

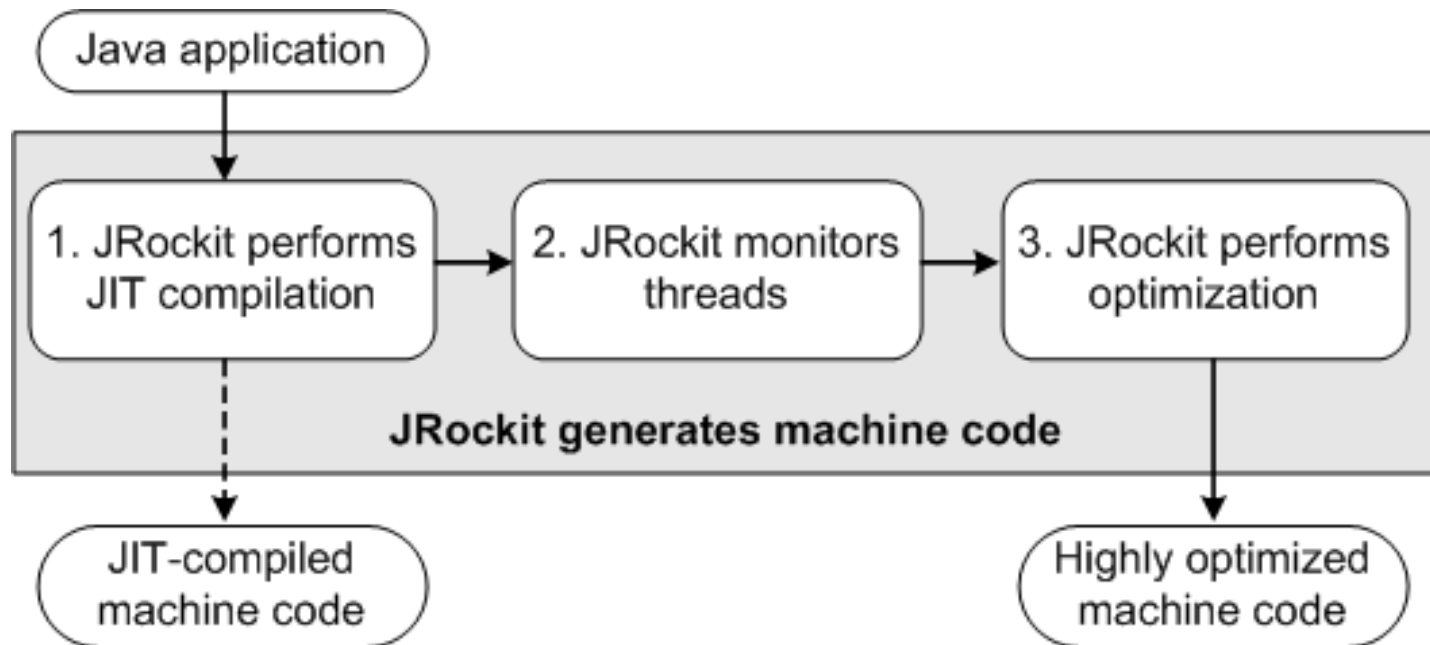
Topics :

Understanding Just-In-Time Compilation and Optimization

Understanding Just-In-Time Compilation and Optimization

How the JRockit JVM Generates Machine Code for Java Applications

The code generator in the JRockit JVM runs in the background during the entire run time of your Java application, automatically adapting the code to run optimally. The code generator works in three steps



Compilation Modes

Inside Java HotSpot VM, there are actually two separate JIT compiler modes, which are known as C1 and C2.

C1 is used for applications where quick startup and rock-solid optimization are required;

Ex : GUI applications

C2, on the other hand, was originally intended for long-running, predominantly server-side applications.

EX : Java Server's

Prior to some of the later Java SE 7 releases, these two modes were available using the -client and -server switches, respectively.

Later Java SE 7 releases, a new feature called *tiered compilation* became available.

This feature uses the C1 compiler mode at the start to provide better startup performance.

Once the application is properly warmed up, the C2 compiler mode takes over to provide more-aggressive optimizations and, usually, better performance.

With the arrival of Java SE 8, tiered compilation is now the default behavior.

The two compilers

The JVM that ships with OpenJDK contains two compiler back-ends:

C1, also known as 'client'

C2, also known as 'server'

The C1 compiler has a number of different modes, and will alter its response to a compilation request given a number of system factors, including, but not limited to, the current workload of the C1 & C2 compiler thread pool.

Given these different modes, the JDK refers to different *tiers*, which can be broken down as follows:

Tier 1- interpreter

Tier1 - client compiler with no profiling information

Tier2 - client compiler with basic counters

Tier3 - client compiler with profiling information

Tier4 - server compiler

jvm flags

To view all the available flags that can be passed to the jvm, run the following command:

```
java -XX:+PrintFlagsFinal
```

Full Logging of JIT Compilation

The switch for enabling full logging is
`-XX:+LogCompilation`

and it must be preceded by the option
`-XX:+UnlockDiagnosticVMOptions`.

Using the `-XX:+LogCompilation` switch produces a separate log file, `hotspot_pid<PID>.log`, in the startup directory.

To change the location of the file :

use `-XX:LogFile=<path to file>`.

Some JIT Compilation Techniques

One of the most common JIT compilation techniques used by Java HotSpot VM is inlining, which is the practice of substituting the body of a method into the places where that method is called. Inlining saves the cost of calling the method; no new stack frames need to be created. By default, Java HotSpot VM will try to inline methods that contain less than 35 bytes of JVM bytecode.

Another common optimization that Java HotSpot VM makes is monomorphic dispatch, which relies on the observed fact that, usually, there aren't paths through a method that cause an object reference to be of one type most of the time but of another type at other times.

-XX:+PrintCompilation flag

The first step to understanding how JIT compilation in Java HotSpot VM is affecting the code is to see which of the methods are getting compiled.

add the -XX:+PrintCompilation flag to the script used to start the Java processes.

Note: The resulting log of compilation events will end up in the standard log (that is, the standard output), and there is currently no way to redirect the entries to another file.

The -XX:+PrintCompilation flag output looks something like this:

```
1 sb java.lang.ClassLoader::loadClassInternal (6 bytes)
2 b  java.lang.String::lastIndexOf (12 bytes)
3 s!b java.lang.ClassLoader::loadClass (58 bytes)
```

Flags correspond to:

- b Blocking compiler (always set for client)
- * Generating a native wrapper
- % On stack replacement
- ! Method has exception handlers
- s Synchronized method

.hotspot_compiler file in the current working directory is used to customized the compiler

.hotspot_compiler

exclude Main main

dontinline Main doTest

compileonly Main doTest

Examples of Code Optimization

The following code examples show how the Hotspot JVM optimizes Java code.

Code Before Optimization:

```
class A {  
    B b;  
    public void newMethod() {  
        y = b.get();  
        ...do stuff...  
        z = b.get();  
        sum = y + y;  
    }  
}  
  
class B {  
    int value;  
    final int get() {  
        return value;  
    }  
}
```

Code After Optimization

```
class A {  
    B b;  
    public void newMethod() {  
        y = b.value;  
        ...do stuff...  
        sum = y + y;  
    }  
}  
  
class B {  
    int value;  
    final int get() {  
        return value;  
    }  
}
```

Inspecting Compilation with jstat

Seeing the output of print compilation requires that the program be started with the `-XX:+PrintCompilation` flag. If the program was started without that flag, you can get some limited visibility into the working of the compiler by using `jstat`.

`jstat` has two options to provide information about the compiler. The `-compiler` option supplies summary information about how many methods have been compiled (here 5003 is the process id of the program to be inspected):

```
% jstat -compiler 5003
```

Compiled	Failed	Invalid	Time	FailedType	FailedMethod
206	0	0	1.97	0	

Alternately, we can use the `-printcompilation` option to get information about the last method that is compiled. Because `jstat` takes an optional argument to repeat its operation, you can see over time which methods are being compiled. In this example, `jstat` repeats the information for process id 5003 every second (1000 milliseconds):

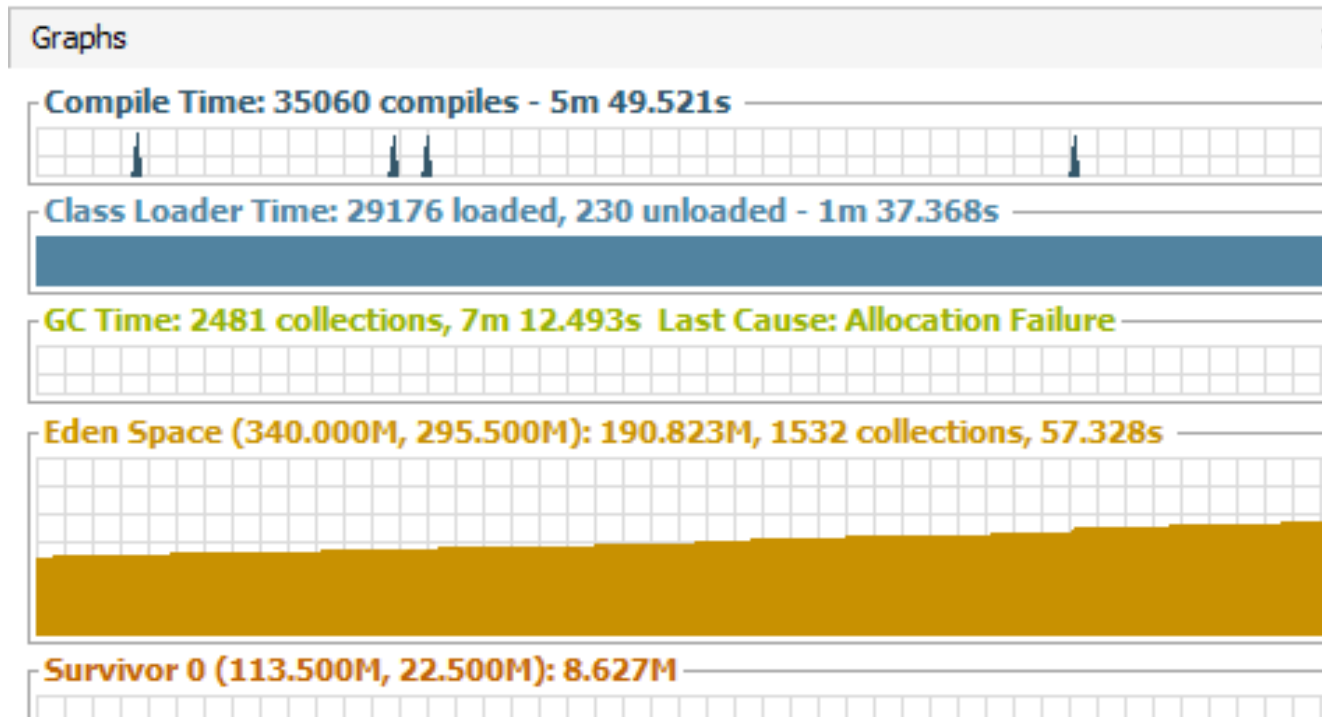
```
% jstat -printcompilation 5003 1000
```

```
Compiled Size Type Method
```

```
207    64    1 java/lang/CharacterDataLatin1 toUpperCase
```

```
208     5    1 java/math/BigDecimal$StringBuilderHelper getCharArray
```


VisualGC -> JIT Compiled classes details



Compilation Thresholds

If the method we are interested in hasn't been compiled yet, JITWatch will notify the same.

We can generate some more load for our application or lower the thresholds for JIT so it will treat the method as hot enough to compile.

`-XX:+CompileThreshold=N` flag.

The default value of N for the client compiler is 1,500; for the server compiler it is 10,000.

Reading the compiler's mind

The `-XX:+LogCompilation` flag produces a low-level XML file about compiler and runtime decisions

`-XX:+UnlockDiagnosticVMOptions -XX:+LogCompilation -XX:+PrintInlining -XX:+PrintCompilation`

Print Assembly:

`-XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly`

Exception:

“Could not load hsdisk-amd64.dll; library not loadable; PrintAssembly is disabled”.

Solution:

Download hsdisk-1.1.1-win32-amd64.zip file from <http://fcml-lib.com/download.html>
and copy hsdisk-amd64.dll file in jre/jdk bin folder.

Ref : x86_64 Assembly to understand generate code

Compilation Log

timestamp (since VM start)	compilation ID	method name	method size
5328	50	n java.io.FileOutputStream::writeBytes (native)	
5330	27	! java.nio.CharBuffer::wrap (20 bytes)	
5333	31	s java.io.BufferedOutputStream::flush (12 bytes)	
5477	56	% CompilationExample::main @ 37 (82 bytes)	

Annotations for the Compilation Log:

- on-stack replacement (points to 's' in row 3)
- synchronized method (points to '!' in row 2)
- exception handler (points to '%' in row 4)
- native (points to 'n' in row 1)
- loop bytecode index (points to '37' in row 4)

Using this reference, we can break down the information in the log output from our test program:

137 48 % 3

demo.C1LoopBackedgeThresholdMain::exerciseTier3LoopBackedgeThreshold @ 5 (25 bytes)

This compile happened 137 milliseconds after JVM startup

Compilation ID was 48

This was an on-stack replacement

This compilation happened at Tier3

The OSR loop bytecode index is 5

The compiled method was 25 bytecodes

On-Stack Replacement

On-Stack replacement is a mechanism that allows the interpreter to take advantage of compiled code, even when it is still executing a loop for that method in interpreted mode. The below flow explains :

- ✓ Start executing a method `loopyMethod` in the interpreter
- ✓ Within `loopyMethod`, we execute an expensive loop body 1,000,000 times
- ✓ The interpreter will see that the loop count has exceeded the
- ✓ `Tier3BackedgeThreshold` setting
- ✓ The interpreter will request compilation of `loopyMethod`
- ✓ The method body is expensive and slow, and we want to start using the compiled version immediately. Without OSR, the interpreter would have to complete the 1,000,000 iterations of slow interpreted code, dispatching to the compiled method on the next call to `loopyMethod()`
- ✓ With OSR, the interpreter can dispatch to the compiled frame at the start of the next loop iteration
- ✓ Execution will now continue in the compiled method body

-XX:-TieredCompilation disables intermediate compilation tiers (1, 2, 3), so that a method is either interpreted or compiled at the maximum optimization level (C2).

As a side effect TieredCompilation flag also changes the number of compiler threads, the compilation policy and the default code cache size. Note that with TieredCompilation disabled

there will be less compiler threads;

simple compilation policy (based on method invocation and backedge counters) will be chosen instead of advanced compilation policy;

default reserved code cache size will be 5 times smaller.

To disable C2 compiler and to leave only C1 with no extra overhead, set `-XX:TieredStopAtLevel=1`.

To disable all JIT compilers and to run everything in interpreter, use `-Xint`.