PrintAssembly

Puzzled by a performance glitch? You might have to look at the generated code.

Examining generated code

The following HotSpot options (with an -XX: prefix on the command line) require OpenJDK 7 and an externally loadable disassembler plugin:

- `+PrintAssembly` print assembly code for bytecoded and native methods
- `+PrintNMethods` print nmethods as they are generated
- `+PrintNativeNMethods` print native method wrappers as they are generated
- `+PrintSignatureHandlers` print native method signature handlers
- `+PrintAdapterHandlers` print adapters (i2c, c2i) as they are generated
- `+PrintStubCode` print stubs: deopt, uncommon trap, exception, safepoint, runtime support
- `+PrintInterpreter` print interpreter code

These flags are "diagnostic", meaning that they must be preceded by `-XX:+UnlockDiagnosticVMOptions`. On the command line, they must all be preceded by `-XX:`. They may also be placed in a flags file, `.hotspotrc` by default, or configurable as `-XX:Flags=myhotspotrc.txt`.

The disassembly output is annotated with various kinds of debugging information, such as field names and source locations. The quality of this information improved markedly in January 2010 (bug fix 6912062).

Examples

```
0x0188272f: mov        ecx, [ecx+0x1A8B8B30]  ;   {oop(cache [42] for constant pool [234]/invokedynamic for
'SumWithIndy' cache=0x1a8b8b30)}
  0x01882735: mov        edx, [ecx+0x10]
  0x01882738: cmp        edx, a 'sun/dyn/ToGeneric$A2'
                                        ;   {oop(a 'sun/dyn/ToGeneric$A2')}
  0x0188273e: jnz        0x018827c1      ;*invokedynamic
                                        ; - SumWithIndy::test@17 (line 90)
  0x01882744: mov        edi, 0x00000150
  0x01882749: mov        edx, [edi+0x1A8B9110]  ;*getstatic cache
                                        ; - java.lang.Integer::valueOf@35 (line 651)
                                        ; - sun.dyn.ToGeneric$A2::targetA2@3 (line 663)
                                        ; - sun.dyn.ToGeneric$A2::invoke_L@4 (line 672)
                                        ; - java.dyn.InvokeDynamic::invokeExact@4
                                        ; - SumWithIndy::test@17 (line 90)
                                        ;   {oop('java/lang/Integer$IntegerCache')}
```

Complete file: sample-disassembly.txt

Plugin Implementations

There are 2 implementations around:

OpenJDK sources (defines the plugin API)

This version of the plugin requires the Gnu disassembler, which is available separately as part of the binutils project.

Look at the README for instructions on building.

With recent binutils version (i.e. binutils-2.23.2) you may get the following build error:

```
WARNING: `makeinfo' is missing on your system.  You should only need it if
         you modified a `.texi' or `.texinfo' file, or any other file
         ...
```

This is because of **"Bug 15345** - [2.23.2 regression] binutils-2.23.2 tarball doesn't build without makeinfo". The simplest workaround is to touch '`bfd/doc/bfd.info`' in the biuntils source directory.

Pre-built binaries do not seem to be available *(help... anyone?)*. Prebuilt binaries for Windows-x86: hsdis-i386.dll

Kenai project base-hsdis

This is a from-scratch implementation which uses code from the Bastard project at SourceForge. The copyrights on this code are non-restrictive.

The Kenai project offers binary downloads.

Installing the Plugin

Once you succeed in building or downloading the `hsdis` binary library (in the following named DLL), you have to install it next to your libjvm.so (jvm.dll on Windows), in the same folder. (Alternatively, you can put it anywhere on your `LD_LIBRARY_PATH`.) The DLL must be given the name that the JVM will be looking for. The core of the name will be `hsdis-i386` for 32-bit Intel JVMs. Other names in use are `hsdis-amd64`, `hsdis-sparc`, and `hsdis-sparcv9`. A prefix and/or suffix will be required, according to system-dependent rules for naming DLLs.

installing the plugin on Solaris

```
$ JDK7=my/copy/of/jre1.7.0
$ cp -p hsdis/.libs/hsdis.so $JDK7/lib/i386/client/hsdis-i386.so
$ cp -p hsdis/.libs/hsdis.so $JDK7/lib/i386/server/hsdis-i386.so
$ XJAVA="$JDK7/bin/java -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly"
$ $XJAVA -Xcomp -cp ~/Classes hello
$ $XJAVA -Xcomp -cp ~/Classes -XX:PrintAssemblyOptions=hsdis-print-bytes hello
$ $XJAVA -XX:-PrintAssembly -XX:+PrintStubCode
$ $XJAVA -XX:-PrintAssembly -XX:+PrintInterpreter
$ $XJAVA -XX:-PrintAssembly -XX:+PrintSignatureHandlers
$ $XJAVA -Xbatch -cp ~/Classes -XX:+PrintCompilation myloopingbenchmark
```

The last line (with `myloopingbenchmark`) is most typical, since it uses the batch execution mode common with benchmarks. The `-XX:+PrintCompilation` flag will let you know which (if any) methods are being compiled.

Filtering Output

The `-XX:+PrintAssembly` option prints everything. If that's too much, drop it and use one of the following options.

Individual methods may be printed:

- `CompileCommand=print,*MyClass.myMethod` prints assembly for just one method
- `CompileCommand=option,*MyClass.myMethod,PrintOptoAssembly` (debug build only) produces the old print command output
- `CompileCommand=option,*MyClass.myMethod,PrintNMethods` produces method dumps

These options accumulate.

If you get no output, use `-XX:+PrintCompilation` to verify that your method is getting compiled at all.

Reading the compiler's mind

The `-XX:+LogCompilation` flag produces a low-level XML file about compiler and runtime decisions, which may be interesting to some. The `-XX:+UnlockDiagnosticVMOptions` must come first. The dump is to `hotspot.log` in the current directory; use `-XX:LogFile=foo.log` to change this.

The `LogCompilation` output is basic line-oriented XML. It can usefully be read in a text editor, and there are also tools for parsing and scanning it.