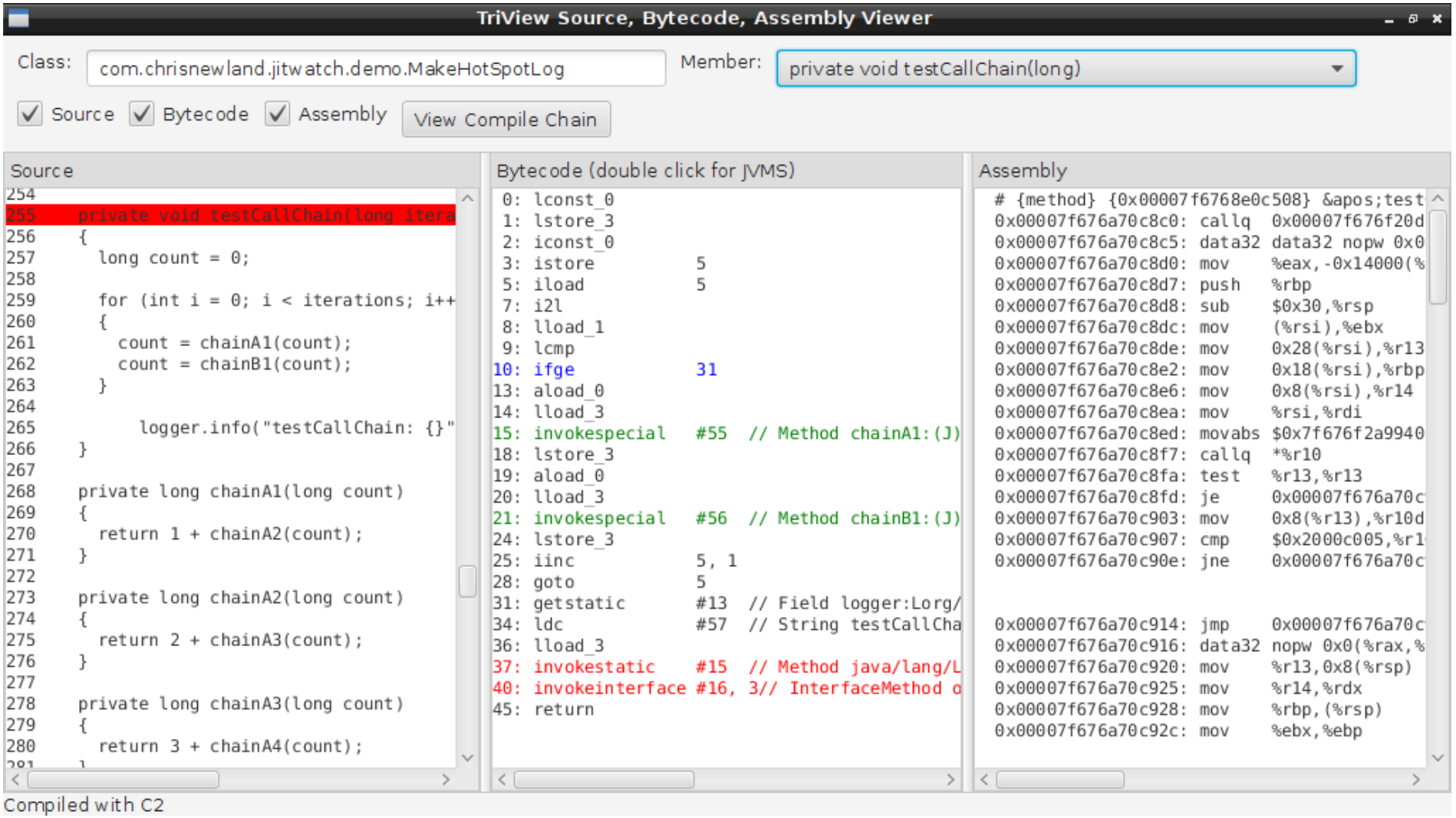
HotSpot profiling with JITWatch



Chris Newland - 16th April 2014

WhatSpot?

- Java HotSpot Virtual Machine
  - Bytecode interpreting stack machine
    - No registers
    - Variables pushed onto stack
  - Just In Time (JIT) compilers
    - Profile Guided Optimisation (PGO)
    - Compile bytecode to native code

Tiered	Non-Tiered	-Xint	-Xcomp
2.9s	2.6s	80.5s	4.4s

*Horrible unscientific benchmark
(com.chrisnewland.jitwatch.demo.MakeHotSpotLog)

Talking JIT

- Client compiler (C1)
  - Starts quickly, simple compilation to native
- Server compiler (C2)
  - Waits until more information available
  - Loop unrolling, **Inlining**, Dead Code Elimination, Escape analysis, Intrinsics, **Branch prediction**
- Tiered Compilation (C1 + C2)
  - Default in Java 8
  - Enable in Java 7 with -XX:+TieredCompilation
  - Best of both worlds?

Explain yourself!

- Enable JIT logging
- -XX:+UnlockDiagnosticVMOptions
- -XX:+LogCompilation
- -XX:+TraceClassLoading (JITWatch)
- -XX:+PrintAssembly
  - Required hsdis binary in jre/lib/<arch>/server
  - Significant performance overhead
  - http://www.chrisnewland.com/building-hsdis-on-linux-amd64-on-debian-369

I heard you like to grep?

```
<task compile_id='23' method='java/util/ArrayList$Itr
checkForComodification ()V' bytes='23' count='9006'
backedge_count='1' iicount='44000' st
amp='1.603'>
<phase name='parse' nodes='3' live='3' stamp='1.603'>
<type id='680' name='void'/>
<klass id='776' name='java/util/ArrayList$Itr' flags='2'/>
<method id='777' holder='776' name='checkForComodification'
return='680' flags='16' bytes='23' iicount='44000'/>
<klass id='781' name='java/util/ConcurrentModificationException'
unloaded='1'/>
<uncommon_trap method='777' bci='14' reason='unloaded'
action='reinterpret' index='47' klass='781'/>
<parse method='777' uses='44000' stamp='1.604'>
<bc code='180' bci='4'/>
...
```

- Logs can be > 50MB
- Much bigger with disassembly!
- Let's build a visualiser!

JITWatch

- https://github.com/AdoptOpenJDK/jitwatch/
- JIT Compilation
  - When? (time, invocations)
  - How? (C1, C2, Tiered, OSR)
- Decompiles
  - Back to bytecode interpretation (Why?)
- Inlining - successes / failures
- Branch probabilities - taken / not taken
- Intrinsics

Inlining (C1 + C2)

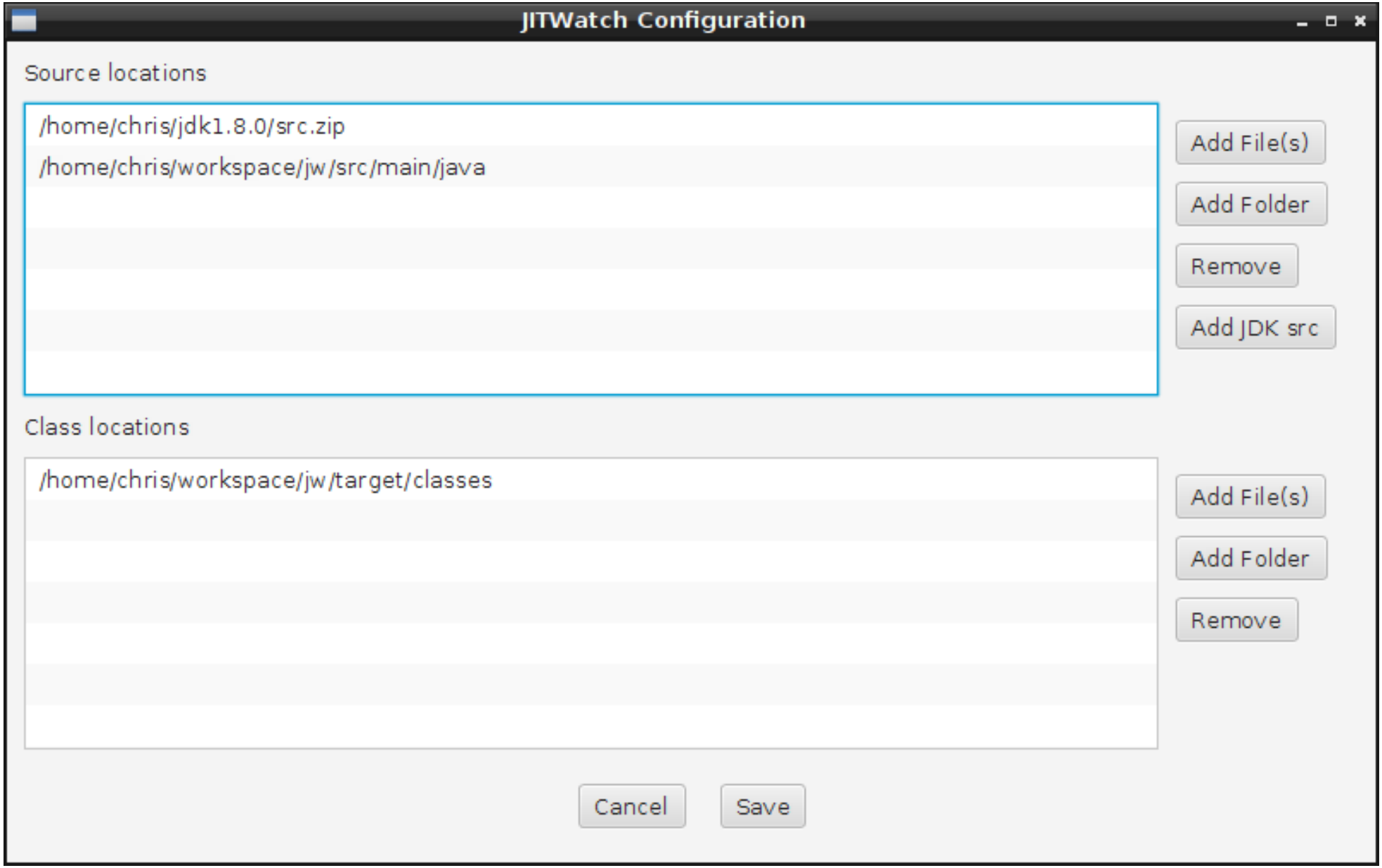
```
int a = 3;
int b = 4;
int result = add(a, b);
...
public int add(int x, int y) {  return x + y;  }
```

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

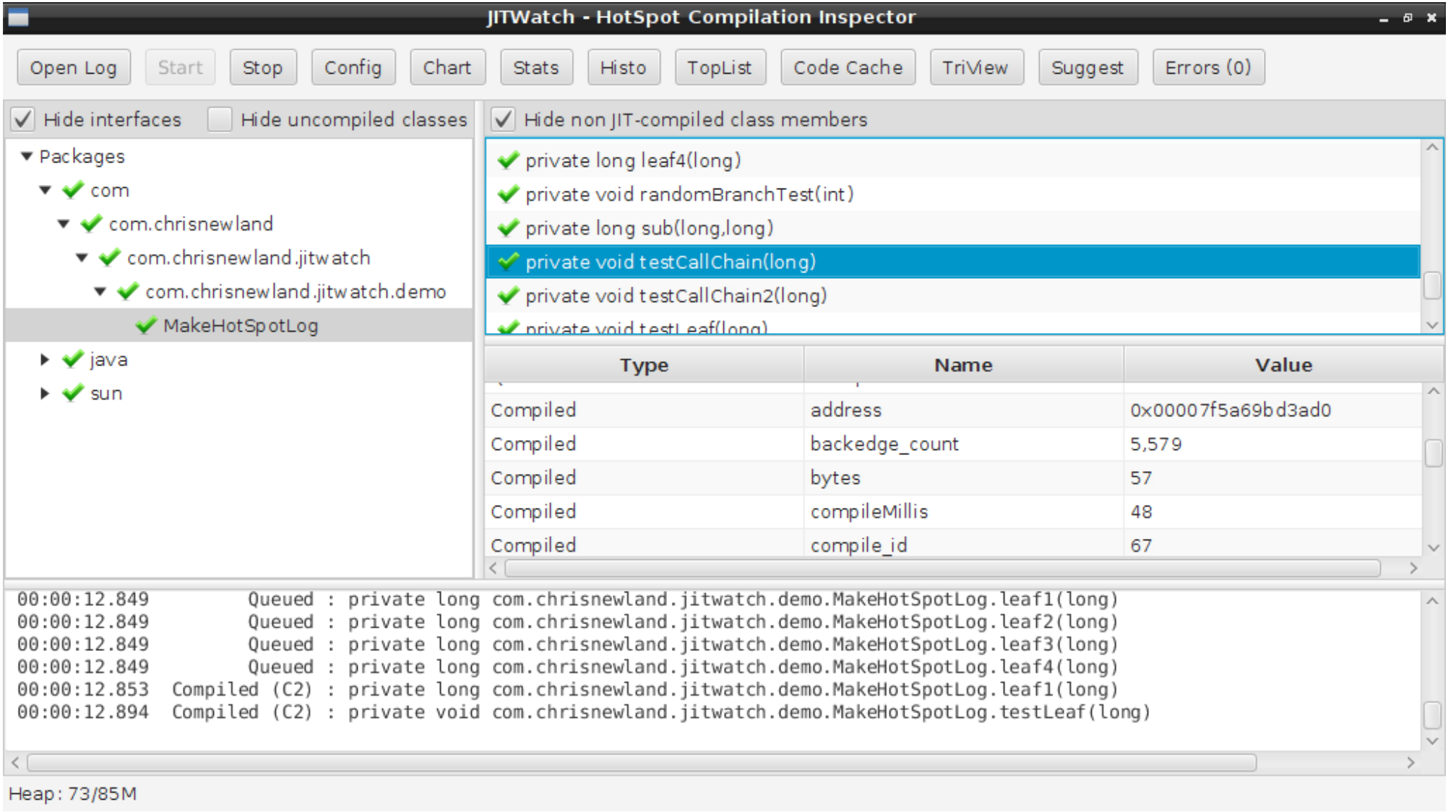
Branch Prediction (C2)

```
// make an array of random doubles 0..1
double[] bigArray = makeBigArray(1_000_000);

for (int i = 0; i < bigArray.length; i++)
{
   double cur = bigArray[i];
   if (cur > 0.5) { doThis();} else { doThat();}
}

// branch will be taken ~50% of time
// sorting the array will make it more predictable
```

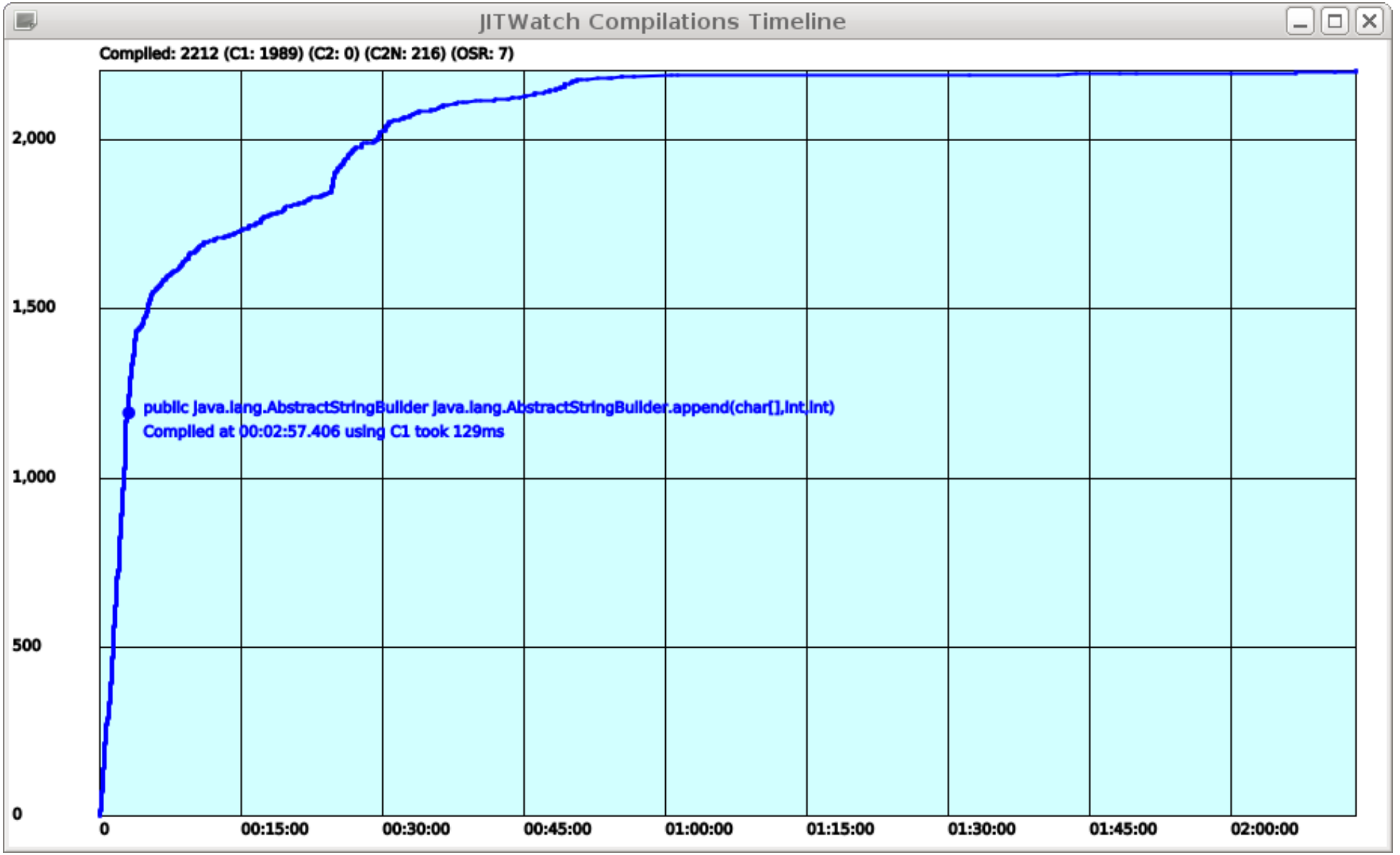
Setting up


JITWatch Configuration
Source locations
/home/chris/jdk1.8.0/src.zip
/home/chris/workspace/jw/src/main/java
Add File(s)
Add Folder
Remove
Add JDK src
Class locations
/home/chris/workspace/jw/target/classes
Add File(s)
Add Folder
Remove
Cancel
Save

Compile tree



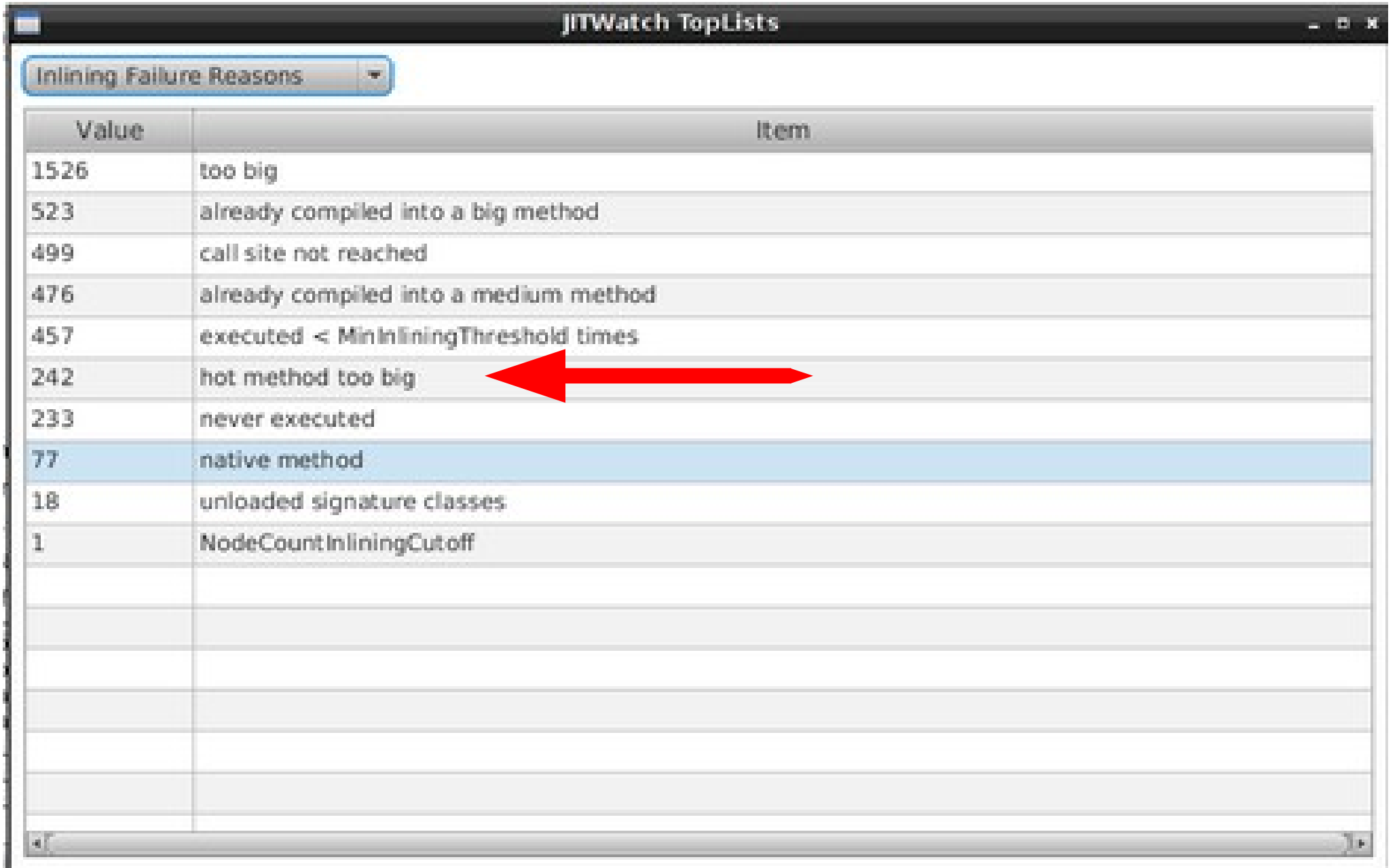
Compilations timeline



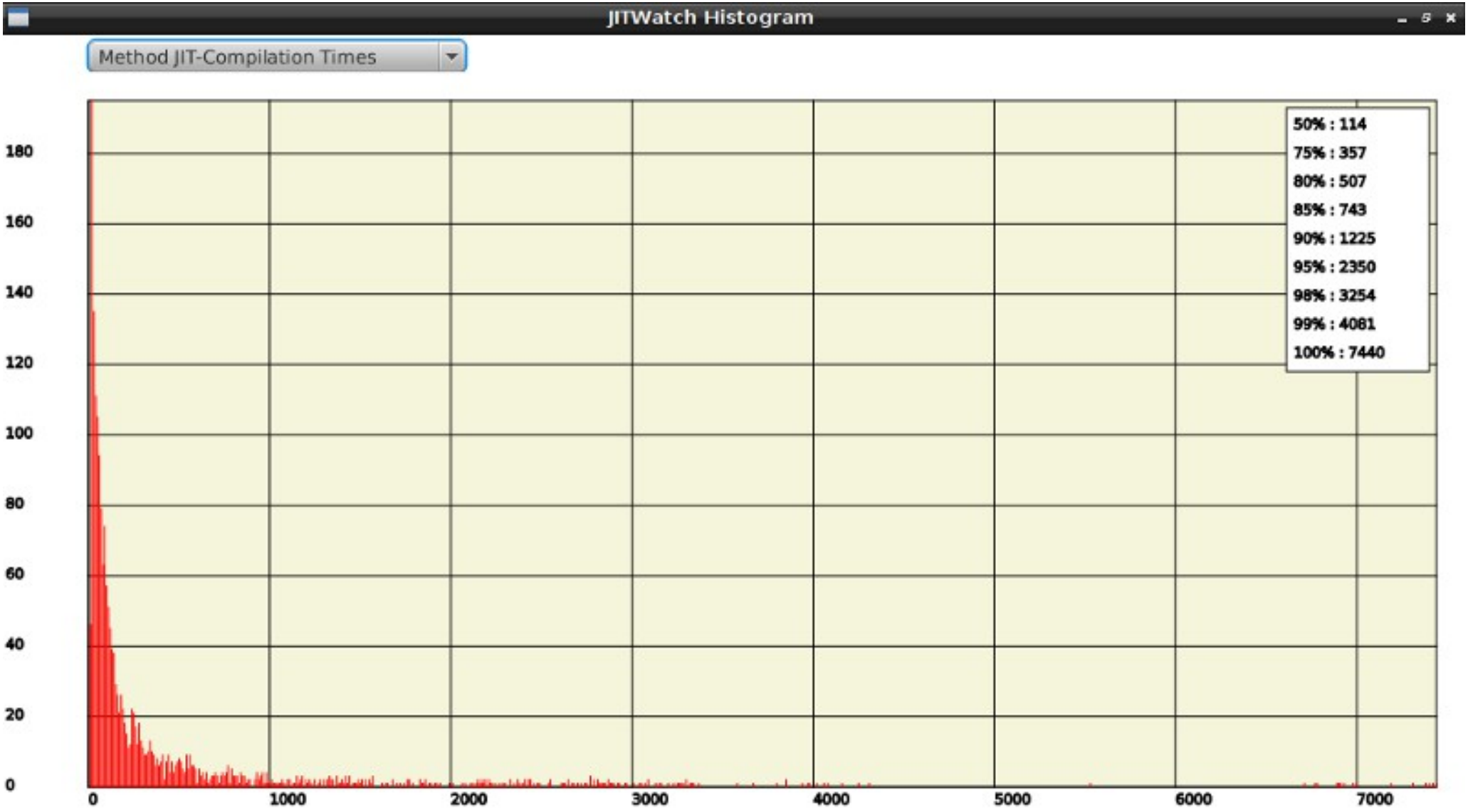
Toplists

- **Bytecode size**
- Native code size
- **Inlining failure reasons**
- Most-used intrinsics
- Compilation order
- Most-decompiled methods
  - Compiler assumption was wrong

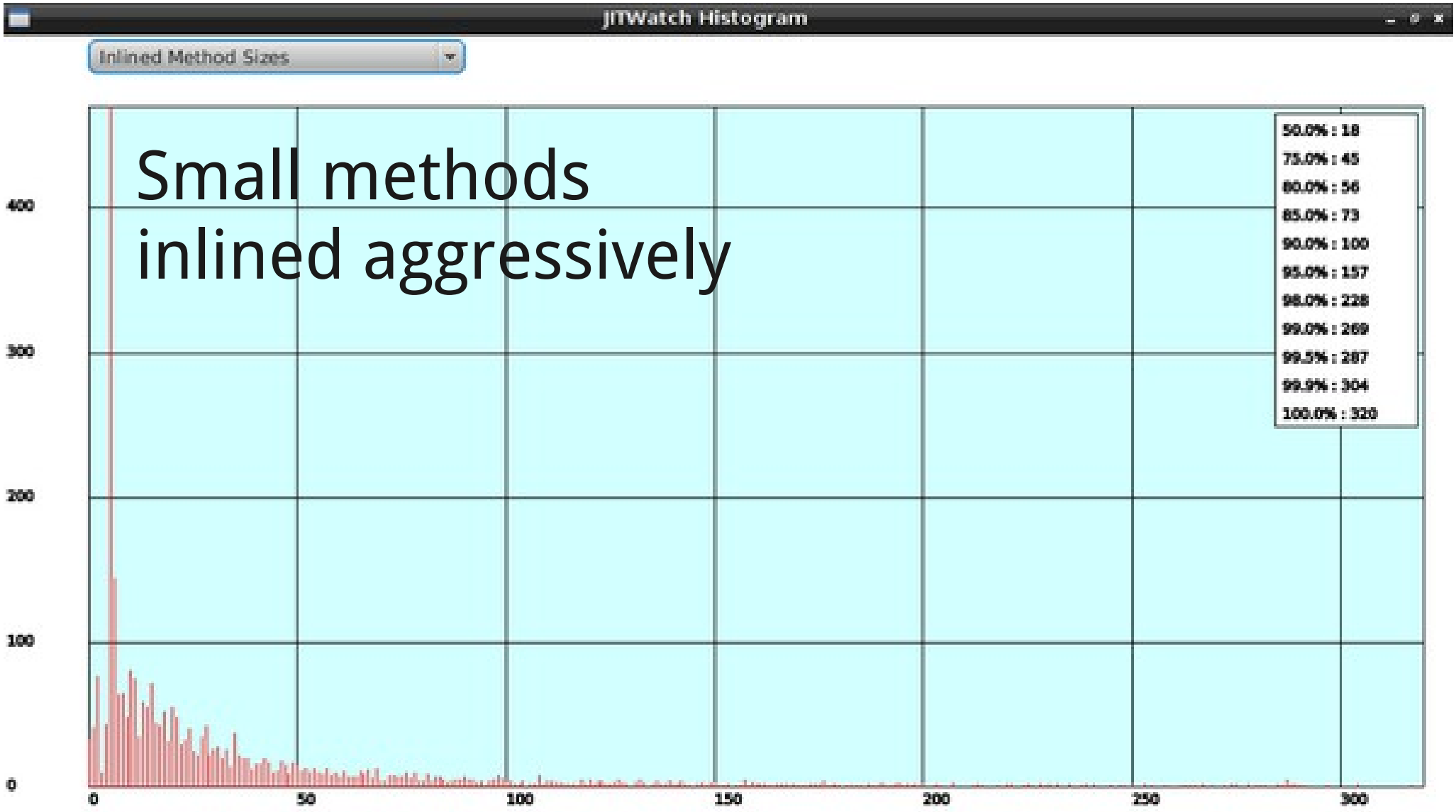
Toplists – Inline failure reasons



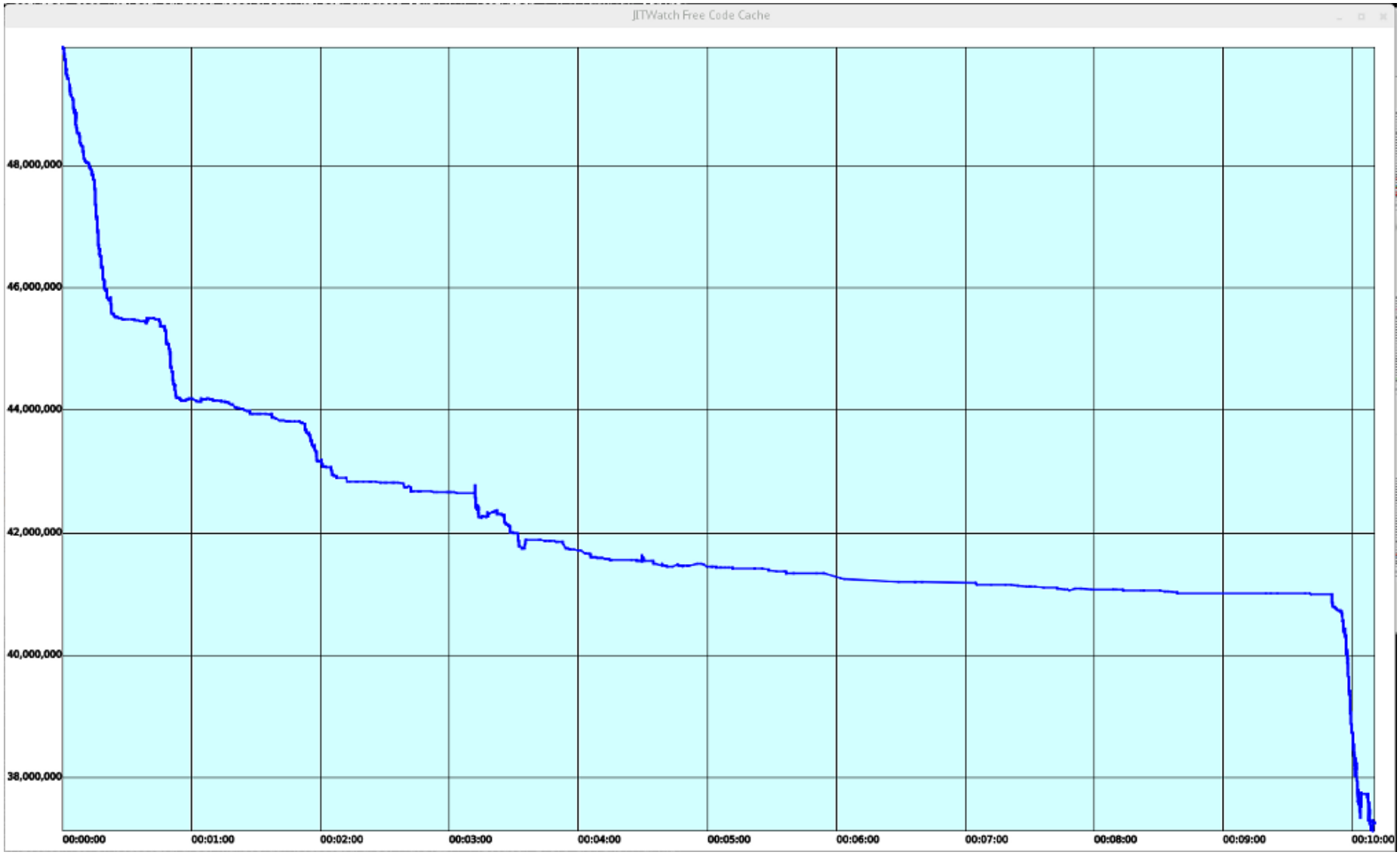
Compile times



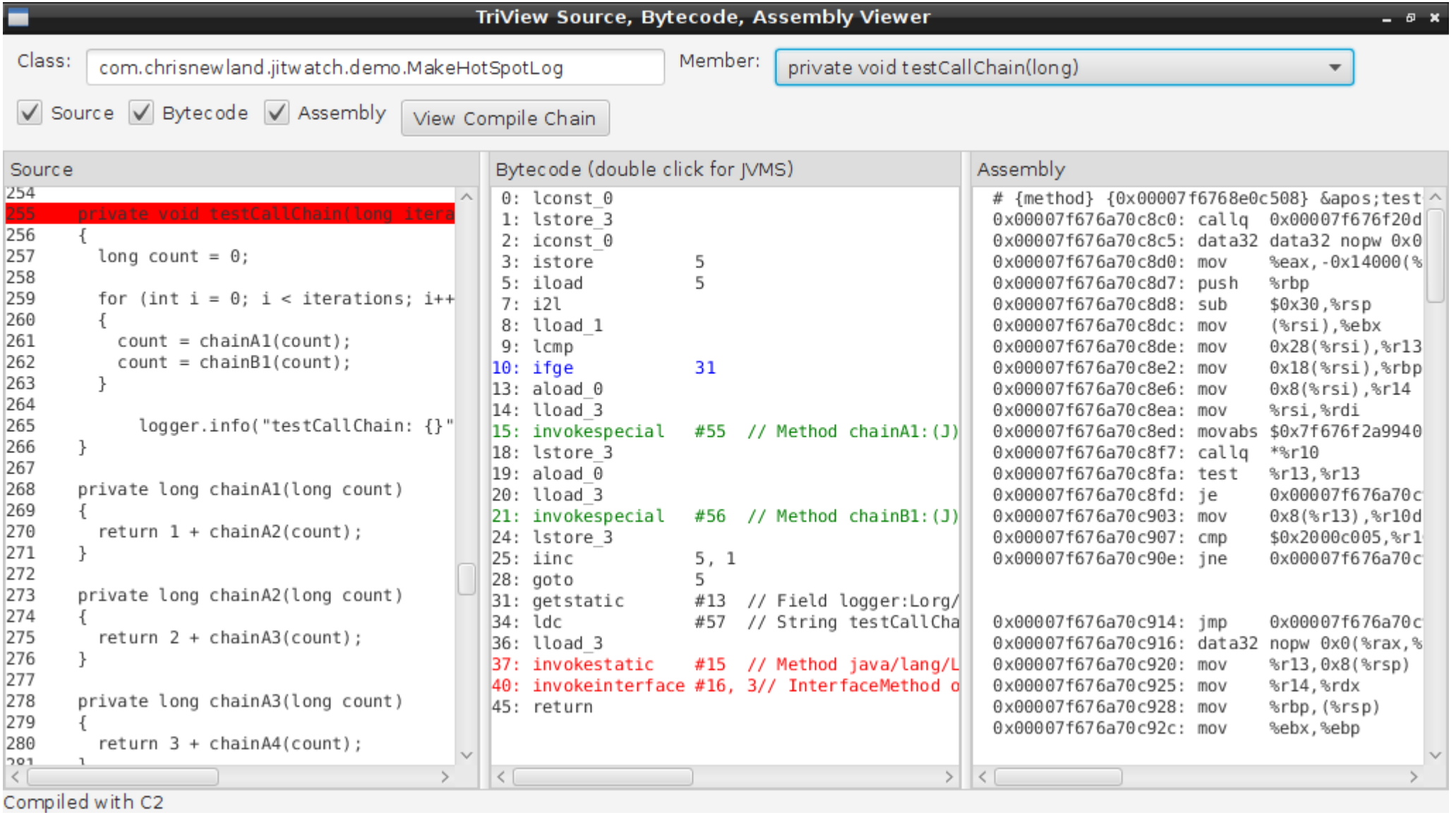
Histogram – Inlined method sizes



Code Cache


JITWatch Free Code Cache
48,000,000
46,000,000
44,000,000
42,000,000
40,000,000
38,000,000
00:00:00
00:01:00
00:02:00
00:03:00
00:04:00
00:05:00
00:06:00
00:07:00
00:08:00
00:09:00
00:10:00

TriView


TriView Source, Bytecode, Assembly Viewer
Class:
com.chrisnewland.jitwatch.demo.MakeHotSpotLog
Member:
private void testCallChain(long)
Source
Bytecode
Assembly
View Compile Chain
Source
254
255 private void testCallChain(long itera
256 {
257 long count = 0;
258
259 for (int i = 0; i < iterations; i++
260 {
261 count = chainA1(count);
262 count = chainB1(count);
263 }
264
265 logger.info("testCallChain: {}"
266 }
267
268 private long chainA1(long count)
269 {
270 return 1 + chainA2(count);
271 }
272
273 private long chainA2(long count)
274 {
275 return 2 + chainA3(count);
276 }
277
278 private long chainA3(long count)
279 {
280 return 3 + chainA4(count);
Bytecode (double click for JVMS)
0: lconst_0
1: lstore_3
2: iconst_0
3: istore 5
5: iload 5
7: i2l
8: lload_1
9: lcmp
10: ifge 31
13: aload_0
14: lload_3
15: invokespecial #55 // Method chainA1:(J)
18: lstore_3
19: aload_0
20: lload_3
21: invokespecial #56 // Method chainB1:(J)
24: lstore_3
25: iinc 5, 1
28: goto 5
31: getstatic #13 // Field logger:Lorg/
34: ldc #57 // String testCallCha
36: lload_3
37: invokestatic #15 // Method java/lang/L
40: invokeinterface #16, 3// InterfaceMethod o
45: return
Assembly
{method} {0x00007f6768e0c508} 'test
0x00007f676a70c8c0: callq 0x00007f676f20d
0x00007f676a70c8c5: data32 data32 nopw 0x0
0x00007f676a70c8d0: mov %eax,-0x14000(%
0x00007f676a70c8d7: push %rbp
0x00007f676a70c8d8: sub $0x30,%rsp
0x00007f676a70c8dc: mov (%rsi),%ebx
0x00007f676a70c8de: mov 0x28(%rsi),%r13
0x00007f676a70c8e2: mov 0x18(%rsi),%rbp
0x00007f676a70c8e6: mov 0x8(%rsi),%r14
0x00007f676a70c8ea: mov %rsi,%rdi
0x00007f676a70c8ed: movabs $0x7f676f2a9940
0x00007f676a70c8f7: callq *%r10
0x00007f676a70c8fa: test %r13,%r13
0x00007f676a70c8fd: je 0x00007f676a70c
0x00007f676a70c903: mov 0x8(%r13),%r10d
0x00007f676a70c907: cmp $0x2000c005,%r1
0x00007f676a70c90e: jne 0x00007f676a70c
0x00007f676a70c914: jmp 0x00007f676a70c
0x00007f676a70c916: data32 nopw 0x0(%rax,%
0x00007f676a70c920: mov %r13,0x8(%rsp)
0x00007f676a70c925: mov %r14,%rdx
0x00007f676a70c928: mov %rbp,(%rsp)
0x00007f676a70c92c: mov %ebx,%ebp
Compiled with C2

JVM Spec Browser

JMVS Browser - if_icmpge

if_icmp<cond>

Operation

Branch if `int` comparison succeeds

Format

```
if_icmp<cond>
branchbyte1
branchbyte2
```

Forms

if_icmpeq = 159 (0x9f)

if_icmpne = 160 (0xa0)

if_icmplt = 161 (0xa1)

if_icmpge = 162 (0xa2)

if_icmpgt = 163 (0xa3)

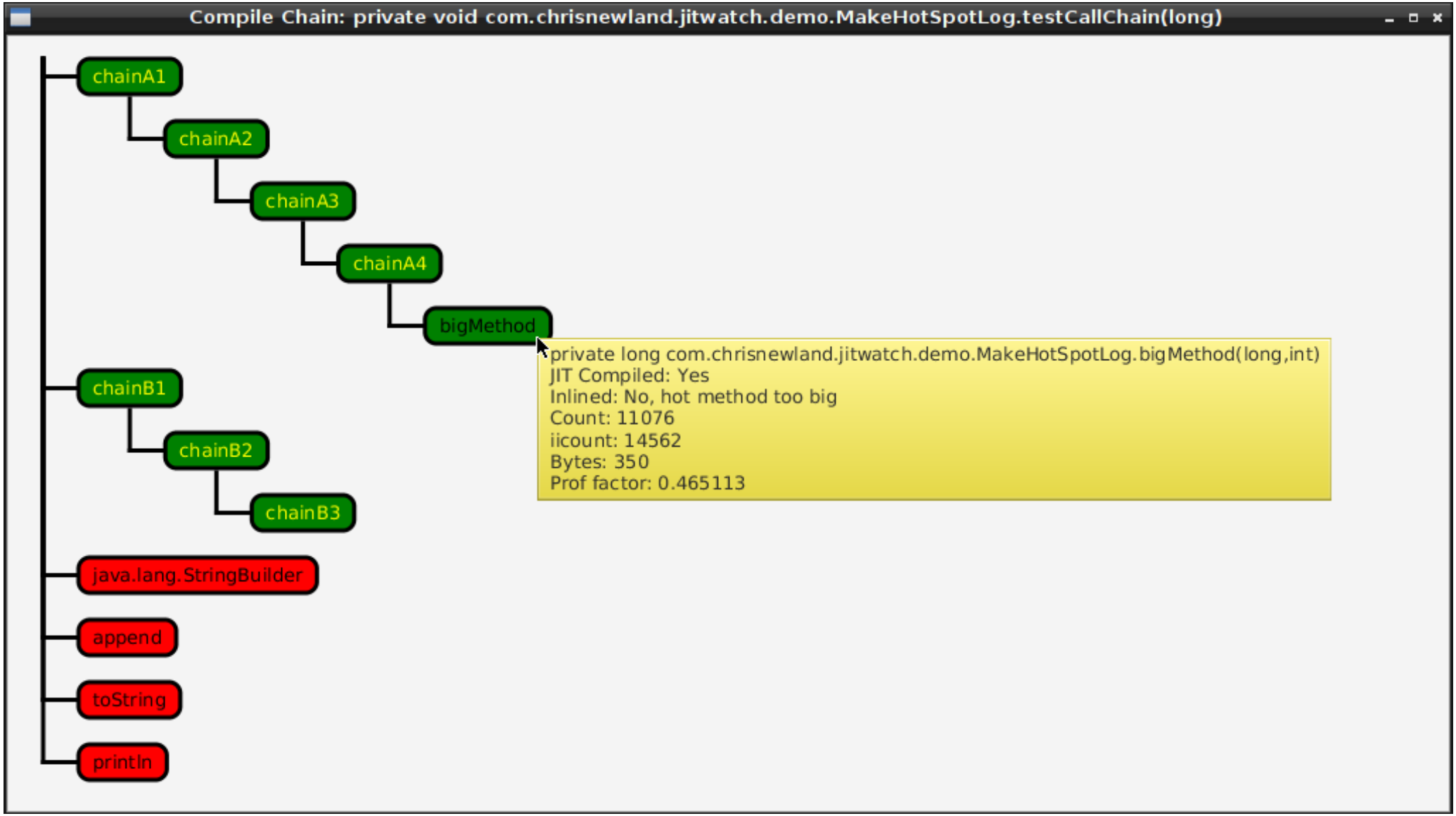
if_icmple = 164 (0xa4)

Operand Stack

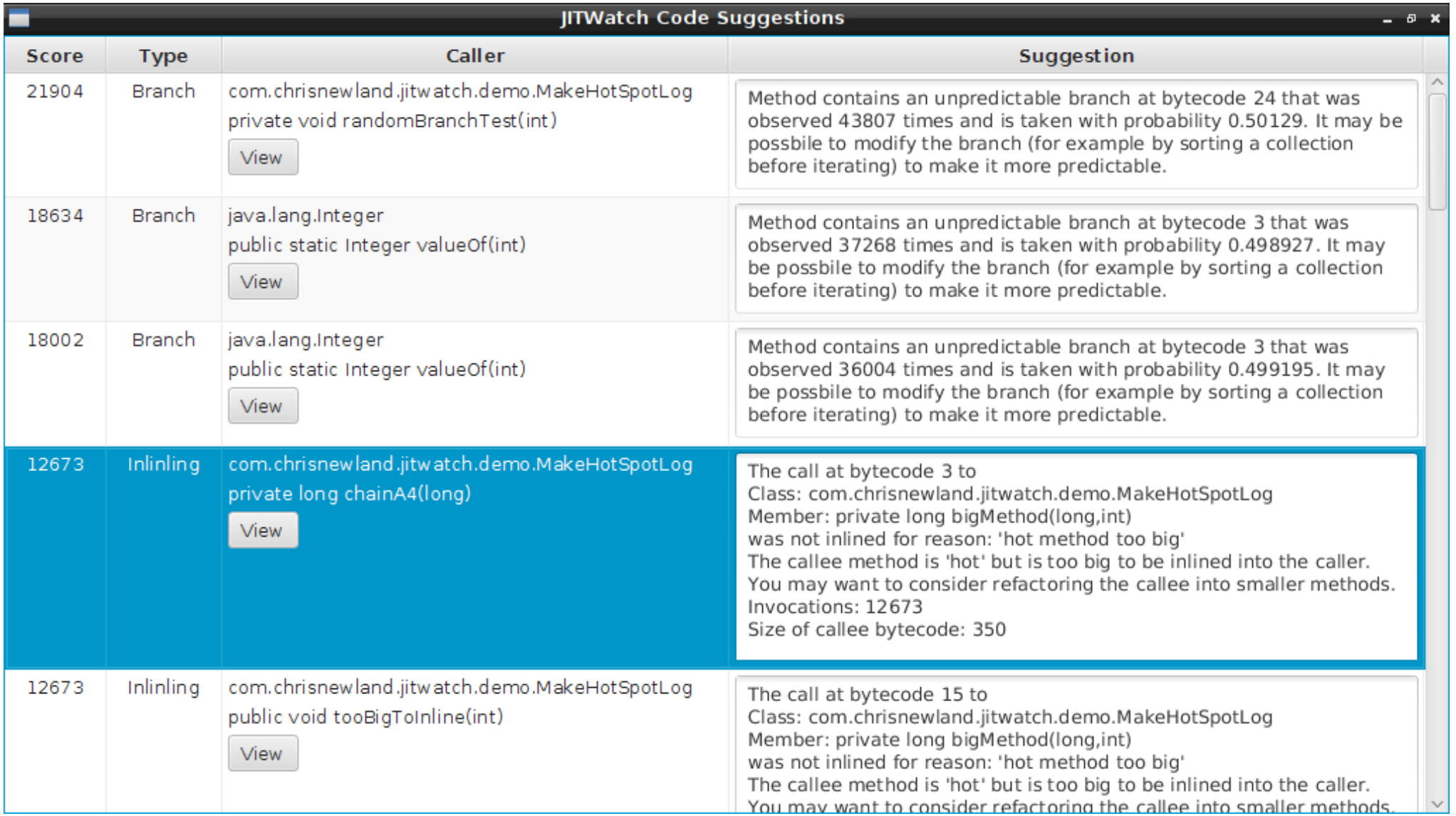
..., value1, value2 →

...

Compile Chains



Code Suggestion Tool



JarScan Tool

- Statical analysis of a jar
- Methods with bytecode > inlining threshold
- These methods might not be hot
- Around 3000 non-inlineable methods in rt.jar
  - String.split
  - String.toUpperCase / toLowerCase
  - Core parts of j.u.ComparableTimSort

TL;DR

- Eliminate other performance issues first
- Keep your methods small for inlining
- Turn on JIT logging
  - JITWatch suggestion tool
    - “hot method too big”
    - Unpredictable branches
- Learn about the JVM :)

Premature optimization is the root of all evil

Donald Knuth

Resources

- JITWatch on GitHub
  - http://www.github.com/AdoptOpenJDK/jitwatch
  - AdoptOpenJDK project
  - Send a pull request!
- Mailing list
  - groups.google.com/jitwatch
- Twitter
  - @chriswhocodes

Thanks!