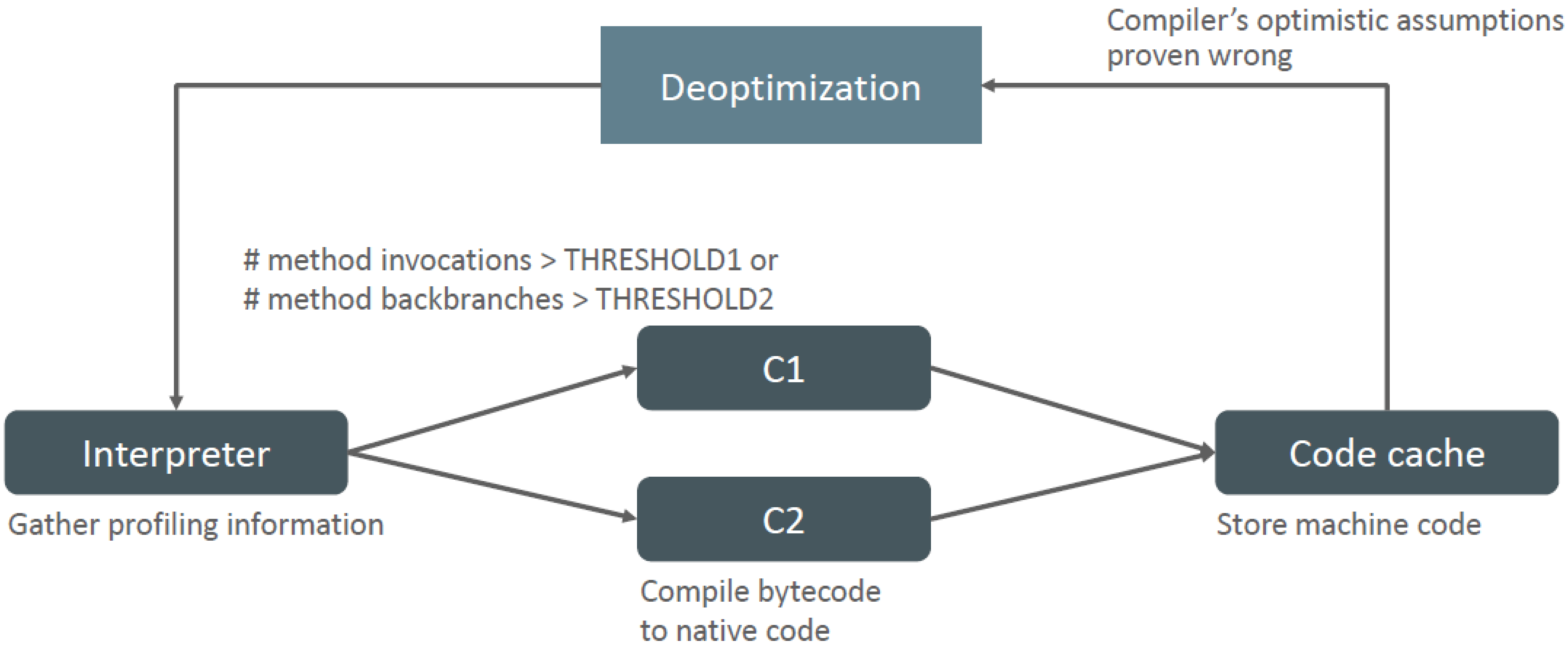
Agenda

- Introduction
- JIT Compilation
  - -c1 compiler
  - -c2 compiler
  - -Tiered compilation
- C1 Compiler Design
  - -HIR
  - -LIR
  - -Optimizations
  - -Garbage Collection
  - -Exception Handling
- C2 Overview

Stages of a Java method's lifetime



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



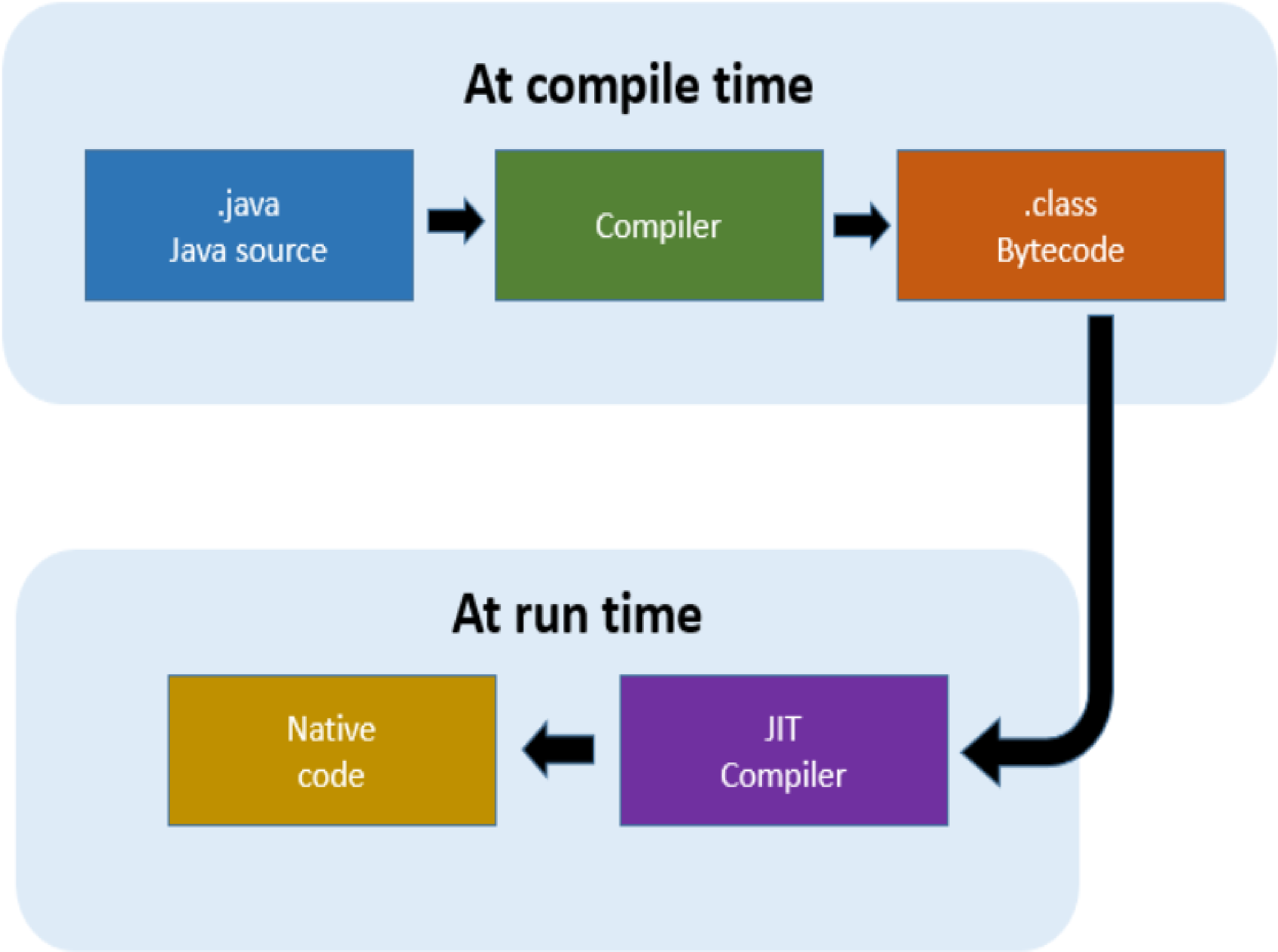
“Just-in-time”

-During Program Execution

-Time is needed to compile **“hot”** methods

-profiling at run time

-optimistic optimizations



Compilers in Hotspot

- **C1 compiler**
  - Fast compilation
  - Small footprint
  - Code could be better
- **C2 compiler**
  - High resource demands
  - High-performance code

Client VM

Server VM

Tiered compilation

SimpleProgram.java

```
public class SimpleProgram {
  static final int CHUNK_SIZE = 1_000;
  public static void main(String[] args) {
    for ( int i = 0; i < 250; ++i ) {
      long startTime = System.nanoTime();
            for ( int j = 0; j < CHUNK_SIZE; ++j ) {
        new Object();
      }
      long endTime = System.nanoTime();
      System.out.printf("%d\t%d%n", i, endTime - startTime);
    }
  }
}
```

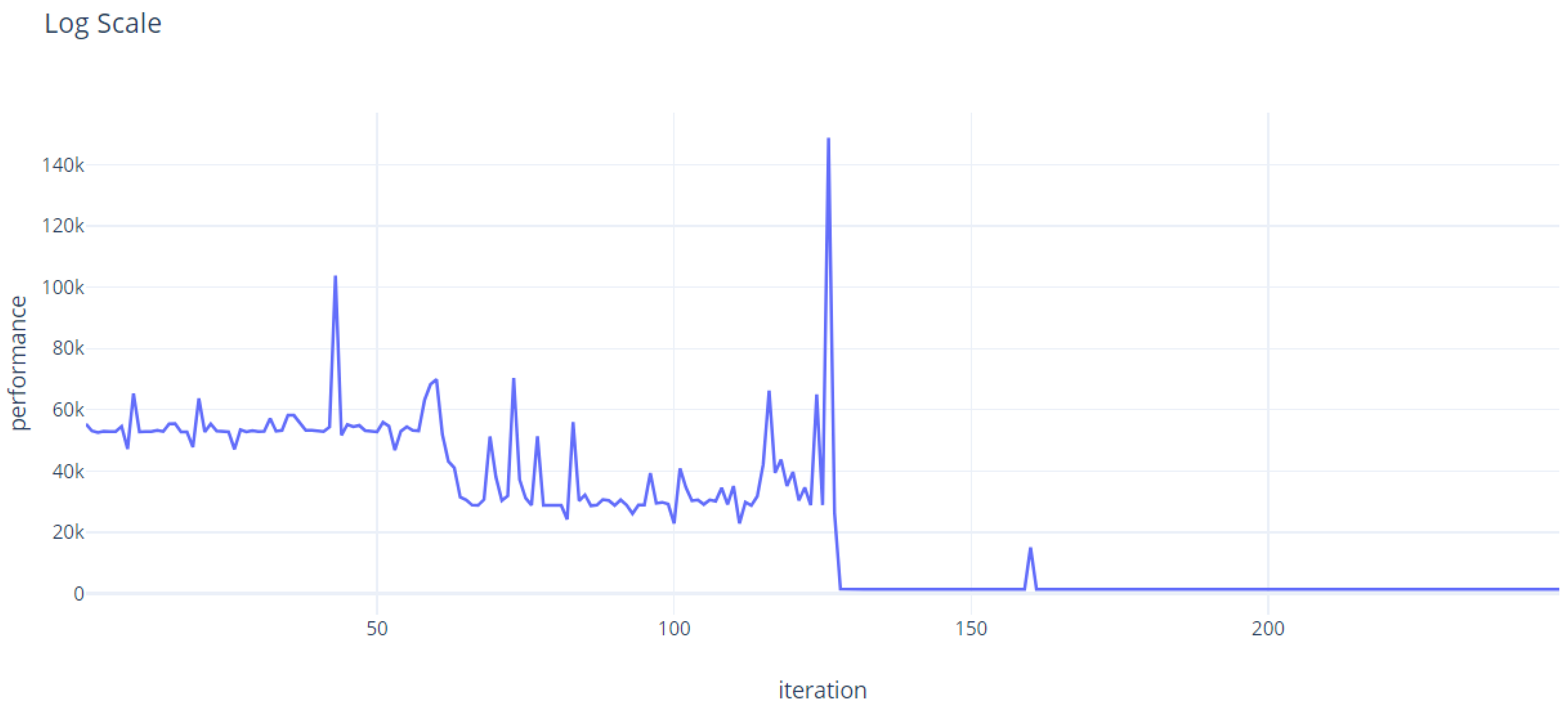
```
idhaya1990@DESKTOP-CKA1S5M:~/JDK/jdk9$ java SimpleProgram
0       72900
1       69800
2       60300
3       61200
4       63800
5       56100
6       72000
7       69800
8       49400
9       58800
10      60100
11      58800
12      57100
13      55800
14      56400
15      106500
16      55700
17      57400
18      58100
19      50800
20      109800
21      93800
22      56800
23      57900
24      58500
25      55500
26      51100
27      56800
28      55600
29      82100
30      82500
31      67100
32      66500
33      60800
34      59200
35      57100
36      139100
37      57900
38      61600
39      77900
40      74400
41      71200
42      76000
43      87000
44      68500
45      59200
46      66900
```

```
67      33200
68      33500
69      58100
70      31700
71      33300
72      38500
73      42200
74      34500
75      31500
76      45300
77      32500
78      29600
79      29900
80      31500
81      31900
82      24200
83      30900
84      29800
85      34700
86      29700
87      30200
88      29700
89      33900
90      42600
91      55200
92      35500
93      35300
94      29700
95      31600
96      31100
97      34700
98      33000
99      31800
100     23900
101     47100
102     30300
103     36000
104     30100
105     31400
106     30400
```

```
109     39400
110     211000
111     13600
112     1700
113     1500
114     1600
115     1500
116     1500
117     1600
118     1500
119     1500
120     1600
121     1600
122     1500
123     1500
124     1600
125     1500
126     1600
127     1800
128     3800
129     1500
130     1500
131     1500
132     1600
133     1500
134     1800
135     1700
136     1700
137     1500
138     1600
139     1500
140     1500
141     1900
142     1900
143     1800
144     1700
145     1500
146     1500
147     1400
148     1500
149     1500
150     1400
151     1500
152     1400
153     1500
154     1500
155     1500
```

```
57      2100
58      2000
59      1900
60      1900
61      1700
62      1600
63      1500
64      1500
65      1500
66      1500
67      1600
68      1500
69      1500
70      1500
71      1500
72      1500
73      1400
74      1500
75      1400
76      1500
77      1900
78      1900
79      1900
80      1700
81      1700
82      1500
83      1500
84      1500
85      1400
86      1400
87      1500
88      1600
89      1500
90      1600
91      1600
92      1700
93      1500
94      1500
95      1500
96      1500
97      1500
98      1500
99      1500
00      1500
01      1500
```

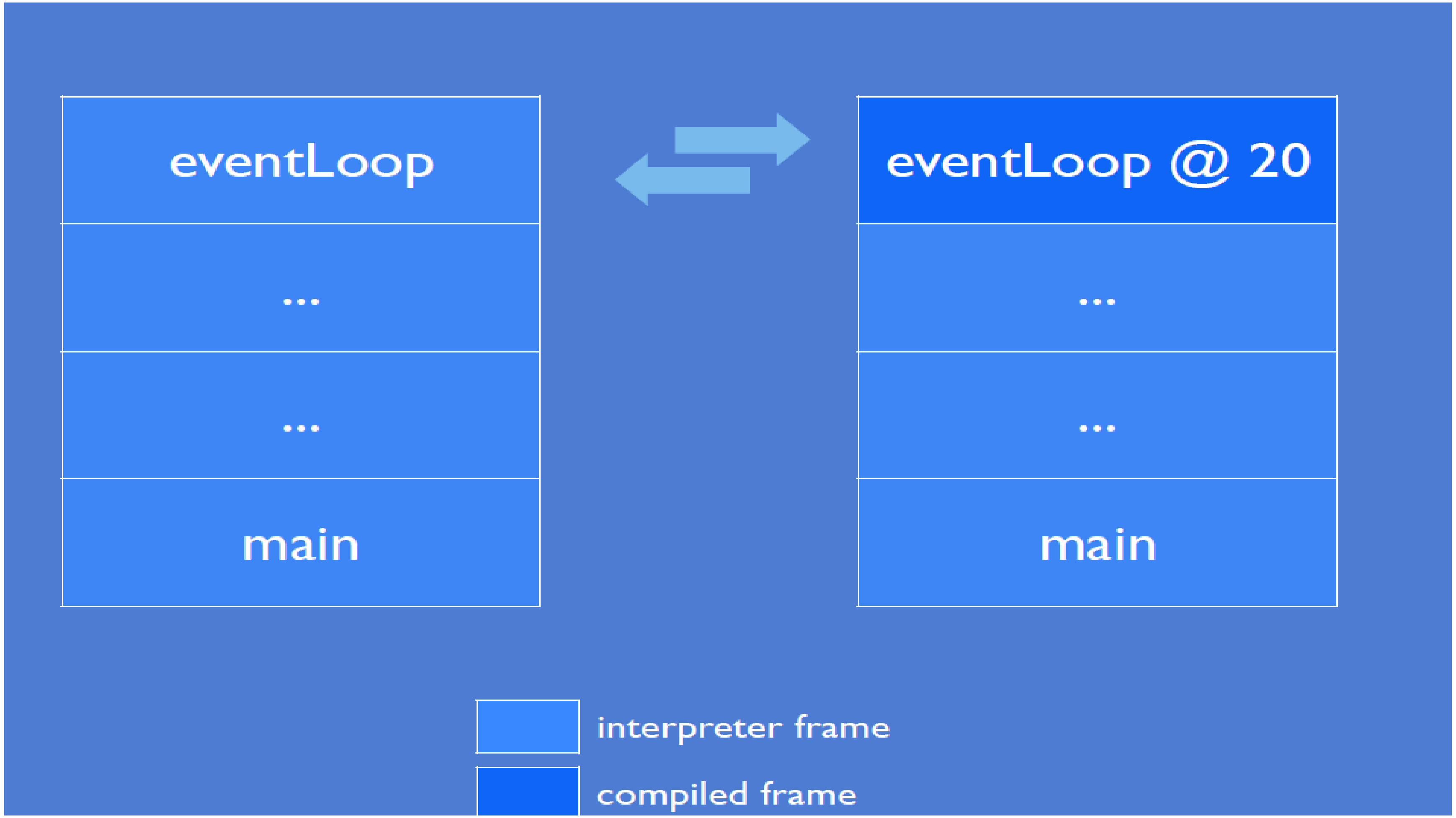
```
202     1500
203     1500
204     1500
205     1500
206     1500
207     1400
208     1500
209     1500
210     1500
211     1500
212     1500
213     1400
214     1400
215     1500
216     1500
217     1400
218     1500
219     1500
220     1500
221     1500
222     1400
223     1500
224     1700
225     1400
226     1900
227     1900
228     1800
229     1700
230     1500
231     1600
232     1500
233     1400
234     1500
235     1500
236     1500
237     1500
238     1800
239     1600
240     1700
241     1500
242     1400
243     1500
244     1500
245     1500
246     1500
247     1500
248     1500
249     1900
```


Log Scale
performance
140k
120k
100k
80k
60k
40k
20k
0
50
100
150
200
iteration

-XX:+PrintCompilation

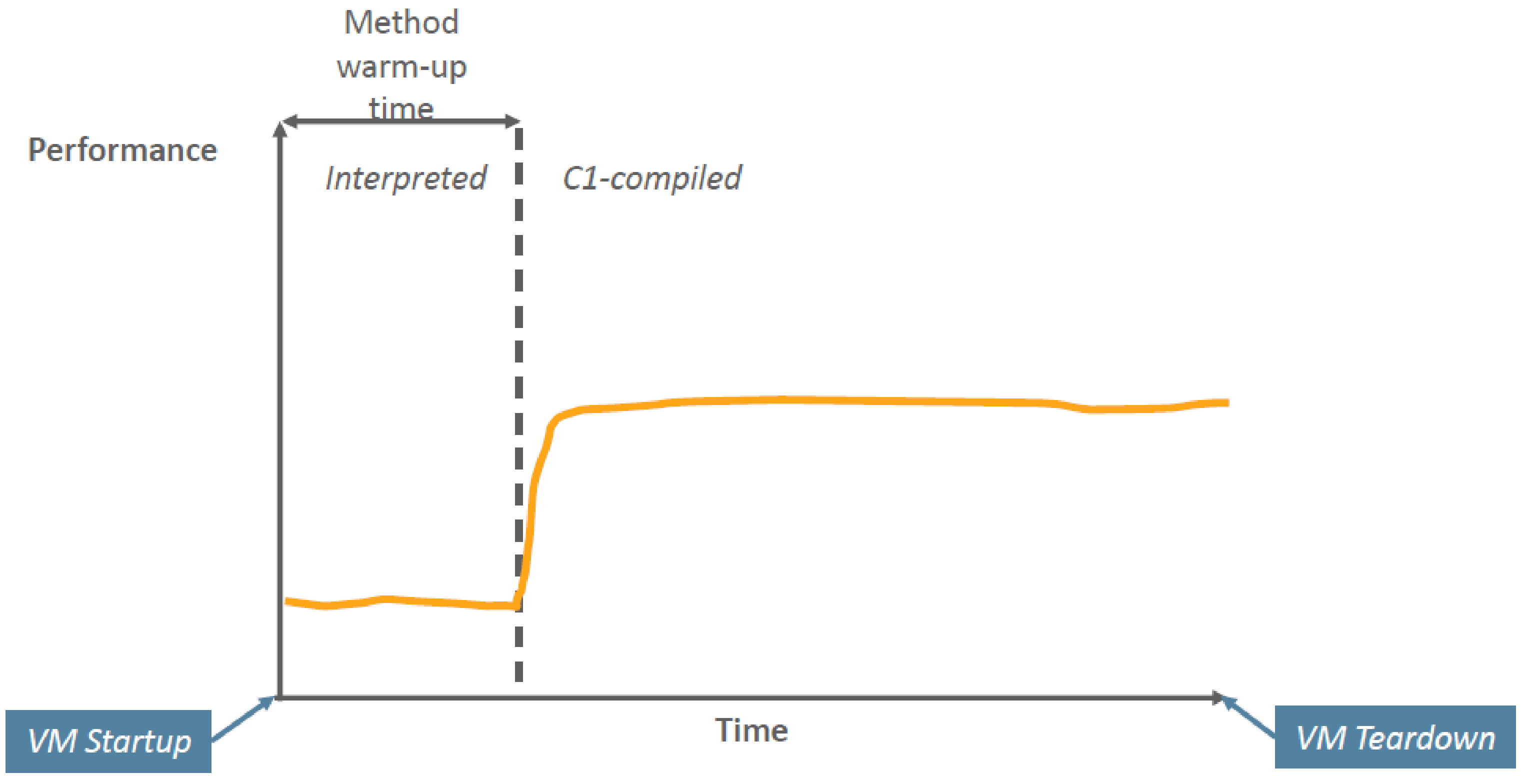
```
idhaya1990@DESKTOP-CKA1S5M:~/JDK/jdk9$ ./build/linux-x86_64-normal-server-fastdebug/jdk/bin/java -XX:+PrintCompilation SimpleProgr
    474    3       3       java.lang.StringUTF16::getChar (60 bytes)
    487    1       3       java.lang.String::charAt (25 bytes)
    491    2       3       java.lang.StringLatin1::charAt (28 bytes)
    511    4       3       java.lang.String::coder (15 bytes)
    525    5       3       java.lang.String::equals (65 bytes)
    534    6       3       java.util.jar.Attributes$Name::isValid (32 bytes)
    536    7       3       java.util.jar.Attributes$Name::isAlpha (30 bytes)
    537    8       3       java.lang.Object::<init> (1 bytes)
    571    9       3       java.lang.StringLatin1::equals (36 bytes)
    572   10       1       java.lang.reflect.Method::getName (5 bytes)
    575   11       3       java.lang.String::length (11 bytes)
    577   12     n 0       java.lang.invoke.MethodHandle::linkToStatic(LLLLLLL)L (native)   (static)
    591   13       3       java.lang.StringLatin1::hashCode (42 bytes)
    601   14       3       java.lang.StringLatin1::canEncode (13 bytes)
    603   15     n 0       java.lang.invoke.MethodHandle::linkToStatic(LLL)L (native)   (static)
    604   16       3       java.lang.String::isLatin1 (19 bytes)
    612   19     n 0       java.lang.invoke.MethodHandle::invokeBasic(LLLLLL)L (native)
    613   17       3       java.lang.String::hashCode (49 bytes)
    618   20     n 0       java.lang.invoke.MethodHandle::linkToSpecial(LLLLLLLL)L (native)   (static)
    619   18       1       java.lang.Class::getClassLoader0 (5 bytes)
    621   24     n 0       java.lang.Object::hashCode (native)
    621   23       1       java.lang.Object::<init> (1 bytes)
    622   25     n 0       java.lang.System::arraycopy (native)   (static)
    623    8       3       java.lang.Object::<init> (1 bytes)   made not entrant
    625   22       3       java.lang.Math::min (11 bytes)
    630   26       4       java.lang.String::charAt (25 bytes)
    635   21       1       java.lang.Enum::ordinal (5 bytes)
    636   27     n 0       java.lang.invoke.MethodHandle::linkToStatic(LLLL)L (native)   (static)
    638   28       3       java.lang.StringLatin1::indexOf (61 bytes)
    639   33     n 0       jdk.internal.misc.Unsafe::getObjectVolatile (native)
    640   29       3       java.lang.AbstractStringBuilder::ensureCapacityInternal (39 bytes)
    645   31       3       java.util.concurrent.ConcurrentHashMap::tabAt (22 bytes)
    649    1       3       java.lang.String::charAt (25 bytes)   made not entrant
    650   36     n 0       java.lang.invoke.MethodHandle::linkToSpecial(LLL)L (native)   (static)
    651   34       3       java.lang.AbstractStringBuilder::isLatin1 (19 bytes)
    652   37     n 0       java.lang.invoke.MethodHandle::invokeBasic(LL)L (native)
    653   30       3       java.lang.String::<init> (15 bytes)
    654   38     n 0       java.lang.invoke.MethodHandle::linkToSpecial(LLLL)L (native)   (static)
    656   32       3       jdk.internal.misc.Unsafe::getObjectAcquire (7 bytes)
    661   42       3       jdk.internal.org.objectweb.asm.ClassWriter::get (49 bytes)
    666   41       3       jdk.internal.org.objectweb.asm.Item::set (219 bytes)
    668   35       3       java.util.Objects::requireNonNull (14 bytes)
    669   39       1       java.lang.invoke.MethodType::returnType (5 bytes)
```

On-Stack-Replacement(OSR)



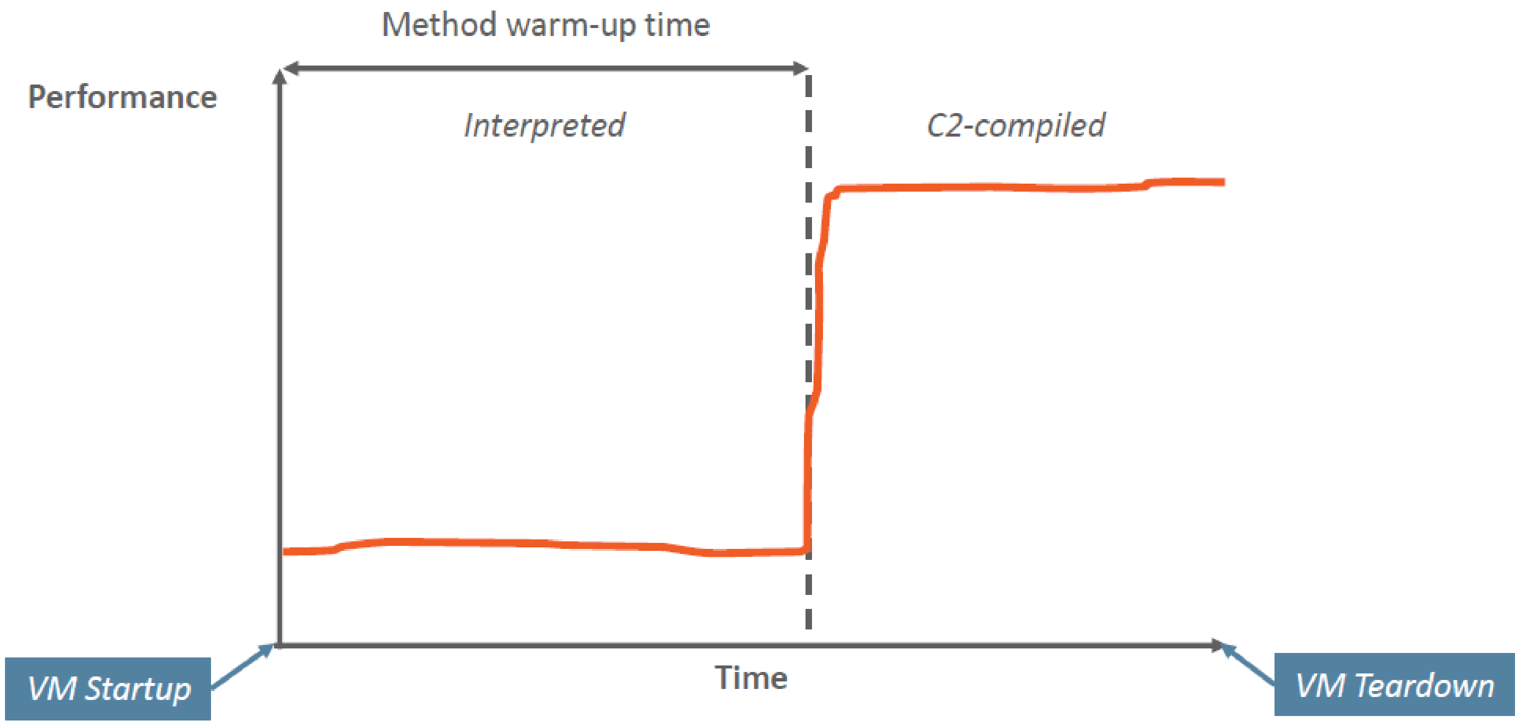
C1 Compiler

Client VM (C1 only)

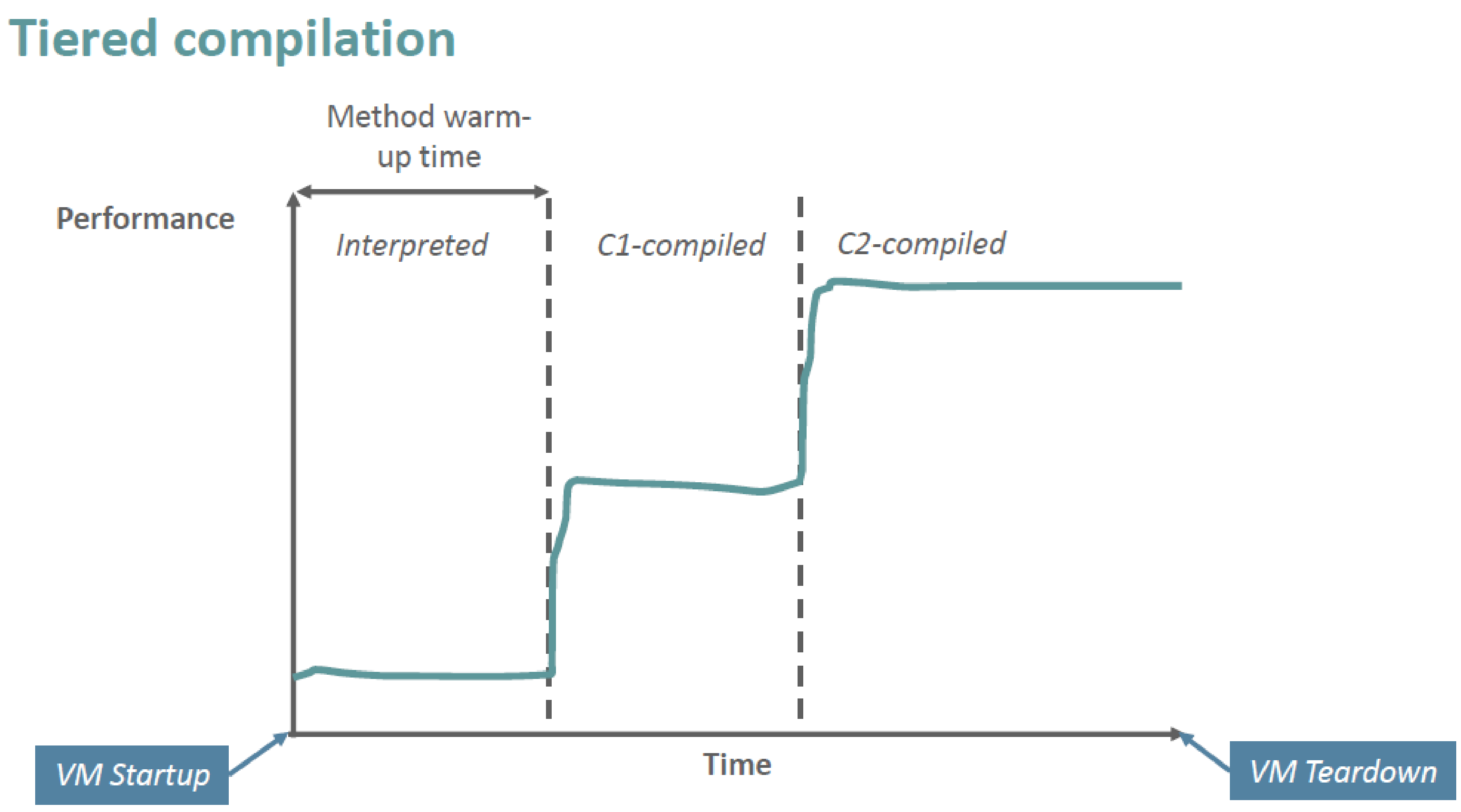


C2 Compiler

Server VM (C2 only)



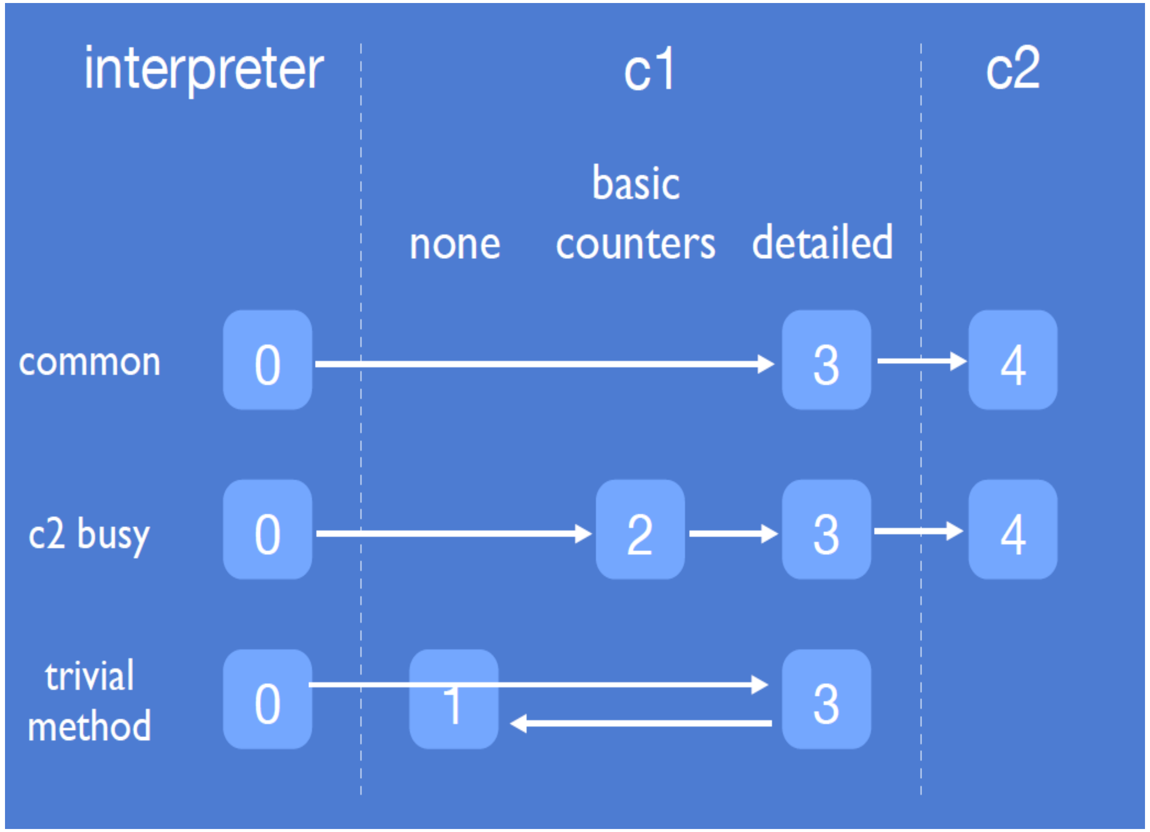
Tiered Compilation

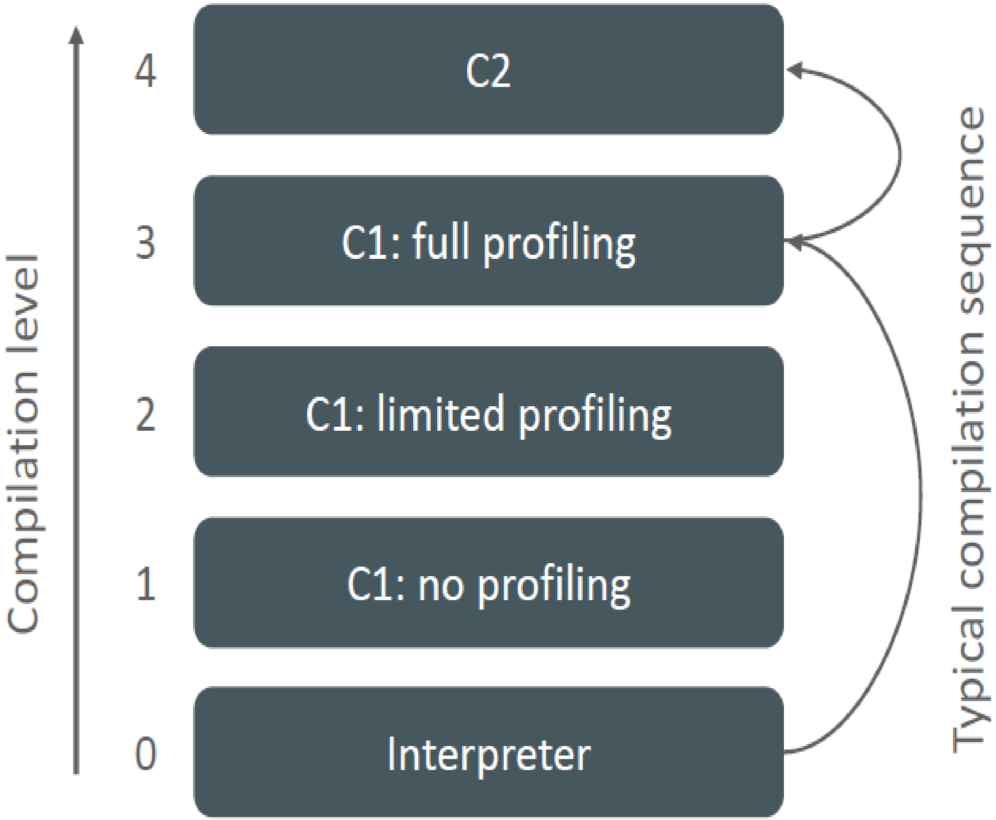


Tiered Compilation (Cont’d)

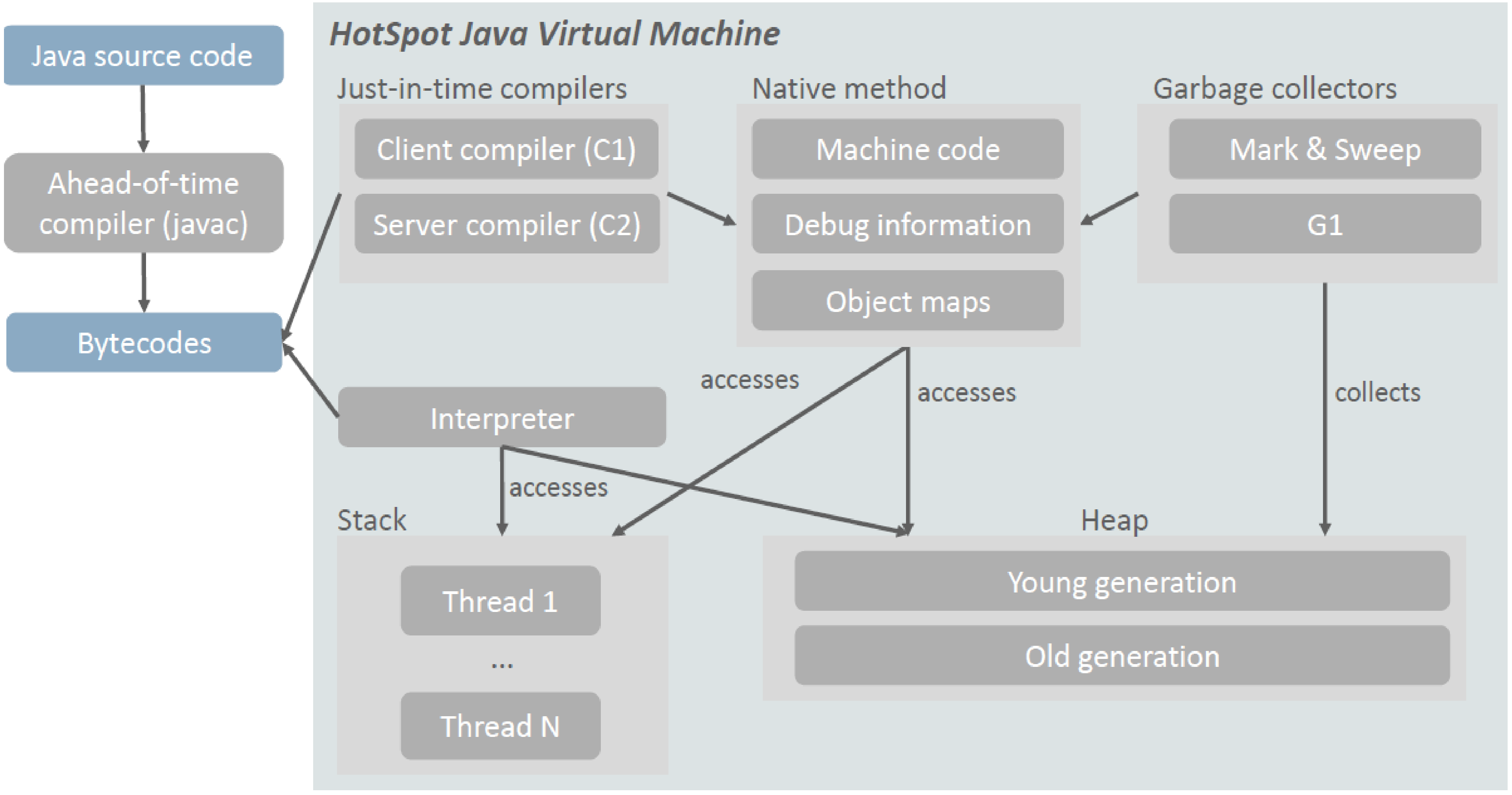
- **Combine the benefits of**
  - Interpreter: Fast startup
  - C1: Fast warmup
  - C2: High peak performance
- **Additional benefits**
  - More accurate profiling information
- **Drawbacks**
  - Complex implementation
  - Careful tuning of compilation thresholds needed

Tiered Compilation in detail

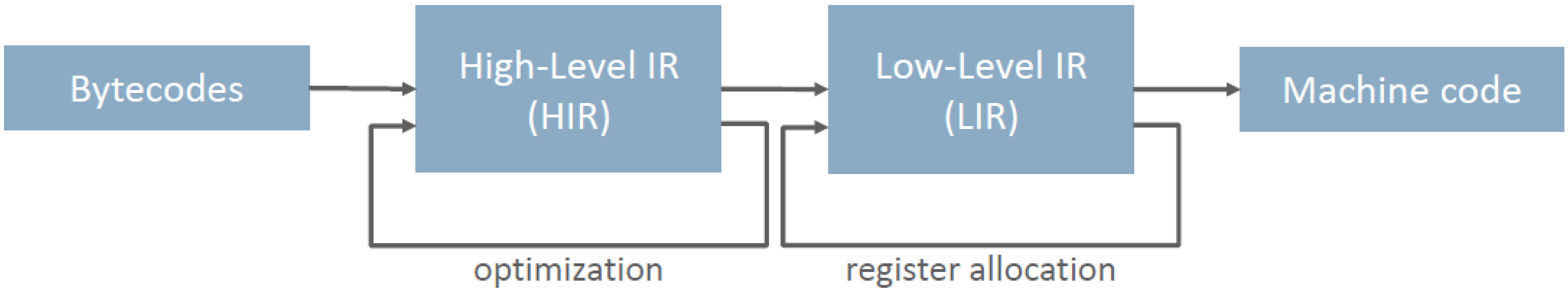




Architecture of Java HotSpot VM



Structure of Java HotSpot Client Compiler



High Level Intermediate Representation(HIR)

- **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**

Static Single Assignment Form

Java code

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

SSA form

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

Java code

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

SSA form

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

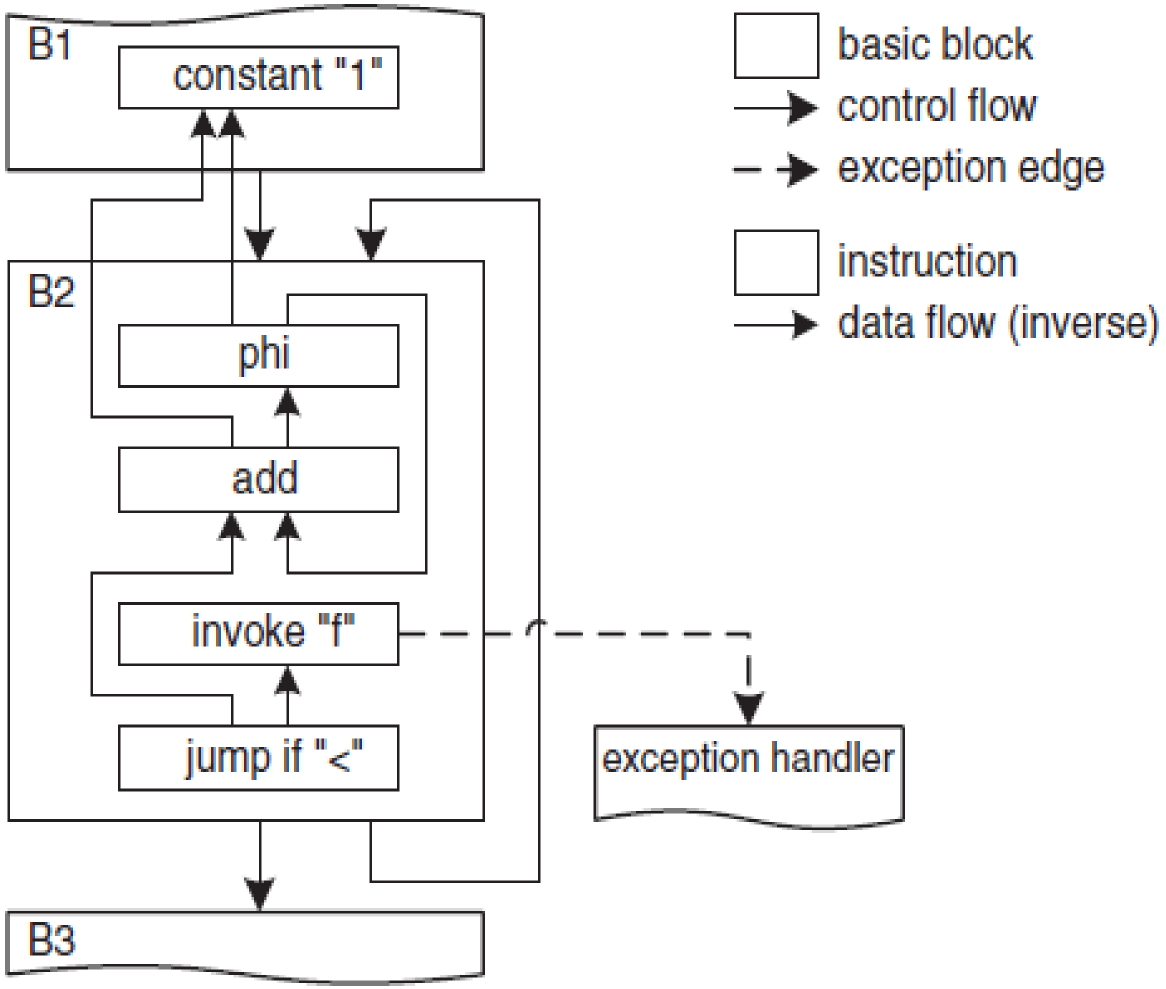
HIR Example with Control and data flow

Java code fragment:

```
int i = 1;
do {
  i++;
} while (i < f())
```

Bytecodes:

```
10: iconst_1
11: istore_0
12: iinc 0, 1
15: iload_0
16: invokestatic f()
19: if_icmplt 12
```



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**

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

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

Garbage Collection

- Uses exact garbage collection technique
- Memory split into three generations

    Young generation –For new object

    Old generation – For long lived objects

Exception Handling

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

Deoptimization

- Stop the machine code
- Undo the compiled optimizations
- Continue execution of method from Interpreter

```
void foo() {
  A p = create();
  p.bar();
}
```

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

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

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

C2 Compiler

- **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**

References

- T. Kotzmann, C. Wimmer, H. M¨ossenb¨ock, T. Rodriguez, K. Russell, and D. Cox. Design of the Java Hotspot client compiler for Java 6. ACM Transactions on Architecture and Code Optimization, 5:7:1–7:32, May 2008. ISSN 1544-3566
- https://github.com/dougqh/jvm-mechanics/tree/d3483e5f54ea3a5ebf3e84caa1b55437f34ee635
- https://www.ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/lst-dam/documents/Education/Classes/Fall2015/210_Compiler_Design/Slides/hotspot.pdf
- https://aboullaite.me/understanding-jit-compiler-just-in-time-compiler/

Questions?