# Understanding HotSpot Logs

What can they tell us?

How to create them?

How to interpret them?

A presentation by Chris Newland (chris@chrisnewland.com) on 22/08/2013 at JCrete2013

# hotspot.log

- Queuing and compilation of methods
- XML format – one tag per line
- A few MB each
- Much larger with disassembly enabled

# Why examine HotSpot logs?

- Application starting slowly?
- Are *your* important methods JIT compiled?
- “Warming up” code for first clients?
- Are you smarter than HotSpot?

# How to Create HotSpot Logs

// enables the required diagnostic options

**-XX:+UnlockDiagnosticVMOptions**

// logs queuing and JIT-compilation of methods

**-XX:+LogCompilation**

These switches will create hotspot.log
in your JVM launch directory.

# Want to See the Assembly?

**-XX:+PrintAssembly**

Requires hsdis (HotSpot disassembler) binary.

Part of Oracle debug JVM builds but can be built from source available in OpenJDK.

Instructions:
**http://dropzone.nfshost.com/hsdis.htm**

Copy binary to  JRE's bin\server or bin\client dir.

# Method Queued for Compilation

<task_queued **compile_id='1234'**

method='java/lang/AbstractStringBuilder append (I)Ljava/lang/AbstractStringBuilder;'

bytes='62'

count='750' backedge_count='1'

iicount='0'

**stamp='184.300'**

comment='count'

hot_count='1500'/>

# Detailed Information

<nmethod **compile_id='1234' compiler='C1'**
entry='0x00ca73c0' size='1704' address='0x00ca7288'
relocation_offset='208' insts_offset='312' stub_offset='680'
scopes_data_offset='760' scopes_pcs_offset='1064'
dependencies_offset='1656' nul_chk_table_offset='1660'
oops_offset='744' method='java/lang/AbstractStringBuilder
append (I)Ljava/lang/AbstractStringBuilder;' bytes='62'
count='796' backedge_count='1' iicount='0' **stamp='184.473'**/>

# Compile Task Information

<task **compile_id='1234'** method='java/lang/AbstractStringBuilder append (I)Ljava/lang/AbstractStringBuilder;' bytes='62' count='750' backedge_count='1' iicount='0' **stamp='184.300'**>

*… dependency information*

<task_done success='1' **nmsize='432'** count='796' backedge_count='1' **inlined_bytes='58'** **stamp='184.473'**/>

</task>

# Compile Entry Attributes

- method (name, arg types and return type)
- bytes (bytecode instructions in the method)
- count (number of calls *not accurate*)
- compiler (C1 client or C2 server)
- compile_kind (c2n or osr)
- nmsize (native instruction bytes produced)
- inlined_bytes (bytecodes from other methods)

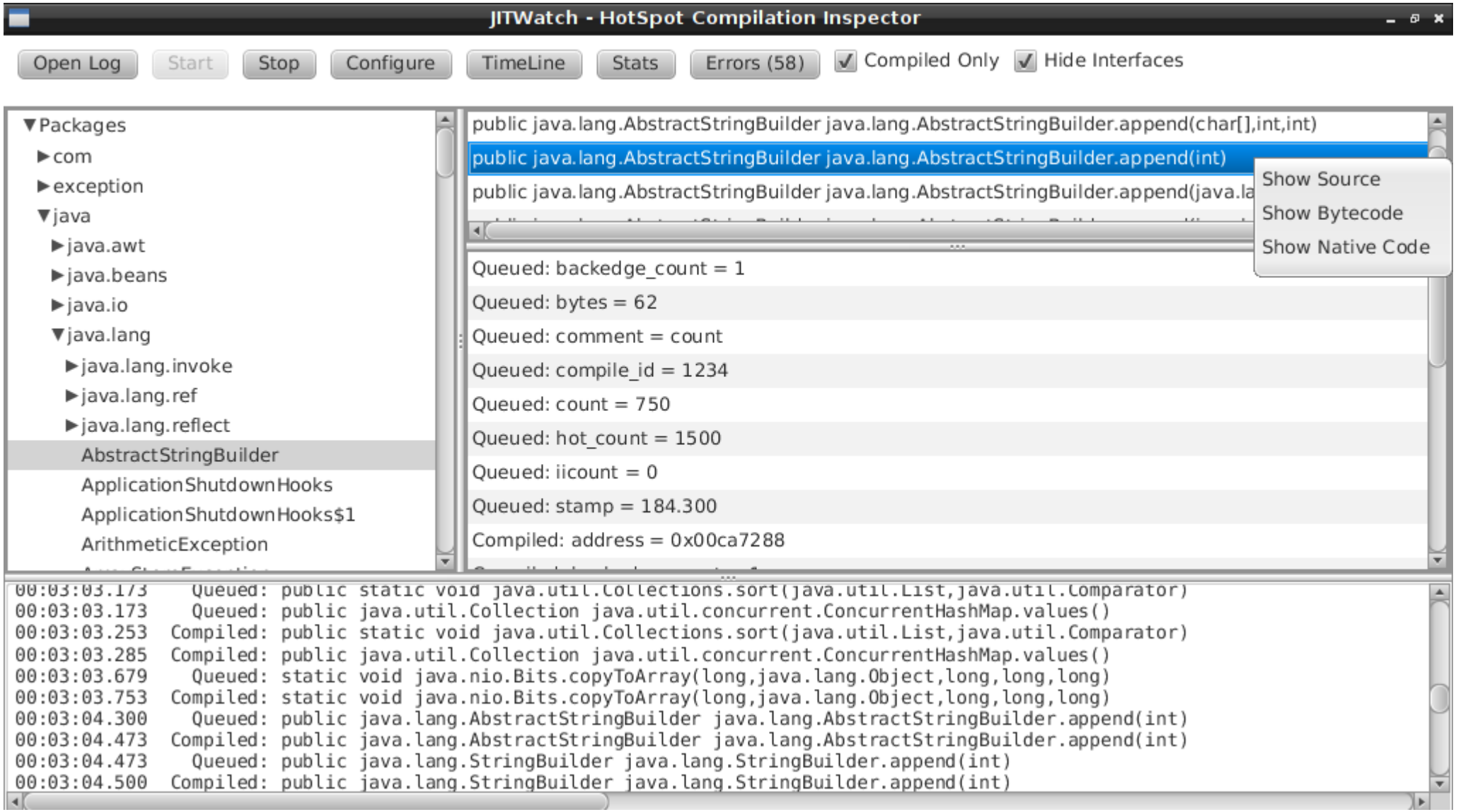
# Getting the Bytecode (javap)

- javap on the command line
- Located in $JDK_HOME/bin/javap
- Usage: javap -c -p <class file>
- Search for the method you want
  - Not a smooth workflow

# Getting the Bytecode (API)

```
String[] args = new String[] { "-c", "-p", "-classpath", cp, clsName };
ByteArrayOutputStream baos = new ByteArrayOutputStream();
try {
com.sun.tools.javap.JavapTask task = new JavapTask();
task.setLog(baos);
task.handleOptions(args);
task.call();
}
catch (BadArgs ba) {}
String bytecode = new String(baos.toByteArray());
```

# Examining a class

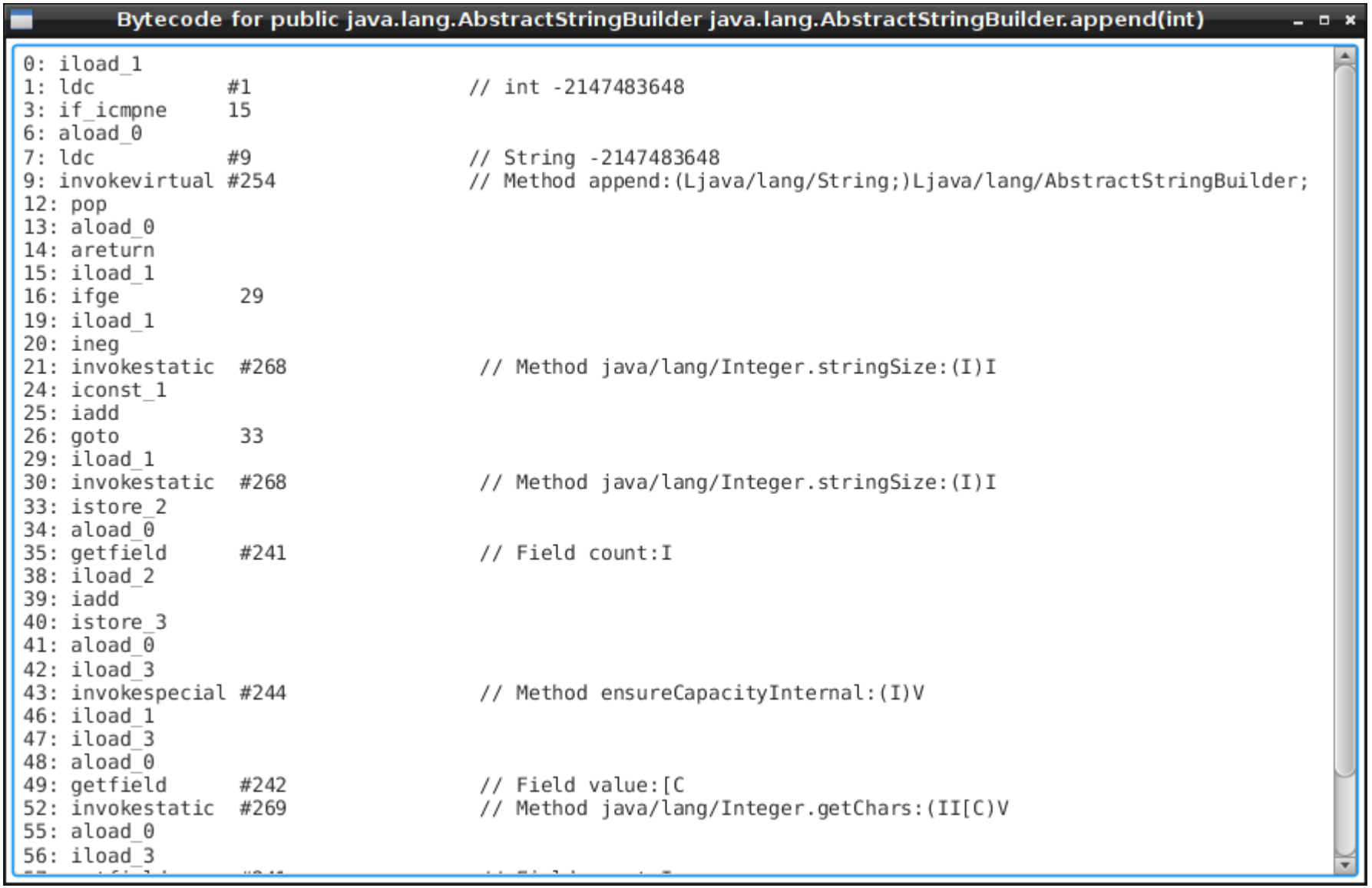


# Source

Source code for java/lang/AbstractStringBuilder.java

```
/**
 * Appends the string representation of the {@code int}
 * argument to this sequence.
 * <p>
 * The overall effect is exactly as if the argument were converted
 * to a string by the method {@link String#valueOf(int)},
 * and the characters of that string were then
 * {@link #append(String) appended} to this character sequence.
 *
 * @param   i   an {@code int}.
 * @return  a reference to this object.
 */
public AbstractStringBuilder append(int i) {
    if (i == Integer.MIN_VALUE) {
        append("-2147483648");
        return this;
    }
    int appendedLength = (i < 0) ? Integer.stringSize(-i) + 1
                                 : Integer.stringSize(i);
    int spaceNeeded = count + appendedLength;
    ensureCapacityInternal(spaceNeeded);
    Integer.getChars(i, spaceNeeded, value);
    count = spaceNeeded;
    return this;
}
```

# Bytecode

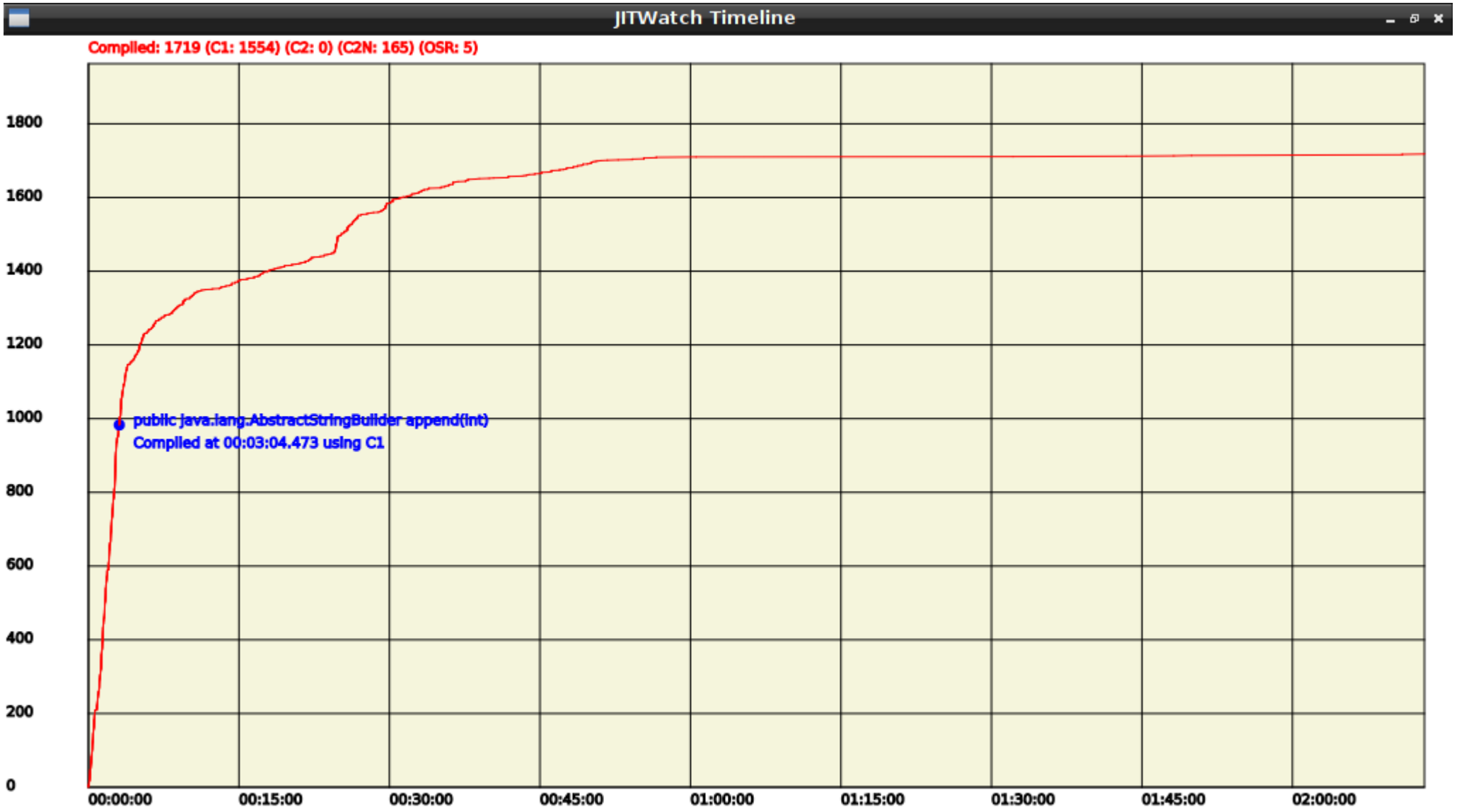

Bytecode for public java.lang.AbstractStringBuilder java.lang.AbstractStringBuilder.append(int)
0: iload_1
1: ldc #1 // int -2147483648
3: if_icmpne 15
6: aload_0
7: ldc #9 // String -2147483648
9: invokevirtual #254 // Method append:(Ljava/lang/String;)Ljava/lang/AbstractStringBuilder;
12: pop
13: aload_0
14: areturn
15: iload_1
16: ifge 29
19: iload_1
20: ineg
21: invokestatic #268 // Method java/lang/Integer.stringSize:(I)I
24: iconst_1
25: iadd
26: goto 33
29: iload_1
30: invokestatic #268 // Method java/lang/Integer.stringSize:(I)I
33: istore_2
34: aload_0
35: getfield #241 // Field count:I
38: iload_2
39: iadd
40: istore_3
41: aload_0
42: iload_3
43: invokespecial #244 // Method ensureCapacityInternal:(I)V
46: iload_1
47: iload_3
48: aload_0
49: getfield #242 // Field value:[C
52: invokestatic #269 // Method java/lang/Integer.getChars:(II[C)V
55: aload_0
56: iload_3

# Assembly

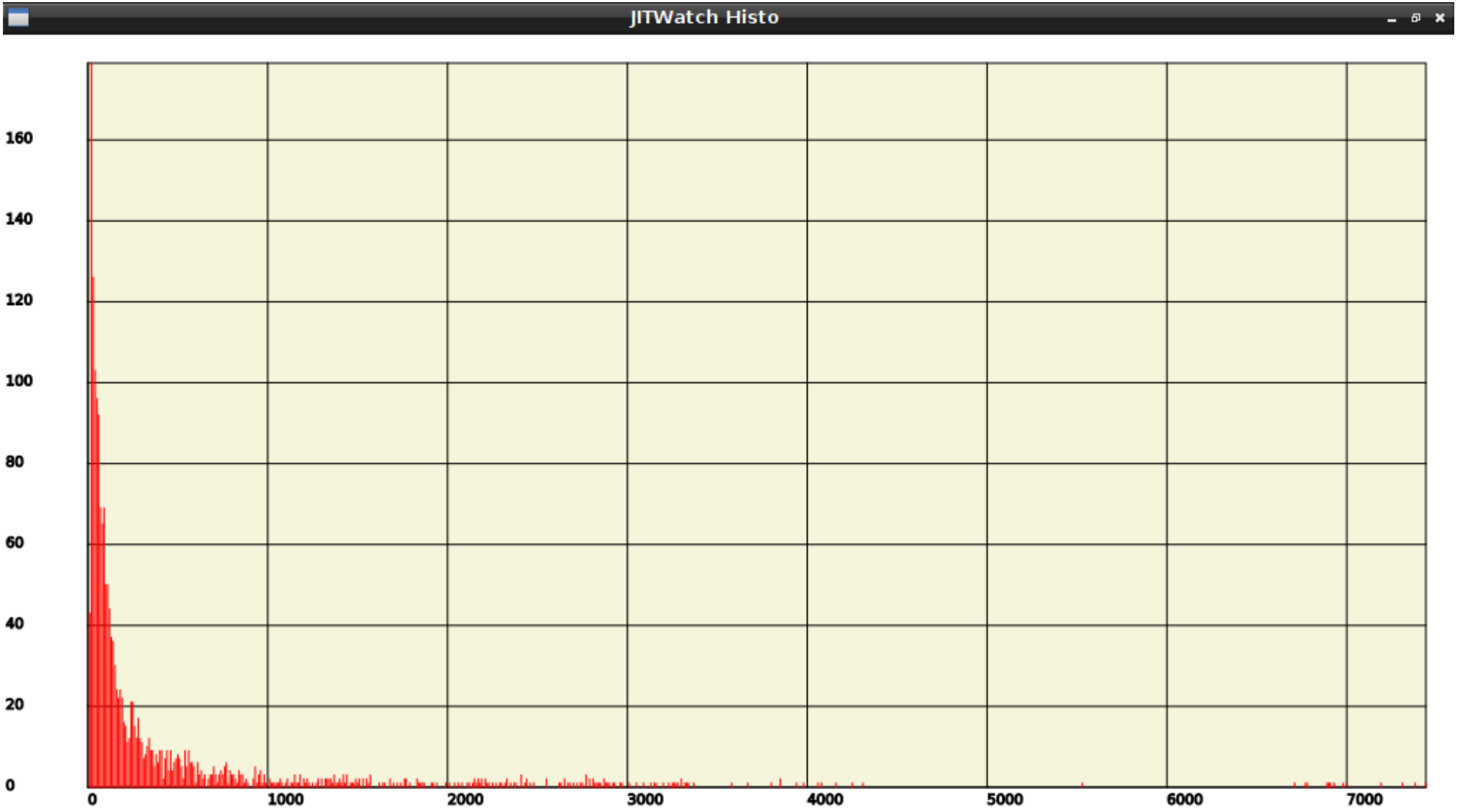
Native code for public java.lang.AbstractString...der java.lang.AbstractStringBuilder.append(int)

```
  # {method} 'append' '(I)Ljava/lang/AbstractStringBuilder;' in 'java/lang/AbstractStringBuilder'
  # this:     ecx       = 'java/lang/AbstractStringBuilder'
  # parm0:    edx       = int
  #           [sp+0x30]  (sp of caller)
  0x00ca73c0: nop
  0x00ca73c1: nop
  0x00ca73c2: nop
  0x00ca73c3: nop
  0x00ca73c4: nop
  0x00ca73c5: nop
  0x00ca73c6: nop
  0x00ca73c7: cmp    0x4(%ecx),%eax
  0x00ca73ca: jne    0x0097d100          ;   {runtime_call}
[Verified Entry Point]
  0x00ca73d0: mov    %eax,-0x4000(%esp)
  0x00ca73d7: push   %ebp
  0x00ca73d8: sub    $0x28,%esp          ;*iload_1
                                         ; - java.lang.AbstractStringBuilder::append@0
  0x00ca73db: mov    %ecx,0x18(%esp)
  0x00ca73df: cmp    $0x80000000,%edx
  0x00ca73e5: je     0x00ca74b2          ;*if_icmpne
                                         ; - java.lang.AbstractStringBuilder::append@3
  0x00ca73eb: cmp    $0x0,%edx
  0x00ca73ee: jge    0x00ca7428          ;*ifge
                                         ; - java.lang.AbstractStringBuilder::append@16
  0x00ca73f4: mov    %edx,%esi
  0x00ca73f6: neg    %esi
  0x00ca73f8: mov    $0x0,%edi
  0x00ca73fd: jmp    0x00ca740b          ;*iload_0
                                         ; - java.lang.Integer::stringSize@2
                                         ; - java.lang.AbstractStringBuilder::append@21
  0x00ca7402: xchg   %ax,%ax
  0x00ca7404: inc    %edi                ; OopMap{ecx=Oop [24]=Oop off=69}
                                         ;*goto
```

# Compilations over Time (C1)



# Compile Times Histogram (C1)



# Some Questions

- How many JIT compilation threads?
- JIT compiler priority?
- Conditions for deleting methods from code cache?
- Is JIT-compiled code worth keeping across JVM instances?
- How much is generated identically each time?
- Anything else important to know?

# Links

- Instructions for building hsdis HotSpot disassembler binary from OpenJDK source
  - **http://dropzone.nfshost.com/hsdis.htm**
- Oracle wiki on HotSpot log files
  - **https://wikis.oracle.com/display/HotSpotInternals/LogCompilation+overview**
- My JITWatch tool for analysing HotSpot log files
  - **http://github.com/chriswhocodes/jitwatch**