Understanding HotSpot JVM Performance with JITWatch

Chris Newland, LJC 2022-07-08


```
git clone https://github.com/AdoptOpenJDK/jitwatch.git

mvn clean package

java -jar ui/target/jitwatch-ui-shaded.jar
```

Chris Newland

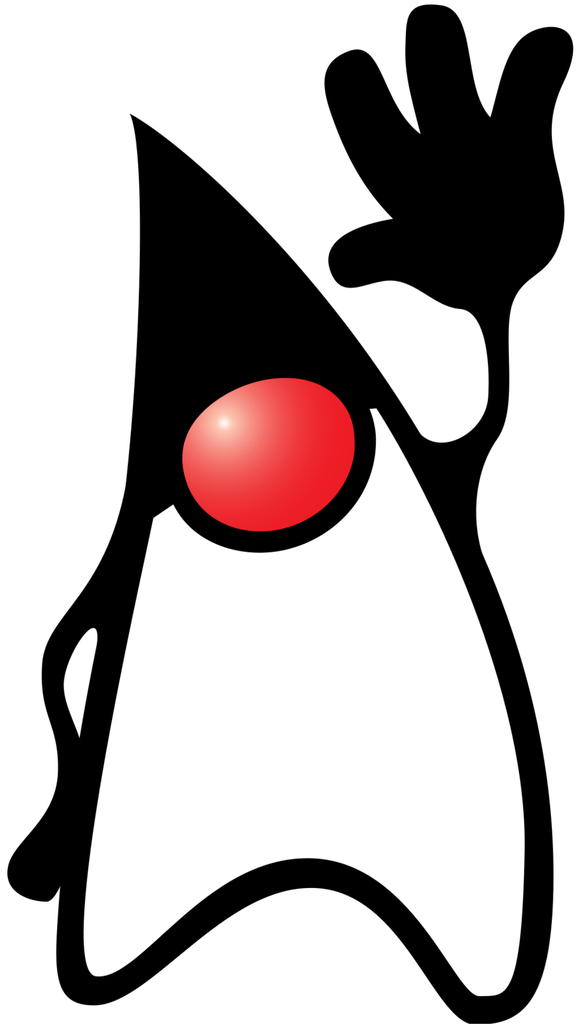
Market data developer at 

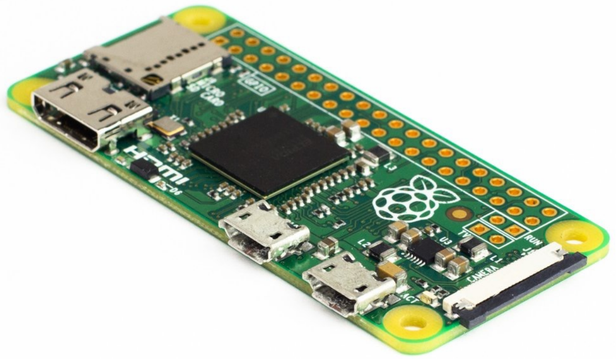
I make tools for understanding the JVM

Java Champion

@chriswhocodes on Twitter

The Amazing JVM





Many Languages

Java, Kotlin, Scala, Groovy, Clojure, JRuby, …

Object-oriented and functional!

Strongly and dynamically typed!

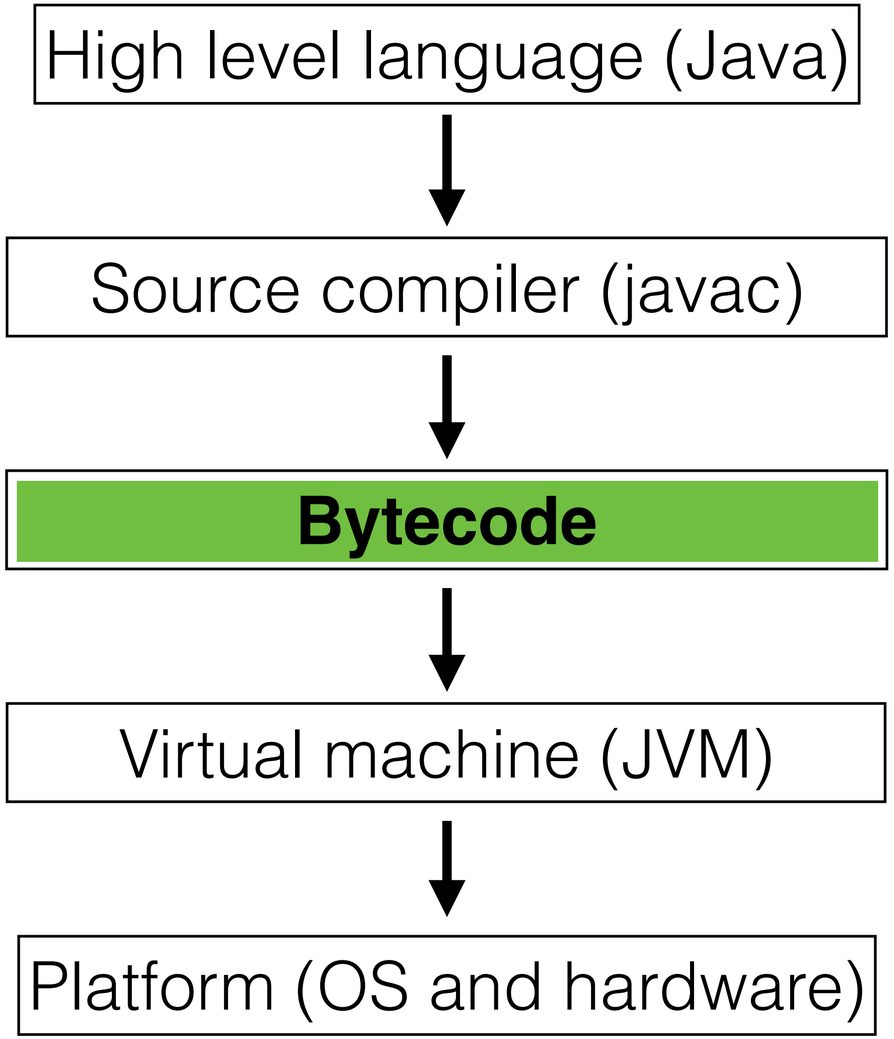
Memory management and concurrency!

Abstraction!

All problems in computer science can be solved by another level of indirection, except of course for the problem of too many indirections.

David Wheeler

A Common Language



Bytecode

(Portable instruction set, 256 possible instructions)

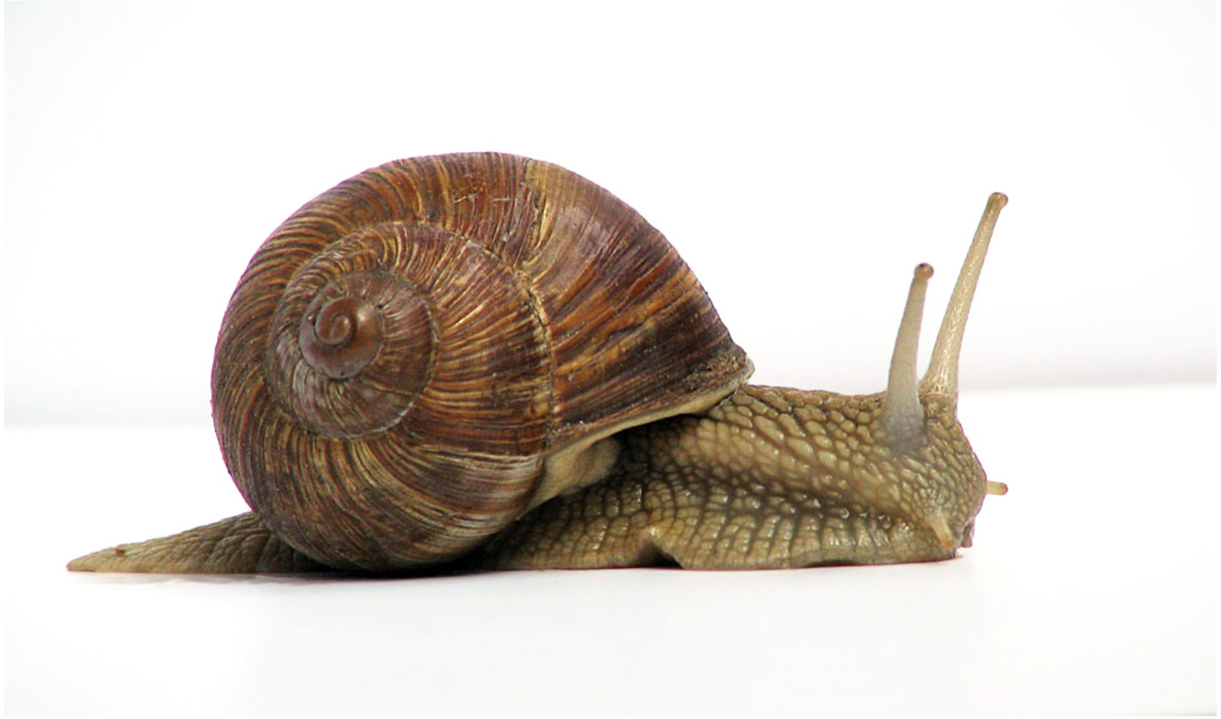
```
public int add(int a, int b)
{
   return a + b;
}
```


```
public int add(int, int);
  descriptor: (II)I
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=3, args_size=3
       0: iload_1
       1: iload_2
       2: iadd
       3: ireturn
```

Interpreted on a virtual stack machine

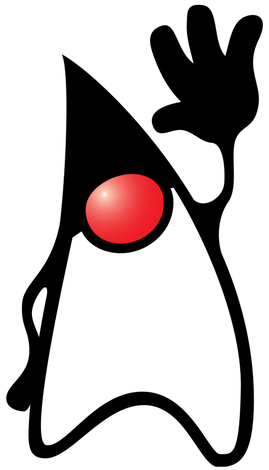
Simple Interpreter

```
while (running)
{
   opcode = getNextOpcode();

   switch(opcode)
   {
   case 0x00:
      // handle
      break;
   case 0x01:
      // handle
      break;

   ...

   case 0xff:
      // handle
      break;
   }
}
```



http://docklandsljc.uk/2016/06/hotspot-hood-microbenchmarking-java.html

Running Faster



Just In Time (JIT)
Profiles running code for Hot Spots
Generates optimised native code
Profiling takes time and resources

GraalVM™

Ahead of Time (AOT)
Produces native executable
Knowledge of target architecture
Full performance from the start

HotSpot Optimisations

lock coarsening

strength reduction

loop unrolling

branch prediction

range check elimination

inlining

CHA

dead code elimination

compiler intrinsics

autobox elimination

copy removal

switch balancing

lock elision

null check elimination

instruction peepholing

devirtualisation

constant propagation

escape analysis

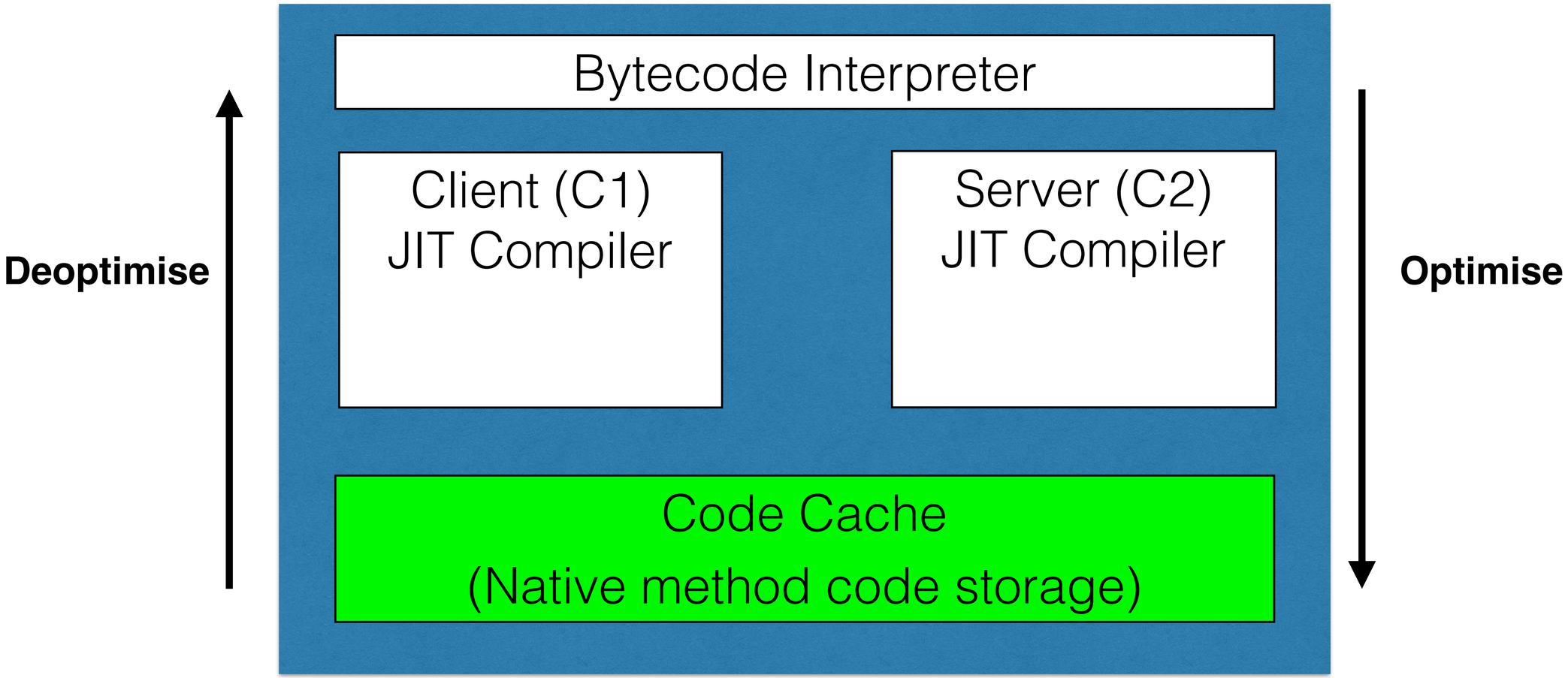
vectorisation

algebraic simplification

register allocation

subexpression elimination

The HotSpot JVM



* plus lots more components!

What triggers the JIT?

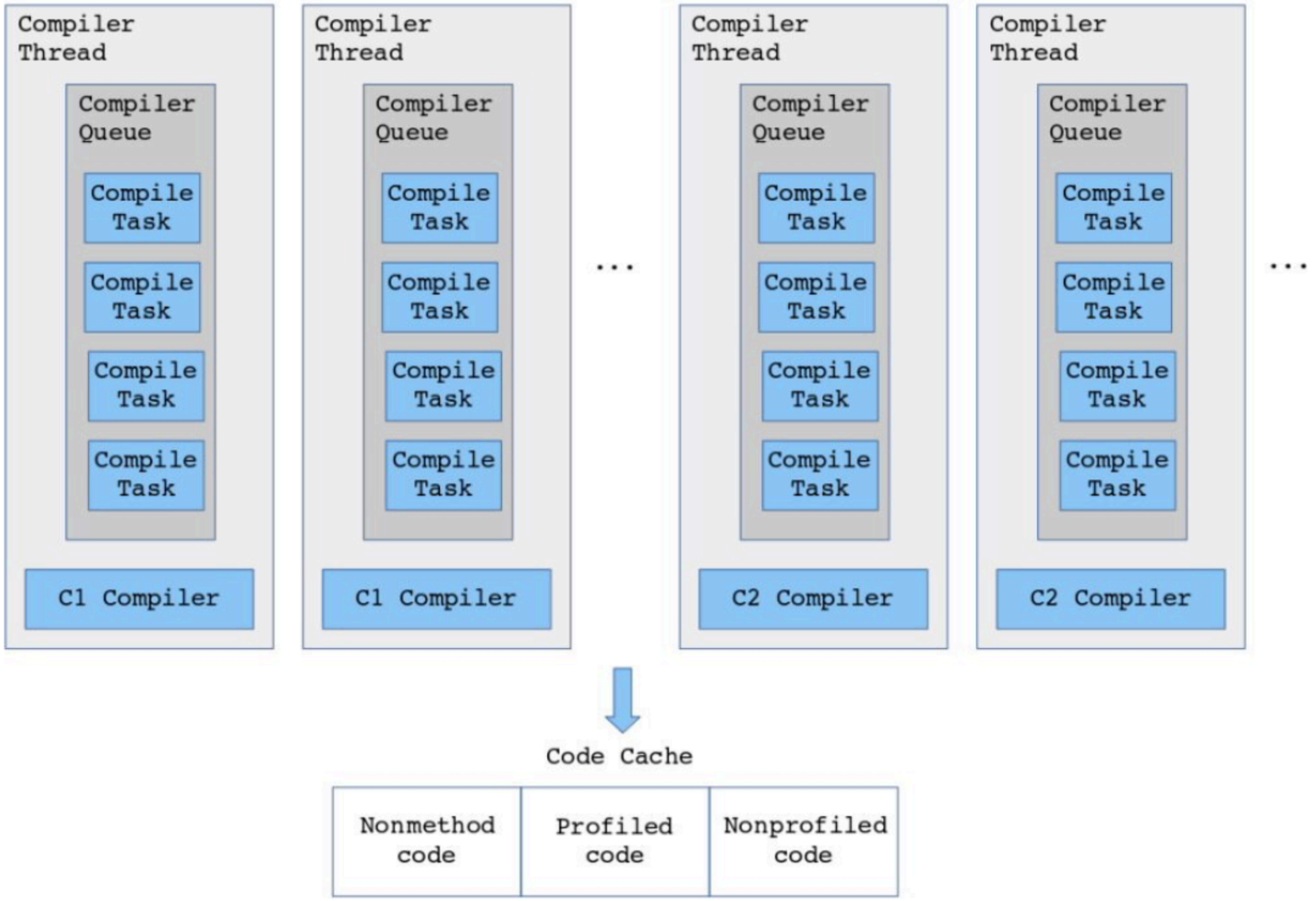
Unit of compilation is the method

JIT counts method invocations

Can also count loop back-edges (OSR)

Compilation task added to a queue

Inside the JIT



Code Cache

JVM memory region for JIT-compiled methods

Can run out of space

Can become fragmented

Since JEP197 (JDK9) - Segmented into

- Profiled code
- Non-profiled code
- JVM internal code

Ergonomics

Garbage collector: G1GC

Initial heap: 1/64th of physical memory

Maximum heap: 1/4th of physical memory

Tiered compilation: Enabled. Code Cache 240MB

Compiler Threads: 4

https://chriswhocodes.com/vm-options-explorer.html

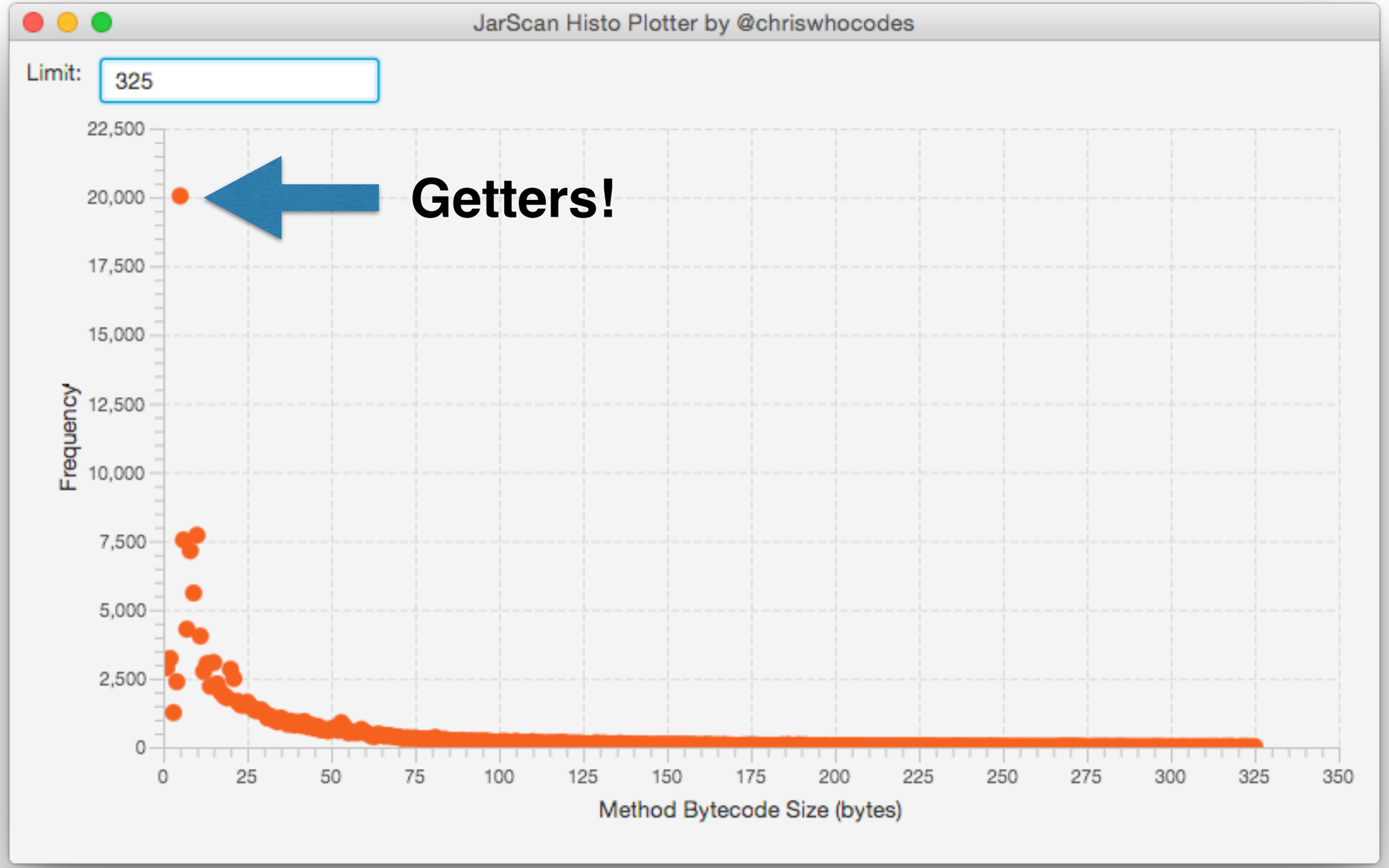
Compilation Levels

Tier	Component	Description	Stored in code cache segment
0	Interpreter	Counts invocations and back edges	Not applicable
1	C1 compiler	Performs full optimization	Nonprofiled code
2	C1 compiler	Counts invocations and back edges	Profiled code
3	C1 compiler	Performs full profiling	Profiled code
4	C2 compiler	Performs full optimization	Nonprofiled code

Compilation patterns

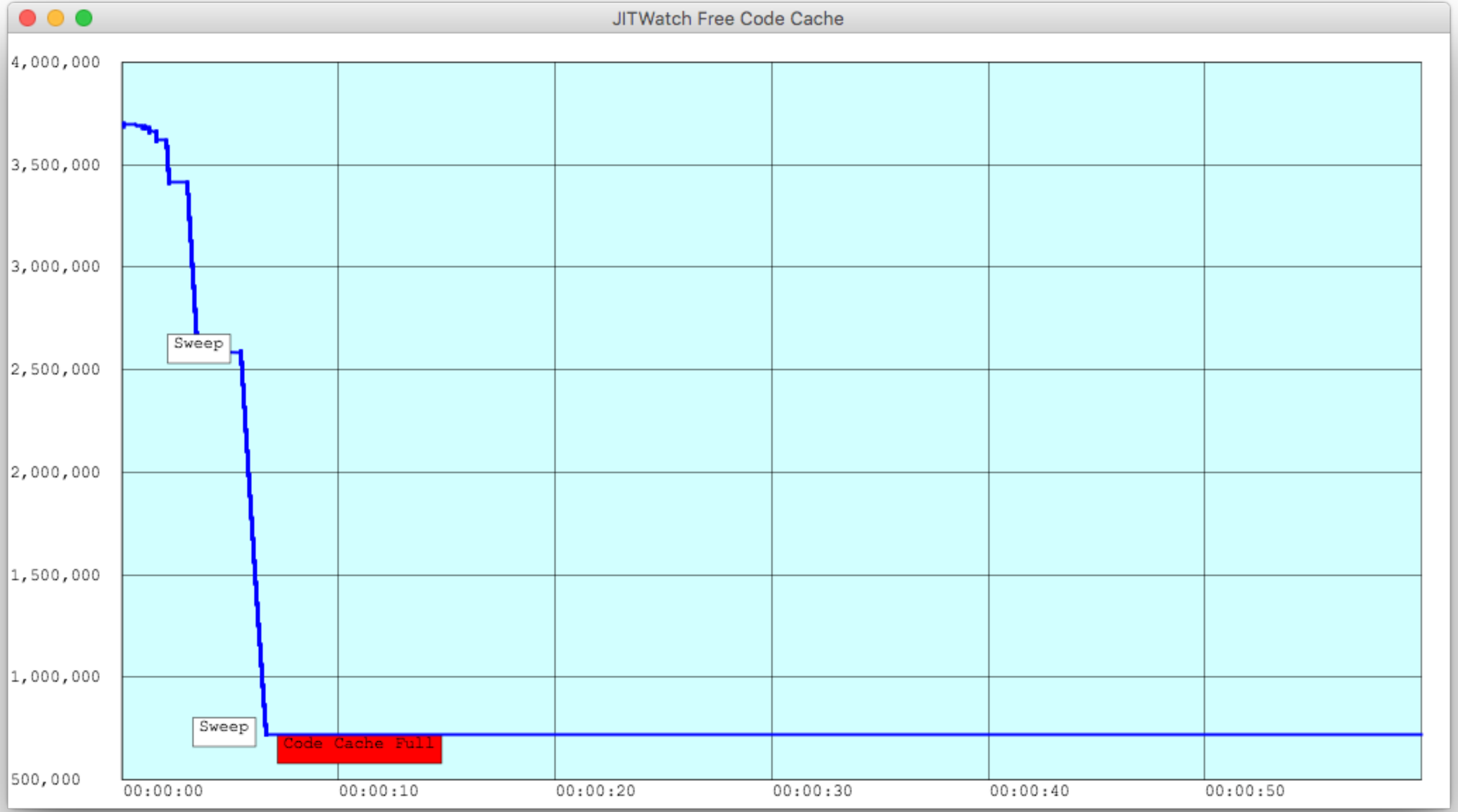
Tier sequence	Explanation
0, 1	Trivial code that will not benefit from C2
0, 3, 4	Tier 3 compilation by C1 with full profiling and then C2 final compilation
0, 2, 3, 4	C2 queues busy; compilation at tier 2 (faster code than tier 3) to minimize the time spent executing tier 3 code
0, 3, 1	Decided during full profiling that method will not benefit from C2 so recompilation at tier 1
0, 4	Tiered compilation disabled; compilation only by C2

Trivial Methods in the JDK

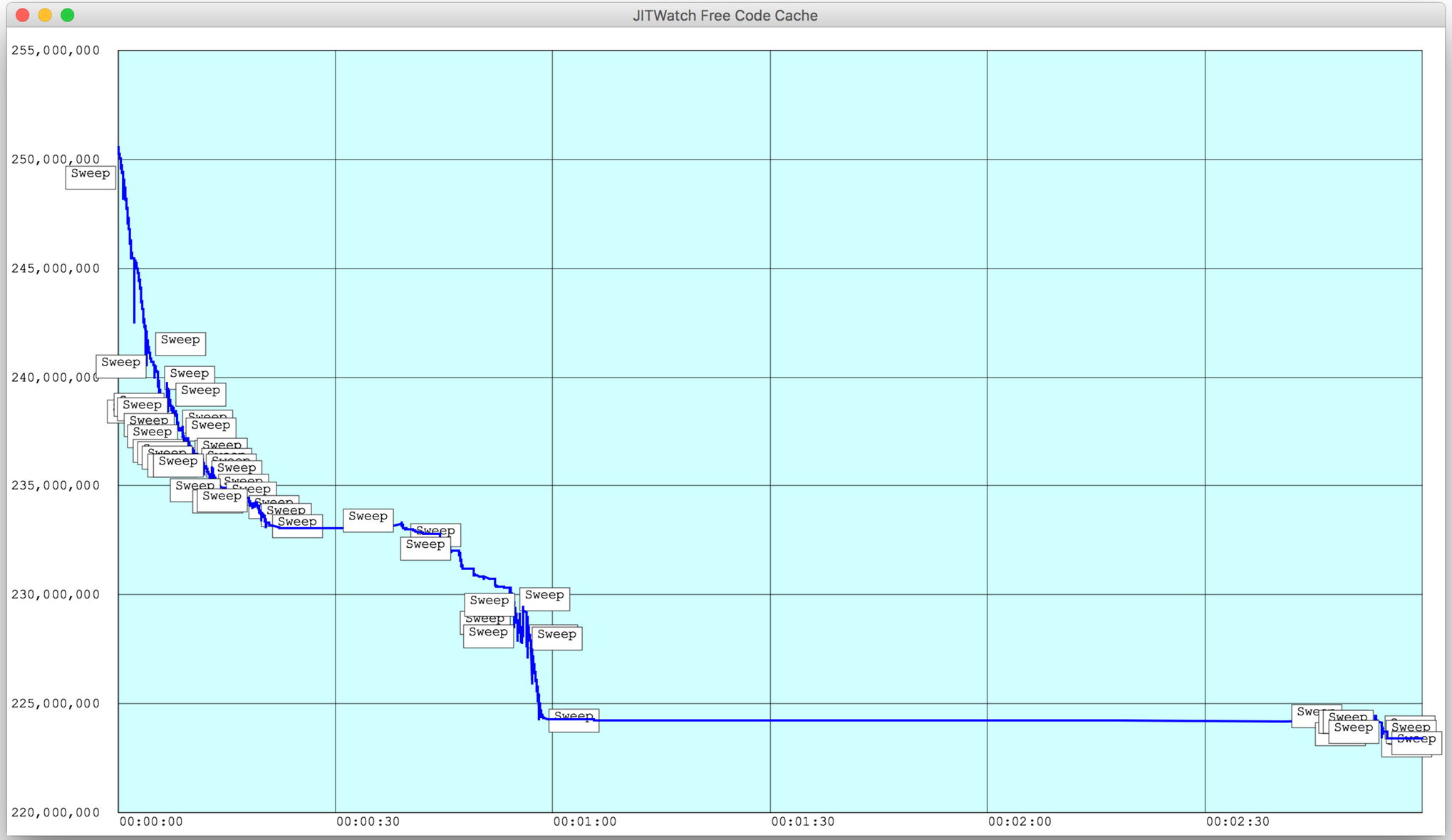


https://www.chrisnewland.com/more-bytecode-geekery-with-jarscan-404

Code Cache Exhaustion



Sweeper activity



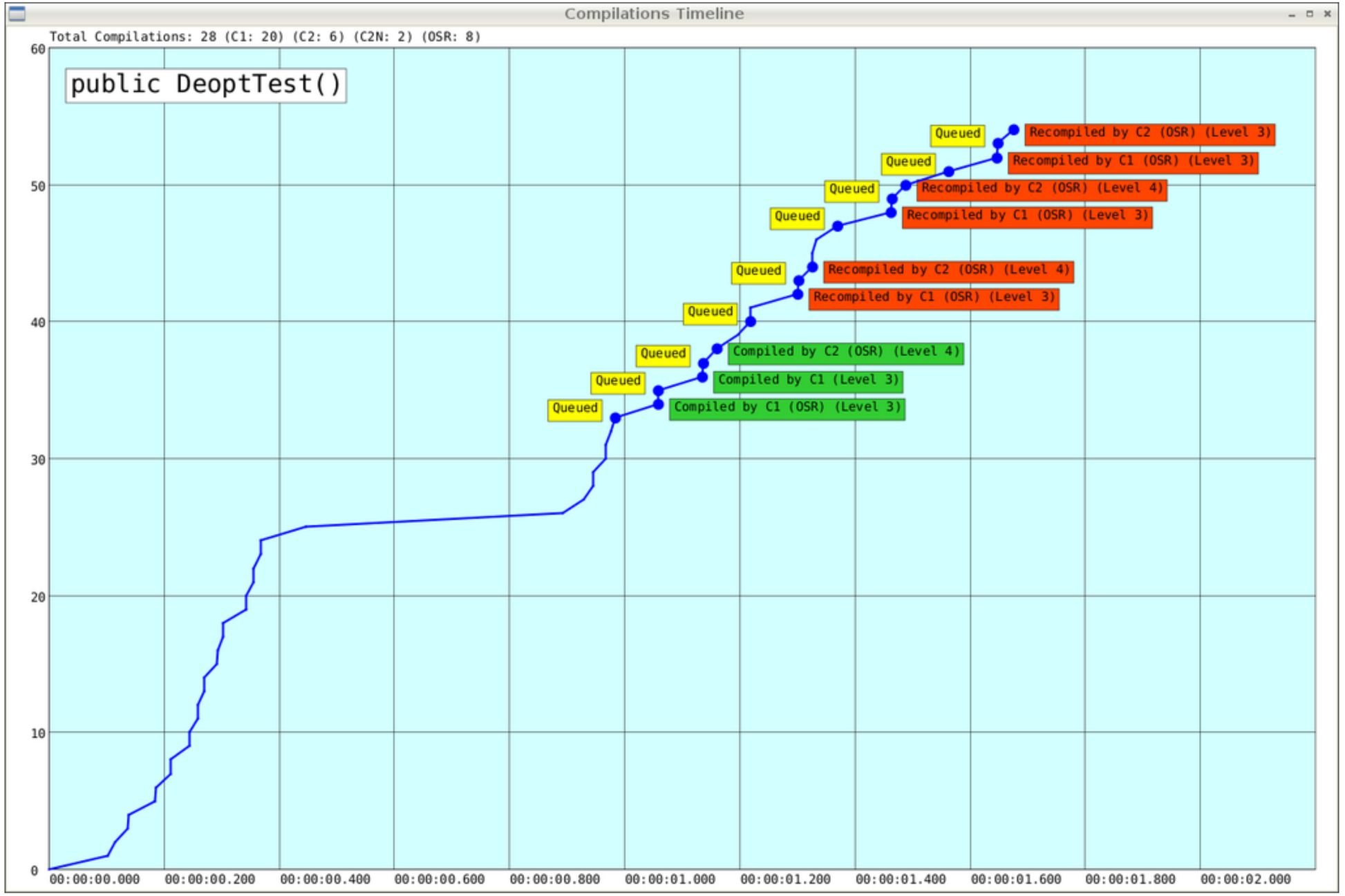
Guess Again?

Many (C2) optimisations are speculative

JVM needs a way back if decision was wrong

Uncommon traps verify if assumption holds

Wrong? Switch back to interpreted code



Repeated deopts can cause poor performance

Logging the JIT

-XX:+UnlockDiagnosticVMOptions

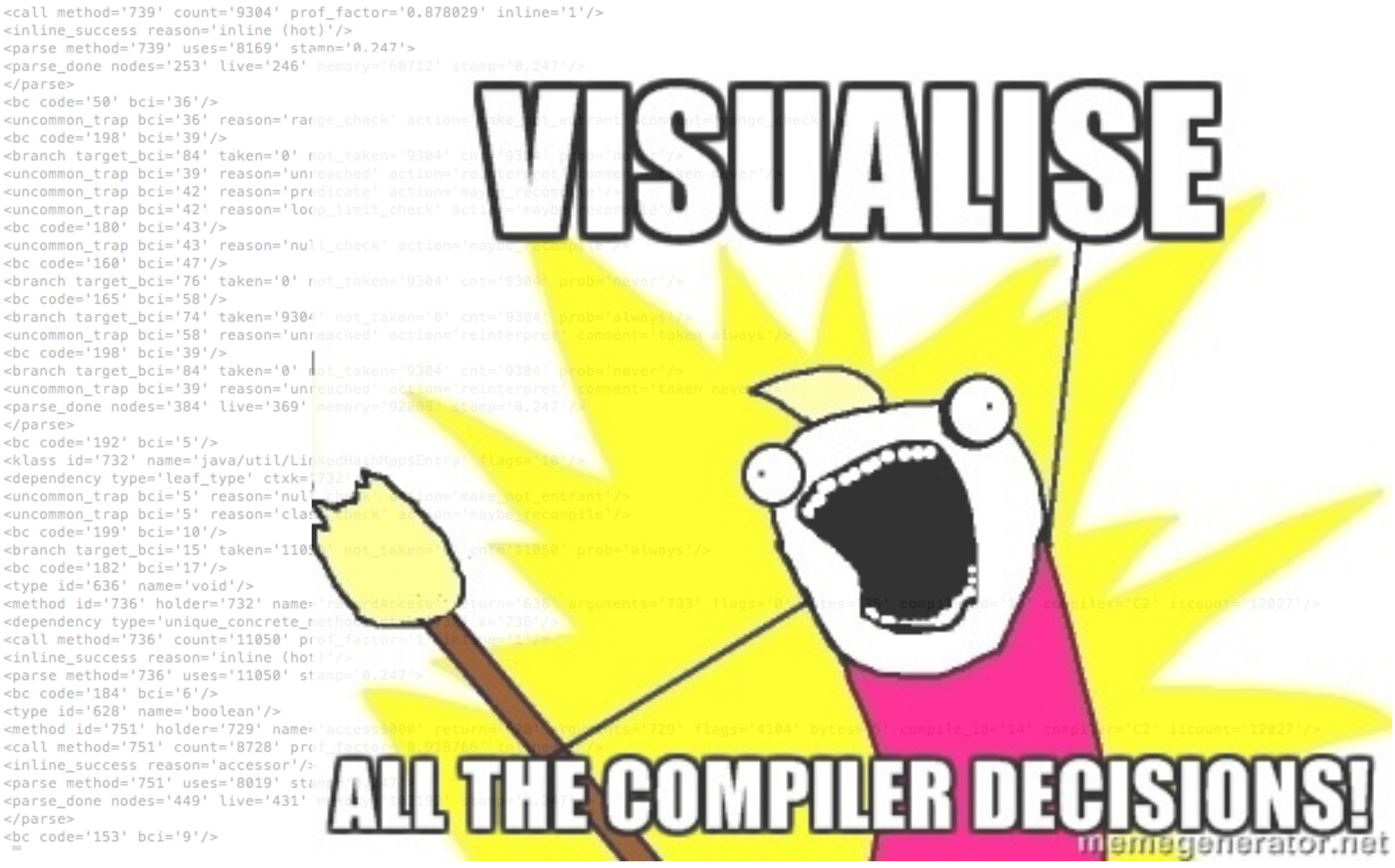
-XX:+LogCompilation

and optionally

-XX:+PrintAssembly

https://blogs.oracle.com/javamagazine/post/java-hotspot-hsdis-disassembler

I heard you like to grep?



JITWatch

Compilations (when, how)

Deoptimisations (why)

Inlining successes and failures

Escape analysis

Branch probabilities

Intrinsics used

Hot throws, stale tasks, vectorisation and more!

Getting Started

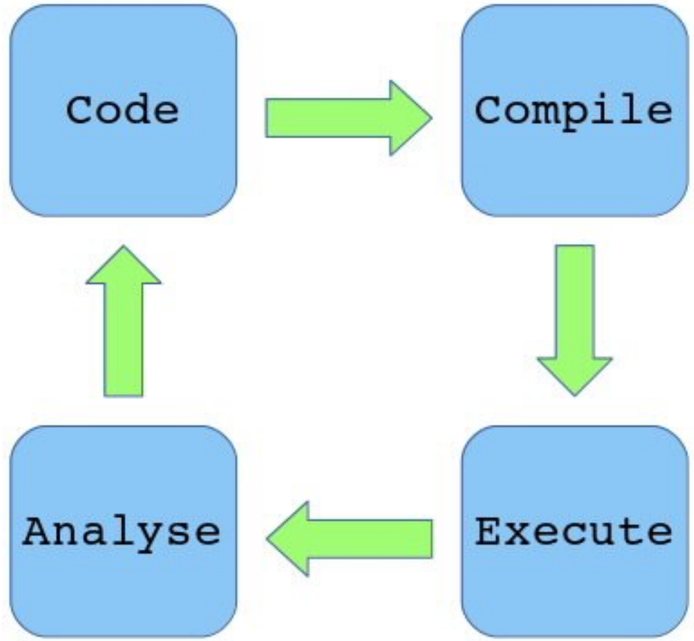
Log File

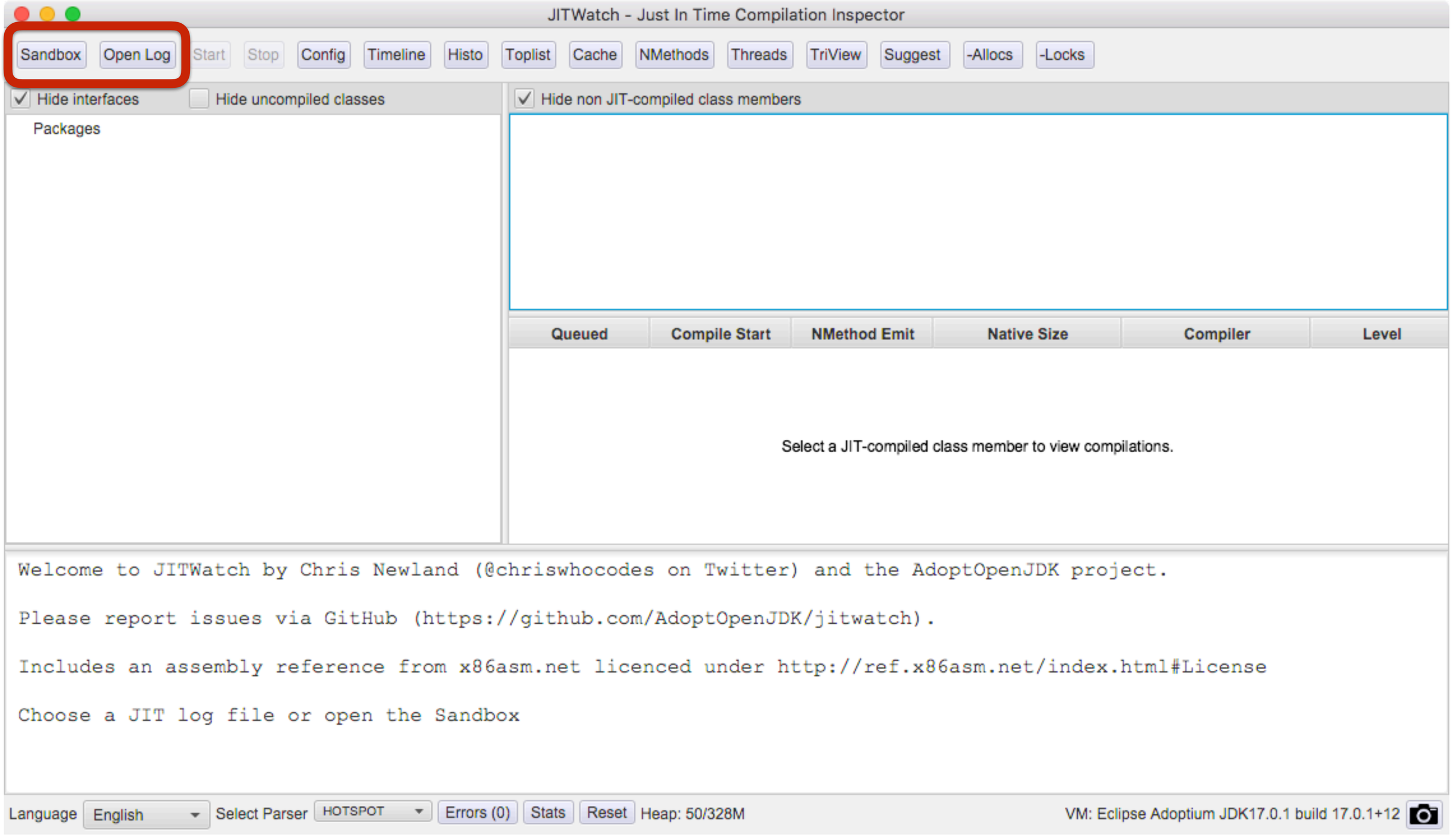
```
</phase>
<phase name='codeemit' stamp='0.352'>
<phase_done stamp='0.352'/>
</phase>
<phase name='codeinstall' stamp='0.35
<phase_done stamp='0.354'/>
</phase>
<code_cache total_blobs='391' nmethod
<task_done success='1' nmsize='216' c
</task>
<task compile_id='61' method='java.la
<phase name='setup' stamp='0.358'>
<phase_done stamp='0.358'/>
</phase>
<phase name='buildIR' stamp='0.358'>
<type id='1109' name='void'/>
<type id='1107' name='int'/>
```

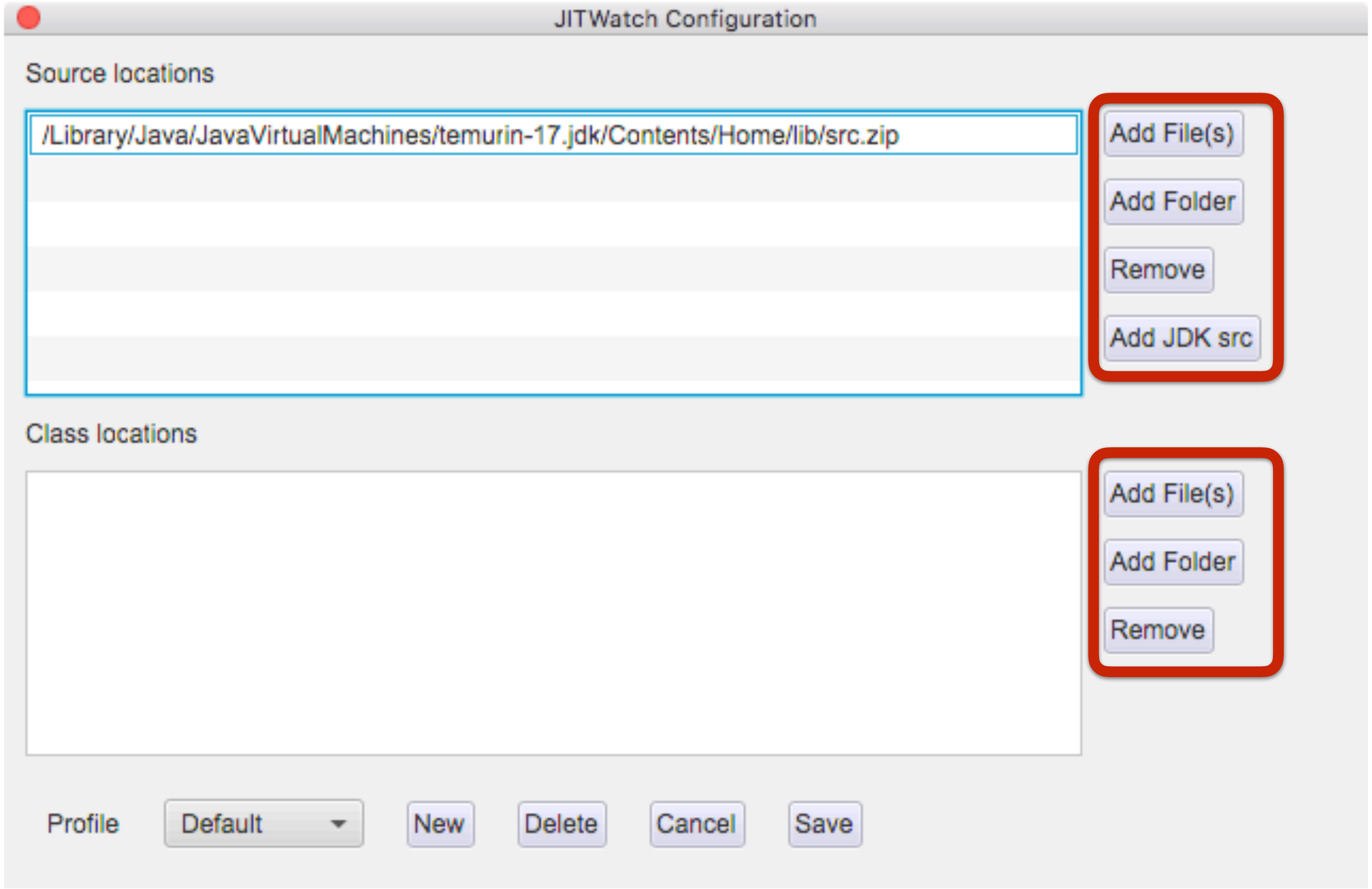
-XX:+UnlockDiagnosticVMOptions

-XX:+LogCompilation

Sandbox



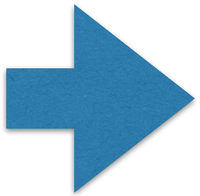

JITWatch - Just In Time Compilation Inspector
Sandbox
Open Log
Start
Stop
Config
Timeline
Histo
Toplist
Cache
NMethods
Threads
TriView
Suggest
-Allocs
-Locks
Hide interfaces
Hide uncompiled classes
Packages
Hide non JIT-compiled class members
Queued
Compile Start
NMethod Emit
Native Size
Compiler
Level
Select a JIT-compiled class member to view compilations.
Welcome to JITWatch by Chris Newland (@chriswhocodes on Twitter) and the AdoptOpenJDK project.
Please report issues via GitHub (https://github.com/AdoptOpenJDK/jitwatch).
Includes an assembly reference from x86asm.net licenced under http://ref.x86asm.net/index.html#License
Choose a JIT log file or open the Sandbox
Language
English
Select Parser
HOTSPOT
Errors (0)
Stats
Reset
Heap: 50/328M
VM: Eclipse Adoptium JDK17.0.1 build 17.0.1+12



Mount source and class locations

Inlining

```
int result = add(a, b);


public int add(int x, int y) {
    return x + y;
}
```



```
int result = a + b;
```

- Copy the body of the callee method into the call site
- Eliminates the cost of method dispatch
- The “Gateway Optimisation”

Sandbox - Code, Compile, Execute, and Analyse JIT logs

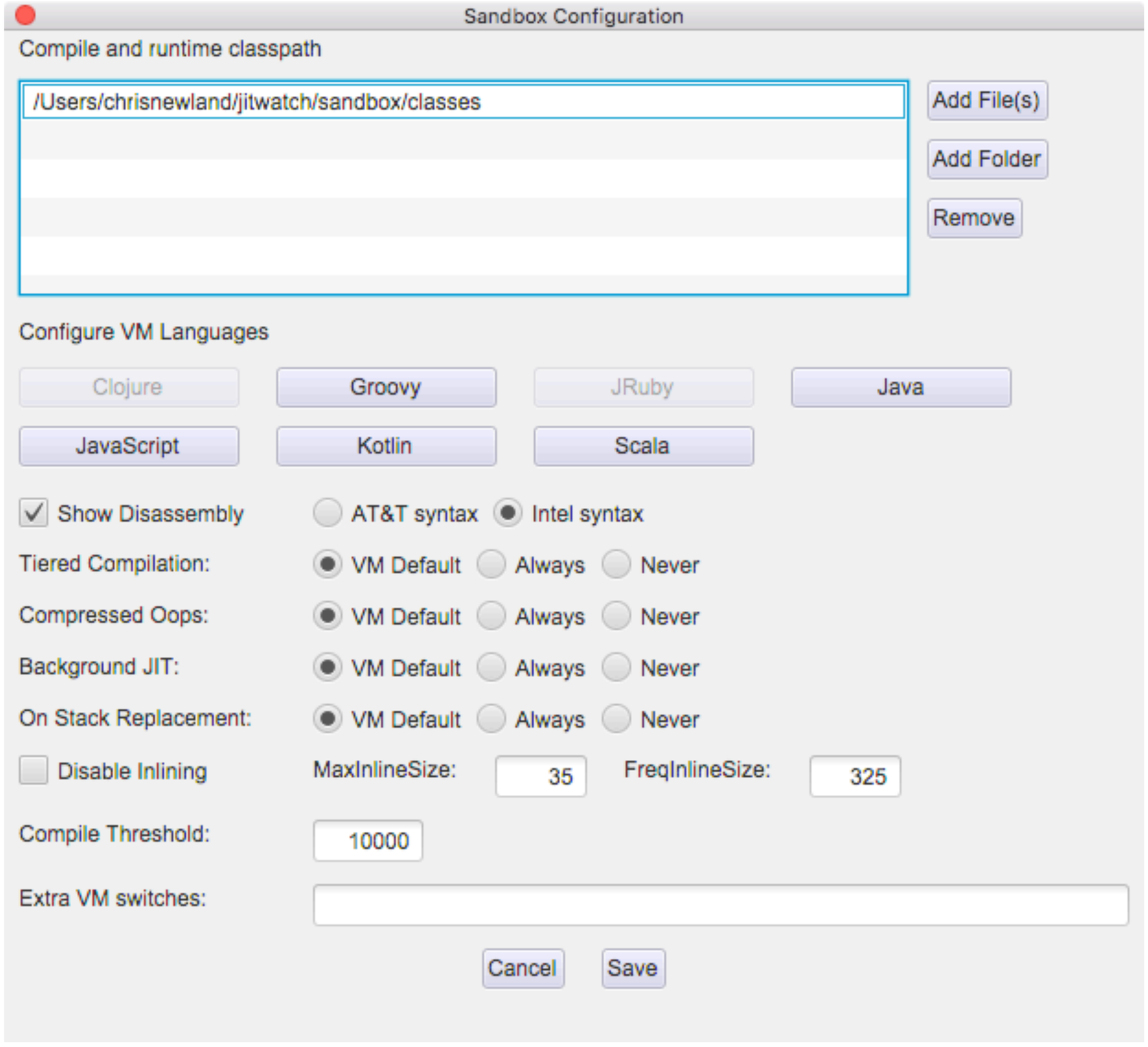
New Editor | Open | Save | Configure Sandbox | Reset Sandbox | Java | Run | View Output

SimpleInliningTest.java

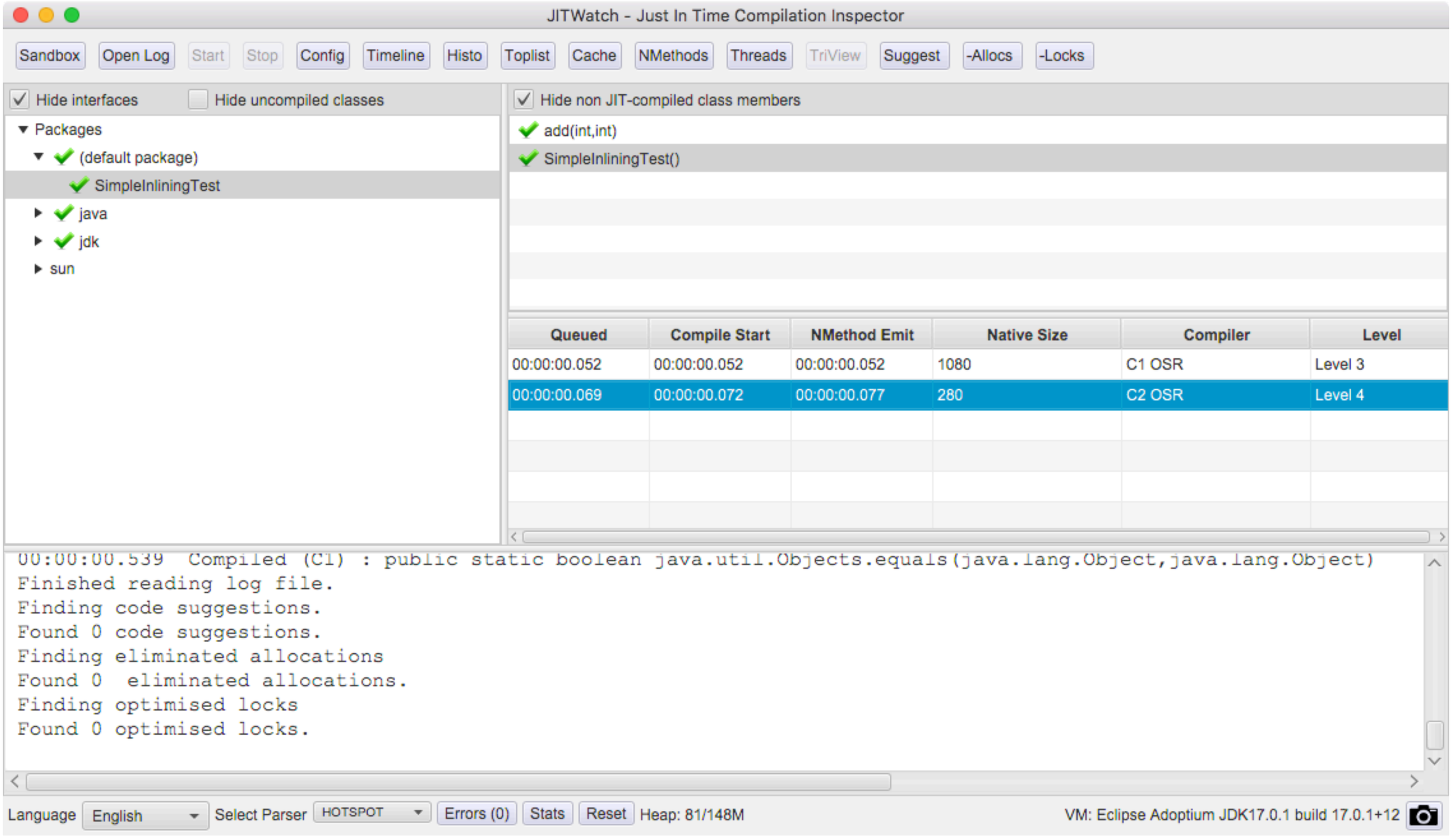
```
1  // The Sandbox is designed to help you learn about the HotSpot JIT compilers.
2  // Please note that the JIT compilers may behave differently when isolating a method
3  // in the Sandbox compared to running your whole application.
4
5  public class SimpleInliningTest
6  {
7      public SimpleInliningTest()
8      {
9          int sum = 0;
10
11         for (int i = 0 ; i < 10_000_000; i++)
12         {
13             sum = this.add(sum, 99); // 0x63 hex
14         }
15
16         System.out.println("Sum:" + sum);
17     }
18
19     public int add(int a, int b)
20     {
21         return a + b;
22     }
23
24     public static void main(String[] args)
25     {
26         new SimpleInliningTest();
27     }
28 }
```

Sandbox Mode

```
Sandbox ready. OS is 'Mac OS X' Architecture is 'x86_64'
Looking for disassembler: hsdis-amd64.dylib
Disassembler available: /Users/chrisnewland/jitwatch/hsdis-amd64.dylib
```



Sandbox Config


JITWatch - Just In Time Compilation Inspector
Sandbox
Open Log
Start
Stop
Config
Timeline
Histo
Toplist
Cache
NMethods
Threads
TriView
Suggest
-Allocs
-Locks
Hide interfaces
Hide uncompiled classes
Packages
(default package)
SimpleInliningTest
java
jdk
sun
Hide non JIT-compiled class members
add(int,int)
SimpleInliningTest()
Queued
Compile Start
NMethod Emit
Native Size
Compiler
Level
00:00:00.052
00:00:00.052
00:00:00.052
1080
C1 OSR
Level 3
00:00:00.069
00:00:00.072
00:00:00.077
280
C2 OSR
Level 4
00:00:00.539 Compiled (C1) : public static boolean java.util.Objects.equals(java.lang.Object,java.lang.Object)
Finished reading log file.
Finding code suggestions.
Found 0 code suggestions.
Finding eliminated allocations
Found 0 eliminated allocations.
Finding optimised locks
Found 0 optimised locks.
Language
English
Select Parser
HOTSPOT
Errors (0)
Stats
Reset
Heap: 81/148M
VM: Eclipse Adoptium JDK17.0.1 build 17.0.1+12

TriView - Source, Bytecode, Assembly Viewer - JITWatch

Class: SimpleInliningTest

Member: SimpleInliningTest()

Source | Bytecode | Assembly | Chain | Journal | LNT | Inlined into | Mouseover

Bytecode size: 41 | Native size: 280 | Compile time: 5ms

Source

```
 1  // The Sandbox is designed to help you learn about the Ho
 2  // Please note that the JIT compilers may behave differer
 3  // in the Sandbox compared to running your whole applicat
 4
 5  public class SimpleInliningTest
 6  {
 7      public SimpleInliningTest()
 8      {
 9          int sum = 0;
10
11          for (int i = 0 ; i < 10_000_000; i++)
12          {
13              sum = this.add(sum, 99); // 63 hex
14          }
15
16          System.out.println("Sum:" + sum);
17      }
18
19      public int add(int a, int b)
20      {
21          return a + b;
22      }
23
24      public static void main(String[] args)
25      {
26          new SimpleInliningTest();
27      }
28  }
```

Bytecode (double click for JVM spec)

```
 0: aload_0
 1: invokespecial   #1   // Method java/lang/Object."<init>"
 4: iconst_0
 5: istore_1
 6: iconst_0
 7: istore_2
 8: iload_2
 9: ldc             #7   // int 10000000
11: if_icmpge       28
14: aload_0
15: iload_1
16: bipush          99
18: invokevirtual   #8   // Method add:(II)I
21: istore_1
22: iinc
25: goto
28: getstatic
31: iload_1
32: invokedynamic
37: invokevirtual   #24  // Method java/io/PrintStream.print
40: return
```

Class: SimpleInliningTest
Method: add
Inlined: Yes, inline (hot)
Count: 4939942
iicount: 5034028
Bytes: 4
Prof factor: 1.000000
Uncommon trap (reason:null_check, action:maybe_recompile)

Ctrl-click to inspect this method
Backspace to return

Assembly

Labels | #2 (C2 / OSR / Level 4)

```
# {method} {0x000000011e800388} '<init>' '()V' in 'SimpleInl
[Entry Point]
0x00000183fae7a4a0: movabs r10,0x106ea9c10  ;   {runtime_cal
0x00000183fae7a4aa: call r10
0x00000183fae7a4ad: data16 xchg ax,ax
0x00000183fae7a4b0: mov DWORD PTR [rsp-0x14000],eax
0x00000183fae7a4b7: push rbp
0x00000183fae7a4b8: sub rsp,0x20
0x00000183fae7a4bc: mov r14,QWORD PTR [rsi+0x10]
0x00000183fae7a4c0: mov ebx,DWORD PTR [rsi+0x8]
0x00000183fae7a4c3: mov r13d,DWORD PTR [rsi]
0x00000183fae7a4c6: mov rdi,rsi
0x00000183fae7a4c9: movabs r10,0x106f48850
0x00000183fae7a4d3: call r10  ;*iload_2 {reexecute=0 rethrow
                                ; - SimpleInliningTest::<init>
0x00000183fae7a4d6: mov r11d,DWORD PTR [r14+0x8]  ; implicit
0x00000183fae7a4da: cmp r11d,0xc00800  ;   {metadata('Simple
0x00000183fae7a4e1: jne L0003
0x00000183fae7a4e3: cmp r13d,0x989680
0x00000183fae7a4ea: jge L0005  ;*if_icmpge {reexecute=0 reth
                                ; - SimpleInliningTest::<init
0x00000183fae7a4f0: imul r10d,r13d,0x63
0x00000183fae7a4f4: mov r11d,ebx
0x00000183fae7a4f7: sub r11d,r10d  ;*iload_2 {reexecute=0 re
                                ; - SimpleInliningTest::<
0x00000183fae7a4fa: inc r13d
0x00000183fae7a4fd: cmp r13d,0x98967f
0x00000183fae7a504: jge L0002
0x00000183fae7a506: movsxd r10,r13d
0x00000183fae7a509: mov r8d,0x989680
0x00000183fae7a50f: sub r8,r10
0x00000183fae7a512: and r8,0xfffffffffffffffe
0x00000183fae7a516: mov ebp,r8d
0x00000183fae7a519: add ebp,r13d  ;*iinc {reexecute=0 rethro
                                ; - SimpleInliningTest::<i
0x00000183fae7a51c: imul ebx,ebp,0x63
0x00000183fae7a51f: add ebx,r11d  ;*iadd {reexecute=0 rethro
                                ; - SimpleInliningTest::ad
                                ; - SimpleInliningTest::<i
          L0000: cmp ebp,0x989680
0x00000183fae7a528: jl L0004  ;*if_icmpge {reexecute=0 retha
```

Mounted class version: 61.0 (Java 17) public SimpleInliningTest() compiled with C2 OSR

invokespecial

Operation

Invoke instance method; special handling for superclass, private, and instance initialization method invocations

Format

```
invokespecial
indexbyte1
indexbyte2
```

Forms

invokespecial = 183 (0xb7)

Operand Stack

..., *objectref*, [*arg1*, [*arg2* ...]] →

...

Description

The unsigned *indexbyte1* and *indexbyte2* are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is (*indexbyte1* << 8) | *indexbyte2*. The run-time constant pool item at that index must be a symbolic reference to a method (§5.1), which gives the name and descriptor (§4.3.3) of the method as well as a symbolic reference to the class in which the method is to be found. The named method is resolved (§5.4.3.3). Finally, if the resolved method is `protected` (§4.6), and it is a member of a superclass of the current class, and the method is not declared in the same run-time package (§5.3) as the current class, then the class of *objectref* must be either the current class or a subclass of the current class

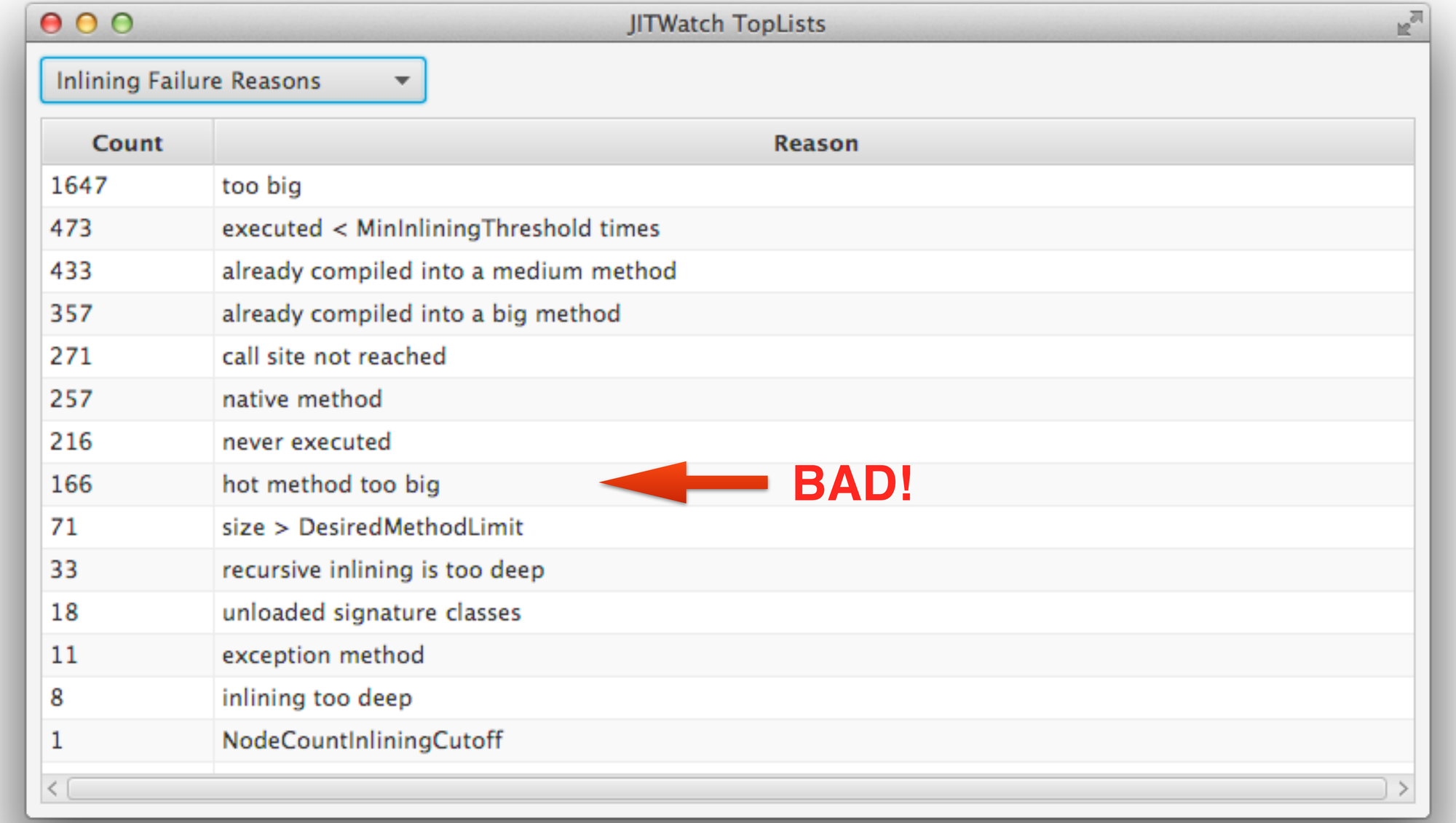
Inlining Limits

Increases size of compiled code

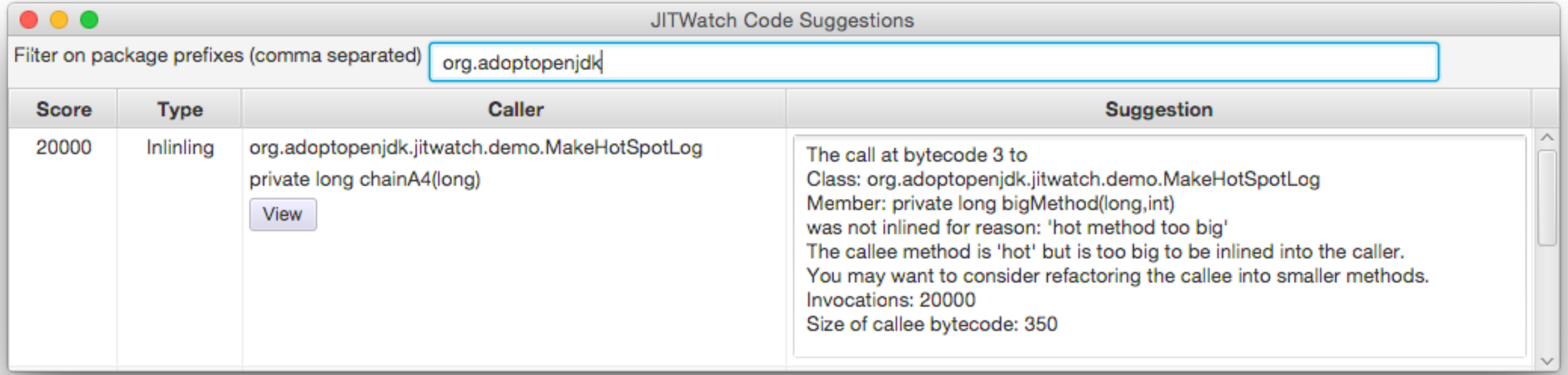
< 35 bytes (**`-XX:MaxInlineSize=n`**)

< 325 bytes and “hot” (**`-XX:FreqInlineSize=n`**)

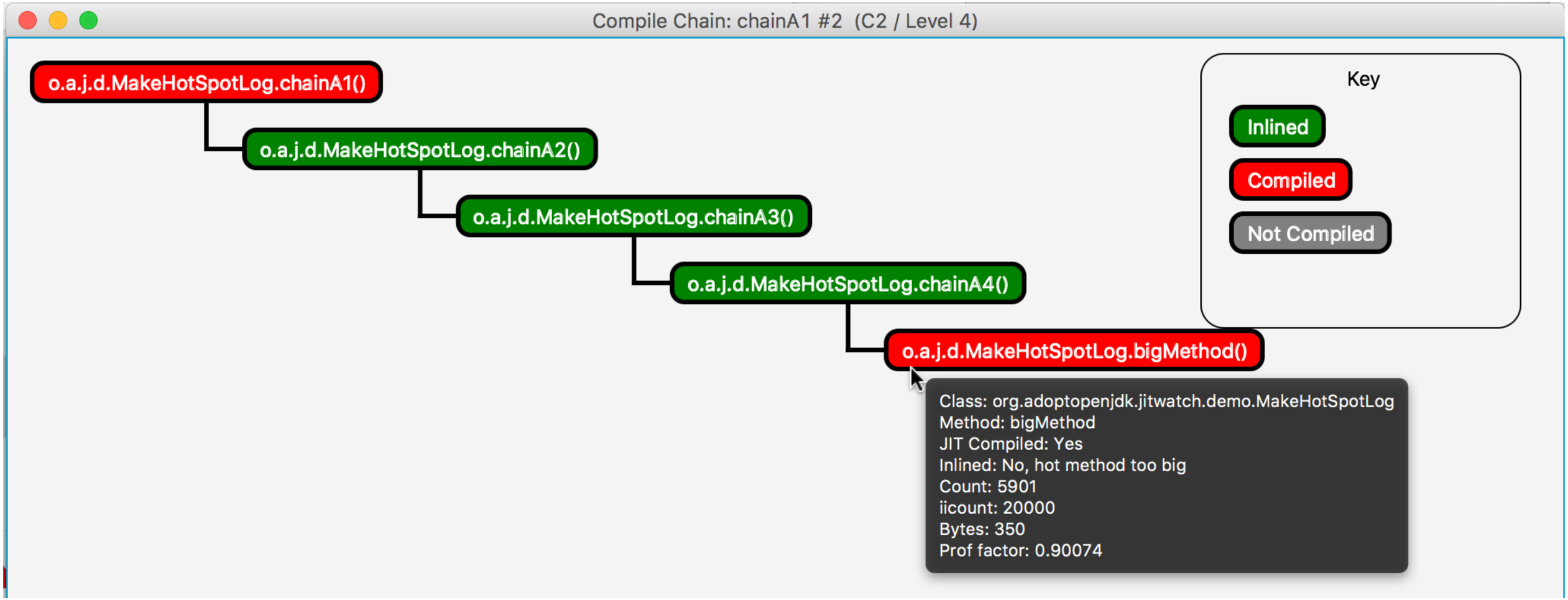
Inlining Failure Modes



Inlining Suggestions



Compile Chain



Look out for inlining failures or deep chains in hot code

JarScan Tool

Static bytecode analysis

Identifies methods above inlining threshold (325 bytes)

Over 3500 above-threshold methods in the JDK

- String.split
- **String.toUpperCase / toLowerCase**

Large JDK methods

```
public String toUpperCase(Locale locale) {
    if (locale == null) {
        throw new NullPointerException();
    }

    int firstLower;
    final int len = value.length;

    /* Now check if there are any characters that need to be changed. */
    scan: {
        for (firstLower = 0 ; firstLower < len; ) {
            int c = (int)value[firstLower];
            int srcCount;
            if ((c >= Character.MIN_HIGH_SURROGATE)
                    && (c <= Character.MAX_HIGH_SURROGATE)) {
                c = codePointAt(firstLower);
                srcCount = Character.charCount(c);
            } else {
                srcCount = 1;
            }
            int upperCaseChar = Character.toUpperCaseEx(c);
            if ((upperCaseChar == Character.ERROR)
                    || (c != upperCaseChar)) {
                break scan;
            }
            firstLower += srcCount;
        }
        return this;
    }

    /* result may grow, so i+resultOffset is the write location in result */
    int resultOffset = 0;
    char[] result = new char[len]; /* may grow */

    /* Just copy the first few upperCase characters. */
    System.arraycopy(value, 0, result, 0, firstLower);

    String lang = locale.getLanguage();
    boolean localeDependent =
            (lang == "tr" || lang == "az" || lang == "lt");
    char[] upperCharArray;
    int upperChar;
    int srcChar;
    int srcCount;
    for (int i = firstLower; i < len; i += srcCount) {
        srcChar = (int)value[i];
        if ((char)srcChar >= Character.MIN_HIGH_SURROGATE &&
            (char)srcChar <= Character.MAX_HIGH_SURROGATE) {
            srcChar = codePointAt(i);
            srcCount = Character.charCount(srcChar);
        } else {
            srcCount = 1;
        }
        if (localeDependent) {
            upperChar = ConditionalSpecialCasing.toUpperCaseEx(this, i, locale);
        } else {
            upperChar = Character.toUpperCaseEx(srcChar);
        }
        if ((upperChar == Character.ERROR)
                || (upperChar >= Character.MIN_SUPPLEMENTARY_CODE_POINT)) {
            if (upperChar == Character.ERROR) {
                if (localeDependent) {
                    upperCharArray =
                            ConditionalSpecialCasing.toUpperCaseCharArray(this, i, locale);
                } else {
```

java.lang.String.toUpperCase()

439 bytes of bytecode

char[] can change size

Too big for inlining

Specialised for ASCII

```
public String toUpperCaseASCII(String source) {
  int len = source.length();

  char[] result = new char[len];

  for (int i = 0; i < len; i++) {
    char c = source.charAt(i);

    if (c >= 'a' && c <= 'z') { c -= 32; }

    result[i] = c;
  }

  return new String(result);
}
```

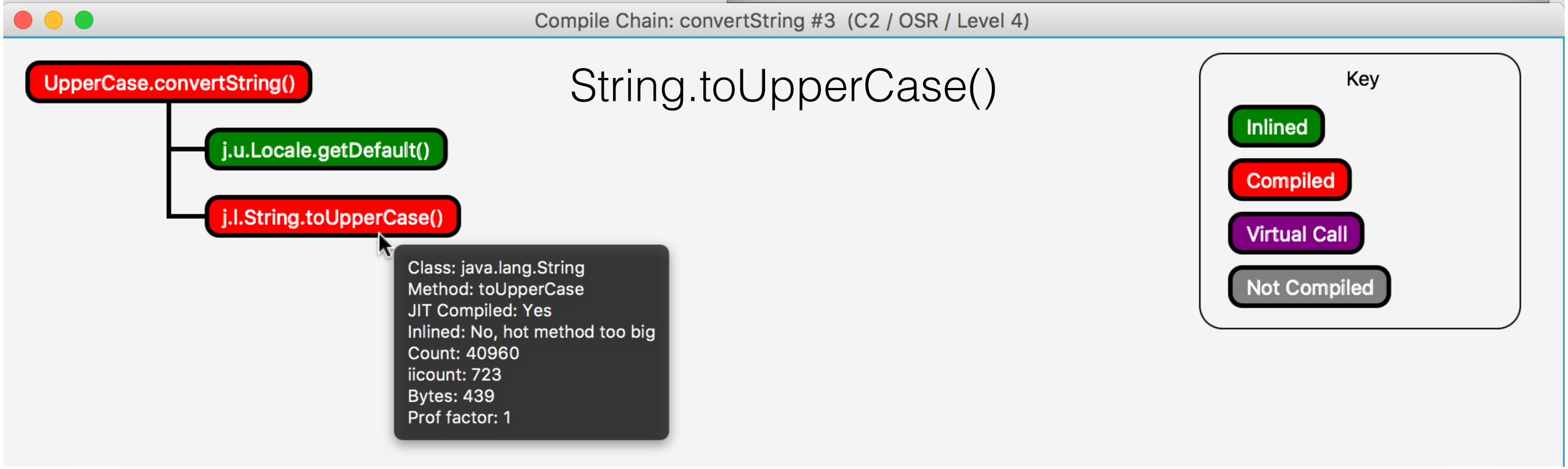
69 bytes of bytecode

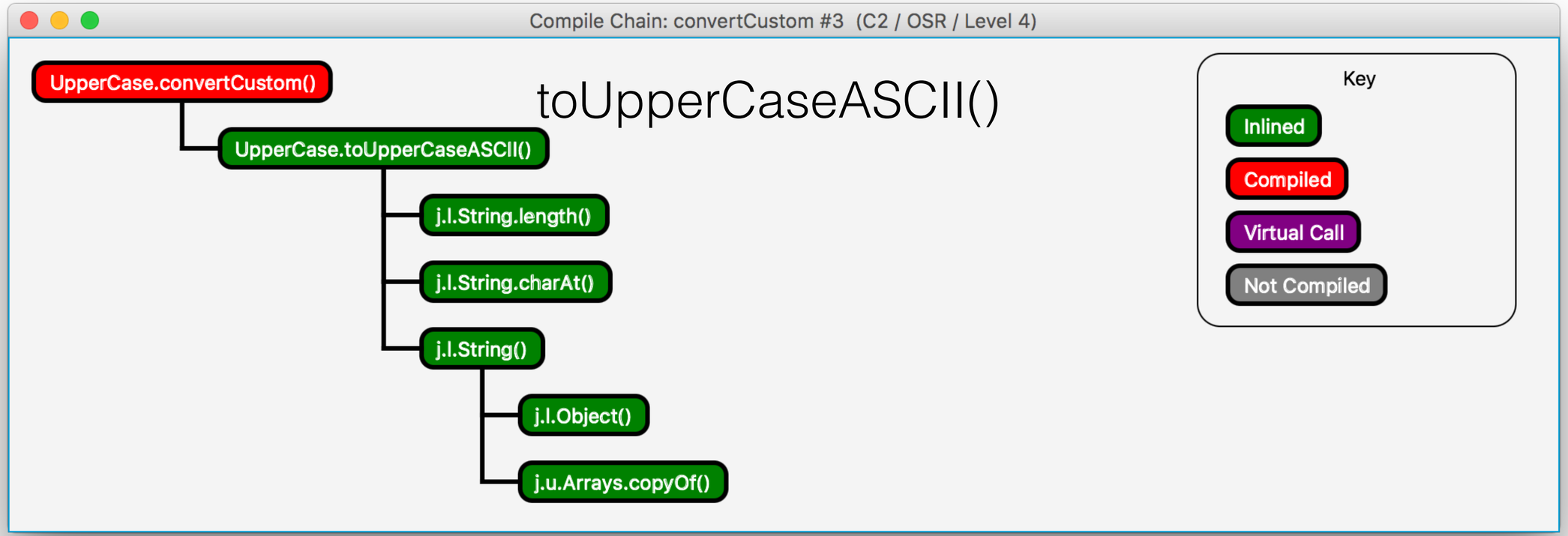
JMH Comparison

```
@State(Scope.Thread)
@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.SECONDS)
public class UpperCase {

    @Benchmark
    public String testStringToUpperCase() {
        return SOURCE.toUpperCase();
    }

    @Benchmark
    public String testCustomToUpperCase() {
        return toUpperCaseASCII(SOURCE);
    }
}
```

```
Benchmark                              Mode  Cnt        Score      Error  Units
UpperCase.testCustomToUpperCase       thrpt  200  1792970.024 ± 8598.436  ops/s
UpperCase.testStringToUpperCase       thrpt  200   820741.756 ± 4346.516  ops/s
```

Custom version is more than twice the ops/second


Compile Chain: convertString #3 (C2 / OSR / Level 4)
UpperCase.convertString()
j.u.Locale.getDefault()
j.l.String.toUpperCase()
Class: java.lang.String
Method: toUpperCase
JIT Compiled: Yes
Inlined: No, hot method too big
Count: 40960
iicount: 723
Bytes: 439
Prof factor: 1
String.toUpperCase()
Key
Inlined
Compiled
Virtual Call
Not Compiled


Compile Chain: convertCustom #3 (C2 / OSR / Level 4)
UpperCase.convertCustom()
UpperCase.toUpperCaseASCII()
j.l.String.length()
j.l.String.charAt()
j.l.String()
j.l.Object()
j.u.Arrays.copyOf()
toUpperCaseASCII()
Key
Inlined
Compiled
Virtual Call
Not Compiled

Callsite Morphism

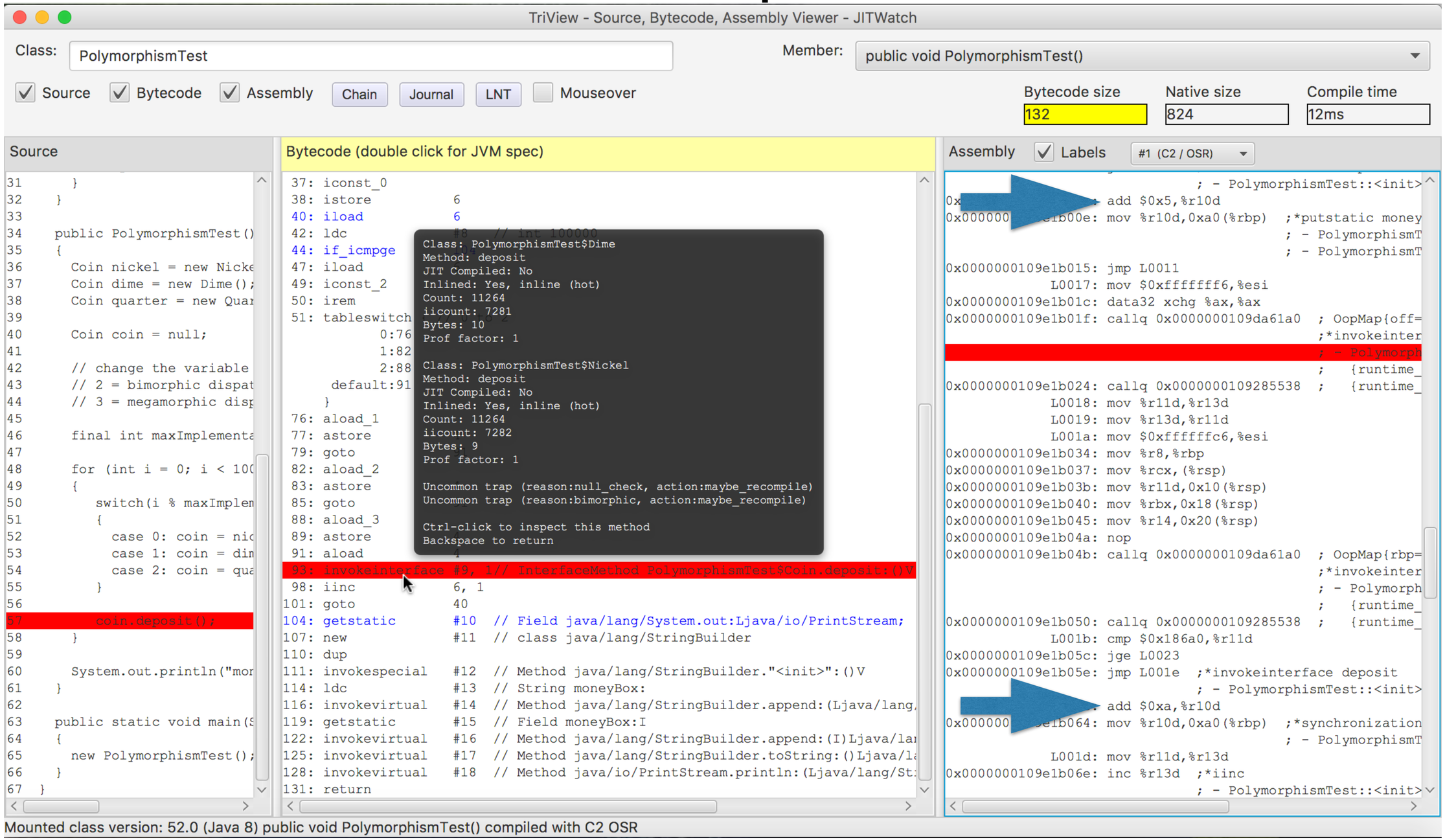
HotSpot counts observed implementations at each callsite.

Too many implementations can prevent inlining.

Implementations	Classification	Inlinable?
1	Monomorphic	Yes
2	Bimorphic	Yes
3+	Megamorphic	No*

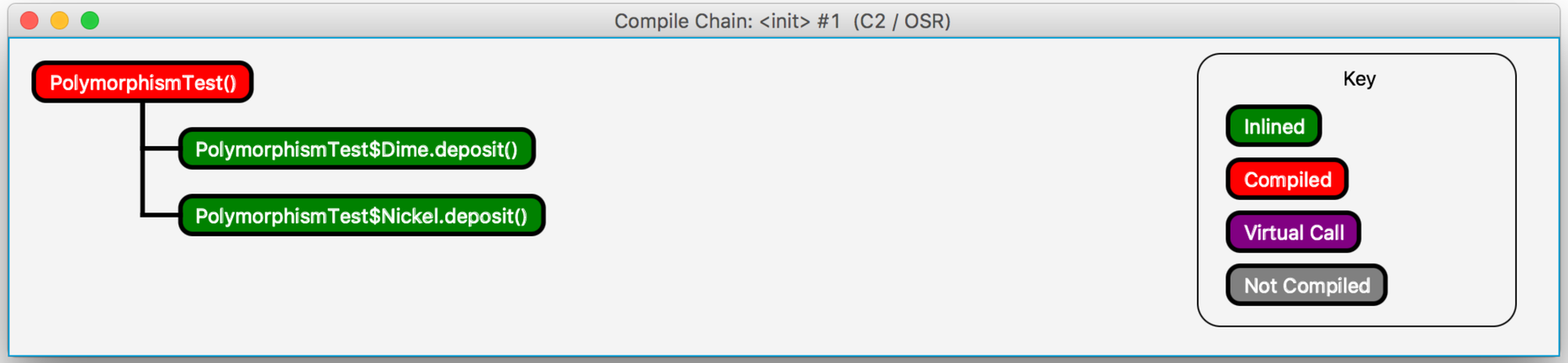
```
public class PolymorphismTest
{
   public interface Coin   { void deposit(); }

   public static int moneyBox = 0;

   public class Nickel  implements Coin { public void deposit() { moneyBox +=  5; } }
   public class Dime    implements Coin { public void deposit() { moneyBox += 10; } }
   public class Quarter implements Coin { public void deposit() { moneyBox += 25; } }

    public PolymorphismTest() {

         Coin nickel = new Nickel();
         Coin dime = new Dime();
         Coin quarter = new Quarter();
         Coin coin = null;

         final int maxImplementations = 2; // 2 OK, 3 Not inlined

         for (int i = 0; i < 100_000; i++) {
             switch(i % maxImplementations) {
                 case 0: coin = nickel;  break;
                 case 1: coin = dime;    break;
                 case 2: coin = quarter; break;
             }

             coin.deposit(); // callsite in question
          }

          System.out.println("moneyBox:" + moneyBox);
    }
}
```

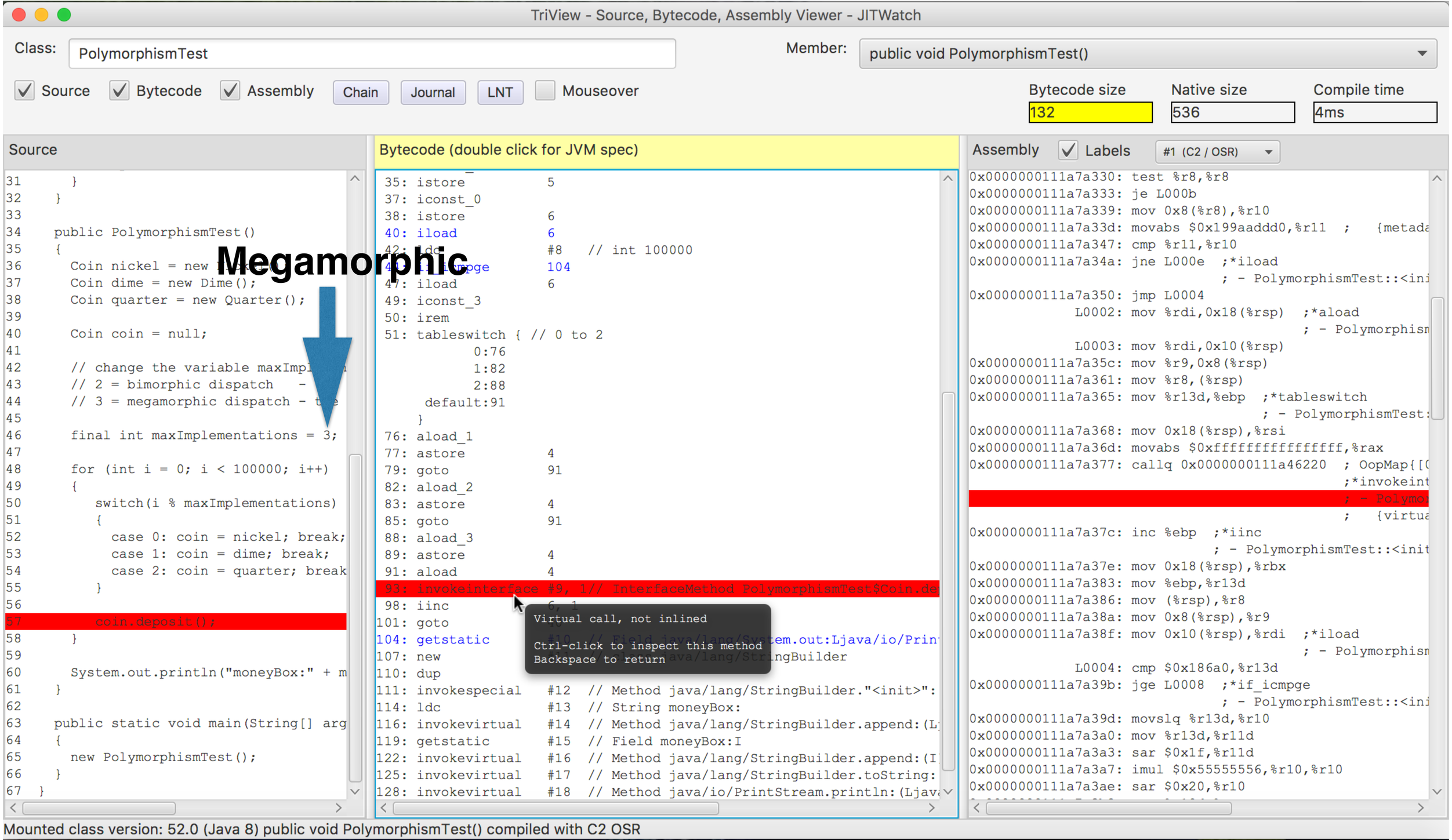
Bimorphic


TriView - Source, Bytecode, Assembly Viewer - JITWatch
Class: PolymorphismTest
Member: public void PolymorphismTest()
Source
Bytecode
Assembly
Chain
Journal
LNT
Mouseover
Bytecode size
Native size
Compile time
132
824
12ms
Source
Bytecode (double click for JVM spec)
Assembly
Labels
#1 (C2 / OSR)
Class: PolymorphismTest$Dime
Method: deposit
JIT Compiled: No
Inlined: Yes, inline (hot)
Count: 11264
iicount: 7281
Bytes: 10
Prof factor: 1
Class: PolymorphismTest$Nickel
Method: deposit
JIT Compiled: No
Inlined: Yes, inline (hot)
Count: 11264
iicount: 7282
Bytes: 9
Prof factor: 1
Uncommon trap (reason:null_check, action:maybe_recompile)
Uncommon trap (reason:bimorphic, action:maybe_recompile)
Ctrl-click to inspect this method
Backspace to return
93: invokeinterface #9, 1// InterfaceMethod PolymorphismTest$Coin.deposit:()V
Mounted class version: 52.0 (Java 8) public void PolymorphismTest() compiled with C2 OSR

Bimorphic

```
Class: PolymorphismTest$Dime
Method: deposit
JIT Compiled: No
Inlined: Yes, inline (hot)
Count: 11264
iicount: 7281
Bytes: 10
Prof factor: 1

Class: PolymorphismTest$Nickel
Method: deposit
JIT Compiled: No
Inlined: Yes, inline (hot)
Count: 11264
iicount: 7282
Bytes: 9
Prof factor: 1

Uncommon trap (reason:null_check, action:maybe_recompile)
Uncommon trap (reason:bimorphic, action:maybe_recompile)

Ctrl-click to inspect this method
Backspace to return
```



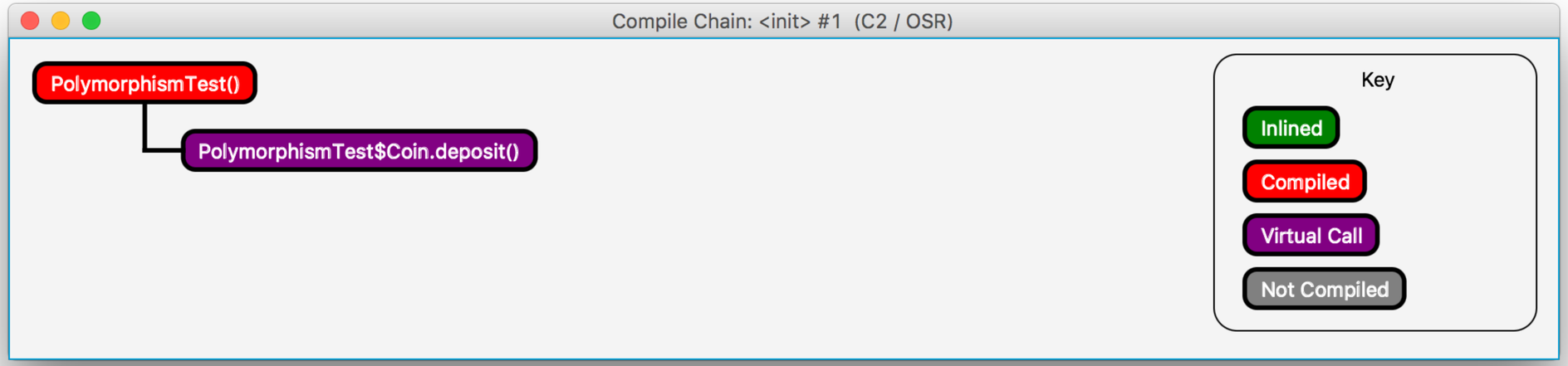

TriView - Source, Bytecode, Assembly Viewer - JITWatch
Class: PolymorphismTest
Member: public void PolymorphismTest()
Source
Bytecode
Assembly
Chain
Journal
LNT
Mouseover
Bytecode size
132
Native size
536
Compile time
4ms
Source
Bytecode (double click for JVM spec)
Assembly
Labels
#1 (C2 / OSR)
Megamorphic
31 }
32 }
33
34 public PolymorphismTest()
35 {
36 Coin nickel = new
37 Coin dime = new Dime();
38 Coin quarter = new Quarter();
39
40 Coin coin = null;
41
42 // change the variable maxImpl
43 // 2 = bimorphic dispatch -
44 // 3 = megamorphic dispatch - t
45
46 final int maxImplementations = 3;
47
48 for (int i = 0; i < 100000; i++)
49 {
50 switch(i % maxImplementations)
51 {
52 case 0: coin = nickel; break;
53 case 1: coin = dime; break;
54 case 2: coin = quarter; break
55 }
56
57 coin.deposit();
58 }
59
60 System.out.println("moneyBox:" + m
61 }
62
63 public static void main(String[] arg
64 {
65 new PolymorphismTest();
66 }
67 }
35: istore 5
37: iconst_0
38: istore 6
40: iload 6
42: ldc #8 // int 100000
45: if_icmpge 104
47: iload 6
49: iconst_3
50: irem
51: tableswitch { // 0 to 2
0:76
1:82
2:88
default:91
}
76: aload_1
77: astore 4
79: goto 91
82: aload_2
83: astore 4
85: goto 91
88: aload_3
89: astore 4
91: aload 4
93: invokeinterface #9, 1// InterfaceMethod PolymorphismTest$Coin.de
98: iinc
101: goto
104: getstatic
107: new
110: dup
111: invokespecial #12 // Method java/lang/StringBuilder."<init>":
114: ldc #13 // String moneyBox:
116: invokevirtual #14 // Method java/lang/StringBuilder.append:(L
119: getstatic #15 // Field moneyBox:I
122: invokevirtual #16 // Method java/lang/StringBuilder.append:(I
125: invokevirtual #17 // Method java/lang/StringBuilder.toString:
128: invokevirtual #18 // Method java/io/PrintStream.println:(Ljava
Virtual call, not inlined
Ctrl-click to inspect this method
Backspace to return
0x0000000111a7a330: test %r8,%r8
0x0000000111a7a333: je L000b
0x0000000111a7a339: mov 0x8(%r8),%r10
0x0000000111a7a33d: movabs $0x199aaddd0,%r11 ; {metada
0x0000000111a7a347: cmp %r11,%r10
0x0000000111a7a34a: jne L000e ;*iload
; - PolymorphismTest::<ini
0x0000000111a7a350: jmp L0004
L0002: mov %rdi,0x18(%rsp) ;*aload
; - Polymorphism
L0003: mov %rdi,0x10(%rsp)
0x0000000111a7a35c: mov %r9,0x8(%rsp)
0x0000000111a7a361: mov %r8,(%rsp)
0x0000000111a7a365: mov %r13d,%ebp ;*tableswitch
; - PolymorphismTest:
0x0000000111a7a368: mov 0x18(%rsp),%rsi
0x0000000111a7a36d: movabs $0xffffffffffffffff,%rax
0x0000000111a7a377: callq 0x0000000111a46220 ; OopMap{[(
;*invokeint
; - Polymo
; {virtua
0x0000000111a7a37c: inc %ebp ;*iinc
; - PolymorphismTest::<init
0x0000000111a7a37e: mov 0x18(%rsp),%rbx
0x0000000111a7a383: mov %ebp,%r13d
0x0000000111a7a386: mov (%rsp),%r8
0x0000000111a7a38a: mov 0x8(%rsp),%r9
0x0000000111a7a38f: mov 0x10(%rsp),%rdi ;*iload
; - Polymorphism
L0004: cmp $0x186a0,%r13d
0x0000000111a7a39b: jge L0008 ;*if_icmpge
; - PolymorphismTest::<ini
0x0000000111a7a39d: movslq %r13d,%r10
0x0000000111a7a3a0: mov %r13d,%r11d
0x0000000111a7a3a3: sar $0x1f,%r11d
0x0000000111a7a3a7: imul $0x55555556,%r10,%r10
0x0000000111a7a3ae: sar $0x20,%r10
Mounted class version: 52.0 (Java 8) public void PolymorphismTest() compiled with C2 OSR

Megamorphic

```
 91: aload           4
 93: invokeinterface #9, 1// InterfaceMethod PolymorphismTest$Coin.de
 98: iinc            6, 1
101: goto
104: getstatic                                    tem.out:Ljava/io/Prin
107: new                                          ringBuilder
110: dup
```

Virtual call, not inlined

Ctrl-click to inspect this method
Backspace to return



Escape Analysis

Scope-based optimisations

Eliminate heap allocations

Lock elision

NoEscape	ArgEscape

`public long noEscape()`
`{`
`  long sum = 0;`
 	
`  for (int i=0; i<BIG; i++)`
`  {`
`  MyObj foo = new MyObj(i);`
 	
`  sum += foo.bar();`
`  }`
 	
`  return sum;`
`}`
	`public long argEscape()`
	`{`
	`  long sum = 0;`
	
	`  for (int i=0; i<BIG; i++)`
	`  {`
	`  MyObj foo = new MyObj(i);`
	
	`  sum += extBar(foo);`
	`  }`
	
	`  return sum;`
	`}`

Object foo doesn't escape the loop scope.	Object foo escapes loop scope by passing as arg to extBar().

Avoid heap allocations

NoEscape objects are “exploded”

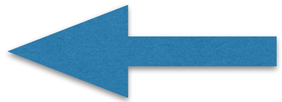
Fields are treated as locals

Register allocator decides where they are stored

Prefer registers

Spill to stack if necessary

```
public class EscapeTest {
   private final int val;

   public EscapeTest(final int val) { this.val = val; }

   public boolean equals(EscapeTest et) { return this.val == et.val; }

   public static int run() {
      int matches = 0;

      java.util.Random random = new java.util.Random();

      for (int i = 0; i < 100_000_000; i++) {
         int v1 = random.nextBoolean() ? 1 : 0;
         int v2 = random.nextBoolean() ? 1 : 0;

         final EscapeTest e1 = new EscapeTest(v1);
         final EscapeTest e2 = new EscapeTest(v2);

         if (e1.equals(e2)) { matches++; }
      }

      return matches;
   }

   public static void main(final String[] args) {
      System.out.println(run());
   }
}
```



Inlining prevents ArgEscape of e2

Hot loop allocations

With Escape Analysis

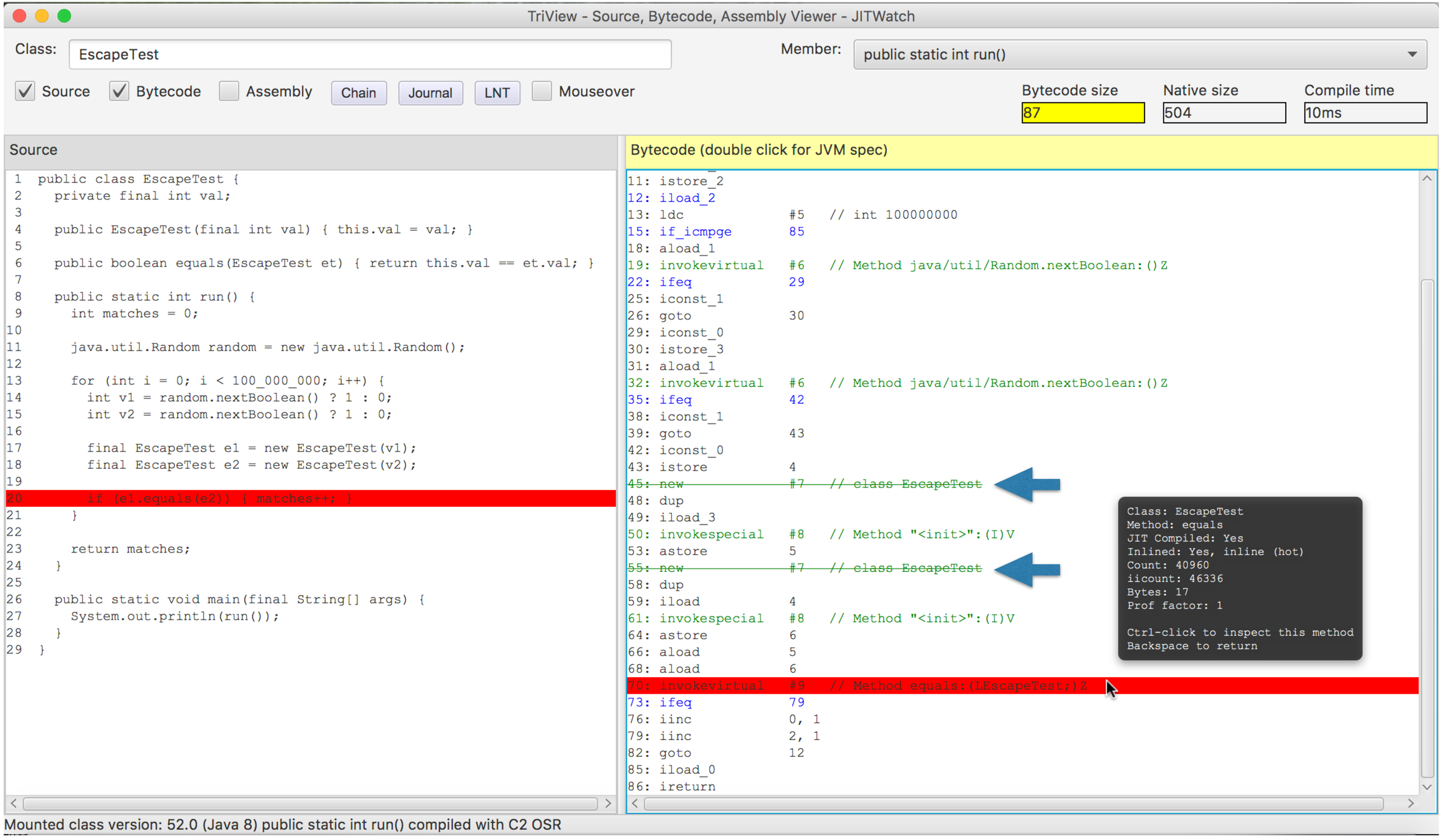
100m loops. No GCs

```
java  -Xms1G -Xmx1G -XX:+PrintGCDetails -verbose:gc  EscapeTest
50001193
Heap
 PSYoungGen      total 305664K, used 20972K [0x00000007aab00000, 0x00000007c0000000, 0x00000007c0000000)
  eden space 262144K, 8% used [0x00000007aab00000,0x00000007abf7b038,0x00000007bab00000)
  from space 43520K, 0% used [0x00000007bd580000,0x00000007bd580000,0x00000007c0000000)
  to   space 43520K, 0% used [0x00000007bab00000,0x00000007bab00000,0x00000007bd580000)
 ParOldGen       total 699392K, used 0K [0x0000000780000000, 0x00000007aab00000, 0x00000007aab00000)
  object space 699392K, 0% used [0x0000000780000000,0x0000000780000000,0x00000007aab00000)
 Metaspace       used 2626K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 285K, capacity 386K, committed 512K, reserved 1048576K
```

Without Escape Analysis

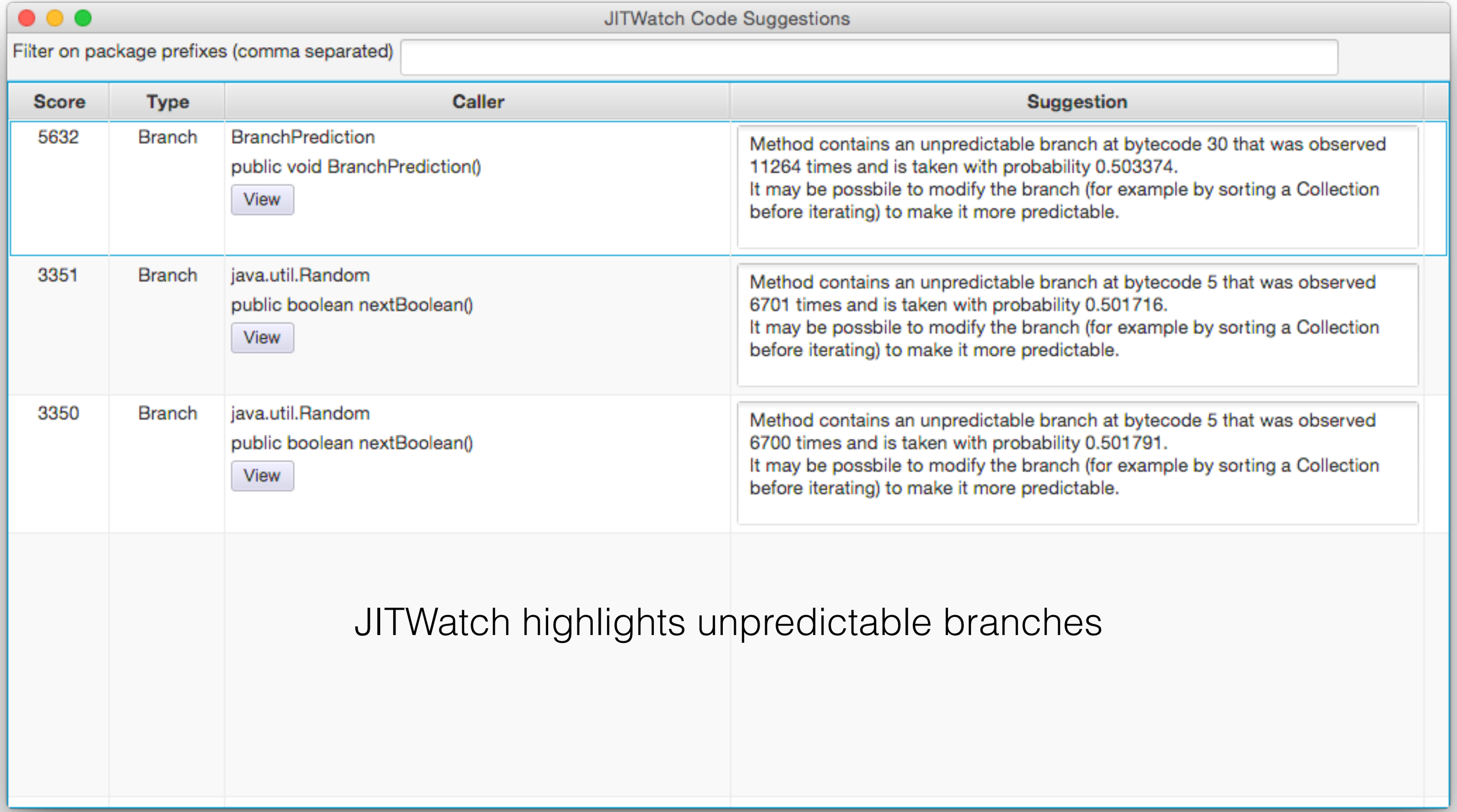
100m loops. 10 minor GCs

```
java  -Xms1G -Xmx1G -XX:+PrintGCDetails -verbose:gc -XX:-DoEscapeAnalysis EscapeTest
[GC (Allocation Failure) [PSYoungGen: 262144K->368K(305664K)] 262144K->376K(1005056K), 0.0006532 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 262512K->432K(305664K)] 262520K->440K(1005056K), 0.0006805 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 262576K->416K(305664K)] 262584K->424K(1005056K), 0.0005623 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 262560K->352K(305664K)] 262568K->360K(1005056K), 0.0006364 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 262496K->400K(305664K)] 262504K->408K(1005056K), 0.0005717 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 262544K->384K(348672K)] 262552K->392K(1048064K), 0.0007290 secs] [Times: user=0.00 sys=0.01, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 348544K->32K(348672K)] 348552K->352K(1048064K), 0.0006297 secs] [Times: user=0.00 sys=0.01, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 348192K->32K(347648K)] 348512K->352K(1047040K), 0.0004195 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 347168K->0K(348160K)] 347488K->320K(1047552K), 0.0004126 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 347136K->0K(348160K)] 347456K->320K(1047552K), 0.0004189 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
50001608
Heap
 PSYoungGen      total 348160K, used 180445K [0x00000007aab00000, 0x00000007c0000000, 0x00000007c0000000)
  eden space 347136K, 51% used [0x00000007aab00000,0x00000007b5b37438,0x00000007bfe00000)
  from space 1024K, 0% used [0x00000007bff00000,0x00000007bff00000,0x00000007c0000000)
  to   space 1024K, 0% used [0x00000007bfe00000,0x00000007bfe00000,0x00000007bff00000)
 ParOldGen       total 699392K, used 320K [0x0000000780000000, 0x00000007aab00000, 0x00000007aab00000)
  object space 699392K, 0% used [0x0000000780000000,0x0000000780050050,0x00000007aab00000)
 Metaspace       used 2626K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 285K, capacity 386K, committed 512K, reserved 1048576K
```

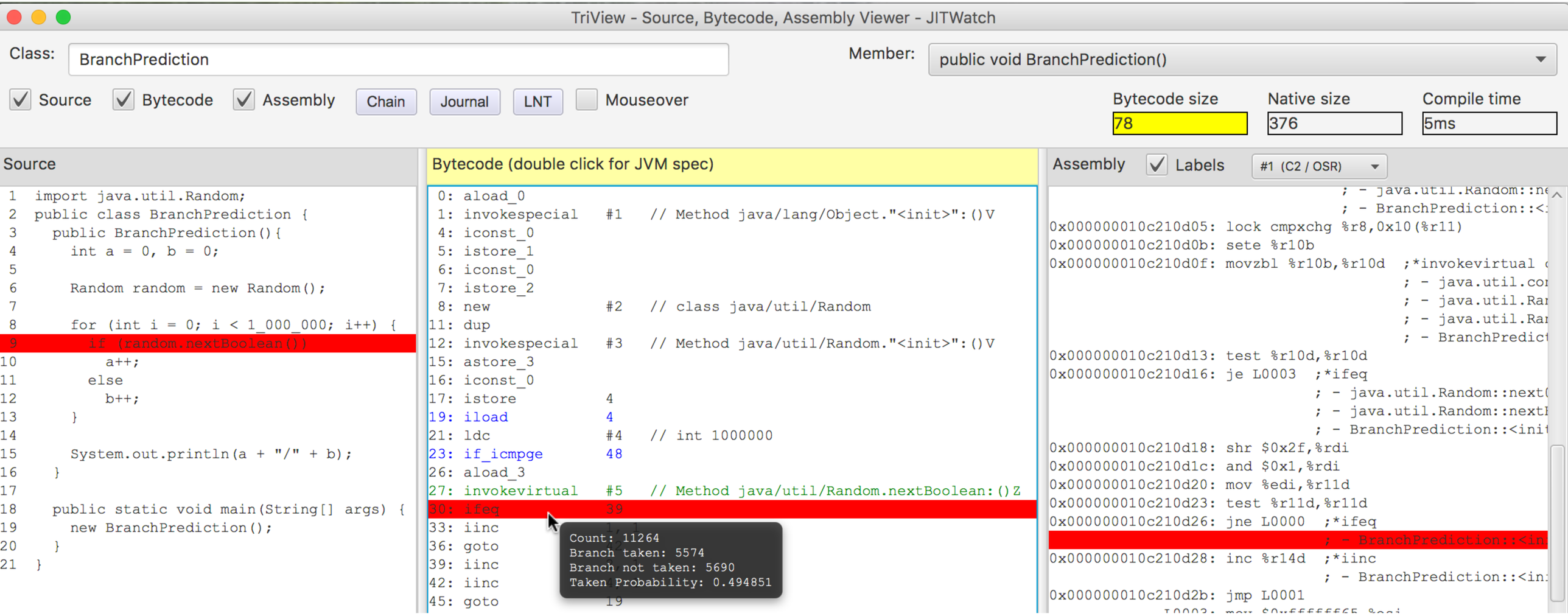

TriView - Source, Bytecode, Assembly Viewer - JITWatch
Class:
EscapeTest
Member:
public static int run()
Source
Bytecode
Assembly
Chain
Journal
LNT
Mouseover
Bytecode size
87
Native size
504
Compile time
10ms
Source
1 public class EscapeTest {
2 private final int val;
3
4 public EscapeTest(final int val) { this.val = val; }
5
6 public boolean equals(EscapeTest et) { return this.val == et.val; }
7
8 public static int run() {
9 int matches = 0;
10
11 java.util.Random random = new java.util.Random();
12
13 for (int i = 0; i < 100_000_000; i++) {
14 int v1 = random.nextBoolean() ? 1 : 0;
15 int v2 = random.nextBoolean() ? 1 : 0;
16
17 final EscapeTest e1 = new EscapeTest(v1);
18 final EscapeTest e2 = new EscapeTest(v2);
19
20 if (e1.equals(e2)) { matches++; }
21 }
22
23 return matches;
24 }
25
26 public static void main(final String[] args) {
27 System.out.println(run());
28 }
29 }
Bytecode (double click for JVM spec)
11: istore_2
12: iload_2
13: ldc #5 // int 100000000
15: if_icmpge 85
18: aload_1
19: invokevirtual #6 // Method java/util/Random.nextBoolean:()Z
22: ifeq 29
25: iconst_1
26: goto 30
29: iconst_0
30: istore_3
31: aload_1
32: invokevirtual #6 // Method java/util/Random.nextBoolean:()Z
35: ifeq 42
38: iconst_1
39: goto 43
42: iconst_0
43: istore 4
45: new #7 // class EscapeTest
48: dup
49: iload_3
50: invokespecial #8 // Method "<init>":(I)V
53: astore 5
55: new #7 // class EscapeTest
58: dup
59: iload 4
61: invokespecial #8 // Method "<init>":(I)V
64: astore 6
66: aload 5
68: aload 6
70: invokevirtual #9 // Method equals:(LEscapeTest;)Z
73: ifeq 79
76: iinc 0, 1
79: iinc 2, 1
82: goto 12
85: iload_0
86: ireturn
Class: EscapeTest
Method: equals
JIT Compiled: Yes
Inlined: Yes, inline (hot)
Count: 40960
iicount: 46336
Bytes: 17
Prof factor: 1
Ctrl-click to inspect this method
Backspace to return
Mounted class version: 52.0 (Java 8) public static int run() compiled with C2 OSR

Branch prediction

```
public class BranchPrediction {
  public BranchPrediction(){
    int a = 0, b = 0;

    Random random = new Random();

    for (int i = 0; i < 1_000_000; i++) {
      if (random.nextBoolean())
        a++;
      else
        b++;
    }

    System.out.println(a + "/" + b);
  }

  public static void main(String[] args) {
    new BranchPrediction();
  }
}
```


JITWatch Code Suggestions
Filter on package prefixes (comma separated)
Score
Type
Caller
Suggestion
5632
Branch
BranchPrediction
public void BranchPrediction()
View
Method contains an unpredictable branch at bytecode 30 that was observed 11264 times and is taken with probability 0.503374.
It may be possbile to modify the branch (for example by sorting a Collection before iterating) to make it more predictable.
3351
Branch
java.util.Random
public boolean nextBoolean()
View
Method contains an unpredictable branch at bytecode 5 that was observed 6701 times and is taken with probability 0.501716.
It may be possbile to modify the branch (for example by sorting a Collection before iterating) to make it more predictable.
3350
Branch
java.util.Random
public boolean nextBoolean()
View
Method contains an unpredictable branch at bytecode 5 that was observed 6700 times and is taken with probability 0.501791.
It may be possbile to modify the branch (for example by sorting a Collection before iterating) to make it more predictable.
JITWatch highlights unpredictable branches

Branch prediction



Intrinsics

Highly optimised native implementations

Use features of target CPU

Intrinsics exist for methods in
`Math, Unsafe, System, Class, Arrays, String, StringBuilder, AESCrypt,` …

393 intrinsics as of JDK20

https://chriswhocodes.com/vm-intrinsics-explorer.html

Intrinsics

`Math.log10(double)` is 2 instructions on x86_64

from: hotspot/src/cpu/x86/vm/x86_64.ad

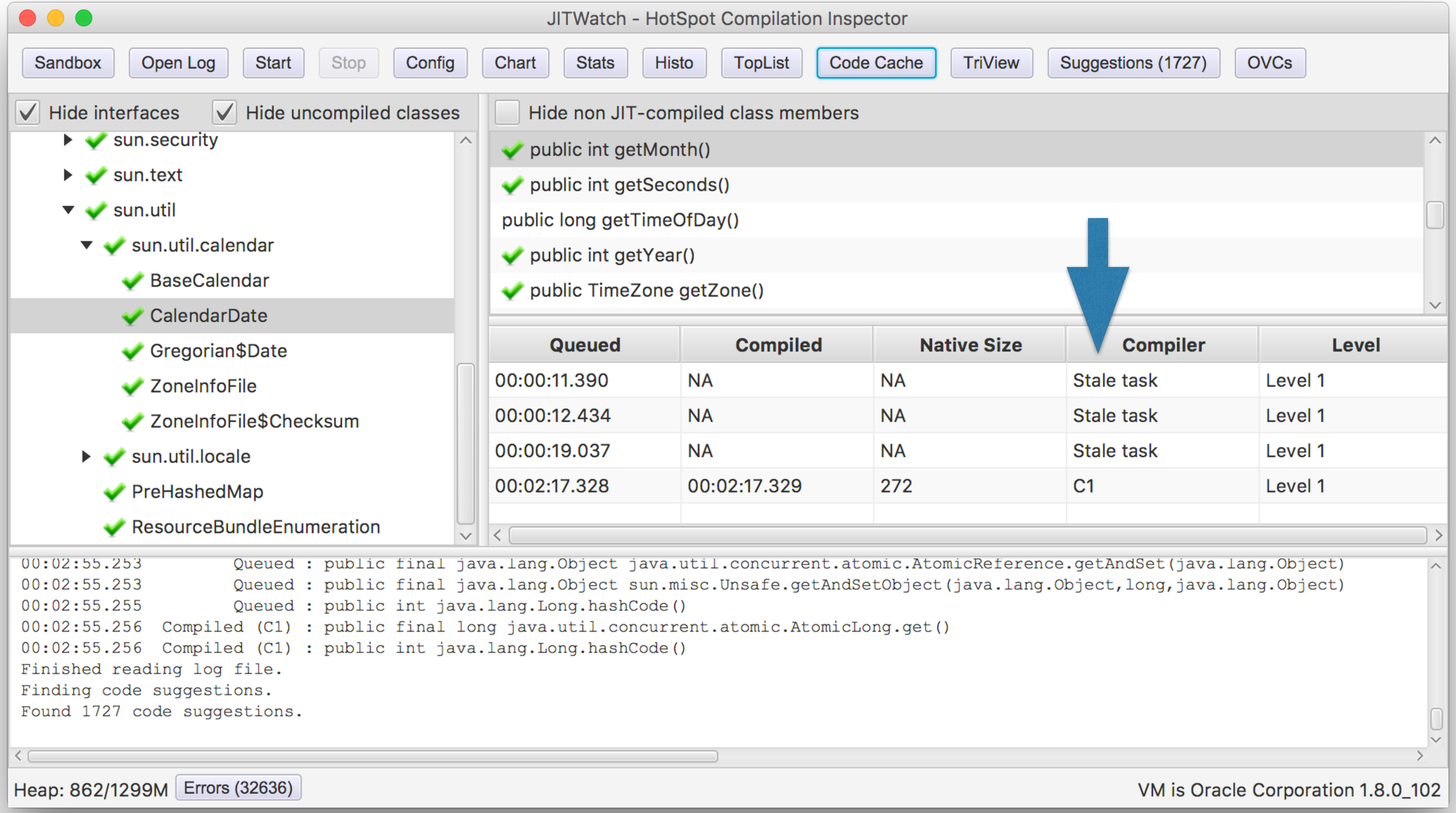
```
instruct log10D_reg(regD dst) %{
  // The source and result Double operands in XMM registers match(Set dst (Log10D dst));
  // fldlg2       ; push log_10(2) on the FPU stack; full 80-bit number
  // fyl2x        ; compute log_10(2) * log_2(x)
  format %{ "fldlg2\t\t\t#Log10\n\t"
            "fyl2x\t\t\t# Q=Log10*Log_2(x)\n\t"
         %}
   ins_encode(Opcode(0xD9), Opcode(0xEC),   // fldlg2
              Push_SrcXD(dst),
              Opcode(0xD9), Opcode(0xF1),   // fyl2x
              Push_ResultXD(dst));

  ins_pipe( pipe_slow );
%}
```

JITWatch TopLists

Most-used Intrinsics

Count	Intrinsic
64	java.lang.System.arraycopy => _arraycopy
45	java.lang.Math.min => _min
25	java.lang.Object.hashCode => _hashCode
23	java.lang.Object.getClass => _getClass
22	java.lang.ref.Reference.get => _Reference_get
18	java.lang.String.equals => _equals
10	java.util.Arrays.copyOf => _copyOf
9	sun.misc.Unsafe.compareAndSwapObject => _compareAndSwapObject
8	java.lang.Object.clone => _clone
6	java.lang.Thread.currentThread => _currentThread
5	java.lang.Math.max => _max
3	sun.reflect.Reflection.getClassAccessFlags => _getClassAccessFlags
3	java.lang.Class.getComponentType => _getComponentType
3	sun.misc.Unsafe.compareAndSwapInt => _compareAndSwapInt
3	java.util.zip.CRC32.updateBytes => _updateBytesCRC32
3	java.lang.String.compareTo => _compareTo
3	sun.misc.Unsafe.getObjectVolatile => _getObjectVolatile
3	java.lang.reflect.Array.newArray => _newArray
2	java.lang.System.identityHashCode => _identityHashCode
2	sun.misc.Unsafe.putObject => _putObject
2	java.lang.System.nanoTime => _nanoTime

Stale Tasks



In the compile queue for >50ms without further invocations / back edges

Vectorisation

```
public void incrementArray(int[] array, int const)
{
    int length = array.length;

    for (int i = 0; i < length; i++)
    {
        array[i] += constant;
    }
}
```

Bytecode for incrementArray

```
 0: aload_1                  // load the reference of 'array'
 1: arraylength              // call the 'arraylength' instruction to get the length of the array
 2: istore_3                 // store the array length into local variable 3 'length'
 3: iconst_0                 // push int 0 onto the stack
 4: istore           4       // store into local variable 4 'i'
 6: iload            4       // load local variable 4 'i' and push onto the stack
 8: iload_3                  // load local variable 3 'length' and push onto the stack
 9: if_icmpge        26      // if (i >= length) jump to BCI 26 (return)
12: aload_1                  // else load the reference of 'array' and push onto the stack
13: iload            4       // load local variable 4 'i' and push onto the stack
15: dup2                     // duplicate the top 2 values on the stack
16: iaload                   // load the value of array[i] and push onto the stack
17: iload_2                  // load local variable 2 'constant' and push onto the stack
18: iadd                     // add array[i] and 'constant' and push result onto stack
19: iastore                  // store the result back into array[i]
20: iinc             4, 1    // increment local variable 4 'i' by 1
23: goto             6       // jump back to BCI 6
26: return
```

```
0x00007f92f523b93f: vmovd xmm0,ecx
0x00007f92f523b943: vpbroadcastd ymm0,xmm0
0x00007f92f523b948: inc eax
0x00007f92f523b94a: mov r8d,0xfa00
            L0002: mov edi,r10d
0x00007f92f523b953: sub edi,eax
0x00007f92f523b955: cmp r10d,eax
0x00007f92f523b958: cmovl edi,esi
0x00007f92f523b95b: cmp edi,0xfa00
0x00007f92f523b961: cmova edi,r8d
0x00007f92f523b965: add edi,eax
0x00007f92f523b967: nop WORD PTR [rax+rax*1+0x0]  ;*aload_1 {reexecute=0 rethrow=0 return_oop=0}
                                                  ;   DoesItVectorise::incrementArray@12 (line 24)
            L0003: vpaddd ymm1,ymm0,YMMWORD PTR [rdx+rax*4+0x10]
0x00007f92f523b976: vmovdqu YMMWORD PTR [rdx+rax*4+0x10],ymm1
0x00007f92f523b97c: vpaddd ymm1,ymm0,YMMWORD PTR [rdx+rax*4+0x30]
0x00007f92f523b982: vmovdqu YMMWORD PTR [rdx+rax*4+0x30],ymm1
0x00007f92f523b988: vpaddd ymm1,ymm0,YMMWORD PTR [rdx+rax*4+0x50]
0x00007f92f523b98e: vmovdqu YMMWORD PTR [rdx+rax*4+0x50],ymm1
0x00007f92f523b994: vpaddd ymm1,ymm0,YMMWORD PTR [rdx+rax*4+0x70]
0x00007f92f523b99a: vmovdqu YMMWORD PTR [rdx+rax*4+0x70],ymm1
0x00007f92f523b9a0: vpaddd ymm1,ymm0,YMMWORD PTR [rdx+rax*4+0x90]
0x00007f92f523b9a9: vmovdqu YMMWORD PTR [rdx+rax*4+0x90],ymm1
0x00007f92f523b9b2: vpaddd ymm1,ymm0,YMMWORD PTR [rdx+rax*4+0xb0]
0x00007f92f523b9bb: vmovdqu YMMWORD PTR [rdx+rax*4+0xb0],ymm1
0x00007f92f523b9c4: vpaddd ymm1,ymm0,YMMWORD PTR [rdx+rax*4+0xd0]
0x00007f92f523b9cd: vmovdqu YMMWORD PTR [rdx+rax*4+0xd0],ymm1
0x00007f92f523b9d6: vpaddd ymm1,ymm0,YMMWORD PTR [rdx+rax*4+0xf0]
0x00007f92f523b9df: vmovdqu YMMWORD PTR [rdx+rax*4+0xf0],ymm1  ;*iastore {reexecute=0 rethrow=0 return_oop=0}
                                                               ;   DoesItVectorise::incrementArray@19 (line 24)
0x00007f92f523b9e8: add eax,0x40  ;*iinc {reexecute=0 rethrow=0 return_oop=0}
                                  ; - DoesItVectorise::incrementArray@20 (line 22)
0x00007f92f523b9eb: cmp eax,edi
0x00007f92f523b9ed: jl L0003  ;*goto {reexecute=0 rethrow=0 return_oop=0}
                              ; - DoesItVectorise::incrementArray@23 (line 22)
```

Broadcast (copy) the 32-bit int constant 8 times across 256-bit SIMD register ymm0

Packed integer addition in 256-bit SIMD registers using vpaddd/vmovdqu instructions and loop unrolling updates 64 array elements per loop

TL;DR

JIT logs can reveal optimisation issues

Keep methods small for inlining (Head Test)

Check inline-ability of 3rd party library methods

Check for unpredictable branches

Use appropriate method visibility (CHA)

Count interface implementations

Check for allocations in hot code (EA)

Epilogue

We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%.

Donald Knuth, Computer Programming as an Art

Thanks for listening!

- JITWatch on GitHub
- https://github.com/AdoptOpenJDK/jitwatch

- VMOptionsExplorer, Intrinsics Explorer, JaCoLine, JEPSearch, …
- https://chriswhocodes.com

- @chriswhocodes on Twitter