About `PrintCompilation`

This note tries to document the output of `PrintCompilation` flag in HotSpot VM. It was originally intended to be a reply to a blog post on PrintCompilation from Stephen Colebourne. It's kind of grown too big to fit as a reply, so I'm putting it here.

Written by: Kris Mok rednaxelafx@gmail.com

Most of the contents in this note are based on my reading of HotSpot source code from OpenJDK and experimenting with the VM flags; otheres come from HotSpot mailing lists and other reading materials listed in the "References" section.

About the examples

There's been some changes to what `PrintCompilation` prints by a HotSpot VM. In this gist, I'm showing a few log examples of running `groovysh` on these 8 configurations:

1. JDK 5 update 22 / HotSpot Client VM

- JDK 6 / HotSpot Client VM
- JDK 6 update 14 / HotSpot Server VM 14.0-b16
- JDK 6 update 18 / HotSpot Server VM 16.0-b13 +TieredCompilation
- JDK 6 update 25 / HotSpot Server VM 20.0-b11
- JDK 6 update 25 / HotSpot Server VM 20.0-b11 +TieredCompilation
- JDK 7 / HotSpot Server VM 21.0-b17
- JDK 7 / HotSpot Server VM 21.0-b17 +TieredCompilation

Common flags

The method attribute flags are common in all these variants:

Symbol	Meaning	Description
%	On stack replacement	*compile type*: * With flag: This `CompileTask` is an OSR compilation task * Without: This `CompileTask` is a standard compilation task
s	Synchronized method	* With flag: The method to compile is declared as `synchronized` * Without: The method to compile is not declared as `synchronized`
!	Method has exception handlers	* With flag: The method to compile has one or more exception handlers * Without: The method to compile doesn't have any exception handlers
b	Blocking compilation	* With flag: Application thread is blocked by this `CompileTask` * Without: This `CompileTask` is executed by a background compiler thread
n	Native wrapper	* With flag: The method to compile is native (not actually compiling the method, but instead generating a native wrapper) * Without: The method to compile is a Java method

In JDK 5's HotSpot VM or earlier, there were 3 possible flags for *compile type*: `b`, `*` or *empty* [1]. The `*` flag has been deprecated by the new `n` format.

Symbol	Meaning

Symbol	Meaning
*	Generating a native wrapper

PrintCompilation's output in JDK 6 or later is mostly affected by these 2 changes:

1. 6953144: Tiered compilation changeset

- 7022998: JSR 292 recursive method handle calls inline themselves infinitely changeset

So let's see what's going on before and after these changes.

JDK 6 ? ~ JDK 6 update 24 (~ HotSpot 19)

This is before 6953144.

A basic line looks like:

```
  36 s!    sun.misc.URLClassPath::getLoader (136 bytes)
```

The format string (in C `printf` format):

```
%3d%c%c%c%c%c %s (%d bytes)
```

and the meaning is:

```
{compile_id}{method_attributes} {short_name} ({size} bytes)
```

Compilation number (`compile_id`)

Every compilation request is given a unique `compile_id` upon submission, and assigned to a newly created `CompileTask`, except for the ones that are trying to compile a native method.

There are 2 incrementing sequences of `compile_id` s, one for standard compilations, the other for OSR compilations; they're counted independently. There used to be a third sequence for native compilations (but not used by default), when HotSpot still compiled native methods, but it's deprecated in JDK 6.

The log snippet below shows the fact that these two sequences are counted independently:

```
  58       org.codehaus.groovy.runtime.metaclass.MetaMethodIndex::copyMethodsToSuper (82 bytes)
  59 s     java.lang.StringBuffer::append (8 bytes)
  60       org.objectweb.asm.Type::a (214 bytes)
   1%      groovy.lang.MetaClassImpl::applyStrayPropertyMethods @ 57 (232 bytes)
  61       groovy.lang.MetaClassImpl$MethodIndexAction::iterate (119 bytes)
 ...
  92       org.codehaus.groovy.runtime.metaclass.MetaMethodIndex::isMatchingMethod (68 bytes)
  93       java.util.HashMap::getEntry (79 bytes)
  94       java.beans.FeatureDescriptor::getObject (13 bytes)
   2%      java.util.Arrays::fill @ 10 (28 bytes)
  95  !    sun.misc.URLClassPath$JarLoader::getResource (91 bytes)
  96       java.util.zip.ZipFile::ensureOpen (37 bytes)
```

The `%` flag indicates an OSR compilation (see the next section).

Method attributes

There are five slots for method attributes, which are:

```
// print method attributes
const bool is_osr = osr_bci() != CompileBroker::standard_entry_bci;
{ const char blocking_char  = is_blocking()                     ? 'b' : ' ';
  const char compile_type   = is_osr                            ? '%' : ' ';
  const char sync_char      = method->is_synchronized()         ? 's' : ' ';
  const char exception_char = method->has_exception_handler()   ? '!' : ' ';
  const char tier_char      =
    is_highest_tier_compile(comp_level())                       ? ' ' : ('0' + comp_level());
  tty->print("%c%c%c%c%c ", compile_type, sync_char, exception_char, blocking_char, tier_char);
}
```

The first 4 are briefly described in the table above. The last one is about the compilation level in "tiered" mode, which will be explained in a moment.

Method name (`short_name`)

This is the method name in the form of `klassname::methodname` . Notice that the signature (argument and return value types) is not included.

Size

This is the size of the original bytecodes of the method. It's **NOT** the size of the generated native code.
The `CompileTask` isn't even executed yet when `PrintCompilation` prints out the log; it couldn't have known the size of the generated code in advance.

Deoptimization ("made not entrant" & "made zombie")

When deoptimization happens [6], if it is decided to invalidate the offending `nmethod` , it will be "made not entrant" first; and then, when the `NMethodSweeper` finds that there are no activations of it on the stacks any more, it is "made zombie"; when the `NMethodSweeper` see a zombie method, it's ready to be reclaimed. This two-phase process is needed because HotSpot doesn't keep track of whether a method is "on-stack" or not, until it's "made not entrant".

So, when does a deoptimization happen?

It's not the same in the Client VM and the Server VM, the latter could deoptimize more than the former. Won't get into all the details here, the reasons and actions of deoptimization are listed in two enums below: (taken from HotSpot 16.0)

```
// What condition caused the deoptimization?
enum DeoptReason {
  Reason_many = -1,             // indicates presence of several reasons
  Reason_none = 0,              // indicates absence of a relevant deopt.
  Reason_null_check,            // saw unexpected null or zero divisor (@bci)
  Reason_null_assert,           // saw unexpected non-null or non-zero (@bci)
  Reason_range_check,           // saw unexpected array index (@bci)
  Reason_class_check,           // saw unexpected object class (@bci)
  Reason_array_check,           // saw unexpected array class (aastore @bci)
  Reason_intrinsic,             // saw unexpected operand to intrinsic (@bci)
  Reason_unloaded,              // unloaded class or constant pool entry
  Reason_uninitialized,         // bad class state (uninitialized)
  Reason_unreached,             // code is not reached, compiler
  Reason_unhandled,             // arbitrary compiler limitation
  Reason_constraint,            // arbitrary runtime constraint violated
  Reason_div0_check,            // a null_check due to division by zero
  Reason_age,                   // nmethod too old; tier threshold reached
```

```
    Reason_LIMIT,
    // Note:  Keep this enum in sync. with _trap_reason_name.
    Reason_RECORDED_LIMIT = Reason_unloaded   // some are not recorded per bc
    // Note:  Reason_RECORDED_LIMIT should be < 8 to fit into 3 bits of
    // DataLayout::trap_bits.  This dependency is enforced indirectly
    // via asserts, to avoid excessive direct header-to-header dependencies.
    // See Deoptimization::trap_state_reason and class DataLayout.
  };

  // What action must be taken by the runtime?
  enum DeoptAction {
    Action_none,                  // just interpret, do not invalidate nmethod
    Action_maybe_recompile,       // recompile the nmethod; need not invalidate
    Action_reinterpret,           // invalidate the nmethod, reset IC, maybe recompile
    Action_make_not_entrant,      // invalidate the nmethod, recompile (probably)
    Action_make_not_compilable,   // invalidate the nmethod and do not compile
    Action_LIMIT
    // Note:  Keep this enum in sync. with _trap_action_name.
  };
```

And, the list is growing.

Examples of `PrintCompilation`'s output for deoptimized methods:

In non-tiered mode:

```
57   made not entrant  java.util.Arrays::mergeSort (234 bytes)
48   made zombie  groovy.lang.MetaClassImpl::applyStrayPropertyMethods (232 bytes)
```

`compile_id` is printed, to tell which compilation it is. The same method may be compiled multiple times, resulting in multiple `nmethod`s, and they're handled independently, so this is necessary.

In tiered mode:

```
58   made zombie  (1)  java.util.ArrayList::RangeCheck (48 bytes)
99   made not entrant  (1)  java.util.ArrayList::add (29 bytes)
```

The `(1)` in the log shows that the method was compiled at tier 1.

Native wrapper

In JDK 6, the compilers in HotSpot don't compile native methods any more. Instead, wrappers for native methods are generated on a per-signature basis, which handles argument shuffling, locking if needed, state transitioning, making the actual call, return value handling, safepoint polling, and so on.

Since they're not compiled, `CompiledTask`s for native methods are not assigned a `compile_id`.

Example:

```
---   n   java.lang.System::arraycopy (static)
```

Remember that the format string for `compile_id` was `%3d`? Since it's expected to be aligned by 3 characters, the same goes for native methods as well -- only that they don't have a `compile_id`, so they get "---" instead. And then follows the `n` flag indicating it's a native wrapper. Then the `short_name` of the method, as usual.

Notice at the end that it doesn't print the size of the method, because native methods don't have a bytecode method body.

Native method declared as `static` will have a `(static)` note at the end of the line.

Compare that with output from JDK 5 update 22's HotSpot:

```
  17*  b   java.lang.System::arraycopy (0 bytes)
```

In JDK 5's HotSpot, `CompileTask` for native methods are assigned a `compile_id`, and defaults to using the same counter sequence as the standard compilations. The `*` flag shows that the task is to generate a native wrapper. At the end of the line it shows `(0 bytes)` for the absense of a bytecode method body.

Another JDK 6 example:

```
---   ns  java.lang.Throwable::fillInStackTrace
```

Notice that native methods can be declared as `synchronized` too, as in this case. The generated wrapper will take care of proper locking upon entry and unlocking upon return.

Background compilation

To make the most out of modern hardware's parallel computing capabilities, HotSpot VM's compilers run on their own background threads, and can compiles Java methods concurrently with Java application threads. With background compilation, compilers gets more budget for optimizations compared with tradition "JIT compilation", while the application stays responsive.

When background compilation is on, the Java application thread that triggered a compilation task doesn't have to block and wait for the task to finish; instead it can keep running in the interpreter for the moment (or some less-optimized version of compiled code, in "tiered" mode), and jump into the compiled version later on.

On the other hand, when background compilation is turned off, the Java application thread that triggered a compilation task will have to block right at the point of triggering, wait for the compilation to finish, and then jump into the compiled code directly. This is indicated by the `b` flag in `PrintCompilation`'s output.

Background compilation is on by default in JDK 6's HotSpot VM, including both the Client and the Server VM. To turn it off directly, use `-Xbatch` or `-XX:-BackgroundCompilation` VM flags. Other flags may also have a side-effect of turning it off, such as: run the VM in a mostly compiled mode (`-Xcomp`), or run CTW (compile-the-world, `-XX:+CompileTheWorld`), or use a somewhat obsolete flag `-Xconcurrentio`. In the current implementation of HotSpot, background compilation is a global setting; it's either always on or always off during a single run.

Prior to JDK 6, it was off by default in HotSpot Client VM [2]. So with a Sun JDK 5 update 22's HotSpot Client VM, a `PrintCompilation` snippet would look like:

```
  318*  b   java.lang.Class::getClassLoader0 (0 bytes)
  319   b   groovy.lang.MetaClassImpl::copyNonPrivateFields (75 bytes)
  320   b   java.lang.ClassLoader::loadClass (7 bytes)
  321   b   java.lang.String::compareTo (9 bytes)
  322   b   java.beans.PropertyDescriptor::getPropertyType0 (11 bytes)
  323   b   java.lang.ref.SoftReference::<init> (13 bytes)
  324 s!b   java.beans.PropertyDescriptor::getPropertyType (33 bytes)
  325  !b   org.codehaus.groovy.util.LazyReference::getLocked (85 bytes)
  326   b   java.io.Win32FileSystem::parentOrNull (171 bytes)
  327 s!b   java.beans.PropertyDescriptor::getWriteMethod (136 bytes)
  328   b   groovy.lang.MetaBeanProperty::<init> (18 bytes)
  329   b   org.objectweb.asm.Item::a (240 bytes)
  330   b   java.io.Win32FileSystem::isLetter (30 bytes)
  331   b   java.util.TreeMap::fixAfterInsertion (287 bytes)
  332   b   org.codehaus.groovy.reflection.CachedClass$8::get (5 bytes)
```

```
  1%  b   java.util.Arrays::fill @ 10 (28 bytes)
333   b   org.codehaus.groovy.reflection.CachedClass::addSubclassExpandos (142 bytes)
```

Notice how all `CompileTask`s show their `b` flag is set, even for OSR compilations and native wrapper generations. It's no longer the case in JDK 6 or later, though.

OSR

To explain a little bit about the difference between a standard compilation and an OSR compilation:

There are two ways a Java method may be compiled by a HotSpot compiler: standard or OSR.

- Standard: The method entry is at the beginning of the method;
- OSR: The method entry point is at the end of some loop's body in the method.

Standard compilation is straightforward, but what is this OSR thing, and why do we need it?

OSR, or On-Stack Replacement, was a mechanism that enables a VM to replace a method while it is still running (and thus the term "on-stack"). It was originally developed for Self-91 [3], and is used by modern JVMs such as HotSpot and Jikes RVM [4]; J9 calls its implementation of OSR "Dynamic Loop Transfer" [5], which is basically the same thing with a different name.

OSR in HotSpot is used to help improve performance of Java methods stuck in loops [6]. Without OSR, a method running in the interpreter can't transfer to its compiled version even if there is one available, until the next time this method is invoked. With OSR, though, a Java method with long-running loops can run in the interpreter, trigger an OSR compilation in one of its loops, keep running in the interpreter until the compilation completes, and jump right into the compiled version without having to wait for "the next invocation".

In HotSpot, there are two `InvocationCounter`s for every Java method: one counts invocations, the other counts "backedges" (which is just a fancy way of saying "loops"). They're not exact; they actually "decade" from time to time, and are just used to indicate the "hotness" of a method.

When the sum of these two counters trip over a certain threshold at method entry, a standard compilation task is triggered; when it trips over some other threshold at a "backedge" (end of a loop body, just about to jump back to the beginning), an OSR compilation task is triggered. The place that triggered the compilation will be where the actual method entry point will be in the compiled method.

If there are multiple loops in a Java method, nested or not, the end of every loop's body is a potential trigger/entry point for OSR compilation. A "place" in a Java method is defined by its bytecode index (BCI), and the place that triggered an OSR compilation is called the "`osr_bci`". An OSR-compiled `nmethod` can only be entered from its `osr_bci`; there can be multiple OSR-compiled versions of the same method at the same time, as long as their `osr_bci` differ.

The format string for an OSR compilation:

```
%3d%c%c%c%c%c %s @ %d (%d bytes)
```

and the meanings are:

```
{compile_id}{method_attributes} {short_name} @ {osr_bci} ({size} bytes)
```

Example:

```
  1%      groovy.lang.MetaClassImpl::applyStrayPropertyMethods @ 57 (232 bytes)
```

Tiered compilation

Starting from early versions of JDK 6, HotSpot incorporates a new mode of execution: the "tiered" mode. In this mode, the interpreter, the client compiler (C1) and the server compiler (C2) would work together in a single VM, to get the best of the world in all phases of execution: start up, warm, steady.

Before tiered compilation, a HotSpot VM could either use C1 (the Client VM) or C2 (the Server VM) as the JIT compiler, but not both in a single VM. With tiered compilation, the HotSpot Server VM could use both compilers at the same time, for different compilation tasks.

Check out Steve Goldman's Weblog [7] to find out more about the early development of HotSpot's tiered compilation.

As an aside, for those not familiar with the guts of modern JVMs, tiered compilation is not unique to HotSpot; for example, IBM's JVMs have had it for a long time [8]; As for JRockit, an except from [9] tell the story:

> Later, as JRockit became a major mainstream JVM, known for its performance, the need to diversify the code pipeline into client and server parts was recognized. No interpreter was added, however. Rather the JIT was modified to differentiate even further between cold and hot code, enabling faster "sloppy" code generation the first time a method was encountered. This greatly improved startup time to a satisfying degree, but of course, getting to pure interpreter speeds with a compile-only approach is still very hard.

In fact it's becoming a common practice in a lot of VMs. Even V8 uses tiered compilation in Chrome 10 or later [10], featuring a baseline compiler (the original compiler) and an optimizing compiler (a lot like HotSpot's C1 compiler).

Back to `PrintCompilation`'s output. Tiered compilation in earlier versions of JDK 6's HotSpot VM was not fully developed yet. Seems like it was only 3 tiers (0 to 2) even though there were VM flags for 5 tiers (0 to 4).

```
// Enumeration to distinguish tiers of compilation
enum CompLevel {
  CompLevel_none               = 0,
  CompLevel_fast_compile       = 1,
  CompLevel_full_optimization  = 2,

  CompLevel_highest_tier       = CompLevel_full_optimization,
#ifdef TIERED
  CompLevel_initial_compile    = CompLevel_fast_compile
#else
  CompLevel_initial_compile    = CompLevel_full_optimization
#endif // TIERED
};
```

A Java method will either be compiled

- at tier 1 by C1 with full profiling (`CompLevel_fast_compile`), or
- at tier 2 by C2 (`CompLevel_full_optimization`),

otherwise it'd be running at tier 0 (`CompLevel_none`) by the interpreter.

For `PrintCompilation`'s output, in "tiered" mode, a compilation task to be carried out by C1 will print `1` at the end of the method attribute flags, and the ones carried out by C2 will print *empty*.

An example of `PrintCompilation` log for a Java method to be compiled by C1 in tiered mode:

```
375 s! 1  java.beans.PropertyDescriptor::getReadMethod (173 bytes)
```

JDK 6 update 25 to 27 (HotSpot 20)

This is after 6953144 and before 7022998. The HotSpot VM in these JDK versions features a much improved implementation of tiered compilation, which finally uses all 5 tiers (0 to 4) together.

The major changes on `PrintCompilation`'s output involve two parts:

- an additional column for time stamps, and
- a new compilation level format for "tiered" mode.

The basic format string for non-tiered mode:

```
%7d %3d%c%c%c%c%c %s (%d bytes)
```

and the meaning is:

```
{timestamp} {compile_id}{method_attributes} {short_name} ({size} bytes)
```

Timestamp

This columns shows the timestamp in milliseconds when this log is printed, which in turn is when a compiler is invoked to run the compilation task.

Compilation number (`compile_id`)

There's no change in format here, might careful readers should quickly notice something is different if in tiered mode:

Non-tiered mode:

```
355   1       java.util.jar.Manifest$FastInputStream::readLine (167 bytes)
356   2       sun.nio.cs.UTF_8$Decoder::decodeArrayLoop (553 bytes)
494   3       java.lang.String::indexOf (151 bytes)
```

Tiered mode:

```
223  10      3  java.lang.String::indexOf (151 bytes)
285   3      3  java.lang.String::hashCode (64 bytes)
286   1      3  java.lang.String::equals (88 bytes)
```

See? `compile_id` used to be in strictly ascending in the logs, but now in tiered mode the order seem to have gone wild.

In previous version of JDK 6's HotSpot VM, there's only one compilation task queue, even in tiered mode; the queue conforms to a FIFO order.

This version of HotSpot VM, however, uses a new `AdvancedThresholdPolicy` by default when in tiered mode. This compilation policy picks a method with the highest rate of events from the compilation task queue as the current task, and it no longer conforms to a FIFO order. The "event count" is just "invocations + loop counts"; in other words, it's trying to pick the hottest method in the queue, so that hottest code gets served first.

(Strangely, `AdvancedThresholdPolicy` doesn't exist in HotSpot Express 20's or OpenJDK 6 build 23's HS20-b11, but exists in JDK6u25's HS20-b11. I wonder why.)

Method attributes

The new set of method attibute flags:

```
// print method attributes
const bool is_osr = bci != InvocationEntryBci;
const char blocking_char  = is_blocking     ? 'b' : ' ';
const char compile_type   = is_osr          ? '%' : ' ';
const char sync_char      = is_synchronized ? 's' : ' ';
const char exception_char = has_xhandler    ? '!' : ' ';
const char native_char    = is_native       ? 'n' : ' ';
st->print("%c%c%c%c%c ", compile_type, sync_char, exception_char, blocking_char, native_char);
if (TieredCompilation) {
  st->print("%d ", comp_level);
}
```

The most significant change is:

- Compilation level is always printed in tiered mode (unlike the previous implementation which only printed the flag for tier 1 compilations).

`is_native` isn't really used here. The `PrintCompilation` output for native wrapper is still done elsewhere in this version.

Deoptimization

A couple of new `DeoptReason`s have been added in HotSpot 20 to facilitate more optimizations:

```
// What condition caused the deoptimization?
enum DeoptReason {
  Reason_many = -1,             // indicates presence of several reasons
  Reason_none = 0,              // indicates absence of a relevant deopt.
  // Next 7 reasons are recorded per bytecode in DataLayout::trap_bits
  Reason_null_check,            // saw unexpected null or zero divisor (@bci)
  Reason_null_assert,           // saw unexpected non-null or non-zero (@bci)
  Reason_range_check,           // saw unexpected array index (@bci)
  Reason_class_check,           // saw unexpected object class (@bci)
  Reason_array_check,           // saw unexpected array class (aastore @bci)
  Reason_intrinsic,             // saw unexpected operand to intrinsic (@bci)
  Reason_bimorphic,             // saw unexpected object class in bimorphic inlining (@bci)

  Reason_unloaded,              // unloaded class or constant pool entry
  Reason_uninitialized,         // bad class state (uninitialized)
  Reason_unreached,             // code is not reached, compiler
  Reason_unhandled,             // arbitrary compiler limitation
  Reason_constraint,            // arbitrary runtime constraint violated
  Reason_div0_check,            // a null_check due to division by zero
  Reason_age,                   // nmethod too old; tier threshold reached
  Reason_predicate,             // compiler generated predicate failed
  Reason_LIMIT,
  // Note:  Keep this enum in sync. with _trap_reason_name.
  Reason_RECORDED_LIMIT = Reason_bimorphic  // some are not recorded per bc
  // Note:  Reason_RECORDED_LIMIT should be < 8 to fit into 3 bits of
  // DataLayout::trap_bits.  This dependency is enforced indirectly
  // via asserts, to avoid excessive direct header-to-header dependencies.
  // See Deoptimization::trap_state_reason and class DataLayout.
};
```

There's a good example of invalidating an OSR-compiled method in the JDK 6 update 25 example output:

```
 2619   2%      groovy.lang.MetaClassImpl::applyStrayPropertyMethods @ 57 (232 bytes)
 2871   2%     made not entrant  groovy.lang.MetaClassImpl::applyStrayPropertyMethods @ -2 (232 bytes)
```

```
   29456   2%     made zombie  groovy.lang.MetaClassImpl::applyStrayPropertyMethods @ -2 (232 bytes)
```

We can confirm these 3 lines are talking about the same `CompileTask` by looking at their `compile_id`. But notice that `osr_bci` changed from `57` to `-2` when the `nmethod` was made not entrant. This is not a bug; it's a side effect from the process of unlinking/invalidating the `nmethod`, by setting its `_entry_bci` to `InvalidOSREntryBci`:

```
// Handy constants for deciding which compiler mode to use.
enum MethodCompilation {
  InvocationEntryBci = -1,     // i.e., not a on-stack replacement compilation
  InvalidOSREntryBci = -2
};
```

In tiered mode, `PrintCompilation`'s output for deoptimizations prints the compilation level in the same way as normal compilation task. Example:

```
    2399 257      3 made not entrant  java.util.Arrays::binarySearch0 (95 bytes)
```

The `3` in this example is the compilation level.

Tiered compilation

The format string in tiered mode:

```
%7d %3d%c%c%c%c%c %d %s (%d bytes)
```

the meaning:

```
{timestamp} {compile_id}{method_attributes} {comp_level} {short_name} ({size} bytes)
```

The new tiered system consists of these compilation levels:

```
// Enumeration to distinguish tiers of compilation
enum CompLevel {
  CompLevel_any               = -1,
  CompLevel_all               = -1,
  CompLevel_none              = 0,         // Interpreter
  CompLevel_simple            = 1,         // C1
  CompLevel_limited_profile   = 2,         // C1, invocation & backedge counters
  CompLevel_full_profile      = 3,         // C1, invocation & backedge counters + mdo
  CompLevel_full_optimization = 4,         // C2

#if defined(COMPILER2)
  CompLevel_highest_tier      = CompLevel_full_optimization,  // pure C2 and tiered
#elif defined(COMPILER1)
  CompLevel_highest_tier      = CompLevel_simple,             // pure C1
#else
  CompLevel_highest_tier      = CompLevel_none,
#endif

#if defined(TIERED)
  CompLevel_initial_compile   = CompLevel_full_profile        // tiered
#elif defined(COMPILER1)
  CompLevel_initial_compile   = CompLevel_simple              // pure C1
#elif defined(COMPILER2)
```

```
    CompLevel_initial_compile   = CompLevel_full_optimization   // pure C2
  #else
    CompLevel_initial_compile   = CompLevel_none
  #endif
  };
```

See [11] for details of `AdvancedThresholdPolicy`. There's a big section of comment in advancedThresholdPolicy.cpp that describes method states and common state transition patterns, very worth reading.

Native wrapper

The `PrintCompilation` output for native wrapper remains unchanged in this version. But because of the additional timestamp column for normal Java methods, the output is not aligned as it was. Doesn't really matter, though.

JDK 7 (HotSpot 21)

This is after 7022998. There's been a lot of changes between HS20 and HS21, including a refactoring of `PrintCompilation`'s output. Handling for normal Java methods and native wrappers are finally unified into a single method.

The basic format string for non-tiered mode:

```
%7d %4d %c%c%c%c%c      %s (%d bytes)
```

and the meaning is:

```
{timestamp} {compile_id} {method_attributes}      {short_name} ({size} bytes)
```

Notice the column for `compile_id` is now aligned to 4 characters, which is a sign of expecting more compilation tasks to be run.

Compilation number (`compile_id`)

Native wrappers are assigned a `compile_id` again, sharing the counter sequence of standard compilations. So there's no more "---" placeholder in the log now.

Method attributes

```
// method attributes
const char compile_type   = is_osr_method                  ? '%' : ' ';
const char sync_char      = is_synchronized                ? 's' : ' ';
const char exception_char = has_exception_handler          ? '!' : ' ';
const char blocking_char  = is_blocking                    ? 'b' : ' ';
const char native_char    = is_native                      ? 'n' : ' ';

// print method attributes
st->print("%c%c%c%c%c ", compile_type, sync_char, exception_char, blocking_char, native_char);

if (TieredCompilation) {
  if (comp_level != -1)  st->print("%d ", comp_level);
  else                   st->print("- ");
}
```

Nothing much new here, except that in tiered mode, for native wrappers it's supposed to print a `-` as a placeholder for compilation level [12], but actually `0` is printed. The actual behavior is unintended; native wrapper `nmethod`s are generated with a compilation level of `CompLevel_none` (0) instead of `-1`, that's why it doesn't work.

Deoptimization

One new `DeoptReason` was added in this version, `Reason_loop_limit_check`. It's a part of the fix for 5091921.

```
// What condition caused the deoptimization?
enum DeoptReason {
  Reason_many = -1,             // indicates presence of several reasons
  Reason_none = 0,              // indicates absence of a relevant deopt.
  // Next 7 reasons are recorded per bytecode in DataLayout::trap_bits
  Reason_null_check,            // saw unexpected null or zero divisor (@bci)
  Reason_null_assert,           // saw unexpected non-null or non-zero (@bci)
  Reason_range_check,           // saw unexpected array index (@bci)
  Reason_class_check,           // saw unexpected object class (@bci)
  Reason_array_check,           // saw unexpected array class (aastore @bci)
  Reason_intrinsic,             // saw unexpected operand to intrinsic (@bci)
  Reason_bimorphic,             // saw unexpected object class in bimorphic inlining (@bci)

  Reason_unloaded,              // unloaded class or constant pool entry
  Reason_uninitialized,         // bad class state (uninitialized)
  Reason_unreached,             // code is not reached, compiler
  Reason_unhandled,             // arbitrary compiler limitation
  Reason_constraint,            // arbitrary runtime constraint violated
  Reason_div0_check,            // a null_check due to division by zero
  Reason_age,                   // nmethod too old; tier threshold reached
  Reason_predicate,             // compiler generated predicate failed
  Reason_loop_limit_check,      // compiler generated loop limits check failed
  Reason_LIMIT,
  // Note:  Keep this enum in sync. with _trap_reason_name.
  Reason_RECORDED_LIMIT = Reason_bimorphic  // some are not recorded per bc
  // Note:  Reason_RECORDED_LIMIT should be < 8 to fit into 3 bits of
  // DataLayout::trap_bits.  This dependency is enforced indirectly
  // via asserts, to avoid excessive direct header-to-header dependencies.
  // See Deoptimization::trap_state_reason and class DataLayout.
};
```

For `PrintCompilation`'s output, deoptimization messages have been moved from the middle to the end of the line. So the new log looks like:

```
 1989   75             groovy.lang.MetaClassImpl::applyStrayPropertyMethods (232 bytes)   made not entrant
20498   75             groovy.lang.MetaClassImpl::applyStrayPropertyMethods (232 bytes)   made zombie
20498    2 %           groovy.lang.MetaClassImpl::applyStrayPropertyMethods @ -2 (232 bytes)   made zombie
20498   94             groovy.lang.MetaClassImpl::applyStrayPropertyMethods (232 bytes)   made zombie
```

This is more aligned with the normal lines, which is both a good and a bad thing: the good thing is the log looks cleaner, and the bad thing is it's a little bit harder to spot the deoptimization message if you're used to the old format.

Native wrappers

In addition to the `compile_id` and compilation level changes, `PrintCompilation`'s output for native wrappers prints `(0 bytes)` at the end again.

Example in tiered mode:

```
       91    7     n 0       java.lang.System::arraycopy (0 bytes)   (static)
```

Using PrintCompilation with other VM flags

//// TODO

`PrintInlining`

Related flags

`LogCompilation` is another VM flag for obtaining statistics on HotSpot's compilations. A tool is included in OpenJDK source distributions to parse `LogCompilation` output into a more human readable format. See [13] for details.

//// TODO

`TraceDeoptimization`

References

[1]: Moazam Rajas, The PrintCompilation flag and how to read its output (Link is dead; an archived version is available at the Internet Archive)

[2]: Sun Microsystems, Java SE 6 Performance White Paper

[3]: Craig Chambers, David Ungar, Making Pure Object-Oriented Languages Practical

[4]: Stephen J. Fink, Feng Qian, Design, Implementation and Evaluation of Adaptive Recompilation with. On-Stack Replacement

[5]: US Pantent 20090064113: Method And System For Dynamic Loop Transfer By Populating Split Variables

[6]: Michael Paleczny, Christopher Vick, Cliff Click, The Java HotSpotTM Server Compiler

[7]: Steve Goldman's Weblog

[8]: Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, Toshio Nakatani, Design and evaluation of dynamic optimizations for a Java just-in-time compiler

[9]: Marcus Hirt, Marcus Lagergren, Oracle JRockit: The Definitive Guide, Page 47, "On-stack replacement"

[10]: Kevin Millikin, The Chromium Blog, A New Crankshaft for V8

[11]: OpenJDK 7 Source Repository, 7020403: Add AdvancedCompilationPolicy for tiered

[12]: Christian Thalinger, hotspot-compiler-dev, Request for review (S): 7022998: JSR 292 recursive method handle calls inline themselves infinitely

[13]: Sun Wikis, LogCompilation overview