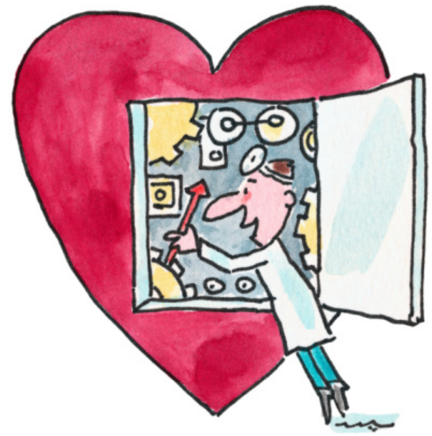


Open Heart Surgery:

Analyzing and Debugging the HotSpot VM at the OS Level

Volker Simonis, SAP

Building a debug version of the JDK

The HotSpot VM is the VM in Oracle's commercial Java SE product and in the OpenJDK.

- Unfortunately Oracle doesn't provide debug versions anymore (that were "only" *fastdebug* versions anyway :)
- But you can easily build it yourself (even on Windows :)

```
> hg clone http://hg.openjdk.java.net/jdk9/dev jdk9-dev
> cd jdk9-dev
> bash get_source.sh
> mkdir ../output-jdk9-dev-dbg && cd ../output-jdk9-dev-dbg
> bash ../jdk9-dev/configure --disable-zip-debug-info --with-debug-level=slowdebug
...
> make images LOG=debug
...
Start 2014-09-24 20:24:53
End   2014-09-24 20:32:54
00:08:01 TOTAL
> ./images/j2sdk-image/bin/java -version
openjdk version "1.9.0-internal-debug"
OpenJDK Runtime Environment (build 1.9.0-internal-debug-simonis_2014_09_23_16_54-b00)
OpenJDK 64-Bit Server VM (build 1.9.0-internal-debug-simonis_2014_09_23_16_54-b00, mixed mode)
```

- The debug-versions knows much more options, traces, and debug output. Use: `-XX:+UnlockExperimentalVMOptions -Xprintflags` or `-XX:+PrintFlagsWithComments` to see them.
- Use `-XX:+PrintFlagsInitial` and `-XX:+PrintFlagsFinal` to see the actual settings

Get the party started

```
public class HelloWorld {
  public static void main(String args[]) {
    System.out.println("Hello World");
  }
}



> javac HelloWorld.java

> javap -c HelloWorld
```

```
public HelloWorld();
  Code:
   0:   aload_0
   1:   invokespecial #1; //Method java/lang/Object."<init>":()V
   4:   return
public static void main(java.lang.String[]);
  Code:
   0:   getstatic     #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   3:   ldc           #3; //String Hello World
   5:   invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   8:   return
```

Let's count the bytecodes the VM executes:

```
> java -XX:+CountBytecodes HelloWorld
Hello World
3128086 bytecodes executed in 1,0s (3,163MHz)
[BytecodeCounter::counter_value = 3128086]
```

Let's see which bytecodes the VM executes:

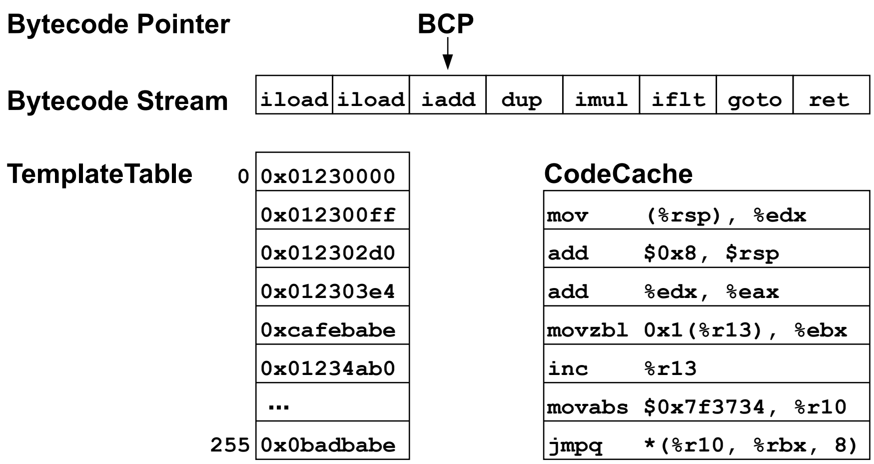
```
> java -XX:+PrintBytecodeHistogram HelloWorld
Hello World
Histogram of 3127972 executed bytecodes:

  absolute  relative  code    name
------------------------------------------
    179266     5,73%    db    fast_aload_0
    158712     5,07%    1b    iload_1
    150836     4,82%    df    fast_iload
    143223     4,58%    84    iinc
...
```

```
> java -XX:+TraceBytecodes HelloWorld

[11931] static void java.lang.Object.<clinit>()
[11931]        1     0  invokestatic 72 <java/lang/Object.registerNatives()V>
[11931]        2     3  return

[11931] static void java.lang.String.<clinit>()
[11931]        3     0  iconst_0
[11931]        4     1  anewarray java/io/ObjectStreamField
[11931]        5     4  putstatic 425 <java/lang/String.serialPersistentFields/[Ljava/io/ObjectStreamField;>
...
```

Bytecode to Assembler - TemplateInterpreter

The HotSpot comes with two interpreters - the Template and the CPP Interpreter

- The Template interpreter is the default and the only officially supported interpreter
- The Template interpreter is much faster than the CPP interpreter (see Template- vs. C++-Interpreter shootout)
- The Template interpreter is generated at VM startup
  - *The VM creates an assembler "template" for each bytecode*
  - *This can be traced with the -XX:+PrintInterpreter option*
  - *Printing the assembly generated by the HotSpot requires the disassembler plugin*



TemplateInterpreter - Demo

Just have a look at the template interpreter. To see the actual disassembly, we need the `hsdis` disassembler library in the library path (it can be easily built from the sources in (`hotspot/src/share/tools/hsdis`) and the GNU binutils).

```
$ export LD_LIBRARY_PATH=/share/OpenJDK/hsdis
$ java -XX:+PrintInterpreter HelloWorld
----------------------------------------------------------------------
Interpreter

code size       =    210K bytes
total space     =   1023K bytes
wasted space    =    813K bytes

# of codelets   =    262
avg codelet size =    824 bytes
...
----------------------------------------------------------------------
iadd  96 iadd  [0x00007f07e5024aa0, 0x00007f07e5024ae0]  64 bytes

  0x00007f07e5024aa0: mov    (%rsp),%eax
  0x00007f07e5024aa3: add    $0x8,%rsp
  0x00007f07e5024aa7: mov    (%rsp),%edx
  0x00007f07e5024aaa: add    $0x8,%rsp
  0x00007f07e5024aae: add    %edx,%eax
  0x00007f07e5024ab0: movzbl 0x1(%r13),%ebx
  0x00007f07e5024ab5: inc    %r13
  0x00007f07e5024ab8: movabs $0x7f07ef8c1540,%r10
  0x00007f07e5024ac2: jmpq   *(%r10,%rbx,8)
  0x00007f07e5024ac6: nop

$ java -XX:+TraceBytecodes HelloWorld | less
```

Grab the address of the `iadd` codelet:

```
gdb java
(gdb) run -XX:+PrintInterpreter HelloWorld | grep iadd
Starting program: /share/OpenJDK/jdk1.7.0_hsx/bin/java -XX:+PrintInterpreter HelloWorld | grep iadd
iadd  96 iadd  [0x00007fffeedeb400, 0x00007fffeedeb440]  64 bytes
During startup program exited normally.
```

Now stop after the Template Interpreter was generated and set a breakpoint in the `iadd` codelet

```
(gdb) b init_globals()
(gdb) run HelloWorld
...
Breakpoint 1, init_globals () at /share/OpenJDK/hsx/hotspot/src/share/vm/runtime/init.cpp:92
(gdb) fin
Run till exit from #0  init_globals () at /share/OpenJDK/hsx/hotspot/src/share/vm/runtime/init.cpp:92
...
(gdb) b *0x00007fffeedeb400
(gdb) c
```

Verify that this is the same assembler code like in the -XX:+PrintInterpreter example

```
Breakpoint 2, 0x00007fffeedeb400 in ?? ()
(gdb) x /12i $pc
=> 0x7fffeedeb400:      mov    (%rsp),%eax
   0x7fffeedeb403:      add    $0x8,%rsp
   0x7fffeedeb407:      mov    (%rsp),%edx
   0x7fffeedeb40a:      add    $0x8,%rsp
   0x7fffeedeb40e:      add    %edx,%eax
   0x7fffeedeb410:      movzbl 0x1(%r13),%ebx
   0x7fffeedeb415:      inc    %r13
   0x7fffeedeb418:      movabs $0x7ffff73fb8e0,%r10
   0x7fffeedeb422:      jmpq   *(%r10,%rbx,8)
   0x7fffeedeb426:      nop
   0x7fffeedeb427:      nop
   0x7fffeedeb428:      int3
```

But notice that gdb has no clue of where we are

```
(gdb) where
#0  0x00007fffeedeb400 in ?? ()
#1  0x0000000000000000 in ?? ()
```

Fortunately, the HotSpot provides some helper functions in the debug build which can be very usefull here:

```
(gdb) call help()
"Executing help"
```

```
basic
  pp(void* p)   - try to make sense of p
  pv(intptr_t p)- ((PrintableResourceObj*) p)->print()
  ps()          - print current thread stack
  pss()         - print all thread stacks
  pm(int pc)    - print Method* given compiled PC
  findm(intptr_t pc) - finds Method*
  find(intptr_t x)   - finds & prints nmethod/stub/bytecode/oop based on pointer into it
  pns(void* sp, void* fp, void* pc)  - print native (i.e. mixed) stack trace. E.g.
                   pns($sp, $rbp, $pc) on Linux/amd64 and Solaris/amd64 or
                   pns($sp, $ebp, $pc) on Linux/x86 or
                   pns($sp, 0, $pc)    on Linux/ppc64 or
                   pns($sp + 0x7ff, 0, $pc) on Solaris/SPARC
                 - in gdb do 'set overload-resolution off' before calling pns()
                 - in dbx do 'frame 1' before calling pns()
misc.
  flush()       - flushes the log file
  events()      - dump events from ring buffers
compiler debugging
  debug()       - to set things up for compiler debugging
  ndebug()      - undo debug
```

Let's try `pns()` first, which will give us a huge, mixed Java/native stack trace:

```
(gdb) call pns($sp, $rbp, $pc)
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
j  java.nio.HeapByteBuffer.ix(I)I+2
j  java.nio.HeapByteBuffer.compact()Ljava/nio/ByteBuffer;+9
j  sun.nio.cs.StreamDecoder.readBytes()I+4
j  sun.nio.cs.StreamDecoder.implRead([CII)I+112
j  sun.nio.cs.StreamDecoder.read([CII)I+180
j  java.io.InputStreamReader.read([CII)I+7
j  java.io.BufferedReader.fill()V+145
j  java.io.BufferedReader.readLine(Z)Ljava/lang/String;+44
j  java.io.BufferedReader.readLine()Ljava/lang/String;+2
j  sun.misc.MetaIndex.registerDirectory(Ljava/io/File;)V+62
j  sun.misc.Launcher$ExtClassLoader$1.run()Lsun/misc/Launcher$ExtClassLoader;+19
j  sun.misc.Launcher$ExtClassLoader$1.run()Ljava/lang/Object;+1
v  ~StubRoutines::call_stub
V  [libjvm.so+0x9b763d]  JavaCalls::call_helper(JavaValue*, methodHandle*, JavaCallArguments*, Thread*)+0x6d9
V  [libjvm.so+0xca00ed]  os::os_exception_wrapper(void (*)(JavaValue*, methodHandle*, JavaCallArguments*, Thread*), JavaValue*, methodHandle*, JavaCallArguments*, Thread*)+0x41
V  [libjvm.so+0x9b6f4d]  JavaCalls::call(JavaValue*, methodHandle, JavaCallArguments*, Thread*)+0x8b
V  [libjvm.so+0xa36ca5]  JVM_DoPrivileged+0x69d
C  [libjava.so+0xc717]  Java_java_security_AccessController_doPrivileged__Ljava_security_PrivilegedExceptionAction_2+0x43
j  java.security.AccessController.doPrivileged(Ljava/security/PrivilegedExceptionAction;)Ljava/lang/Object;+0
j  sun.misc.Launcher$ExtClassLoader.getExtClassLoader()Lsun/misc/Launcher$ExtClassLoader;+12
j  sun.misc.Launcher.()V+4
j  sun.misc.Launcher.()V+15
v  ~StubRoutines::call_stub
V  [libjvm.so+0x9b763d]  JavaCalls::call_helper(JavaValue*, methodHandle*, JavaCallArguments*, Thread*)+0x6d9
V  [libjvm.so+0xca00ed]  os::os_exception_wrapper(void (*)(JavaValue*, methodHandle*, JavaCallArguments*, Thread*), JavaValue*, methodHandle*, JavaCallArguments*, Thread*)+0x41
V  [libjvm.so+0x9b6f4d]  JavaCalls::call(JavaValue*, methodHandle, JavaCallArguments*, Thread*)+0x8b
V  [libjvm.so+0x940be4]  InstanceKlass::call_class_initializer_impl(instanceKlassHandle, Thread*)+0x22e
V  [libjvm.so+0x940910]  InstanceKlass::call_class_initializer(Thread*)+0x5c
V  [libjvm.so+0x93f52b]  InstanceKlass::initialize_impl(instanceKlassHandle, Thread*)+0x83d
V  [libjvm.so+0x93dc5b]  InstanceKlass::initialize(Thread*)+0x91
V  [libjvm.so+0xb32f9d]  LinkResolver::resolve_static_call(CallInfo&, KlassHandle&, Symbol*, Symbol*, KlassHandle, bool, bool, Thread*)+0x179
V  [libjvm.so+0xb371d0]  LinkResolver::resolve_invokestatic(CallInfo&, constantPoolHandle, int, Thread*)+0xec
V  [libjvm.so+0xb36e02]  LinkResolver::resolve_invoke(CallInfo&, Handle, constantPoolHandle, int, Bytecodes::Code, Thread*)+0x8c
V  [libjvm.so+0x9a92fc]  InterpreterRuntime::resolve_invoke(JavaThread*, Bytecodes::Code)+0x438
j  java.lang.ClassLoader.initSystemClassLoader()V+22
j  java.lang.ClassLoader.getSystemClassLoader()Ljava/lang/ClassLoader;+0
v  ~StubRoutines::call_stub
V  [libjvm.so+0x9b763d]  JavaCalls::call_helper(JavaValue*, methodHandle*, JavaCallArguments*, Thread*)+0x6d9
V  [libjvm.so+0xca00ed]  os::os_exception_wrapper(void (*)(JavaValue*, methodHandle*, JavaCallArguments*, Thread*), JavaValue*, methodHandle*, JavaCallArguments*, Thread*)+0x41
V  [libjvm.so+0x9b6f4d]  JavaCalls::call(JavaValue*, methodHandle, JavaCallArguments*, Thread*)+0x8b
V  [libjvm.so+0x9b6ba1]  JavaCalls::call_static(JavaValue*, KlassHandle, Symbol*, Symbol*, JavaCallArguments*, Thread*)+0x14b
V  [libjvm.so+0x9b6cbd]  JavaCalls::call_static(JavaValue*, KlassHandle, Symbol*, Symbol*, Thread*)+0x9d
V  [libjvm.so+0xdf4c38]  SystemDictionary::compute_java_system_loader(Thread*)+0x90
V  [libjvm.so+0xe331b4]  Threads::create_vm(JavaVMInitArgs*, bool*)+0x57e
V  [libjvm.so+0xa00096]  JNI_CreateJavaVM+0xce
C  [libjli.so+0xa50d]  InitializeJVM+0x13b
C  [libjli.so+0x8044]  JavaMain+0xd3
C  [libpthread.so.0+0x8182]  start_thread+0xc2
```

You can now verify that the stack is the same like for the first occurence of `iadd` we see with -XX:+TraceBytecodes (actually it isn't, it's the stack at the second occurence - see below why)

The Template Interpreter maintains a Top Of Stack Cache (TOSCA) in a register. Some bytecodes leave their result in this register to avoid writing it to the stack and other bytecodes can potentially consume the top of stack value directly from that register without reading it from the stack. Therefore there's actually not just one *active* active template dispatch table, but a an array of dispatch tables, one for each type of TOSCA output state.

The template codelets have different entry points, based on the TOSCA state of the previous bytecode. It is therefore not enough to put a breakpoint only on the first instruction of a bytecode. If we want to stop at every occurence of a bytecode, we have to put breakpoints at every entry point.

```
(gdb) p /d 'Bytecodes::_iadd'
$23 = 96
((gdb) p 'TemplateTable::_template_table'['Bytecodes::_iadd']
$24 = {_flags = 0, _tos_in = itos, _tos_out = itos, _gen = 0x7ffff6b90f7c , _arg = 0}
(gdb) p 'TemplateTable::_template_table'['Bytecodes::_getfield']
$25 = {_flags = 5, _tos_in = vtos, _tos_out = vtos, _gen = 0x7ffff6b98404 , _arg = 1}
(gdb) p 'TemplateTable::_template_table'['Bytecodes::_iconst_1']
$26 = {_flags = 0, _tos_in = vtos, _tos_out = itos, _gen = 0x7ffff6b8b8f8 , _arg = 1}
```

With our `iadd` example, it happend that the very first occurence of an `iadd` bytecode happened after a `iconst_1` bytecode which has a TOSCA out state of *itos* (i.e. it places its result right into the TOSCA register). So after the `iconst_1` codelet, the interpreter dispatches right to the second entry point of the `iadd` codelet bacause it only has to load the second paramter from the stack (and we didn't stop at our breakpoint!).

Let's see what GDB thinks about the address we want to jump at the end of the codelet:

```
(gdb) x /12i $pc
=> 0x7fffeedeb400:      mov    (%rsp),%eax
   0x7fffeedeb403:      add    $0x8,%rsp
   0x7fffeedeb407:      mov    (%rsp),%edx
   0x7fffeedeb40a:      add    $0x8,%rsp
   0x7fffeedeb40e:      add    %edx,%eax
   0x7fffeedeb410:      movzbl 0x1(%r13),%ebx
   0x7fffeedeb415:      inc    %r13
   0x7fffeedeb418:      movabs $0x7ffff73fb8e0,%r10
   0x7fffeedeb422:      jmpq   *(%r10,%rbx,8)
   0x7fffeedeb426:      nop
   0x7fffeedeb427:      nop
   0x7fffeedeb428:      int3
(gdb) info symbol(0x7ffff73fb8e0)
TemplateInterpreter::_active_table + 6144 in section .bss of /share/OpenJDK/jdk1.7.0_hsx/jre/lib/amd64/server/libjvm.so
```

It's actually computed from the active Bytecode Table!

And how is it computed (i.e. what is `$r13`)?

```
(gdb) call find($r13)
"Executing find"
0x00000000bce065fe is an oop
{constMethod}
 - klass: {other class}
 - method:       0x00000000bce06608 {method} 'charAt' '(I)C' in 'java/lang/String'
 - exceptions:   0x00000000bce01c10
bci_from(0xbce065fe) = 30; print_codes():
0 iload_1
1 iflt 12
4 iload_1
5 aload_0
6 fast_igetfield 395
9 if_icmplt 21
12 new 234
15 dup
16 iload_1
17 invokespecial 476 (I)V>
20 athrow
21 aload_0
22 fast_agetfield 398
25 iload_1
26 aload_0
27 fast_igetfield 397
30 iadd
31 caload
32 ireturn

(gdb) x /2b $r13
0xbce065fe:     0x60    0x34
```

Unfortunately, `call find($r13)` won't return the method anymore after the PermGen removal. It only recognizes that `$r13` now points into the MetaSpace.

So `$r13` is actually the Bytecode Pointer (BCP). It points to `0x60` which is the bytecode for `iadd`. The next byte is `0x34` which is the bytecode for `caload`.

We can now easily compute the address of the next codelet manually and verify that's the right one (the one for `caload`)

```
(gdb) p ((void**)0x7ffff73fb8e0)[0x34]
$6 = (void *) 0x7fffeede9cc7
(gdb) call find(0x7fffeede9cc7)
"Executing find"
0x00007fffeede9cc7 is an Interpreter codelet
caload  52 caload  [0x00007fffeede9cc0, 0x00007fffeede9d00]  64 bytes
```

There are other/better ways to stop at a specific bytecode: `-XX:StopInterpreterAt=116`

```
(gdb) b breakpoint
(gdb) run -XX:StopInterpreterAt=174 HelloWorld
(gdb) where
#0  breakpoint () at /share/OpenJDK/hsx/hotspot/src/os/linux/vm/os_linux.cpp:518
#1  0x00007ffff6a4a147 in os::breakpoint () at /share/OpenJDK/hsx/hotspot/src/os/linux/vm/os_linux.cpp:513
#2  0x00007fffeedebf02 in ?? ()
#3  0x000000000000001f in ?? ()
#4  0x00000000eb564790 in ?? ()
#5  0x00007ffff7fe33c8 in ?? ()
...
(gdb) call mixed_ps($sp, $rbp, $pc)
"Executing mixed_ps"
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
V  [libjvm.so+0x96c14d]  breakpoint+0x4
V  [libjvm.so+0x96c147]  os::breakpoint()+0x9
j  java.lang.String.charAt(I)C+27
j  java.security.BasicPermission.init(Ljava/lang/String;)V+37
j  java.security.BasicPermission.(Ljava/lang/String;)V+7
j  java.lang.RuntimePermission.(Ljava/lang/String;)V+2
j  java.lang.Thread.()V+16
v  ~StubRoutines::call_stub
V  [libjvm.so+0x7305d8]  JavaCalls::call_helper(JavaValue*, methodHandle*, JavaCallArguments*, Thread*)+0x51a
V  [libjvm.so+0x9749ee]  os::os_exception_wrapper(void (*)(JavaValue*, methodHandle*, JavaCallArguments*, Thread*), JavaValue*, methodHandle*, JavaCallArguments*, Thread*)+0x3a
V  [libjvm.so+0x7300b7]  JavaCalls::call(JavaValue*, methodHandle, JavaCallArguments*, Thread*)+0x7d
V  [libjvm.so+0x6cae31]  instanceKlass::call_class_initializer_impl(instanceKlassHandle, Thread*)+0x199
V  [libjvm.so+0x6cac29]  instanceKlass::call_class_initializer(Thread*)+0x45
V  [libjvm.so+0x6c9ab9]  instanceKlass::initialize_impl(instanceKlassHandle, Thread*)+0x57d
V  [libjvm.so+0x6c8920]  instanceKlass::initialize(Thread*)+0x74
V  [libjvm.so+0xac5adf]  initialize_class(Symbol*, Thread*)+0x5b
V  [libjvm.so+0xacd2ba]  Threads::create_vm(JavaVMInitArgs*, bool*)+0xa08
V  [libjvm.so+0x76fe83]  JNI_CreateJavaVM+0xb0
C  [libjli.so+0x31b7]  JavaMain+0x97

(gdb) up
(gdb) up
(gdb) call find($pc)
"Executing find"
0x00007fffeedebf02 is an Interpreter codelet
iadd  96 iadd  [0x00007fffeedebee0, 0x00007fffeedebf40]  96 bytes

(gdb) x /16i 0x00007fffeedebee0
   0x7fffeedebee0:      mov    (%rsp),%eax
   0x7fffeedebee3:      add    $0x8,%rsp
   0x7fffeedebee7:      incl   0x85a9713(%rip)        # 0x7ffff7395600 <_ZN15BytecodeCounter14_counter_valueE>
   0x7fffeedebeed:      cmpl   $0xae,0x85a9709(%rip)        # 0x7ffff7395600 <_ZN15BytecodeCounter14_counter_valueE>
   0x7fffeedebef7:      jne    0x7fffeedebf02
   0x7fffeedebefd:      callq  0x7ffff6a4a13e <_ZN2os10breakpointEv>
=> 0x7fffeedebf02:      mov    (%rsp),%edx
   0x7fffeedebf05:      add    $0x8,%rsp
   0x7fffeedebf09:      add    %edx,%eax
   0x7fffeedebf0b:      movzbl 0x1(%r13),%ebx
   0x7fffeedebf10:      inc    %r13
   0x7fffeedebf13:      movabs $0x7ffff73fb8e0,%r10
   0x7fffeedebf1d:      jmpq   *(%r10,%rbx,8)
   0x7fffeedebf21:      nop
   0x7fffeedebf22:      nop
   0x7fffeedebf23:      nop
(gdb) info symbol 0x7ffff7395600
BytecodeCounter::_counter_value in section .bss of /share/OpenJDK/jdk1.7.0_hsx/jre/lib/amd64/server/libjvm.so
```

But as you can see, the `iadd` interpreter codelet looks different now, because the interpreter had to generate extra code for the -XX:StopInterpreterAt feature.

Bytecode to Assembler - JIT Compiler

The HotSpot comes with two interpreters - the C1 client and the C2 server compiler

- C1 is optimised for compilation speed while C2 is optimised for maximal performance of the generated code
- In the past, the VM could only use one JIT compiler (*client* vs. *server* VM)
- Nowadays, the VM can used both compilers with -XX:+TieredCompilation

The C2 server compiler is a high-end fully optimising compiler:

- Null and range check elimination, implicit null checks
- Sophisticated deep inlining based on profiling (use -XX:+PrintInlining to see it)
- Dead code and common subexpression elimination
- Loop unrolloing and invariant hoisting
- Use -XX:+PrintOptoAssembly / -XX:+PrintAssembly to see the compiler output
  Requires the `hsdis` library (`hotspot/src/share/tools/hsdis`) and the GNU binutils

Notice that you can usually not debug/profile fully optimized C2 code with Java debuggers/profilers

- To do this, you need a C debugger (e.g. gdb) and a system profiler (e.g. VTune, oprofile)

JIT Compiler Demo - Hunting a Bug

In this demo I will show how to find and fix a HotSpot bug. As example I'll take a real bug (JDK-8011901 which was introduced in Java 8 but luckily fixed before the first release.

To make the following examples easier to run we set some global Java options with the help of the `_JAVA_OPTIONS` environment variable. We also set `LD_LIBRARY_PATH` to point to the location where we've placed the `hsdis-amd64.so` library.

```
$ export _JAVA_OPTIONS="-XX:-TieredCompilation -XX:-UseOnStackReplacement -XX:CICompilerCount=1 -XX:-UseCompressedOops -Xbatch"
$ export LD_LIBRARY_PATH=/share/OpenJDK/hsdis
```

So lets start with the following simple Java program:

```java
import java.util.concurrent.atomic.AtomicLong;

public class AtomicLongTest {

  static AtomicLong al = new AtomicLong(0);
  static long count, l;
  static final long f = 4_294_967_296L;

  static void update() {
    l += f;
    al.addAndGet(f);

    if (l == al.longValue()) {
      printOK();
    }
    else {
      printError();
    }
  }

  static public void main(String args[]) {
    for (count = 0; count < Integer.parseInt(args[0]); count++) {
      update();
    }
  }
  static boolean ok = false, error = false;

  static void printOK() {
    if (!ok) {
      System.out.println("OK (iteration " + count +")");
      ok = true;
      error = false;
    }
  }

  static void printError() {
    if (!error) {
      System.out.println("Error (iteration " + count + ", " + l + " != " + al.longValue());
      error = true;
      ok = false;
    }
    l = al.longValue(); // make them equal again
  }
}
```

.. which increments both, a `long` and an `AtomicLong` variable at the same time.
We would expect that condition '`(l == al.longValue())`' is always true:

```
$ java AtomicLongTest 10000
OK (iteration 1)
$ java AtomicLongTest 10001
OK (iteration 1)
Error (iteration 10001, 42953967927296 != 42949672960000
$ java AtomicLongTest 10002
OK (iteration 1)
Error (iteration 10001, 42953967927296 != 42949672960000
OK (iteration 10002)
$ java AtomicLongTest 20000
OK (iteration 1)
Error (iteration 10001, 42953967927296 != 42949672960000
OK (iteration 10002)
Error (iteration 20000, 85895050952704 != 85890755985408
```

The result is strange: the condition fails one time, in iteration 10.001, and it always fails if we iterate more than 20.000 times!
Let's try to debug the program to see what happens:

```
$ jdb -launch AtomicLongTest 20000
...
main[1] stop in AtomicLongTest:17
main[1] cont
> Set deferred breakpoint AtomicLongTest:17
OK (iteration 1)
```

```
The application exited
```

So that didn't worked out quite well. It seems the bug isn't triggered when we're in the debugger. Let's try to attach when we're already in an error state:

```
$ java -agentlib:jdwp=transport=dt_socket,address=8000,\
       server=y,suspend=n AtomicLongTest 200000000
Listening for transport dt_socket at address: 8000
OK (iteration 1)
Error (iteration 10001, 42953967927296 != 42949672960000
OK (iteration 10002)
Error (iteration 20000, 85895050952704 != 85890755985408

OK (iteration 138809530)  <===
Error (iteration 138814529, 107365592465408 != 107361297498112
```

```
$ jdb -attach 8000
...
> stop in AtomicLongTest:13
Set breakpoint AtomicLongTest:13
>
Breakpoint hit: "thread=main", AtomicLongTest.update(), line=13 bci=20
13        if (l == al.longValue()) {

main[1] print AtomicLongTest.l
 AtomicLongTest.l = 85895050952704
main[1] print AtomicLongTest.al
 AtomicLongTest.al = "85895050952704"
main[1] print AtomicLongTest.count
 AtomicLongTest.count = 138809530  <===
main[1] clear AtomicLongTest:13
Removed: breakpoint AtomicLongTest:13
main[1] cont
>

The application exited
```

So a Java debugger isn't of much help here. Maybe this is related to JIT compilation (notice that Java debuggers usually can not debug or set breakpoints in JIT-compiled methods). Let's see:

```
$ java -XX:+PrintCompilation AtomicLongTest 20000 | egrep "^OK|^Error|update"

OK (iteration 1)
   1145   49    b        AtomicLongTest::update (43 bytes)
   1169   49             AtomicLongTest::update (43 bytes)   made not entrant
Error (iteration 10001, 42953967927296 != 42949672960000
OK (iteration 10002)
   1173   56    b        AtomicLongTest::update (43 bytes)
Error (iteration 20000, 85895050952704 != 85890755985408
```

So it seems that every time after the JIT-compilation of `AtomicLongTest::update()` we get an error! Let's try a real debugger:)

```
$ gdb java
...
(gdb) break breakpoint
Breakpoint 1 at 0x7ffff68a3bfe: file /share/OpenJDK/jdk9-dev/hotspot/src/os/linux/vm/os_linux.cpp, line 457.
(gdb) run -XX:CompileCommand="break AtomicLongTest.update" AtomicLongTest 20000
...
OK (iteration 1)
### Breaking when compiling:  AtomicLongTest::update
[Switching to Thread 0x7fff9e924700 (LWP 10742)]

Breakpoint 1, breakpoint ()
    at /share/OpenJDK/jdk9-dev/hotspot/src/os/linux/vm/os_linux.cpp:457
457     extern "C" void breakpoint() {
(gdb) where
#0  breakpoint () at /share/OpenJDK/jdk9-dev/hotspot/src/os/linux/vm/os_linux.cpp:457
#1  0x00007ffff62dd8d9 in Compile::print_compile_messages (this=0x7fff9e922c80)
    at /share/OpenJDK/jdk9-dev/hotspot/src/share/vm/opto/compile.cpp:500
#2  0x00007ffff62de822 in Compile::Compile (this=0x7fff9e922c80, ci_env=0x7fff9e923940,
    compiler=0x7ffff01266e0, target=0x7ffff014f190, osr_bci=-1, subsume_loads=true,
    do_escape_analysis=true, eliminate_boxing=true)
    at /share/OpenJDK/jdk9-dev/hotspot/src/share/vm/opto/compile.cpp:709
#3  0x00007ffff61ea0d6 in C2Compiler::compile_method (this=0x7ffff01266e0,
    env=0x7fff9e923940, target=0x7ffff014f190, entry_bci=-1)
    at /share/OpenJDK/jdk9-dev/hotspot/src/share/vm/opto/c2compiler.cpp:101
#4  0x00007ffff62fb174 in CompileBroker::invoke_compiler_on_method (task=0x7ffff01817b0)
    at /share/OpenJDK/jdk9-dev/hotspot/src/share/vm/compiler/compileBroker.cpp:1974
#5  0x00007ffff62fa69c in CompileBroker::compiler_thread_loop ()
    at /share/OpenJDK/jdk9-dev/hotspot/src/share/vm/compiler/compileBroker.cpp:1796
#6  0x00007ffff6a414df in compiler_thread_entry (thread=0x7ffff0137800,
    __the_thread__=0x7ffff0137800)
    at /share/OpenJDK/jdk9-dev/hotspot/src/share/vm/runtime/thread.cpp:3164
#7  0x00007ffff6a3c387 in JavaThread::thread_main_inner (this=0x7ffff0137800)
    at /share/OpenJDK/jdk9-dev/hotspot/src/share/vm/runtime/thread.cpp:1678
#8  0x00007ffff6a3c232 in JavaThread::run (this=0x7ffff0137800)
    at /share/OpenJDK/jdk9-dev/hotspot/src/share/vm/runtime/thread.cpp:1658
#9  0x00007ffff68a4823 in java_start (thread=0x7ffff0137800)
    at /share/OpenJDK/jdk9-dev/hotspot/src/os/linux/vm/os_linux.cpp:830
#10 0x00007ffff73dd182 in start_thread (arg=0x7fff9e924700) at pthread_create.c:312
#11 0x00007ffff78f1fbd in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:111
(gdb) cont
Continuing.
...
```

```
(gdb) where
#0  breakpoint () at /share/OpenJDK/jdk9-dev/hotspot/src/os/linux/vm/os_linux.cpp:457
#1  0x00007ffff62e4c6a in Compile::Optimize (this=0x7fff9e922c80)
    at /share/OpenJDK/jdk9-dev/hotspot/src/share/vm/opto/compile.cpp:2041
#2  0x00007ffff62df09e in Compile::Compile (this=0x7fff9e922c80, ci_env=0x7fff9e923940,
    compiler=0x7ffff01266e0, target=0x7ffff014f190, osr_bci=-1, subsume_loads=true,
    do_escape_analysis=true, eliminate_boxing=true)
    at /share/OpenJDK/jdk9-dev/hotspot/src/share/vm/opto/compile.cpp:837
...
(gdb) cont
Continuing.
...
(gdb) where
#0  breakpoint () at /share/OpenJDK/jdk9-dev/hotspot/src/os/linux/vm/os_linux.cpp:457
#1  0x00007ffff68a3be0 in os::breakpoint ()
    at /share/OpenJDK/jdk9-dev/hotspot/src/os/linux/vm/os_linux.cpp:454
#2  0x00007fffed1a9671 in ?? ()
#3  0x00007fffacd5c1b0 in ?? ()
```

We use the `-XX:CompileCommand` option to instruct the VM to stop when a certain method is compiled AND at the beginning of its execution. For this option to have any effect, we also need to set a breakpoint in the helper function `breakpoint()` whithin the VM.

The first two times, the debugger stops at the beginning of the method compilation (at `Compile::Compile()`) and when the VM starts the optimzation passes (at `Compile::Optimze()`. The third time it stops, when the compiled method is ionvoked for the first time. Obviously, the debugger doesn't know where we are, because he has no debugging information for generated code. Fortunately we can use the various HotSpot debugging helpers to find out what's going on:

```
(gdb) call pns($sp, $rbp, $pc)

"Executing pns"
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
V  [libjvm.so+0xc94bfe]  breakpoint+0x8
V  [libjvm.so+0xc94be0]  os::breakpoint()+0x1c
J 49 C2 AtomicLongTest.update()V (43 bytes) @ 0x00007fffed1a9671 [0x00007fffed1a9660+0x11]
j  AtomicLongTest.main([Ljava/lang/String;)V+18
v  ~StubRoutines::call_stub
V  [libjvm.so+0x9b763d]  JavaCalls::call_helper(JavaValue*, methodHandle*, JavaCallArguments*, Thread*)+0x6d9
V  [libjvm.so+0xca00ed]  os::os_exception_wrapper(void (*)(JavaValue*, methodHandle*, JavaCallArguments*, Thread*), JavaValue*, methodHandle*, JavaCallArguments*, Thread*)+0x41
V  [libjvm.so+0x9b6f4d]  JavaCalls::call(JavaValue*, methodHandle, JavaCallArguments*, Thread*)+0x8b
V  [libjvm.so+0x9cfdb1]  jni_invoke_static(JNIEnv_*, JavaValue*, _jobject*, JNICallType, _jmethodID*, JNI_ArgumentPusher*, Thread*)+0x205
V  [libjvm.so+0x9e72df]  jni_CallStaticVoidMethod+0x358
C  [libjli.so+0x8820]  JavaMain+0x8af
C  [libpthread.so.0+0x8182]  start_thread+0xc2

(gdb) up
#1  0x00007ffff68a3be0 in os::breakpoint ()
    at /share/OpenJDK/jdk9-dev/hotspot/src/os/linux/vm/os_linux.cpp:454
454       BREAKPOINT;
(gdb) up
#2  0x00007fffed1a9671 in ?? ()
(gdb) call find($pc)

"Executing find"
0x00007fffed1a9671 is at entry_point+17 in (nmethod*)0x00007fffed1a9510
Compiled method (c2) 8089115   49             AtomicLongTest::update (43 bytes)
 total in heap  [0x00007fffed1a9510,0x00007fffed1a9960] = 1104
 relocation     [0x00007fffed1a9640,0x00007fffed1a9660] = 32
 main code      [0x00007fffed1a9660,0x00007fffed1a9740] = 224
 stub code      [0x00007fffed1a9740,0x00007fffed1a9758] = 24
 oops           [0x00007fffed1a9758,0x00007fffed1a9760] = 8
 metadata       [0x00007fffed1a9760,0x00007fffed1a9788] = 40
 scopes data    [0x00007fffed1a9788,0x00007fffed1a9808] = 128
 scopes pcs     [0x00007fffed1a9808,0x00007fffed1a9948] = 320
 dependencies   [0x00007fffed1a9948,0x00007fffed1a9950] = 8
 nul chk table  [0x00007fffed1a9950,0x00007fffed1a9960] = 16
```

So we're indeed in the code the JIT compiler has generated for `AtomicLongTest::update()`. Let's see how it looks like:

```
(gdb) x /12i $pc
=> 0x7fffed1a9671:      movabs $0x7fffd878f8a8,%r10 // AtomicLongTest Class
   0x7fffed1a967b:      mov    0xa0(%r10),%r11      // AtomicLongTest.al
   0x7fffed1a9682:      movabs $0x100000000,%r8     // copy g to r8
   0x7fffed1a968c:      add    %r8,0xb0(%r10)       // AtomicLongTest.l += 4_294_967_296L
   0x7fffed1a9693:      test   %r11,%r11            // Null-check for AtomicLongTest.al
   0x7fffed1a9696:      je     0x7fffed1a96cd
   0x7fffed1a9698:      lock addq $0x0,0x10(%r11)   // AtomicLongTest.al.value += 0
   0x7fffed1a969e:      mov    0xa0(%r10),%r11      // reload AtomicLongTest.al
   0x7fffed1a96a5:      mov    0x10(%r11),%r11      // reload AtomicLongTest.al.value
   0x7fffed1a96a9:      mov    0xb0(%r10),%r8       // copy AtomicLongTest.l to r8
   0x7fffed1a96b0:      cmp    %r11,%r8             // (l == al.longValue())
   0x7fffed1a96b3:      jne    0x7fffed1a96dd

(gdb) call find(0x7fffd878f8a8)
"Executing find"
0x00007fffd878f8a8 is an oop
```

```
java.lang.Class
 - klass: 'java/lang/Class'
 - ---- fields (total size 25 words):
 - private volatile transient strict 'cachedConstructor' 'Ljava/lang/reflect/Constructor;' @16  NULL (0 0)
 - private volatile transient strict 'newInstanceCallerCache' 'Ljava/lang/Class;' @24  NULL (0 0)
 - private transient 'name' 'Ljava/lang/String;' @32  "AtomicLongTest" (d878f988 7fff)  <===
 - private final 'classLoader' 'Ljava/lang/ClassLoader;' @40  a 'sun/misc/Launcher$AppClassLoader' (d865ebd8 7fff)
 - private final strict 'componentType' 'Ljava/lang/Class;' @48  NULL (0 0)
 - private volatile transient strict 'reflectionData' 'Ljava/lang/ref/SoftReference;' @56  a 'java/lang/ref/SoftReference' (d878fbc0 7fff)
 - private volatile transient 'genericInfo' 'Lsun/reflect/generics/repository/ClassRepository;' @64  NULL (0 0)
 - private volatile transient strict 'enumConstants' '[Ljava/lang/Object;' @72  NULL (0 0)
 - private volatile transient strict 'enumConstantDirectory' 'Ljava/util/Map;' @80  NULL (0 0)
 - private volatile transient 'annotationData' 'Ljava/lang/Class$AnnotationData;' @88  NULL (0 0)
 - private volatile transient 'annotationType' 'Lsun/reflect/annotation/AnnotationType;' @96  NULL (0 0)
 - transient 'classValueMap' 'Ljava/lang/ClassValue$ClassValueMap;' @104  NULL (0 0)
 - private volatile transient 'classRedefinedCount' 'I' @144  0
 - signature: LAtomicLongTest;
 - fake entry for mirror: 'AtomicLongTest'
 - fake entry for array: NULL
 - fake entry for oop_size: 25
 - fake entry for static_oop_field_count: 1
 - static 'al' 'Ljava/util/concurrent/atomic/AtomicLong;' @160  a 'java/util/concurrent/atomic/AtomicLong' (d87901f8 7fff)  <===
 - static 'count' 'J' @168  10001 (2711 0)
 - static 'l' 'J' @176  42949672960000 (0 2710)  <===
 - static final 'f' 'J' @184  4294967296 (0 1)
 - static 'ok' 'Z' @192  true
 - static 'error' 'Z' @193  false
```

If we step instruction-wise till the compare we can verify, that `AtomicLongTest.l` and the long value of `AtomicLongTest.al` do indeed differ.

```
(gdb) print $r11
$5 = 42949672960000
(gdb) print $r8
$6 = 42953967927296
```

And if we continue the exection, we'll get the error printed out - however just for a single iteration:

```
(gdb) cont
Continuing.
Error (iteration 10001, 42953967927296 != 42949672960000
OK (iteration 10002)
```

For some reason we seem to have been just one time in the JIT compiled method (i.e. we don't stop at the breakpoint anymore. Instead, the method seems to be recompiled again:

```
### Breaking when compiling:  AtomicLongTest::update
[Switching to Thread 0x7fff9e924700 (LWP 10742)]

Breakpoint 1, breakpoint ()
    at /share/OpenJDK/jdk9-dev/hotspot/src/os/linux/vm/os_linux.cpp:457
457     extern "C" void breakpoint() {

(gdb) cont
(gdb) cont
(gdb) call pns($sp, $rbp, $pc)

"Executing pns"
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
V  [libjvm.so+0xc94bfe]  breakpoint+0x8
V  [libjvm.so+0xc94be0]  os::breakpoint()+0x1c
J 56 C2 AtomicLongTest.update()V (43 bytes) @ 0x00007fffed1ad571 [0x00007fffed1ad560+0x11]
j  AtomicLongTest.main([Ljava/lang/String;)V+18
v  ~StubRoutines::call_stub
V  [libjvm.so+0x9b763d]  JavaCalls::call_helper(JavaValue*, methodHandle*, JavaCallArguments*, Thread*)+0x6d9
V  [libjvm.so+0xca00ed]  os::os_exception_wrapper(void (*)(JavaValue*, methodHandle*, JavaCallArguments*, Thread*), JavaValue*, methodHandle*, JavaCallArguments*, Thread*)+0x41
V  [libjvm.so+0x9b6f4d]  JavaCalls::call(JavaValue*, methodHandle, JavaCallArguments*, Thread*)+0x8b
V  [libjvm.so+0x9cfdb1]  jni_invoke_static(JNIEnv_*, JavaValue*, _jobject*, JNICallType, _jmethodID*, JNI_ArgumentPusher*, Thread*)+0x205
V  [libjvm.so+0x9e72df]  jni_CallStaticVoidMethod+0x358
C  [libjli.so+0x8820]  JavaMain+0x8af
C  [libpthread.so.0+0x8182]  start_thread+0xc2
```

We can easily verify from the address of the function that we're in a different compiled version of `AtomicLongTest.update()` now. And we observe, that after the method has been compiled for a secons time, we now permanently get the error until the program exits:

```
(gdb) cont
Continuing.
Error (iteration 20000, 85895050952704 != 85890755985408
...
[Inferior 1 (process 12698) exited normally]
```

So the problem why we only got the error one time for the first version of the compiled method and didn't run into the breakpoint anymore is because that version conatained a so called "uncomman trap" for the part of the branch which printed the error. That means that the JIT compiler didn't generate any code for that part, because he knew from the previous runs (from the profiling data collected by the interpreter), that the condition never failed. Now, when the condition suddently fails during the first exection of the compiled code, the VM has to deoptimze the compiled method and compile it one more time. The new version will now contain code for both parts of the bracnch.

And once we get in to the compiled code again, the condition starts to be false again. As we can easily see from the assembly, the instruction which increments the `AtomicLong` object is clearly wrong, because it adds '0' to the AtomicLong objects instead of '4_294_967_296L' as expected.

In such a case it can help to take a look at the assembly/opto-assembly produced by the JIT himself to see what happens, because these assembly outputs contain some more meta-information:

```
(gdb) run -XX:+PrintOptoAssembly -XX:CompileCommand="print AtomicLongTest.update" AtomicLongTest 20000 | less
...
{method}
 - this oop:          0x00007fffacd5c1b0
 - method holder:     'AtomicLongTest'
 - constants:         0x00007fffacd5bca0 constant pool [103] {0x00007fffacd5bca0} for 'AtomicLongTe
st' cache=0x00007fffacd5c788
 - access:            0x81000008  static
 - name:              'update'
 - signature:         '()V'
...
033     ADDQ  [[R11 + #16 (8-bit)]],#4294967296
```

So while the opto-assembly looks still OK, the generated machine code (see blow) already adds '0' to the AtomicLong objects instead of '4_294_967_296L' which is clearly wrong:

```
...
 0x00007fffed1a9613: lock addq $0x0,0x10(%r11)  ;*invokevirtual getAndAddLong
                                                ; - java.util.concurrent.atomic.AtomicLong::addAndGet@8 (line 219)
                                                ; - AtomicLongTest::update@16 (line 11)
...
```

If you're familiar with x64 assembly, you may kow that the `addq` instruction only supports 32-bit immediats, but our constant is 2^32 which is too big and wraps around to 0 if casted to a 32-bit value.

If we grep for "ADDQ" in the HotSpot sources we only find one hit in the x86_64.ad file:

```
$ grep -r ADDQ src/
src/cpu/x86/vm/x86_64.ad:  format %{ "ADDQ  [$mem],$add" %}
```

And if we look at this file which is used by the C2-JIT for code generation, we can see that this instruction indeed takes an immediate long argument and emits an `addq` instruction with this long immediate value as argument:

```
instruct xaddL_no_res( memory mem, Universe dummy, immL add, rFlagsReg cr) %{
  predicate(n->as_LoadStore()->result_not_used());
  match(Set dummy (GetAndAddL mem add));
  effect(KILL cr);
  format %{ "ADDQ  [$mem],$add" %}
  ins_encode %{
    if (os::is_MP()) { __ lock(); }
    __ addq($mem$$Address, $add$$constant);
  %}
  ins_pipe( pipe_cmpxchg );
%}
```

The fix is trivial - we can just change the `immL` argument to `immL32`, rebuild and hopefully everything should work jsut fine!

As a side note I want to mention that it is also possibly to identify the before mentioned "uncommon trap" in the opt-assembly:

```
...
044   B3: #     B7 B4 <- B2  Freq: 0,999998
044     movq    R8, [R10 + #176 (32-bit)]       # long ! Field: AtomicLongTest.l
04b     MEMBAR-acquire ! (empty encoding)
04b     cmpq    R8, R11
04e     jne,s   B7  P=0,000000 C=6700,000000    // Jump to uncommon trap in block 7 if condtion fails.
...
079   B7: #     N1 <- B3  Freq: 4,99999e-07
079     cmpq    R8, R11 # CmpL3
        movl    RBP, -1
        jl,s    done
        setne   RBP
        movzbl  RBP, RBP
        done:
08b     movl    RSI, #-163      # int
        nop     # 3 bytes pad for loops and calls
093     call,static  wrapper for: uncommon_trap(reason='unstable_if' action='reinterpret')  <===
        # AtomicLongTest::update @ bci:30  STK[0]=RBP
        # OopMap{off=152}
```

```
098     int3    # ShouldNotReachHere
...
```

JIT Compiler Demo - Simple Loops

```
public class IntLoop {

  static void loop(int count) {
    for (int i = 0; i < count; i++)
      for (int j = 0; j < 1_000_000; j++);
  }

  public static void main(String[] args) {

    for (long i = 0; i < 100; i++)
      loop(2);
    System.out.println("Warmup done");

    long start  = System.currentTimeMillis();
    loop(Integer.parseInt(args[0]));
    long end = System.currentTimeMillis();

    System.out.println(end - start + "ms");
  }
}
```

```
$ java -Xint IntLoop 10
Warmup done
112ms
$ java -Xint IntLoop 100
Warmup done
1111ms
$ java -Xint IntLoop 1000
Warmup done
11202ms
```

For the interprter we get nice linear scaling - no surprise!

Now lets try with JIT:

```
$ export MY_OPTS="-XX:-TieredCompilation -XX:-UseOnStackReplacement -XX:CICompilerCount=1 -XX:LoopUnrollLimit=0"
$ java -XX:+PrintCompilation $MY_OPTS IntLoop 10
    583    1             IntLoop::loop (28 bytes)
Warmup done
0ms
$ java -XX:+PrintCompilation $MY_OPTS IntLoop 100000000
    598    1             IntLoop::loop (28 bytes)
Warmup done
0ms
```

That's really impressing - couldn't have been better!

Now lets change the example slightly..

```
public class LongLoop {

  static void loop(int count) {
    for (long i = 0; i < count; i++)
      for (long j = 0; j < 1_000_000; j++);
  }

  public static void main(String[] args) {

    for (long i = 0; i < 100; i++)
      loop(2);
    System.out.println("Warmup done");

    long start  = System.currentTimeMillis();
    loop(Integer.parseInt(args[0]));
    long end = System.currentTimeMillis();

    System.out.println(end - start + "ms");
  }
}
```

```
$ java -Xint LongLoop 10
Warmup done
155ms
$ java -Xint LongLoop 100
Warmup done
1535ms
$ java -Xint LongLoop 1000
Warmup done
15345ms
```

Again no surprise for the interpreted version. It needs a little longer than the int version, but it's still linear. So let's try the JITed version:

```
$ java -XX:+PrintCompilation $MY_OPTS LongLoop 10
    583    1             LongLoop::loop (34 bytes)
Warmup done
3ms
$ java -XX:+PrintCompilation $MY_OPTS LongLoop 100
    635    1             LongLoop::loop (34 bytes)
Warmup done
35ms
$ java -XX:+PrintCompilation $MY_OPTS LongLoop 1000
    620    1             LongLoop::loop (34 bytes)
Warmup done
330ms
$ java -XX:+PrintCompilation $MY_OPTS LongLoop 10000
    590    1             LongLoop::loop (34 bytes)
Warmup done
3235ms
```

That's a little suprising. The JITed version is considerably faster than the interpreted one, but it bocomes linear with regard to the input parmater (whereas the int version has been constant!)

Let's have a look at the generated code with the help of the -XX:+PrintAssembly option and the hsdis library. First we examine the integer version:

```
$ java -XX:+PrintAssembly -XX:+PrintCompilation $MY_OPTS IntLoop 10
    622    1             IntLoop::loop (28 bytes)
Loaded disassembler from /share/OpenJDK/jdk1.7.0_hsx/jre/lib/amd64/hsdis-amd64.so
Decoding compiled method 0x00007f8c47b4af90:
Code:
[Disassembling for mach='i386:x86-64']
[Entry Point]
[Verified Entry Point]
[Constants]
  # {method} 'loop' '(I)V' in 'IntLoop'
  # parm0:    rsi       = int
  #           [sp+0x20]  (sp of caller)
  ;; N1: #      B1 <- B1  Freq: 1

  ;; B1: #      N1 <- BLOCK HEAD IS JUNK   Freq: 1

  0x00007f8c47b4b0c0: sub    $0x18,%rsp
  0x00007f8c47b4b0c7: mov    %rbp,0x10(%rsp)    ;*synchronization entry
                                                ; - IntLoop::loop@-1 (line 4)
  0x00007f8c47b4b0cc: add    $0x10,%rsp
  0x00007f8c47b4b0d0: pop    %rbp
  0x00007f8c47b4b0d1: test   %eax,0x6140f29(%rip)        # 0x00007f8c4dc8c000
                                                ;   {poll_return}
  0x00007f8c47b4b0d7: retq
  ...
```

And compare it with the code generated for the long version:

```
$ java -XX:+PrintAssembly -XX:+PrintCompilation $MY_OPTS LongLoop 10
    638    1             LongLoop::loop (34 bytes)
Loaded disassembler from /share/OpenJDK/jdk1.7.0_hsx/jre/lib/amd64/hsdis-amd64.so
Decoding compiled method 0x00007f9c80264e50:
Code:
[Disassembling for mach='i386:x86-64']
[Entry Point]
[Verified Entry Point]
[Constants]
  # {method} 'loop' '(I)V' in 'LongLoop'
  # parm0:    rsi       = int
  #           [sp+0x20]  (sp of caller)
  ;; N1: #      B1 <- B3  Freq: 1

  ;; B1: #      B3 B2 <- BLOCK HEAD IS JUNK   Freq: 1

  0x00007f9c80264f80: sub    $0x18,%rsp
  0x00007f9c80264f87: mov    %rbp,0x10(%rsp)    ;*synchronization entry
                                                ; - LongLoop::loop@-1 (line 4)
  0x00007f9c80264f8c: movslq %esi,%r10          ;*i2l  ; - LongLoop::loop@4 (line 4)
  0x00007f9c80264f8f: test   %r10,%r10
  0x00007f9c80264f92: jle    0x00007f9c80264f99  ;*ifge
                                                ; - LongLoop::loop@6 (line 4)
  ;; B2: #      B7 <- B1  Freq: 0.5

  0x00007f9c80264f94: xor    %r11d,%r11d
  0x00007f9c80264f97: jmp    0x00007f9c80264fd2  ;*return
                                                ; - LongLoop::loop@33 (line 6)
  ;; B3: #      N1 <- B6 B1  Freq: 1

  0x00007f9c80264f99: add    $0x10,%rsp
  0x00007f9c80264f9d: pop    %rbp
  0x00007f9c80264f9e: test   %eax,0x614105c(%rip)        # 0x00007f9c863a6000
                                                ;   {poll_return}
  0x00007f9c80264fa4: retq
  ...
```

```
0x00007f9c80264faf: nop
;; B4: #      B5 <- B5  top-of-loop Freq: 4.93447e+06

0x00007f9c80264fb0: add    $0x1,%r8           ; OopMap{off=52}
                                              ;*goto
                                              ; - LongLoop::loop@23 (line 5)
;; B5: #      B4 B6 <- B7 B4  Loop: B5-B4 inner  Freq: 4.93447e+06

0x00007f9c80264fb4: test   %eax,0x6141046(%rip)        # 0x00007f9c863a6000
                                              ;*goto
                                              ; - LongLoop::loop@23 (line 5)
                                              ;   {poll}
0x00007f9c80264fba: cmp    $0xf4240,%r8
0x00007f9c80264fc1: jl     0x00007f9c80264fb0  ;*ifge
                                              ; - LongLoop::loop@16 (line 5)
;; B6: #      B3 B7 <- B5  Freq: 5

0x00007f9c80264fc3: add    $0x1,%r11          ; OopMap{off=71}
                                              ;*goto
                                              ; - LongLoop::loop@30 (line 4)
0x00007f9c80264fc7: test   %eax,0x6141033(%rip)        # 0x00007f9c863a6000
                                              ;*goto
                                              ; - LongLoop::loop@30 (line 4)
                                              ;   {poll}
0x00007f9c80264fcd: cmp    %r10,%r11
0x00007f9c80264fd0: jge    0x00007f9c80264f99  ;*lconst_0
                                              ; - LongLoop::loop@9 (line 5)
;; B7: #      B5 <- B2 B6     Loop: B7-B6  Freq: 5

0x00007f9c80264fd2: mov    $0x1,%r8d
0x00007f9c80264fd8: jmp    0x00007f9c80264fb4
0x00007f9c80264fda: hlt
...
```

So that's a lot mor code for only the fact that we've changed an integer into a long! We will see some of the implications in the next example.

JIT Compiler Demo - Simple loops considered harmfull

In this part we use the same example, but additionally, we start a concurrent thread which triggers a GC every second while we are looping in out main thread. Let's start this time with the long version:

```
public class LongLoopWithGC {

  static long tmp;

  static void loop(int count) {
    for (long i = 1; i < count; i++)
      for (long j = 1; j < 1_000_000; j++)
        tmp++;
  }

  public static void main(String[] args) {

    for (long i = 0; i < 100; i++)
      loop(2);
    System.out.println("Warmup done");

    new Thread() {
      public void run() {
        while(true) {
          try { Thread.sleep(1_000); } catch (InterruptedException e) {}
          System.gc();
        }
      }
    }.start();

    long start  = System.currentTimeMillis();
    loop(Integer.parseInt(args[0]));
    long end = System.currentTimeMillis();

    System.out.println(end - start + "ms");
    System.exit(0);
  }
}
```

```
$ java -XX:+PrintCompilation -verbose:gc $MY_OPTS LongLoopWithGC 100
    567    1             LongLoopWithGC::loop (42 bytes)
Warmup done
68ms
$ java -XX:+PrintCompilation -verbose:gc $MY_OPTS LongLoopWithGC 1000
    568    1             LongLoopWithGC::loop (42 bytes)
Warmup done
646ms
$ java -XX:+PrintCompilation -verbose:gc $MY_OPTS LongLoopWithGC 10000
    560    1             LongLoopWithGC::loop (42 bytes)
Warmup done
[GC 317K->304K(60800K), 0.0112460 secs]
[Full GC 304K->217K(60800K), 0.0799480 secs]
[GC 852K->281K(60800K), 0.0007890 secs]
[Full GC 281K->217K(60800K), 0.0562010 secs]
[GC 217K->217K(60800K), 0.0008090 secs]
[Full GC 217K->217K(60800K), 0.0570710 secs]
[GC 217K->217K(60800K), 0.0007410 secs]
[Full GC 217K->217K(60800K), 0.0541550 secs]
[GC 217K->217K(60800K), 0.0006150 secs]
[Full GC 217K->217K(60800K), 0.0482390 secs]
[GC 217K->217K(60800K), 0.0006570 secs]
[Full GC 217K->217K(60800K), 0.0527470 secs]
7014ms
```

No surprise until now - everything looks as expected! Now lets change the example in the same way as before (i.e. replace the long iterators by integer ones). Should we see any differences?

```
...
  static void loop(int count) {
    for (int i = 1; i < count; i++)
      for (int j = 1; j < 1_000_000; j++)
        tmp++;
  }
...
```

```
$ java -XX:+PrintCompilation -verbose:gc $MY_OPTS IntLoopWithGC 1000
    543    1             IntLoopWithGC::loop (36 bytes)
Warmup done
341ms
$ java -XX:+PrintCompilation -verbose:gc $MY_OPTS IntLoopWithGC 10000
    537    1             IntLoopWithGC::loop (36 bytes)
Warmup done
[GC 317K->272K(60800K), 0.0109640 secs]
[Full GC 272K->217K(60800K), 0.0659020 secs]
3809ms
$ java -XX:+PrintCompilation -verbose:gc $MY_OPTS IntLoopWithGC 100000
    560    1             IntLoopWithGC::loop (36 bytes)
Warmup done
```

```
[GC 317K->320K(60800K), 0.0112450 secs]
[Full GC 320K->217K(60800K), 0.0708950 secs]
37818ms
```

Very strange! No difference how long our main thread is running, we always only see a single GC!

But why? Let's try to attach with a Java debugger to see what happens..

```
$ java -XX:+PrintCompilation -verbose:gc -agentlib:jdwp=transport=dt_socket,address=8000,server=y,suspend=n $MY_OPTS IntLoopWithGC 100000 &
$ jdb -attach 8000
java.io.IOException
        at com.sun.tools.jdi.VirtualMachineManagerImpl.createVirtualMachine(VirtualMachineManagerImpl.java:234)
        at com.sun.tools.jdi.VirtualMachineManagerImpl.createVirtualMachine(VirtualMachineManagerImpl.java:241)
        at com.sun.tools.jdi.GenericAttachingConnector.attach(GenericAttachingConnector.java:117)
        at com.sun.tools.jdi.SocketAttachingConnector.attach(SocketAttachingConnector.java:90)
        at com.sun.tools.example.debug.tty.VMConnection.attachTarget(VMConnection.java:347)
        at com.sun.tools.example.debug.tty.VMConnection.open(VMConnection.java:156)
        at com.sun.tools.example.debug.tty.Env.init(Env.java:54)
        at com.sun.tools.example.debug.tty.TTY.main(TTY.java:1057)

Fatal error:
Unable to attach to target VM.
```

It doesn't work! We can not attach to 'IntLoopWithGC'. But it works with 'LongLoopWithGC':

```
$ java -XX:+PrintCompilation -verbose:gc -agentlib:jdwp=transport=dt_socket,address=8000,server=y,suspend=n $MY_OPTS LongLoopWithGC 100000 &
$ jdb -attach 8000
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
> threads
Group system:
  (java.lang.ref.Reference$ReferenceHandler)0x16e Reference Handler cond. waiting
  (java.lang.ref.Finalizer$FinalizerThread)0x16f  Finalizer         cond. waiting
  (java.lang.Thread)0x170                         Signal Dispatcher running
Group main:
  (java.lang.Thread)0x1                           main              running
  (LongLoopWithGC$1)0x172                         Thread-0          sleeping
> suspend 1
> where 1
  [1] LongLoopWithGC.loop (LongLoopWithGC.java:7)
  [2] LongLoopWithGC.main (LongLoopWithGC.java:27)
```

Ok, let's see what we can find out with GDB:

```
$ gdb java
(gdb) run -verbose:gc -XX:+PrintAssembly -XX:+PrintCompilation $MY_OPTS IntLoopWithGC 100000
Disassembling for mach='i386:x86-64']
[Entry Point]
[Verified Entry Point]
[Constants]
  # {method} 'loop' '(I)V' in 'IntLoopWithGC'
  # parm0:    rsi       = int
  #           [sp+0x20]  (sp of caller)
  ;; N1: #      B1 <- B6  Freq: 1

  ;; B1: #      B6 B2 <- BLOCK HEAD IS JUNK   Freq: 1

  0x00007ffff1eb6fc0: sub    $0x18,%rsp
  0x00007ffff1eb6fc7: mov    %rbp,0x10(%rsp)    ;*synchronization entry
                                                ; - IntLoopWithGC::loop@-1 (line 6)
  0x00007ffff1eb6fcc: cmp    $0x1,%esi
  0x00007ffff1eb6fcf: jle    0x00007ffff1eb700c  ;*if_icmpge
                                                ; - IntLoopWithGC::loop@4 (line 6)
  ;; B2: #      B3 <- B1  Freq: 0.5

  0x00007ffff1eb6fd1: movabs $0xeb5aa680,%r10   ;   {oop(a 'java/lang/Class' = 'IntLoopWithGC')}
  0x00007ffff1eb6fdb: mov    0x70(%r10),%r11
  0x00007ffff1eb6fdf: mov    $0x1,%r9d
  ;; B3: #      B4 <- B2 B5     Loop: B3-B5  Freq: 5

  0x00007ffff1eb6fe5: mov    $0x1,%r8d
  ...
  0x00007ffff1eb6fef: nop                       ;*getstatic tmp
                                                ; - IntLoopWithGC::loop@15 (line 8)
  ;; B4: #      B4 B5 <- B3 B4  Loop: B4-B4 inner  Freq: 4.93447e+06

  0x00007ffff1eb6ff0: inc    %r8d               ;*iinc
                                                ; - IntLoopWithGC::loop@23 (line 7)
  0x00007ffff1eb6ff3: add    $0x1,%r11          ;*ladd
                                                ; - IntLoopWithGC::loop@19 (line 8)
  0x00007ffff1eb6ff7: mov    %r11,0x70(%r10)    ;*putstatic tmp
                                                ; - IntLoopWithGC::loop@20 (line 8)
  0x00007ffff1eb6ffb: cmp    $0xf4240,%r8d
  0x00007ffff1eb7002: jl     0x00007ffff1eb6ff0  ;*if_icmpge
                                                ; - IntLoopWithGC::loop@12 (line 7)
  ;; B5: #      B3 B6 <- B4  Freq: 5
```

```
  0x00007ffff1eb7004: inc    %r9d               ;*iinc
                                                ; - IntLoopWithGC::loop@29 (line 6)
  0x00007ffff1eb7007: cmp    %esi,%r9d
  0x00007ffff1eb700a: jl     0x00007ffff1eb6fe5  ;*return
                                                ; - IntLoopWithGC::loop@35 (line 9)
  ;; B6: #      N1 <- B5 B1  Freq: 1

  0x00007ffff1eb700c: add    $0x10,%rsp
  0x00007ffff1eb7010: pop    %rbp
  0x00007ffff1eb7011: test   %eax,0x613ffe9(%rip)        # 0x00007ffff7ff7000
                                                ;   {poll_return}
  0x00007ffff1eb7017: retq                      ;*getstatic tmp
                                                ; - IntLoopWithGC::loop@15 (line 8)
  0x00007ffff1eb7018: hlt
  ...
^C
Program received signal SIGINT, Interrupt.
0x00007ffff7bc803d in pthread_join () from /lib/libpthread.so.0
(gdb) info threads
  12 Thread 0x7fffecfb0710 (LWP 7255)  0x00007ffff7bcb85c in pthread_cond_wait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
  11 Thread 0x7fffed44a710 (LWP 7254)  0x00007ffff7bcbbc9 in pthread_cond_timedwait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
  10 Thread 0x7fffed54b710 (LWP 7253)  0x00007ffff7bcb85c in pthread_cond_wait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
  9 Thread 0x7fffed64c710 (LWP 7252)  0x00007ffff7bcb85c in pthread_cond_wait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
  8 Thread 0x7fffed74d710 (LWP 7251)  0x00007ffff7bcdb50 in sem_wait () from /lib/libpthread.so.0
  7 Thread 0x7fffed9ab710 (LWP 7250)  0x00007ffff7bcb85c in pthread_cond_wait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
  6 Thread 0x7fffedaac710 (LWP 7249)  0x00007ffff7bcb85c in pthread_cond_wait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
  5 Thread 0x7fffedbad710 (LWP 7248)  0x00007ffff74d9437 in sched_yield () from /lib/libc.so.6
  4 Thread 0x7ffff1932710 (LWP 7247)  0x00007ffff7bcb85c in pthread_cond_wait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
  3 Thread 0x7ffff1a33710 (LWP 7246)  0x00007ffff7bcb85c in pthread_cond_wait@@GLIBC_2.3.2 () from /lib/libpthread.so.0
  2 Thread 0x7ffff7fe4710 (LWP 7245)  0x00007ffff1eb6ffb in ?? ()
* 1 Thread 0x7ffff7fe6700 (LWP 7244)  0x00007ffff7bc803d in pthread_join () from /lib/libpthread.so.0
(gdb) thread 2
[Switching to thread 2 (Thread 0x7ffff7fe4710 (LWP 7245))]#0  0x00007ffff1eb6ffb in ?? ()
(gdb) where
#0  0x00007ffff1eb6ffb in ?? ()
#1  0x0000000000000000 in ?? ()
(gdb) call mixed_ps($sp, $rbp, $pc)
"Executing mixed_ps"
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
J  IntLoopWithGC.loop(I)V
v  ~StubRoutines::call_stub
V  [libjvm.so+0x7305d8]  JavaCalls::call_helper(JavaValue*, methodHandle*, JavaCallArguments*, Thread*)+0x51a
V  [libjvm.so+0x9749ee]  os::os_exception_wrapper(void (*)(JavaValue*, methodHandle*, JavaCallArguments*, Thread*), JavaValue*, methodHandle*, JavaCallArguments*, Thread*)+0x3a
V  [libjvm.so+0x7300b7]  JavaCalls::call(JavaValue*, methodHandle, JavaCallArguments*, Thread*)+0x7d
V  [libjvm.so+0x7431dd]  jni_invoke_static(JNIEnv_*, JavaValue*, _jobject*, JNICallType, _jmethodID*, JNI_ArgumentPusher*, Thread*)+0x186
V  [libjvm.so+0x758ea1]  jni_CallStaticVoidMethod+0x37b
C  [libjli.so+0x39c3]  JavaMain+0x8a3
(gdb)  continue
Continuing.
```

Everything looks normal, but if we continue, we get a segmentation fault. Should we be scared about that?

```
Program received signal SIGSEGV, Segmentation fault.
0x00007ffff1eb7011 in ?? ()
(gdb)  call mixed_ps($sp, $rbp, $pc)

"Executing mixed_ps"
Native frames: (J=compiled Java code, j=interpreted, Vv=VM code, C=native code)
J  IntLoopWithGC.loop(I)V
v  ~StubRoutines::call_stub
V  [libjvm.so+0x7305d8]  JavaCalls::call_helper(JavaValue*, methodHandle*, JavaCallArguments*, Thread*)+0x51a
V  [libjvm.so+0x9749ee]  os::os_exception_wrapper(void (*)(JavaValue*, methodHandle*, JavaCallArguments*, Thread*), JavaValue*, methodHandle*, JavaCallArguments*, Thread*)+0x3a
V  [libjvm.so+0x7300b7]  JavaCalls::call(JavaValue*, methodHandle, JavaCallArguments*, Thread*)+0x7d
V  [libjvm.so+0x7431dd]  jni_invoke_static(JNIEnv_*, JavaValue*, _jobject*, JNICallType, _jmethodID*, JNI_ArgumentPusher*, Thread*)+0x186
V  [libjvm.so+0x758ea1]  jni_CallStaticVoidMethod+0x37b
C  [libjli.so+0x39c3]  JavaMain+0x8a3

(gdb)  x /1i $pc
=> 0x7ffff1eb7011:      test   %eax,0x613ffe9(%rip)        # 0x7ffff7ff7000
(gdb)  continue
Continuing.
[GC 317K->320K(60800K), 0.0438280 secs]
[Full GC 320K->217K(60800K), 0.0690220 secs]
8060ms

Program exited normally.
```

It doesn't seems to be critical! If we continue, the program terminates normally.

If we try the same with LongLoopWithGC, we will see, that we get even more segmentation faults (actually every time before a GC happens).

```
(gdb) run -verbose:gc -XX:+PrintOptoAssembly -XX:+PrintCompilation $MY_OPTS LongLoopWithGC 10000
...
Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0x7ffff7fe4710 (LWP 7302)]
```

```
0x00007ffff1eb6f4c in ?? ()
(gdb) continue
Continuing.
[GC 317K->288K(60800K), 0.0248930 secs]
[Full GC 288K->217K(60800K), 0.0708540 secs]

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff1eb6f4c in ?? ()
(gdb) continue
Continuing.
[GC 852K->281K(60800K), 0.0027870 secs]
[Full GC 281K->217K(60800K), 0.0575250 secs]
...
```

Notice that this time we used -XX:+PrintOptoAssembly instead of -XX:+PrintAssembly. It provides some interesting information:

```
044   B5: #     B4 B6 <- B7 B4  Loop: B5-B4 inner  Freq: 4.93447e+06
044     addq    R8, #1  # long
048     movq    [R11 + #112 (8-bit)], R8        # long ! Field LongLoopWithGC.tmp
04c     testl  rax, [rip + #offset_to_poll_page]        # Safepoint: poll for GC        # LongLoopWithGC::loop @ bci:31  L[0]=RSI L[1]=R9 L[2]=_ L[3]=RCX L[4]=_
        # OopMap{r11=Oop off=76}
052     cmpq    RCX, #1000000
059     jl,s   B4  P=0.999999 C=1610592.000000
059
05b   B6: #     B3 B7 <- B5  Freq: 5
05b     addq    R9, #1  # long
05f     testl  rax, [rip + #offset_to_poll_page]        # Safepoint: poll for GC        # LongLoopWithGC::loop @ bci:38  L[0]=RSI L[1]=R9 L[2]=_ L[3]=_ L[4]=_
        # OopMap{r11=Oop off=95}
065     cmpq    R9, R10
068     jge,s   B3  P=0.100000 C=-1.000000
```

So the segmentation faults are in fact Safepoints!

```
(gdb) p 'os::_polling_page'
$1 = (address) 0x7ffff7ff8000 ""
```