

# HotSpot profiling with JITWatch

The screenshot displays the TriView Source, Bytecode, Assembly Viewer interface. The Class field is set to `com.chrisnewland.jitwatch.demo.MakeHotSpotLog` and the Member field is `private void testCallChain(long)`. The Source, Bytecode, and Assembly checkboxes are all checked. The Source pane shows the following code:

```
254
255 private void testCallChain(long iterations)
256 {
257     long count = 0;
258     for (int i = 0; i < iterations; i++)
259     {
260         count = chainA1(count);
261         count = chainB1(count);
262     }
263     logger.info("testCallChain: {}");
264 }
265
266 private long chainA1(long count)
267 {
268     return 1 + chainA2(count);
269 }
270
271 private long chainA2(long count)
272 {
273     return 2 + chainA3(count);
274 }
275
276 private long chainA3(long count)
277 {
278     return 3 + chainA4(count);
279 }
280
281
```

The Bytecode pane shows the following instructions:

```
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
```

The Assembly pane shows the following instructions:

```
# {method} {0x00007f6768e0c508} &apos;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

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

<b>Tiered</b>	<b>Non-Tiered</b>	<b>-Xint</b>	<b>-Xcomp</b>
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, Intrinsic, **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 hsdisk binary in `jre/lib/<arch>/server`
  - Significant performance overhead
  - <http://www.chrisnewland.com/building-hsdisk-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
- Intrinsic

# 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)

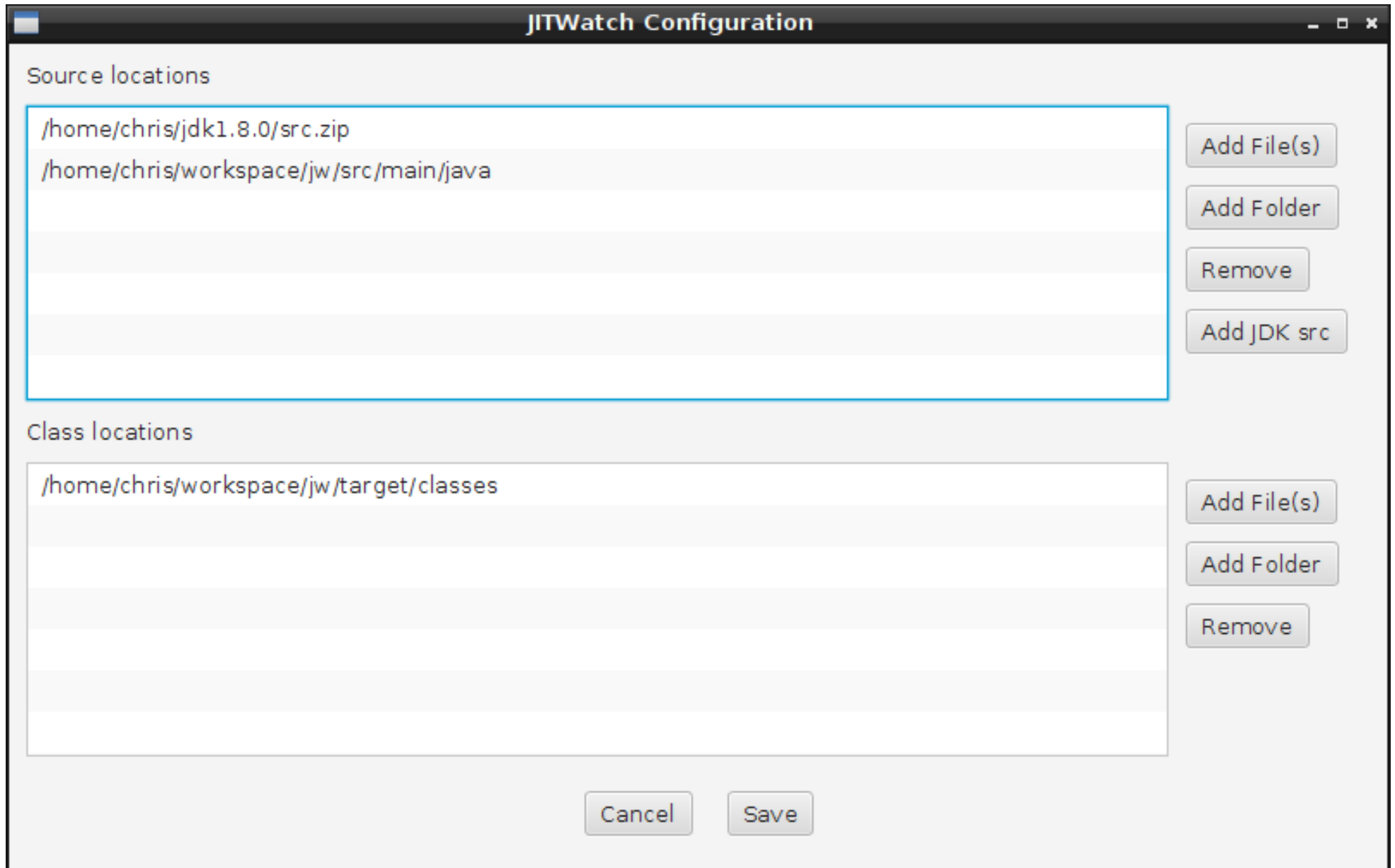
```
// make an array of random doubles 0..1
double[] bigArray = makeBigArray(1_000_000);

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



# Compile tree

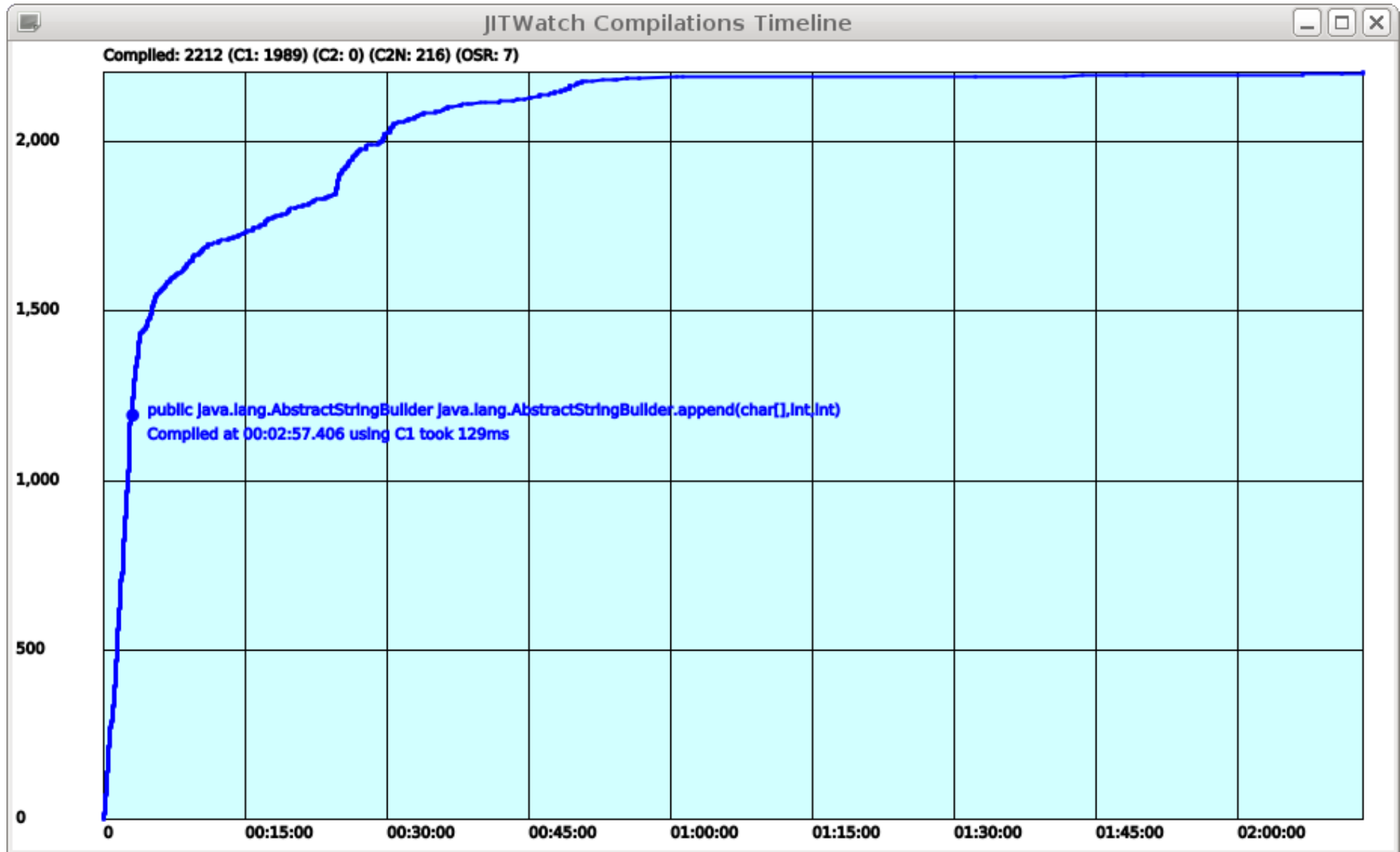
The screenshot shows the JITWatch - HotSpot Compilation Inspector interface. The top toolbar includes buttons for Open Log, Start, Stop, Config, Chart, Stats, Histo, TopList, Code Cache, TriView, Suggest, and Errors (0). The left sidebar shows a package tree with 'com.chrisnewland.jitwatch.demo.MakeHotSpotLog' selected. The main area displays a list of private methods, with 'private void testCallChain(long)' highlighted. Below this is a table of compiled methods with columns for Type, Name, and Value.

Type	Name	Value
Compiled	address	0x00007f5a69bd3ad0
Compiled	backedge_count	5,579
Compiled	bytes	57
Compiled	compileMillis	48
Compiled	compile_id	67

00:00:12.849 Queued : private long com.chrisnewland.jitwatch.demo.MakeHotSpotLog.leaf1(long)  
00:00:12.849 Queued : private long com.chrisnewland.jitwatch.demo.MakeHotSpotLog.leaf2(long)  
00:00:12.849 Queued : private long com.chrisnewland.jitwatch.demo.MakeHotSpotLog.leaf3(long)  
00:00:12.849 Queued : private long com.chrisnewland.jitwatch.demo.MakeHotSpotLog.leaf4(long)  
00:00:12.853 Compiled (C2) : private long com.chrisnewland.jitwatch.demo.MakeHotSpotLog.leaf1(long)  
00:00:12.894 Compiled (C2) : private void com.chrisnewland.jitwatch.demo.MakeHotSpotLog.testLeaf(long)

Heap: 73/85M

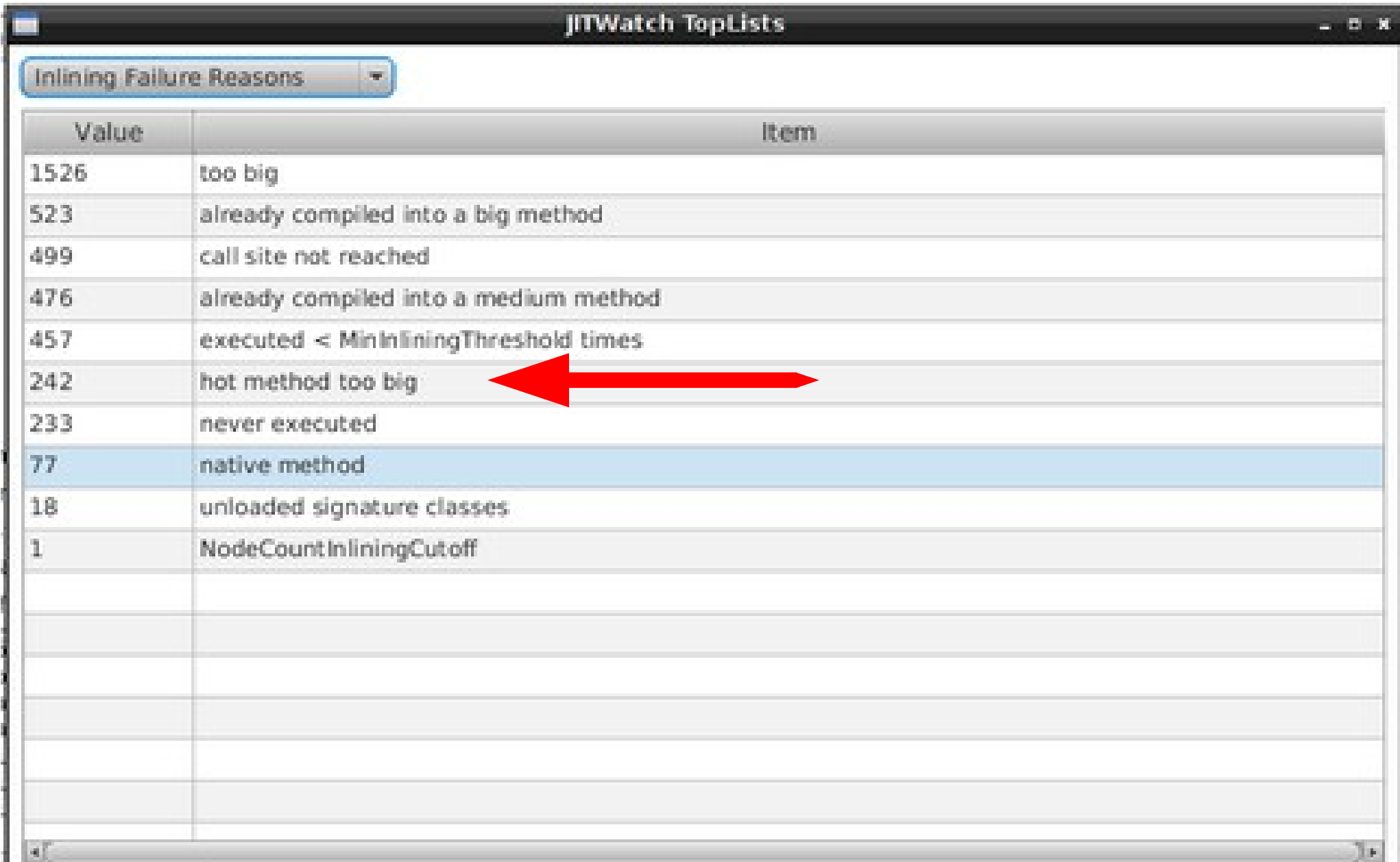
# Compilations timeline



# 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



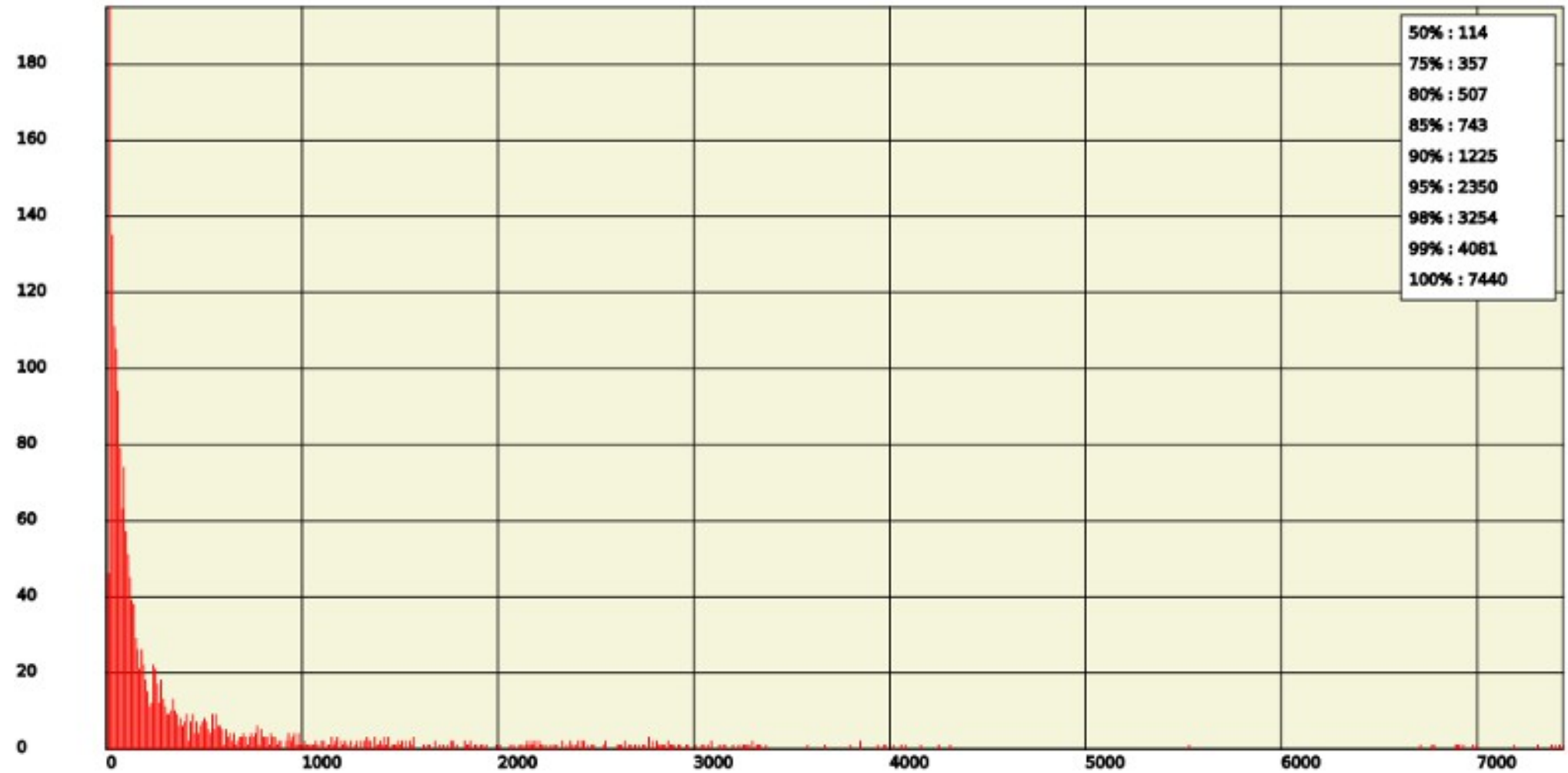
The screenshot shows a window titled "JITWatch TopLists" with a dropdown menu set to "Inlining Failure Reasons". The window displays a table with two columns: "Value" and "Item". The table lists various reasons for inlining failures, with a red arrow pointing to the entry "hot method too big".

Value	Item
1526	too big
523	already compiled into a big method
499	call site not reached
476	already compiled into a medium method
457	executed < MinInliningThreshold times
242	hot method too big
233	never executed
77	native method
18	unloaded signature classes
1	NodeCountInliningCutoff

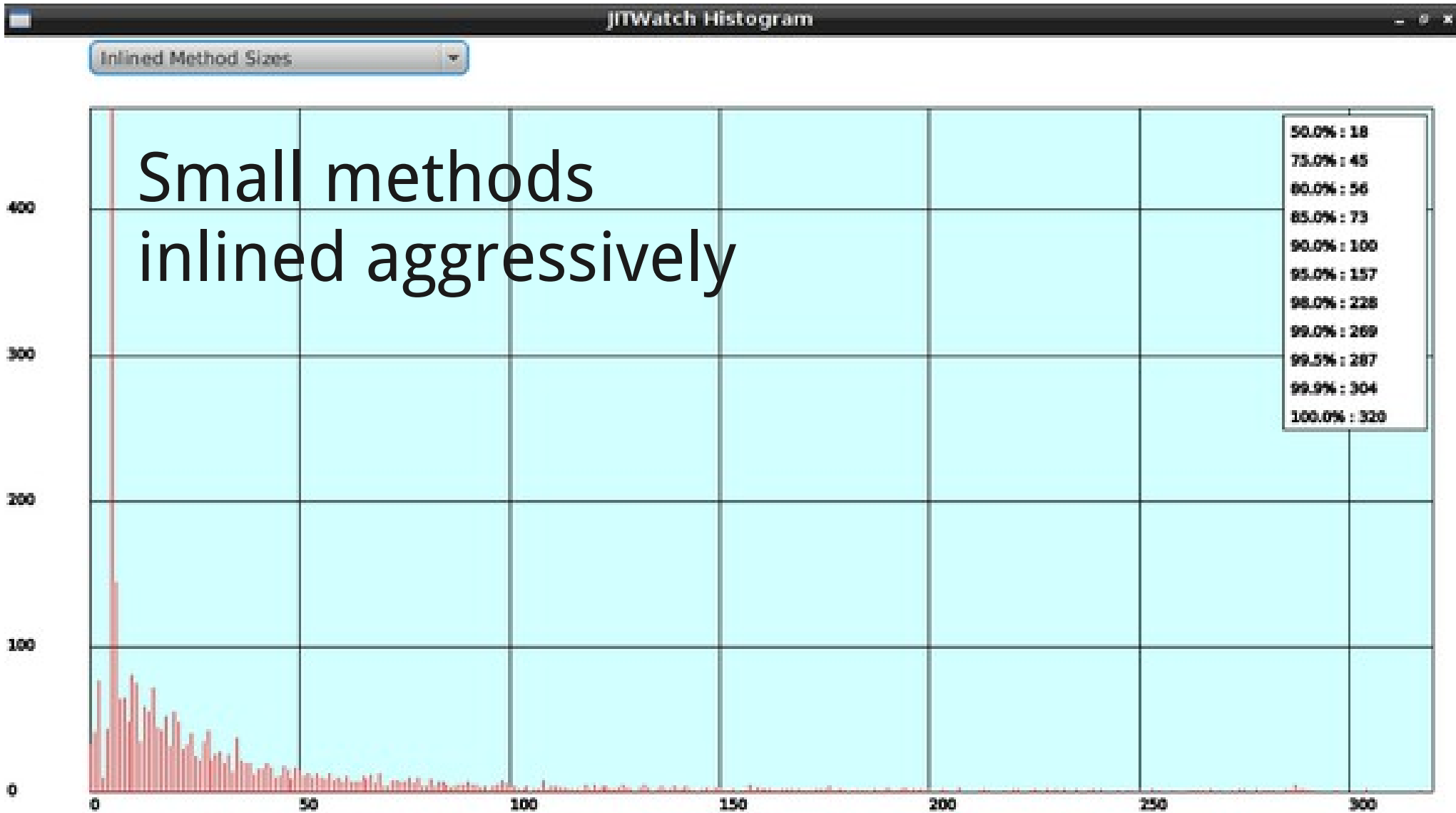
# Compile times

JITWatch Histogram

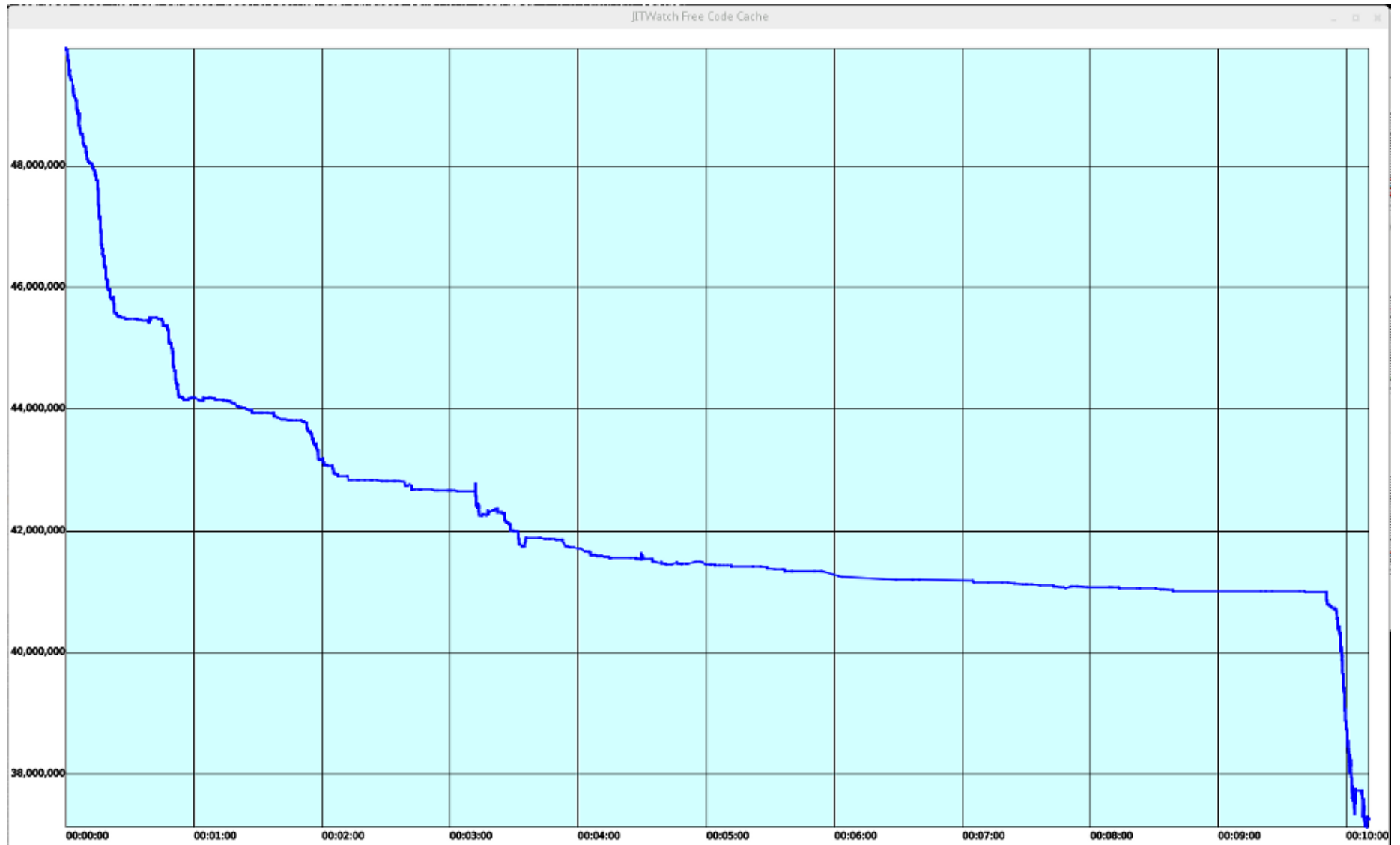
Method JIT-Compilation Times



# Histogram - Inlined method sizes



# Code Cache





# TriView

TriView Source, Bytecode, Assembly Viewer

Class:  Member:

Source  Bytecode  Assembly

Source	Bytecode (double click for JVMs)	Assembly
254	0: lconst_0	# {method} {0x00007f6768e0c508} &apos;test
255 private void testCallChain(long iterations)	1: lstore_3	0x00007f676a70c8c0: callq 0x00007f676f20d
256 {	2: iconst_0	0x00007f676a70c8c5: data32 data32 nopw 0x0
257     long count = 0;	3: istore      5	0x00007f676a70c8d0: mov    %eax, -0x14000(%
258	5: iload      5	0x00007f676a70c8d7: push  %rbp
259     for (int i = 0; i < iterations; i++)	7: i2l	0x00007f676a70c8d8: sub   \$0x30,%rsp
260     {	8: lload_1	0x00007f676a70c8dc: mov   (%rsi),%ebx
261         count = chainA1(count);	9: lcmp	0x00007f676a70c8de: mov   0x28(%rsi),%r13
262         count = chainB1(count);	10: ifge      31	0x00007f676a70c8e2: mov   0x18(%rsi),%rbp
263     }	13: aload_0	0x00007f676a70c8e6: mov   0x8(%rsi),%r14
264         logger.info("testCallChain: {}")	14: lload_3	0x00007f676a70c8ea: mov   %rsi,%rdi
265     }	15: invokespecial #55 // Method chainA1:(J)	0x00007f676a70c8ed: movabs \$0x7f676f2a9940
266 }	18: lstore_3	0x00007f676a70c8f7: callq *%r10
267	19: aload_0	0x00007f676a70c8fa: test  %r13,%r13
268 private long chainA1(long count)	20: lload_3	0x00007f676a70c8fd: je    0x00007f676a70c
269 {	21: invokespecial #56 // Method chainB1:(J)	0x00007f676a70c903: mov   0x8(%r13),%r10d
270     return 1 + chainA2(count);	24: lstore_3	0x00007f676a70c907: cmp   \$0x2000c005,%r1
271 }	25: iinc      5, 1	0x00007f676a70c90e: jne  0x00007f676a70c
272	28: goto      5	
273 private long chainA2(long count)	31: getstatic  #13 // Field logger:Lorg/	
274 {	34: ldc       #57 // String testCallCha	0x00007f676a70c914: jmp  0x00007f676a70c
275     return 2 + chainA3(count);	36: lload_3	0x00007f676a70c916: data32 nopw 0x0(%rax,%
276 }	37: invokestatic #15 // Method java/lang/L	0x00007f676a70c920: mov   %r13,0x8(%rsp)
277	40: invokeinterface #16, 3// InterfaceMethod o	0x00007f676a70c925: mov   %r14,%rdx
278 private long chainA3(long count)	45: return	0x00007f676a70c928: mov   %rbp, (%rsp)
279 {		0x00007f676a70c92c: mov   %ebx,%ebp
280     return 3 + chainA4(count);		
281 }		

Compiled with C2

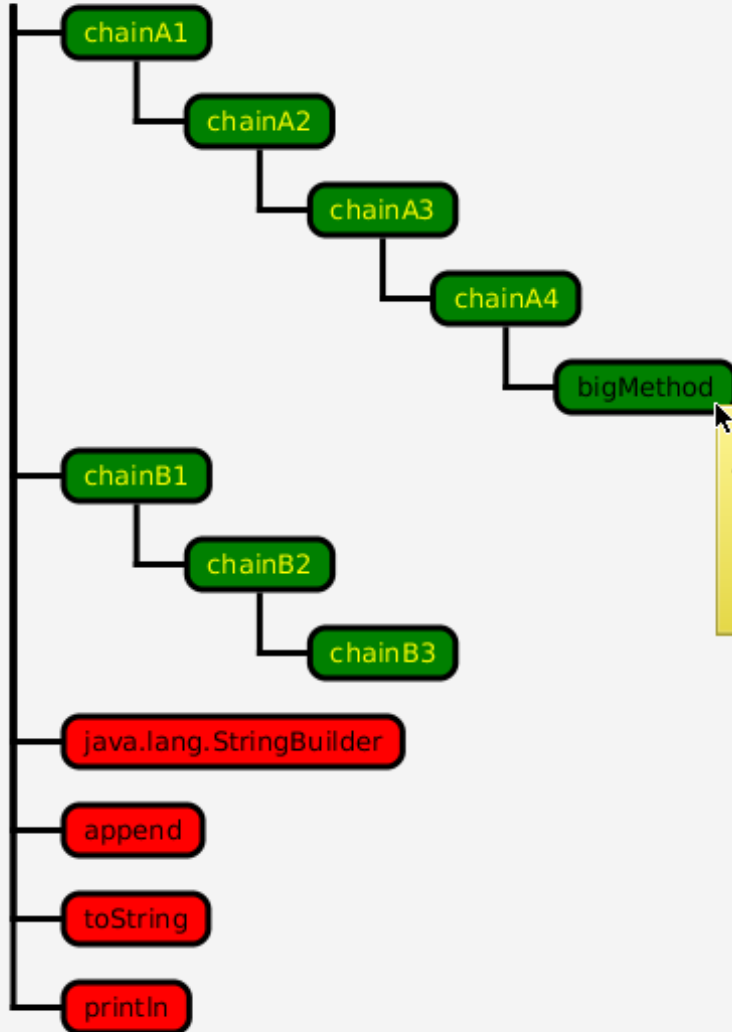
# JVM Spec Browser

The screenshot shows a window titled "JMVS Browser - if\_icmpge". The content is organized into several sections:

- if\_icmp<cond>**: The instruction name, displayed in a grey header bar.
- Operation**: "Branch if int comparison succeeds".
- Format**: A grey box containing the instruction format: `if_icmp<cond>`, `branchbyte1`, and `branchbyte2`.
- Forms**: A list of instruction forms with their corresponding opcodes in hexadecimal:
  - `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**: Shows the stack state as `..., value1, value2 →`.

# Compile Chains

Compile Chain: private void com.chrisnewland.jitwatch.demo.MakeHotSpotLog.testCallChain(long)



private long com.chrisnewland.jitwatch.demo.MakeHotSpotLog.bigMethod(long,int)  
JIT Compiled: Yes  
Inlined: No, hot method too big  
Count: 11076  
iicount: 14562  
Bytes: 350  
Prof factor: 0.465113

# Code Suggestion Tool

Score	Type	Caller	Suggestion
21904	Branch	com.chrisnewland.jitwatch.demo.MakeHotSpotLog private void randomBranchTest(int) <a href="#">View</a>	Method contains an unpredictable branch at bytecode 24 that was observed 43807 times and is taken with probability 0.50129. It may be possible to modify the branch (for example by sorting a collection before iterating) to make it more predictable.
18634	Branch	java.lang.Integer public static Integer valueOf(int) <a href="#">View</a>	Method contains an unpredictable branch at bytecode 3 that was observed 37268 times and is taken with probability 0.498927. It may be possible to modify the branch (for example by sorting a collection before iterating) to make it more predictable.
18002	Branch	java.lang.Integer public static Integer valueOf(int) <a href="#">View</a>	Method contains an unpredictable branch at bytecode 3 that was observed 36004 times and is taken with probability 0.499195. It may be possible to modify the branch (for example by sorting a collection before iterating) to make it more predictable.
12673	Inlining	com.chrisnewland.jitwatch.demo.MakeHotSpotLog private long chainA4(long) <a href="#">View</a>	The call at bytecode 3 to Class: com.chrisnewland.jitwatch.demo.MakeHotSpotLog Member: private long bigMethod(long,int) was not inlined for reason: 'hot method too big' The callee method is 'hot' but is too big to be inlined into the caller. You may want to consider refactoring the callee into smaller methods. Invocations: 12673 Size of callee bytecode: 350
12673	Inlining	com.chrisnewland.jitwatch.demo.MakeHotSpotLog public void tooBigToInline(int) <a href="#">View</a>	The call at bytecode 15 to Class: com.chrisnewland.jitwatch.demo.MakeHotSpotLog Member: private long bigMethod(long,int) was not inlined for reason: 'hot method too big' The callee method is 'hot' but is too big to be inlined into the caller. You may want to consider refactoring the callee into smaller methods.

# 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](https://groups.google.com/jitwatch)
- Twitter
  - @chriswhocodes

Thanks!