Understanding HotSpot JVM Performance with JITWatch

Chris Newland, LJC (Docklands) 2015-11-10

Bio

Chris Newland

ADVFN / Throgmorton Street Capital

Market data / FIX / trading systems

@chriswhocodes on Twitter

Play along at home!

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

```
mvn clean install exec:java
```

Caveat Emptor

Measure, don’t guess

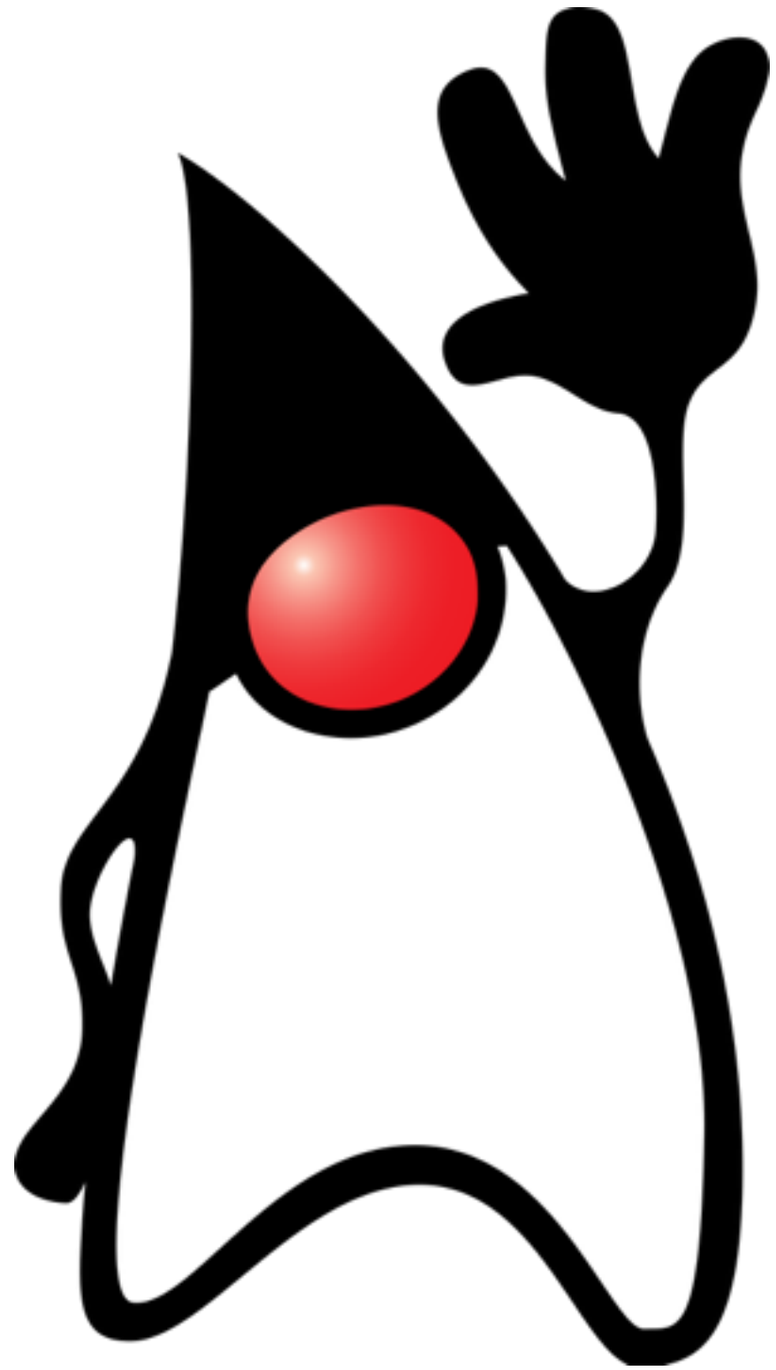
Stay SOLID

90%+ does not need optimising!

Premature optimization is the root of all evil

Donald Knuth

The Amazing JVM



50+ Languages

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

Object-Oriented and Functional

Strongly and Dynamically Typed

On Many Platforms

x86, ARM, SPARC, PPC, MIPS, Vega, …

Linux, Windows, Solaris, OSX, AIX, …

Single CPU to 1000s of cores

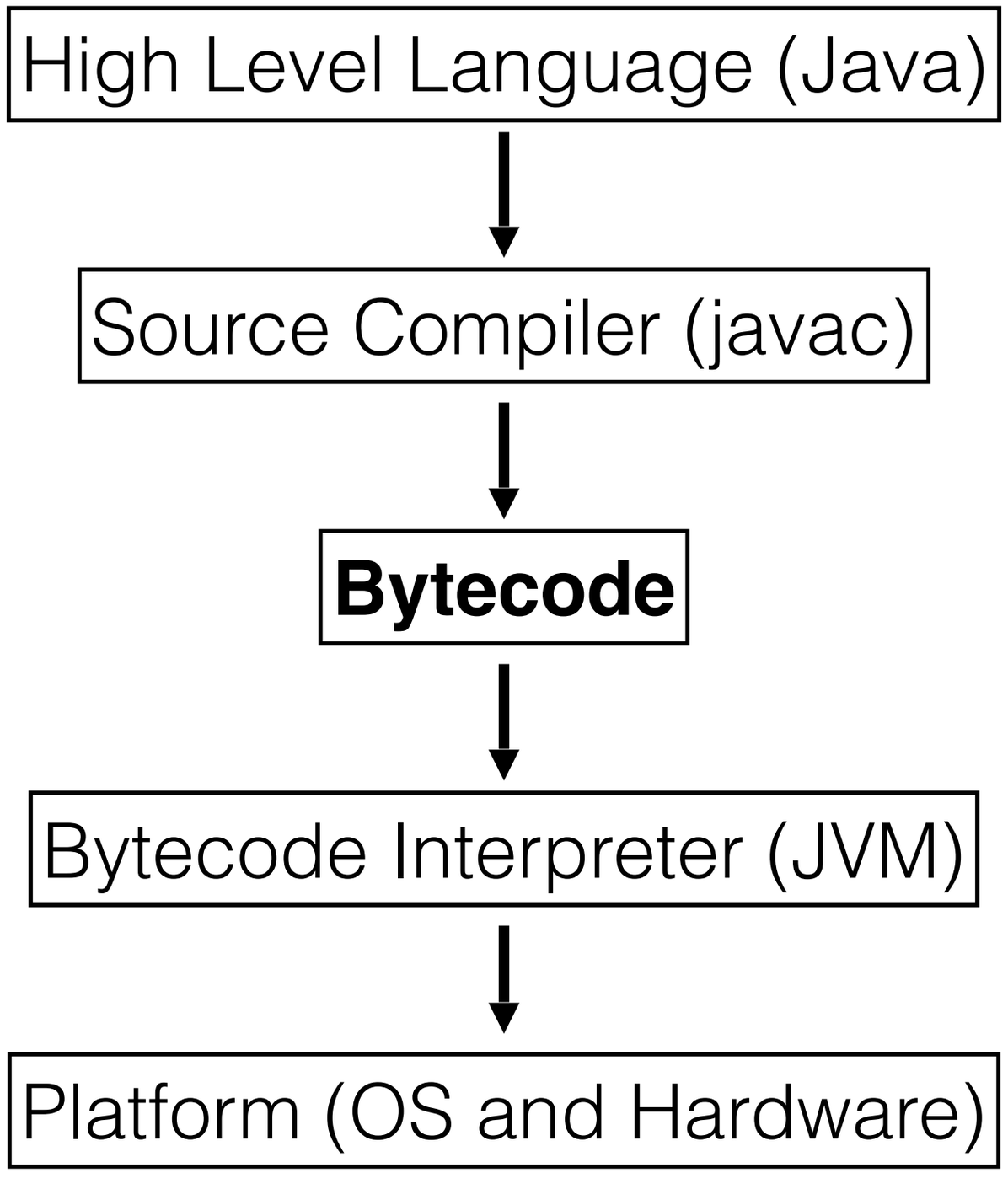
Tiny SoC to TeraBytes of RAM

How does the JVM support such diversity?

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


High Level Language (Java)
Source Compiler (javac)
Bytecode
Bytecode Interpreter (JVM)
Platform (OS and Hardware)

Bytecode

Portable instruction set

All opcodes represented in 1 byte

Executed on a **virtual stack machine**

javac compiler

javac

```
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
```

Virtual Stack Machine?

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

    switch(opcode)
    {
    case 00:
        // stuff
        break;
    case 01:
        // stuff
        break;

    ...

    case ff:
        // stuff
        break;
    }
}
```

Types of Compiler

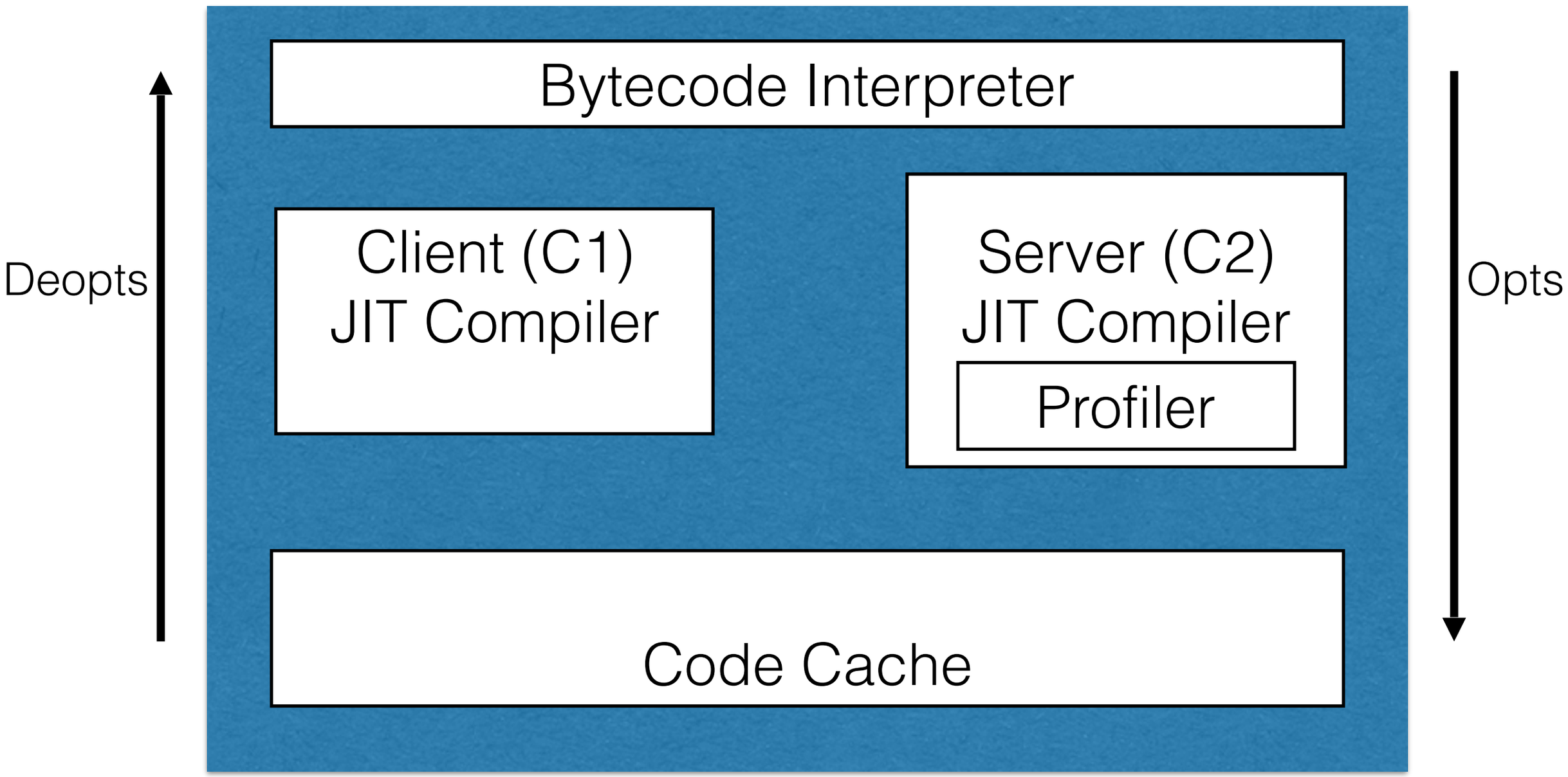
Ahead of Time (AOT)

- Produces native code
- Knowledge of target architecture
- Full performance from the start

Just In Time (JIT)

- Profiles running code
- Adaptive optimisations
- Takes time to build a profile

The HotSpot JVM



Talking JIT

Client compiler (C1)

- Starts quickly, simple optimisations

Server compiler (C2)

- Profile guided, waits for more information
- Advanced optimisations

Tiered Compilation (C1 + C2)

- Default in Java 8
- Enable in Java 7 with **`-XX:+TieredCompilation`**

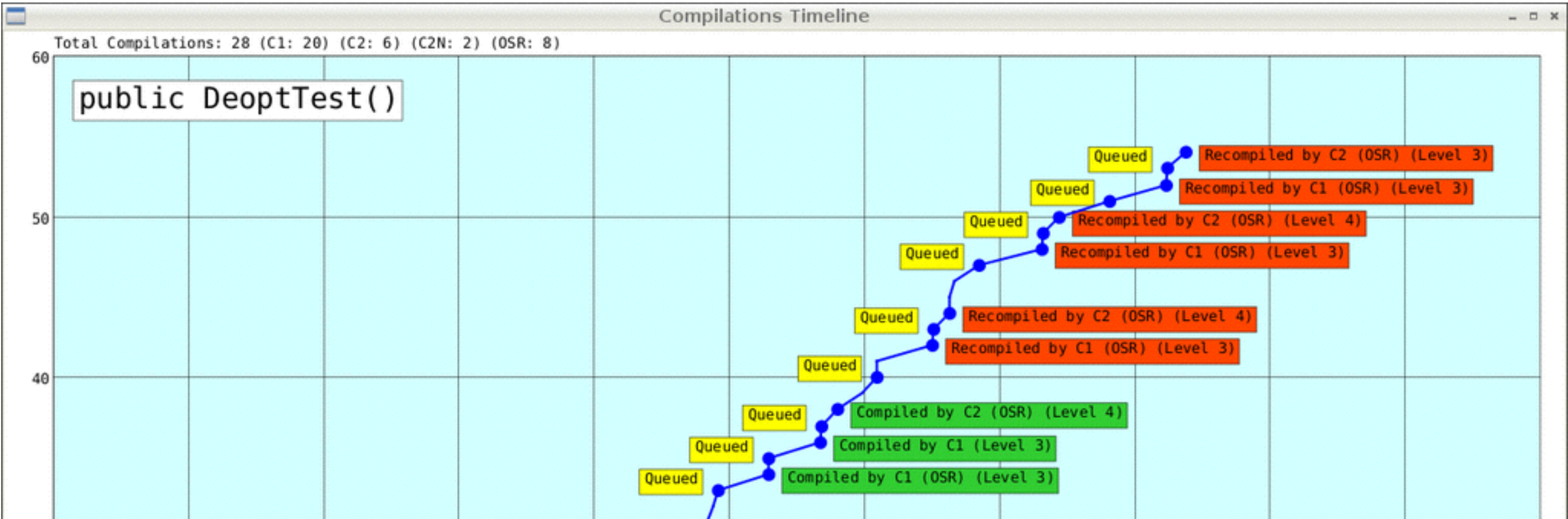
Guess Again?

Most server (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

The Code Cache

JVM region for JIT-compiled methods

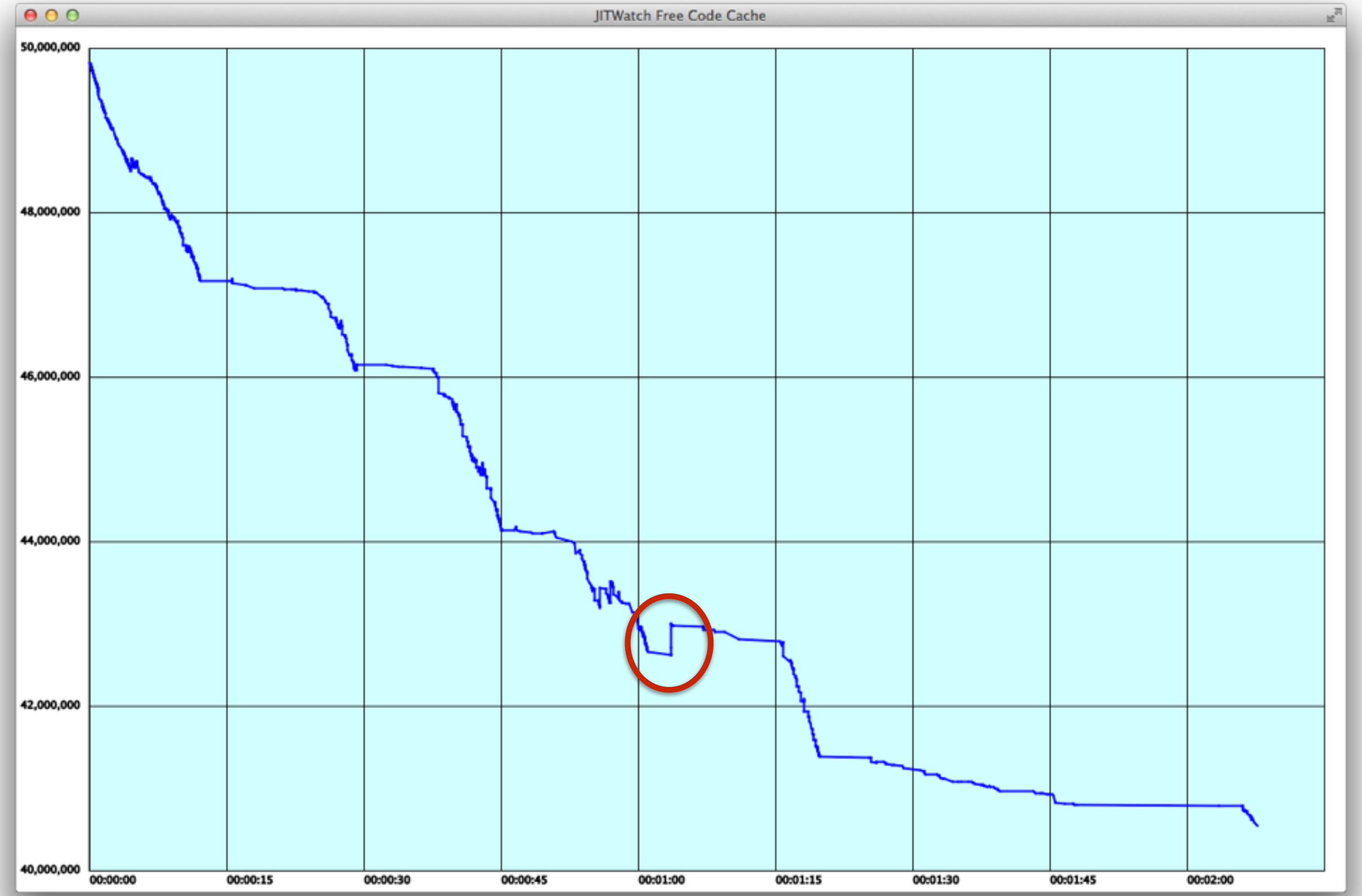
Can run out of space! (*TieredCompilation*?)

Can get fragmented!

Removals (deopts / class unloading)

`-XX:ReservedCodeCacheSize=<size>m`

Code Cache


JITWatch Free Code Cache
50,000,000
48,000,000
46,000,000
44,000,000
42,000,000
40,000,000
00:00:00
00:00:15
00:00:30
00:00:45
00:01:00
00:01:15
00:01:30
00:01:45
00:02:00

I Know Kung-Fu!

HotSpot JIT compilers know ~100 optimisations

lock coarsening

strength reduction

loop unrolling

branch prediction

range check elimination

inlining

dead code elimination

CHA

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

Show Me

-XX:+UnlockDiagnosticVMOptions

-XX:+LogCompilation

-XX:+TraceClassLoading

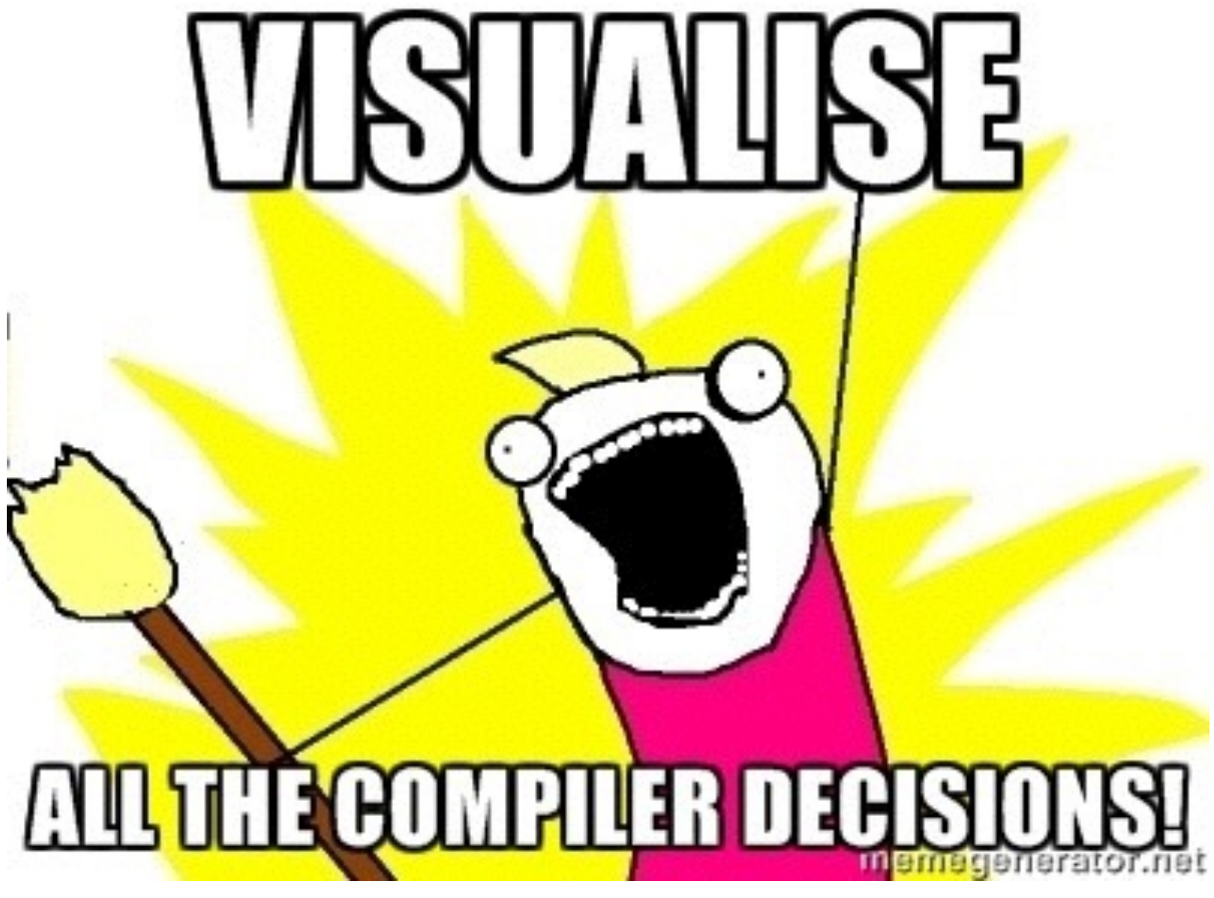
-XX:+PrintAssembly

hsdis binary in jre/lib/amd64/server

Significant performance overhead

I heard you like to grep?

Program	Size of HotSpot log
Microbenchmark	A few hundred KB
Desktop application	A few MB
Large application with assembly	50MB+



JITWatch

https://github.com/AdoptOpenJDK/jitwatch/

Compilations (when, how)

Deoptimisations (why)

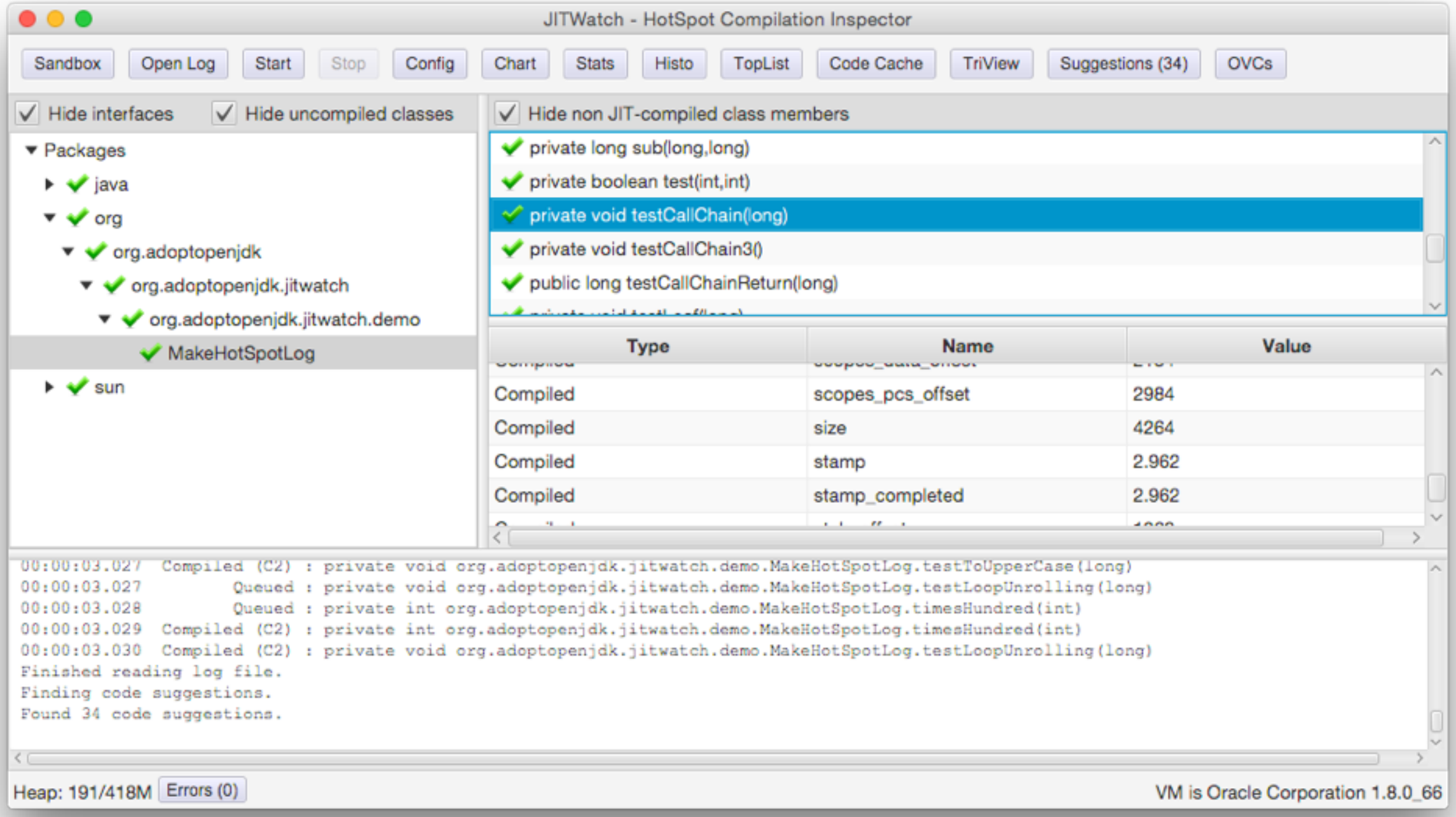
Inlining successes and failures

Escape analysis (heap alloc, lock elision)

Branch probabilities

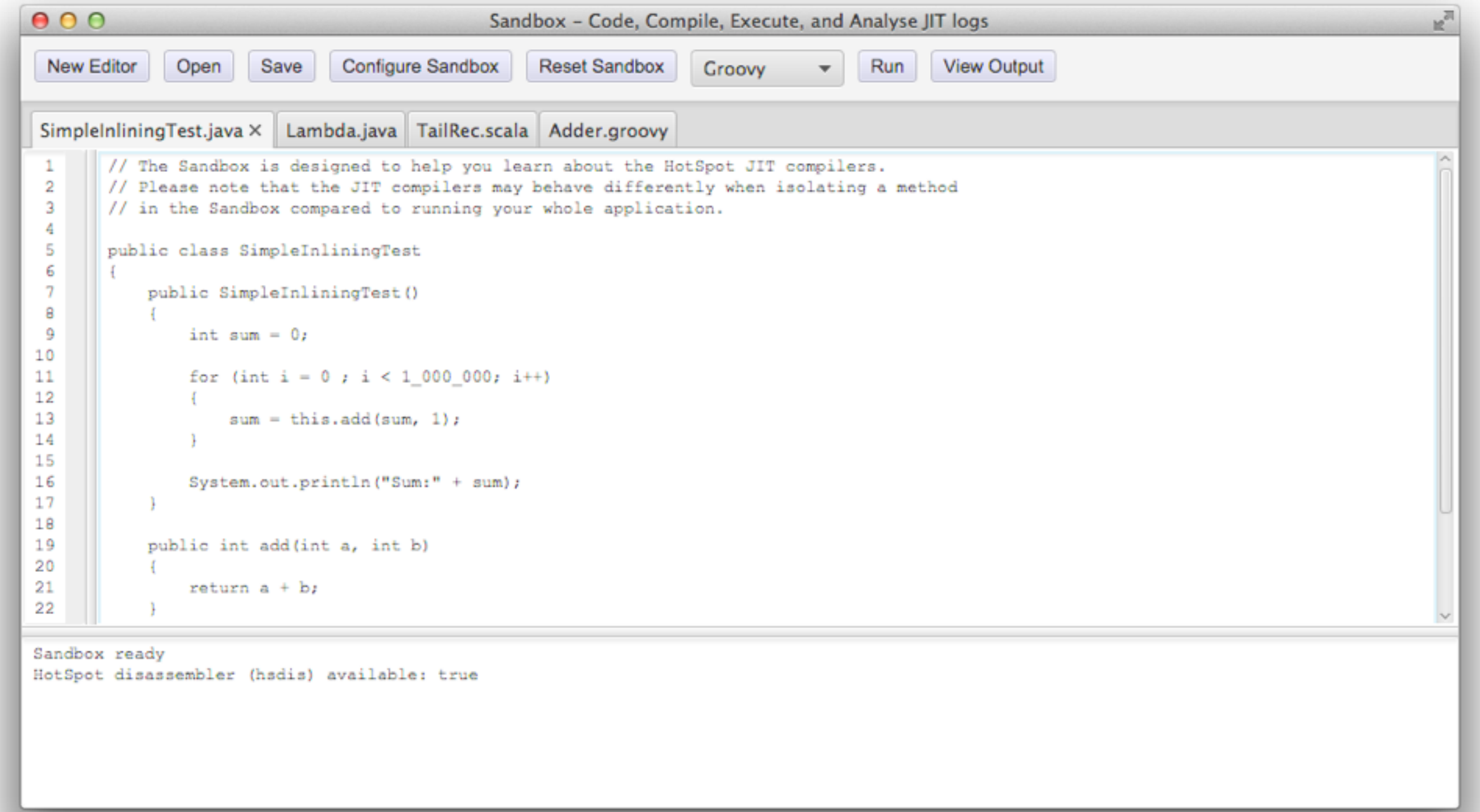
Intrinsics used

Getting Started



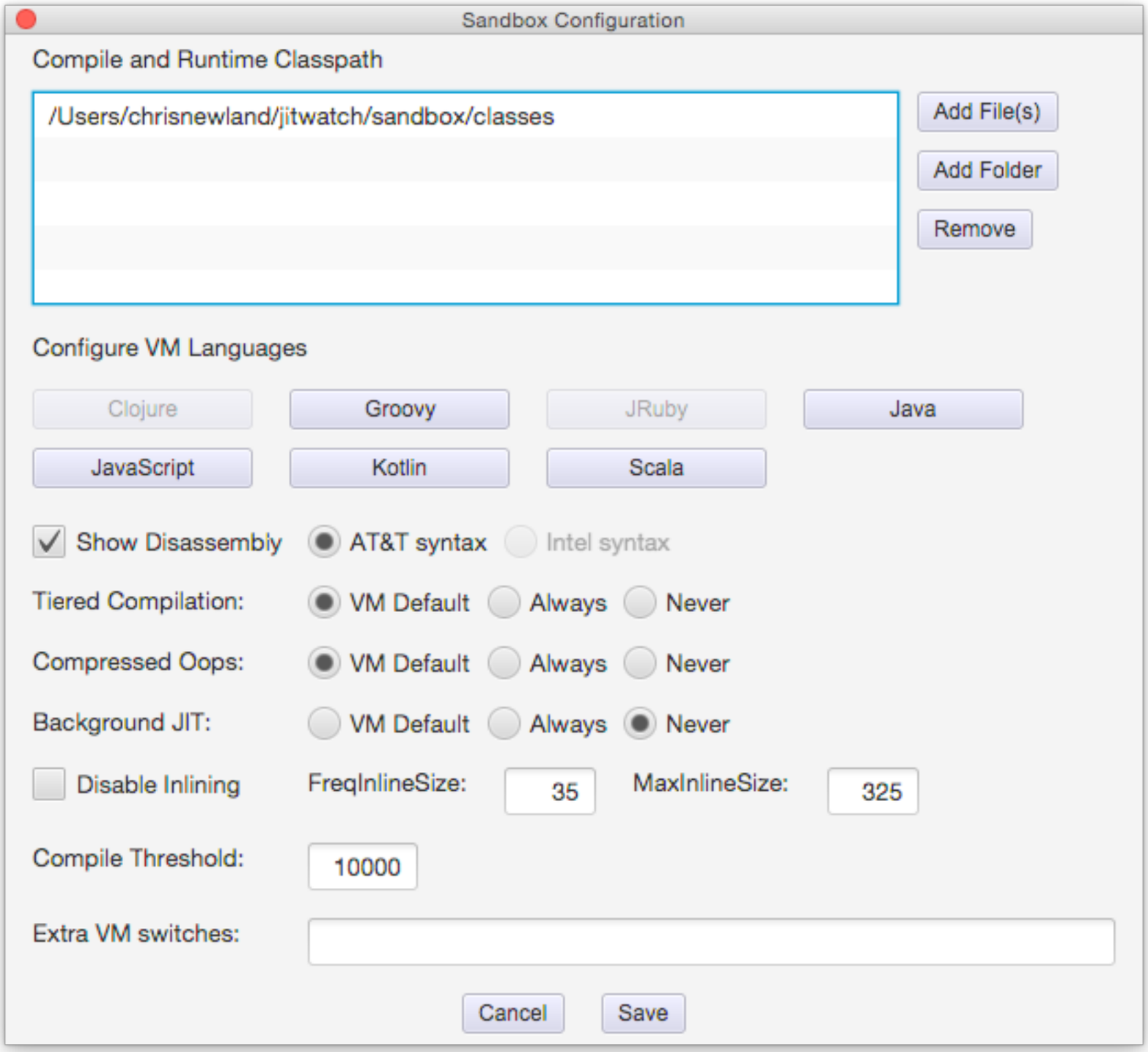
Build with maven, ant, or gradle

Sandbox Mode

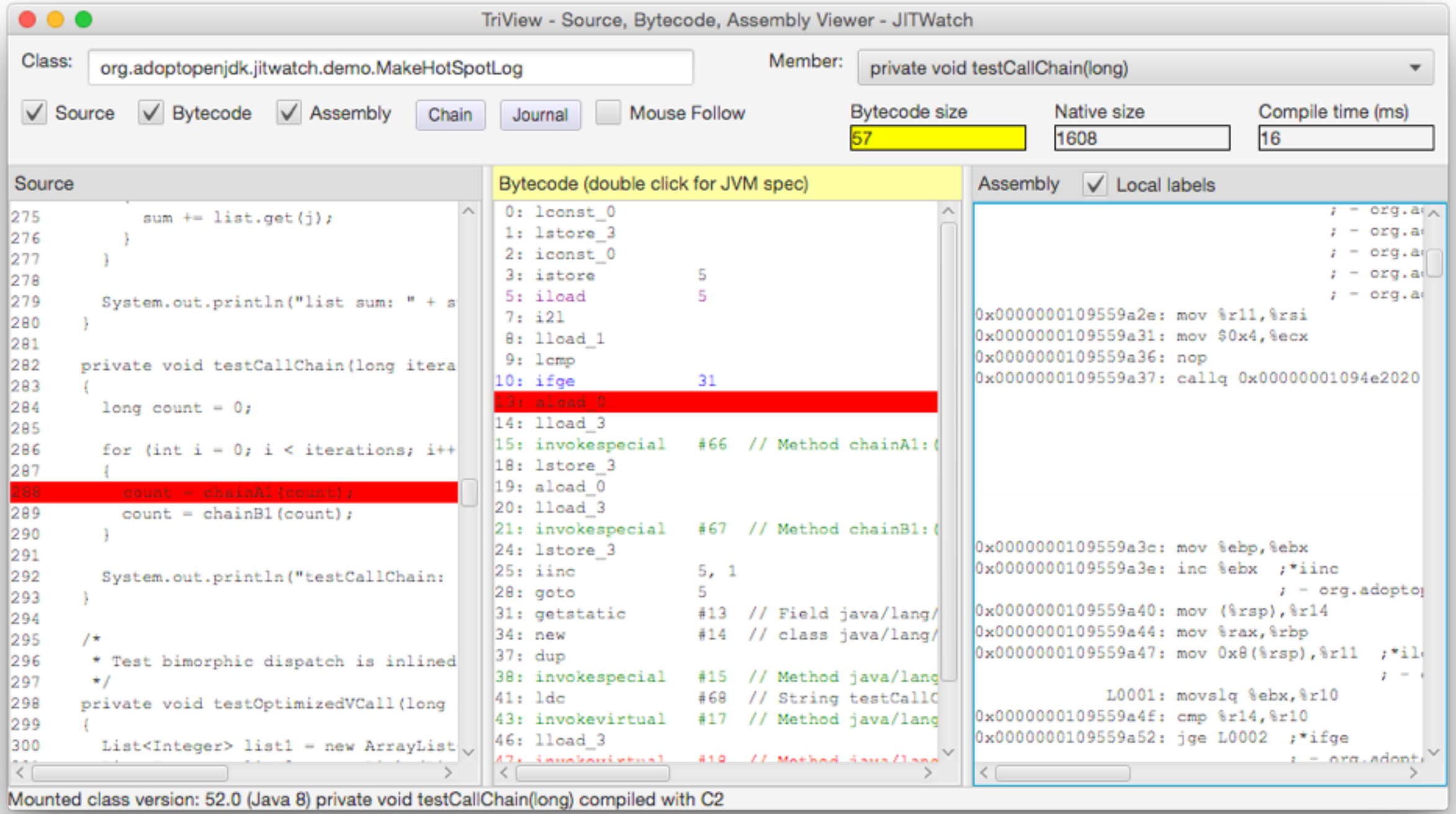


Examples that exercise JIT behaviours

Sandbox Config



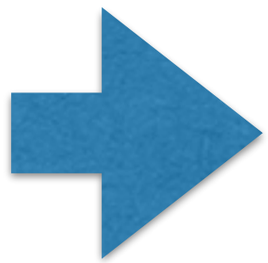
TriView Screen



Inlining, traps, branches, escape analysis

Inlining

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


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



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

Copy the body of the callee method into the call site

Eliminates the cost of method dispatch

Opens the door to other optimisations

Inlining Limits

Increases size of compiled code

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

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

Inlining Failure Modes

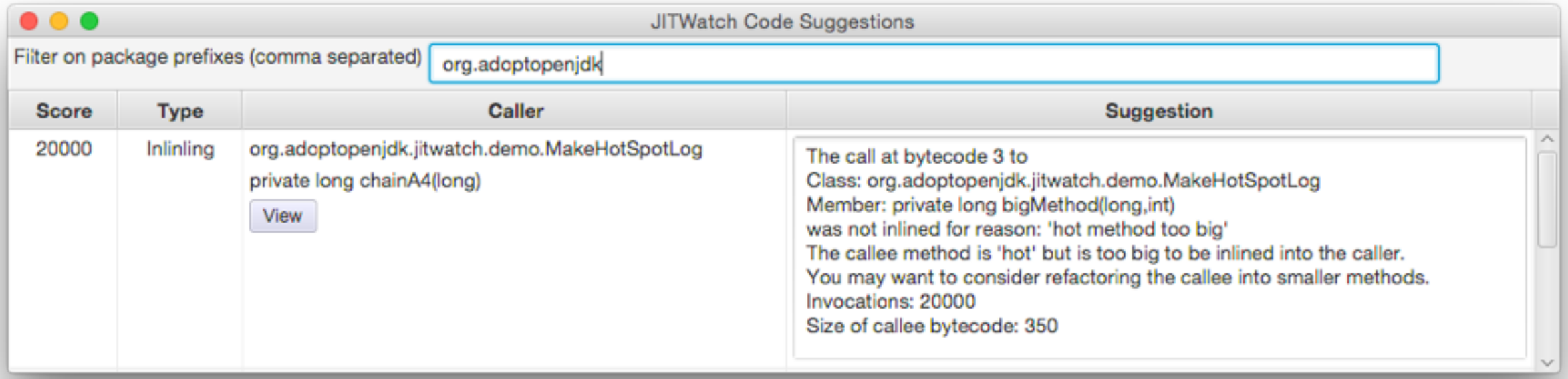
JITWatch TopLists

Inlining Failure Reasons

Count	Reason
1647	too big
473	executed < MinInliningThreshold times
433	already compiled into a medium method
357	already compiled into a big method
271	call site not reached
257	native method
216	never executed
166	hot method too big
71	size > DesiredMethodLimit
33	recursive inlining is too deep
18	unloaded signature classes
11	exception method
8	inlining too deep
1	NodeCountInliningCutoff

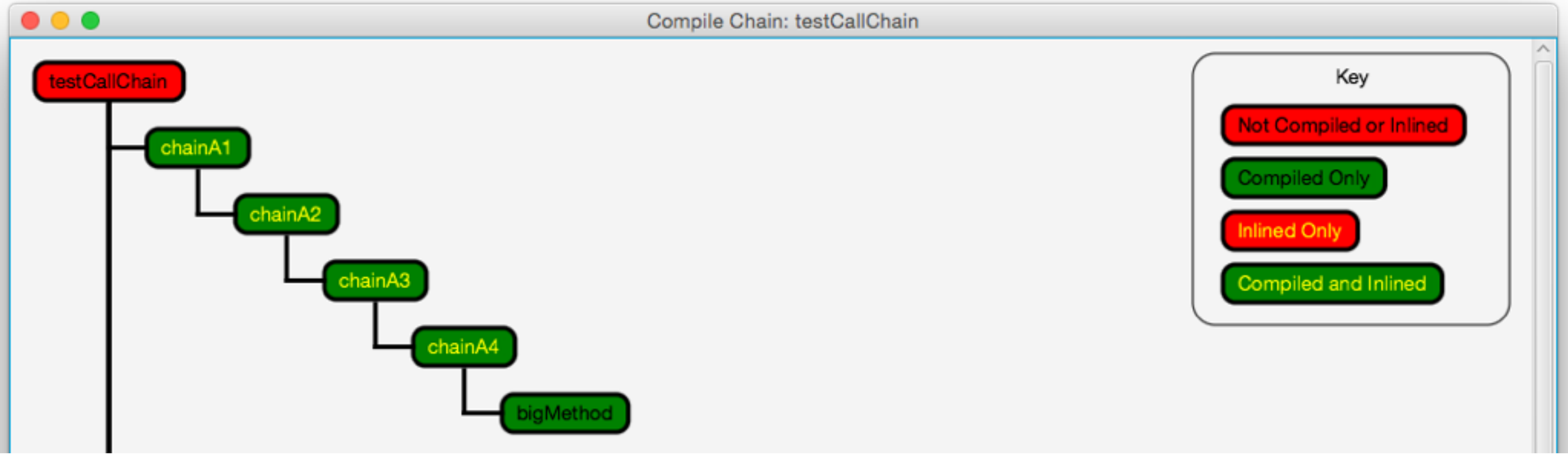
BAD!

Inlining Suggestions



Filter by package to exclude 3rd party and core libs

Call Chain



Look out for inlining failures in the call chain

Watch for overly deep call chains in hot code

JarScan Tool

- Static analysis tool, maven plugin
- Identifies methods > inlining threshold
- 3,613 above-threshold methods in 8u66 rt.jar
  - String.split
  - **String.toUpperCase / toLowerCase**
  - Parts of java.util.ComparableTimSort

```
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 {
```

String.toUpperCase()

439 bytes of bytecode

char[] can change size

Too big for inlining

ASCII Optimised

```
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);
}
```

68 bytes of bytecode

Can be inlined

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 as fast

Assertions

Disabled by default (enable with **`-ea`**)

Core-lib assertion code baked into bytecode

Counted in inlining budget

Can push a method over the inlining limit!

j.u.ComparableTimSort

Method	Size with assertions	Size without assertions	Saving
gallopLeft	327	244	25.4%
gallopRight	327	244	25.4%
mergeLo	652	517	18.6%
mergeHi	716	583	20.7%

Possible to create an rt.jar without assertions using OpenJDK

Modify javac to suppress assertion bytecode generation!

Callsite Polymorphism

HotSpot tracks observed implementations at each callsite.

Too many implementations can prevent inlining.

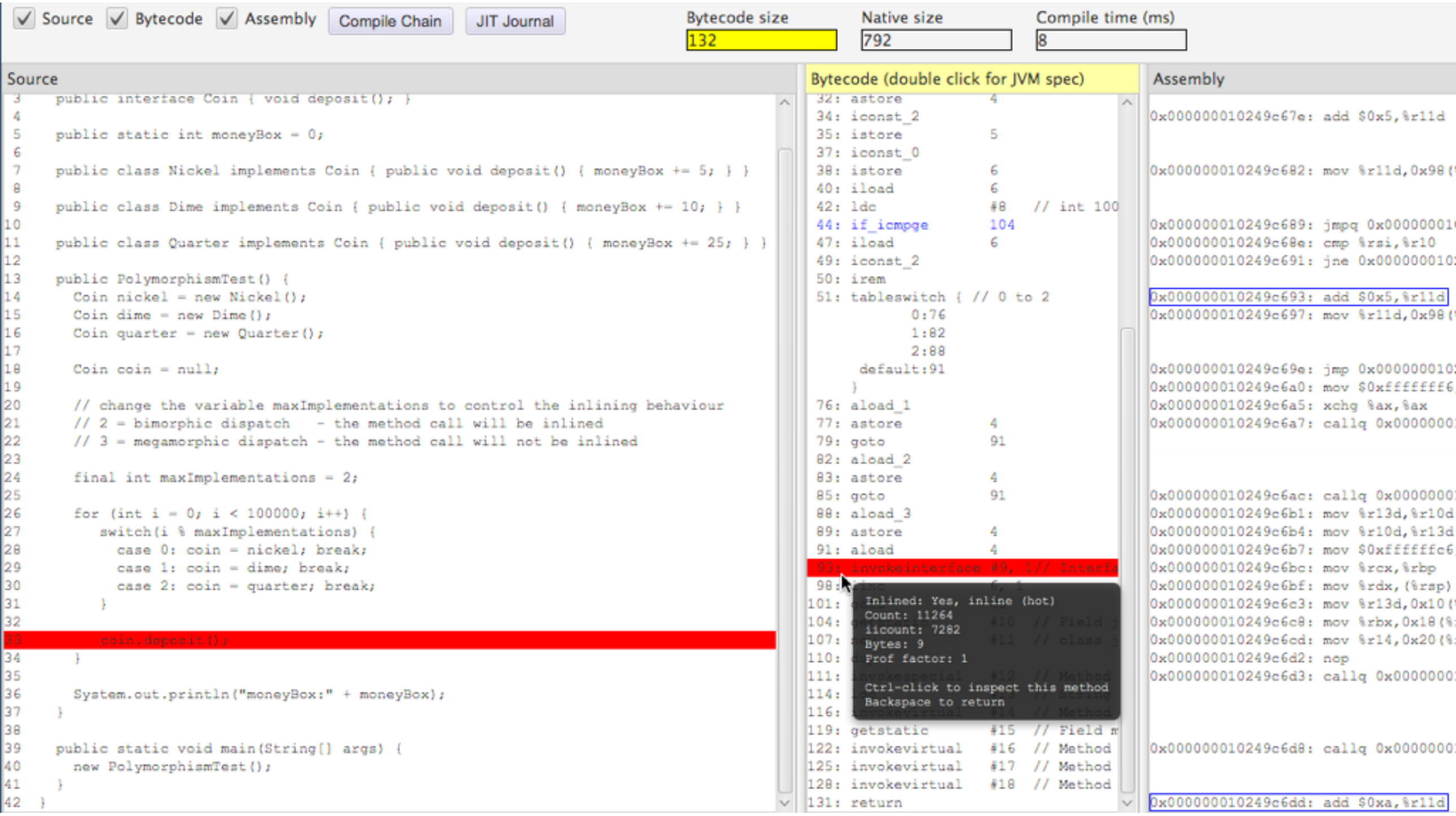
Implementations	Classification	Inlinable?
1	Monomorphic	Yes
2	Bimorphic	Yes
3+	Megamorphic	No*

`-XX:TypeProfileMajorReceiverPercent=90`

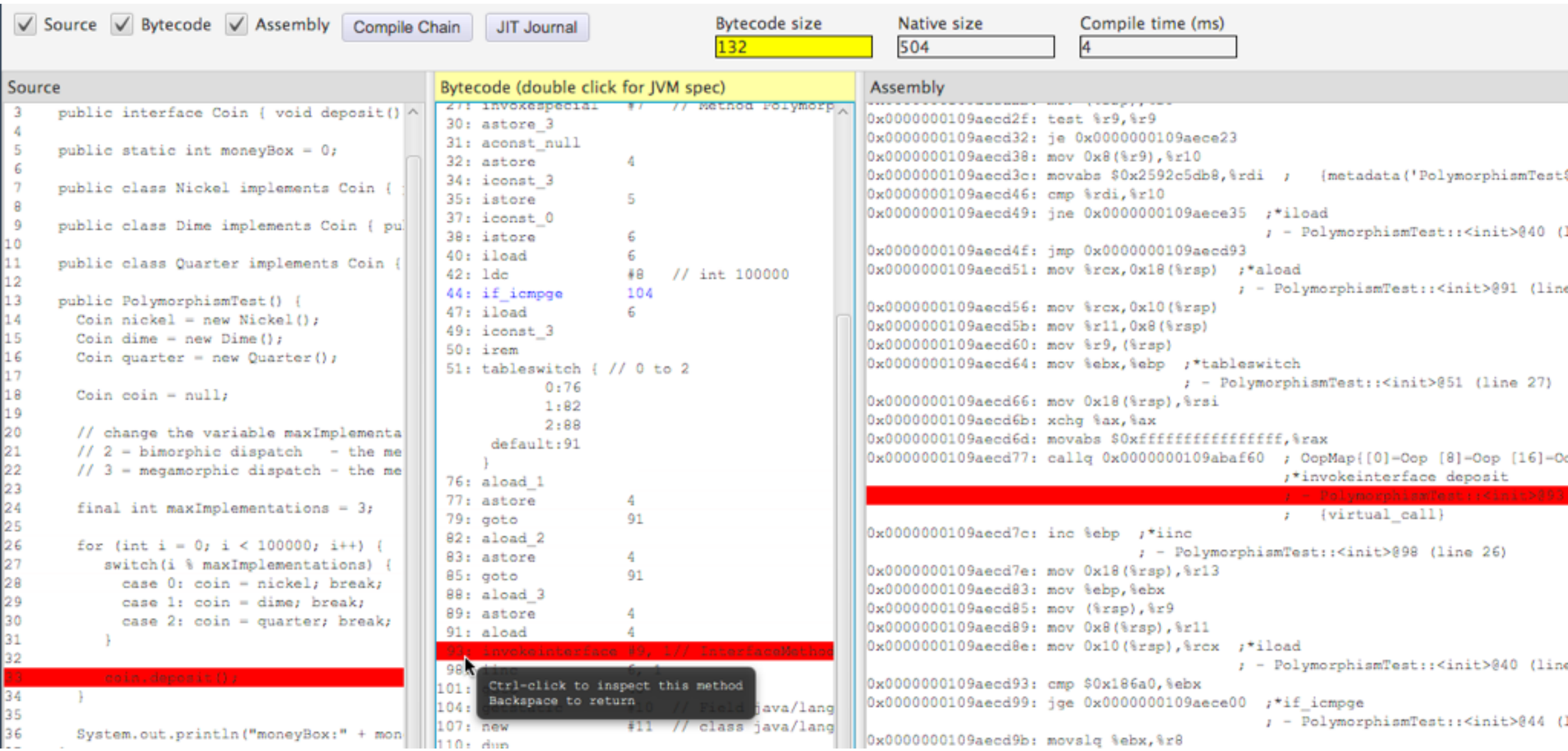
```
public class PolymorphismTest
{
    public interface Coin { void deposit(); }

    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;

        // 1 = monomorphic dispatch - the method call will be inlined
        // 2 = bimorphic dispatch   - the method call will be inlined
        // 3 = megamorphic dispatch - the method call will not be inlined
        final int maxImplementations = 3;

        for (int i = 0; i < 100000; i++) {
            switch(i % maxImplementations) {
                case 0: coin = nickel;  break;
                case 1: coin = dime;    break;
                case 2: coin = quarter; break;
            }

            coin.deposit();
        }

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

Implementation calls are balanced so megamorphic callsite will not be inlined.

Bimorphic


Source
Bytecode
Assembly
Compile Chain
JIT Journal
Bytecode size
132
Native size
792
Compile time (ms)
8
Source
Bytecode (double click for JVM spec)
Assembly
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;
// change the variable maxImplementations to control the inlining behaviour
// 2 = bimorphic dispatch - the method call will be inlined
// 3 = megamorphic dispatch - the method call will not be inlined
final int maxImplementations = 2;
for (int i = 0; i < 100000; i++) {
switch(i % maxImplementations) {
case 0: coin = nickel; break;
case 1: coin = dime; break;
case 2: coin = quarter; break;
}
coin.deposit();
}
System.out.println("moneyBox:" + moneyBox);
}
public static void main(String[] args) {
new PolymorphismTest();
}
}
32: astore 4
34: iconst_2
35: istore 5
37: iconst_0
38: istore 6
40: iload 6
42: ldc #8 // int 100
44: if_icmpge 104
47: iload 6
49: iconst_2
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// Interfa
98: iinc 6, 1
Inlined: Yes, inline (hot)
Count: 11264
iicount: 7282
Bytes: 9
Prof factor: 1
Ctrl-click to inspect this method
Backspace to return
119: getstatic #15 // Field
122: invokevirtual #16 // Method
125: invokevirtual #17 // Method
128: invokevirtual #18 // Method
131: return
0x000000010249c67e: add $0x5,%r11d
0x000000010249c682: mov %r11d,0x98(
0x000000010249c689: jmpq 0x00000001
0x000000010249c68e: cmp %rsi,%r10
0x000000010249c691: jne 0x000000010
0x000000010249c693: add $0x5,%r11d
0x000000010249c697: mov %r11d,0x98(
0x000000010249c69e: jmp 0x000000010
0x000000010249c6a0: mov $0xffffff6
0x000000010249c6a5: xchg %ax,%ax
0x000000010249c6a7: callq 0x000000
0x000000010249c6ac: callq 0x000000
0x000000010249c6b1: mov %r13d,%r10d
0x000000010249c6b4: mov %r10d,%r13d
0x000000010249c6b7: mov $0xfffffc6
0x000000010249c6bc: mov %rcx,%rbp
0x000000010249c6bf: mov %rdx,(%rsp)
0x000000010249c6c3: mov %r13d,0x10(
0x000000010249c6c8: mov %rbx,0x18(%
0x000000010249c6cd: mov %r14,0x20(%
0x000000010249c6d2: nop
0x000000010249c6d3: callq 0x000000
0x000000010249c6d8: callq 0x000000
0x000000010249c6dd: add $0xa,%r11d

Megamorphic



Escape Analysis

Scope analysis optimisations for eliminating heap allocations and object locks

NoEscape

Object does not escape method scope.

Can avoid memory allocation on the JVM heap.

Reduce overhead of high-churn objects.

ArgEscape

Object escapes method scope (as arg?)

Does not escape current thread.

Cannot eliminate heap allocation.

Locks on this object may be elided.

GlobalEscape

Accessible by other methods and threads.

Not available for heap elimination or lock elision.

Garbage collected at end of life.

Allocation Elimination

Object explosion

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 value;

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

     public boolean equals(EscapeTest other)
     {
          return this.value == other.value;
     }

     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 of equals() prevents ArgEscape

Hot Loop Allocations

With Escape Analysis

```
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

```
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 ☐ Mouse Follow

Bytecode size: 87 | Native size: 3080 | Compile time (ms): 8

Source

```
1  public class EscapeTest
2  {
3    private final int value;
4
5    public EscapeTest(final int value)
6    {
7      this.value = value;
8    }
9
10   public boolean equals(EscapeTest other)
11   {
12     return this.value == other.value;
13   }
14
15   public static int run()
16   {
17     int matches = 0;
18
19     java.util.Random random = new java.util.Random();
20
21     for (int i = 0; i < 100_000_000; i++)
22     {
23       int v1 = random.nextBoolean() ? 1 : 0;
24       int v2 = random.nextBoolean() ? 1 : 0;
25
26       final EscapeTest e1 = new EscapeTest(v1);
27       final EscapeTest e2 = new EscapeTest(v2);
28
29       if (e1.equals(e2))
30       {
31         matches++;
32       }
33     }
34
35     return matches;
36   }
37
38   public static void main(final String[] args)
39   {
40     System.out.println(run());
41   }
42 }
```

Bytecode (double click for JVM spec)

```
 0: iconst_0
 1: istore_0
 2: new           #3   // class java/util/Random
 5: dup
 6: invokespecial #4   // Method java/util/Random."<init>":()V
 9: astore_1
10: iconst_0
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
46: 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
```

Mounted class version: 52.0 (Java 8) public static int run() compiled with C2

JITWatch shows both heap allocations were eliminated

Branch Prediction

```
public class BranchPrediction
{
    public BranchPrediction()
    {
        int thingOne = 0;
        int thingTwo = 0;

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

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

        System.out.println(thingOne + "/" + thingTwo);
    }

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

HotSpot measures branches

Speculative execution

Code elimination

Puts in a trap

Branch Prediction

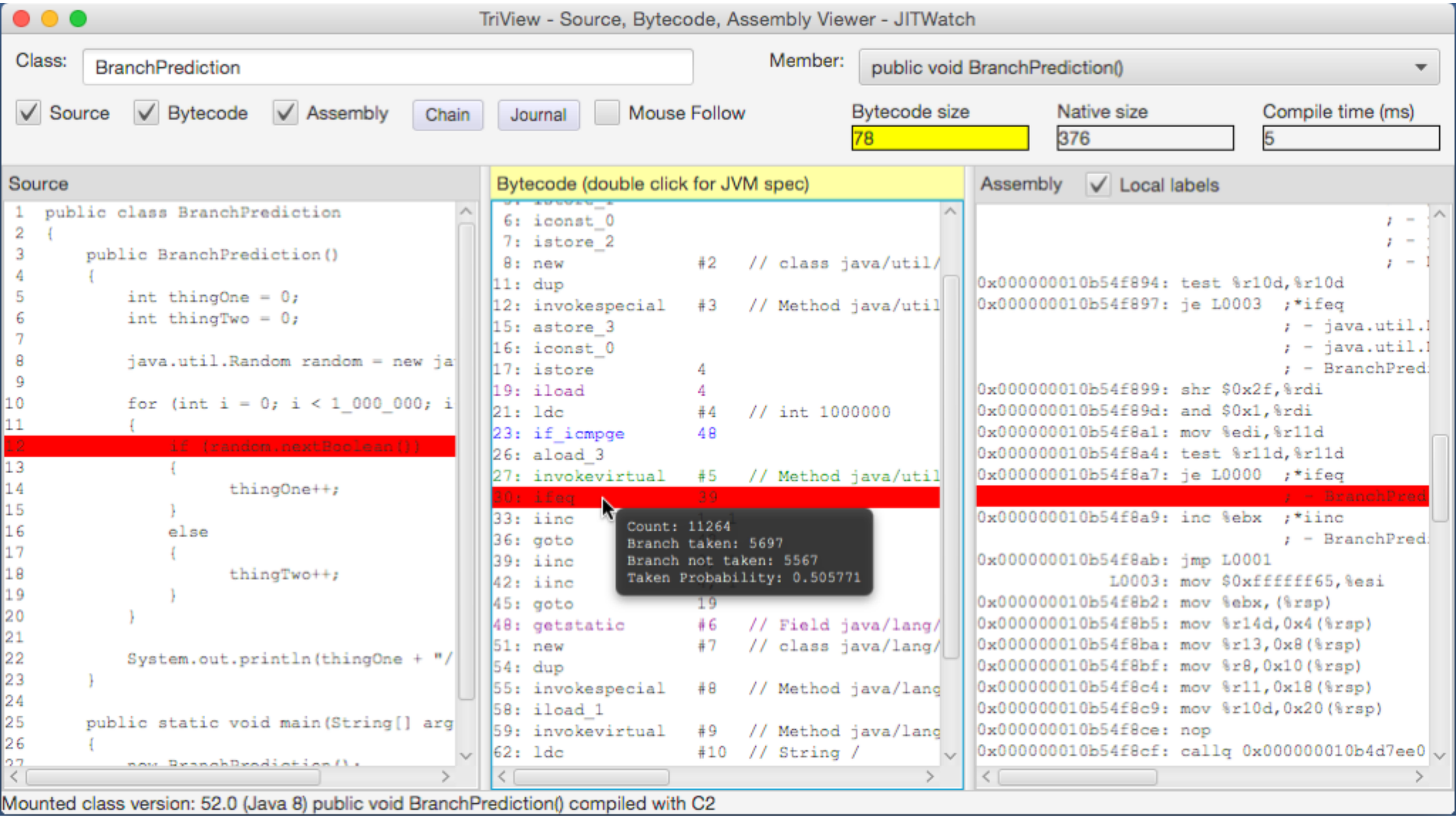
JITWatch Code Suggestions

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

Intrinsic optimised code found in
`Math`, `Unsafe`, `System`, `Class`, `Arrays`, `String`, `StringBuilder`, `AESCrypt`, …

Full list in
`hotspot/src/share/vm/classfile/vmSymbols.hpp`

Intrinsics

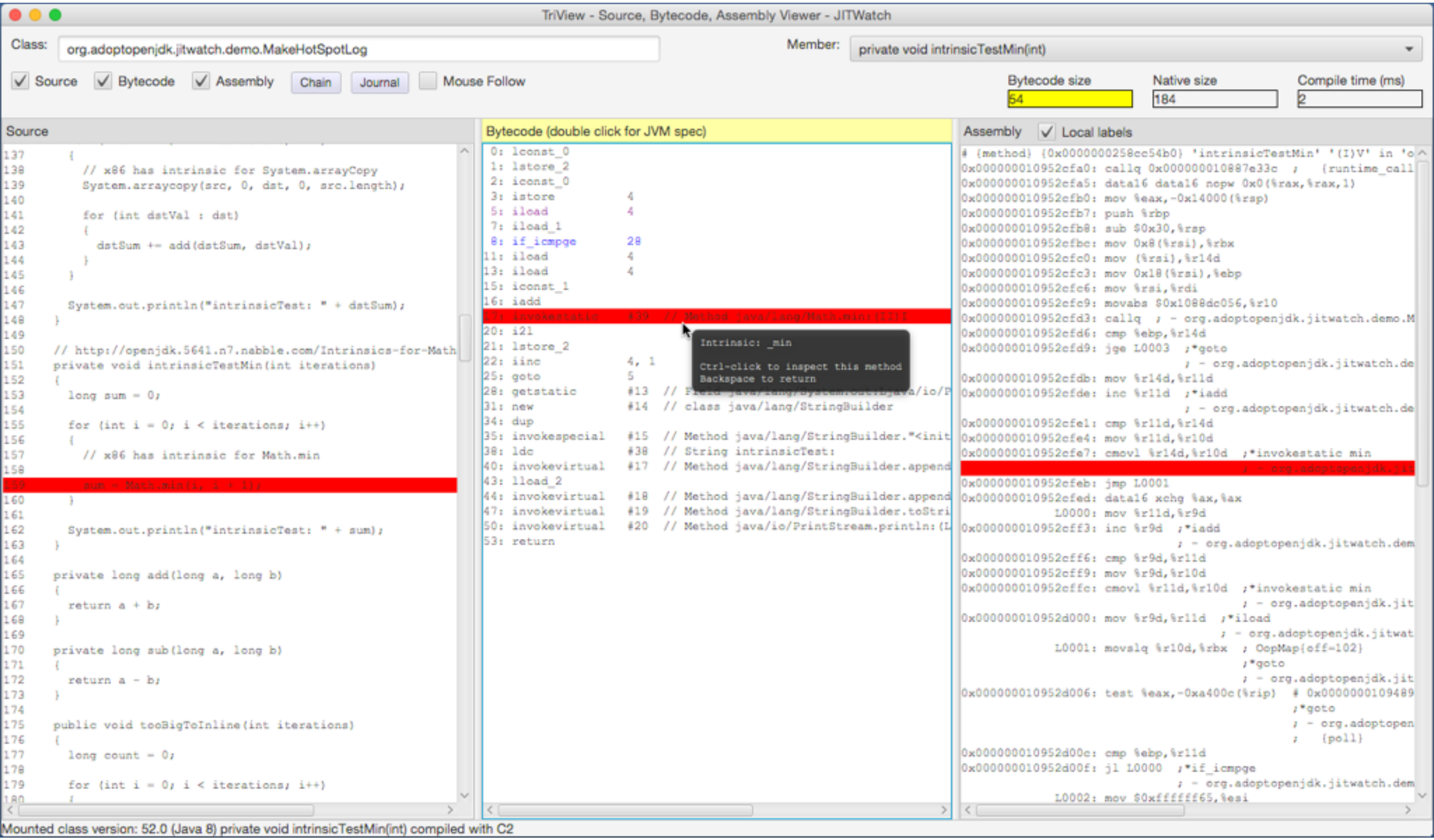
Don’t assume core-libs code is unoptimised

`Math.log10(double)` is 2 CPU instructions

```
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 );
%}
```

(hotspot/src/cpu/x86/vm/x86_64.ad)

Intrinsics



JITWatch highlights use of intrinsics

TL;DR

JIT logs should be part of perf toolkit

Keep methods small for inlining (Head Test)

Use inlineable core lib methods

Look out for unpredictable branches

Use appropriate method visibility (CHA)

Count interface implementations

Check allocations in hot code are EA’d

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**

Resources

- JITWatch on GitHub
  - http://www.github.com/AdoptOpenJDK/jitwatch
  - AdoptOpenJDK project
  - Pull requests are welcome!
- Mailing list
  - groups.google.com/jitwatch
- Twitter
  - @chriswhocodes