LogCompilation overview

This is a very rough overview of the LogCompilation output of HotSpot. It probably deserves more detail and maybe a schema for the XML would make it all clearer.

The diagnostic options -XX:+LogCompilation emits a structured XML log of compilation related activity during a run of the virtual machine. By default it ends up in the standard hotspot.log file, though this can be changed using the -XX:LogFile= option. Note that both of these are considered diagnostic options and have to be enabled using -XX:+UnlockDiagnosticVMOptions. Note that support for this by the client compiler is incomplete so it will not report any inlining decisions.

The log actually consists of two parts. The main log is output from normal Java threads and contains events for methods being enqueued for compilation, uncommon traps that invalidate a method and other events. The other part of the log comes from the compiler threads themselves. During the execution the compiler threads write their logs to temporary files with names of the form hs_c#_<pid>.log where # is the number of the thread using by the compiler thread. At VM shutdown the separate logs are appended onto the main hotspot.log to form the final complete log.

The output from the compiler is basically a log of the stages of the compile along with the high level decisions made during the compile such as inlining. The individual log for a compiler thread is wrapped by the compilation_log element which has a single attribute 'thread' which is the id thread in which the compiler runs.

<compilation_log thread=''>
..
</compilation_log>

There's a start_compile_thread element which mainly gives a timestamp for the start time.

<start_compile_thread thread='1090857296' process='6053' stamp='1.241'/>

Many elements contain time stamps as the end which can be used to order them relative to events in other threads and the measure elapsed time. The time stamp is in seconds since the start of the VM and the start time of the VM is recorded in the hotspot_log element in the time_ms attribute.

Methods to be compiled are placed into the compile queue and the compiler threads dequeue them and compile them as long as there are elements in the queue. Each individual compile is shown as a 'task' element.

<task compile_id='1' method='java/lang/Thread <init> (Ljava/lang/ThreadGroup;Ljava/lang/String;)V' bytes='49' count='6' iicount='6' blocking='1' stamp='1.242'>
...
<task_done success='1' nmsize='1216' count='0' inlined_bytes='111' stamp='1.550'/>
</task>

Every compile is assigned a compile id by the system at the point is is enqueued and that's recorded as the compile_id attribute. Some compiles are on stack replacement (OSR) compile where the code is going to be used to replace an already existing activation. These are tagged by the compile_kind attribute as compile_kind='osr'. It can also have the value c2n for compiled native wrappers which are used to call JNI methods.

The 'method' attribute is a string version of the VM name of the method with spaces separating the class, method name signature.

The 'bytes' attribute is number of bytecodes in the method.

'count' is the invocation count as recorded by the method invocation counters. This may not appear in more recent versions of the log output. Note that these counters are mainly used for triggering compiles and are not guaranteed to be an accurate reflection of the number of times a method has actually executed. Multiple threads may be updated these counters and sometimes the VM will reset a counter to a lower value to delay retriggering of compiles.

'iicount' is the interpreter invocation count. This is a separate
copy of the invocation count which is maintained by the profiling
support. Again it's not guaranteed to be accurate since multiple
threads may update it but it's never reset so it's reasonably
accurate.

'blocking' indicates whether the thread that requested the compile is
waiting for the compile to finish before proceeding. By default
threads don't wait but using -XX:-BackgroundCompilation or -Xbatch
will cause them to wait.

The task_done element indicates completion of the compile and includes
a 'success' element to indicate whether the compile succeeded. If it
succeeded there will be an 'nmsize' attribute which indicates the
number of bytes of instruction produced by the compile. The main log
includes a more detailed element describing the resulting nmethod and
that is described elsewhere. 'inlined_bytes' indicates how many
bytecodes where inlined from other methods during the compile.

Each phase of the compilation is wrapped by a 'phase' element which
records the name of the phase, the maximum number of nodes in the IR
at that point and the timestamp.

<phase name='parse' nodes='3' stamp='1.543'>
...
<phase_done nodes='403' stamp='1.545'/>
</phase>

Phases may nest and may be repeated. Some phases may only run if
certain flags are on or if certain conditions are met.

In C2 the rough structure of the phases is this:

- parse
- escapeAnalysis
- optimizer
    - iterGVN
    - idealLoop
    - ccp
    - iterGVN2
    - idealLoop
    - macroExpand
    - graphReshape
- matcher
- scheduler
- regalloc
- output
- install_code

The parse phase may be repeated recursively when handling inlining.

<type id='482' name='void'/>

Many of the elements have to reference classes or methods and this is
done by emitting elements that describe "id" to "name" mappings for
them. "type" elements describe primitive types.

<klass id='575' name='[B' flags='1041'/>
<klass id='608' name='sun/misc/URLClassPath$FileLoader' unloaded='1'/>

"klass" elements describe instance and array types. These either have
a "flags" attribute which corresponds to the class flags from the
class file or an "unloaded" attribute that indicates the class wasn't
loaded at that point.

<method id='586' holder='521' name='checkBounds' return='482' arguments='575 480 480' flags='10' bytes='46' iicount='51'/>

"method" elements refers back to previously mentioned type or klass elements.

"holder" is the id of klass that declares the method. "name" is the
method name. "arguments" is a list of ids that correspond to the
argument types. "flags" is the method access flags encoded as the
classfile would. "bytes" is the number of bytecodes. "iicount" is
the interpreter invocation count at the point it was emitted into the log.

<bc code='190' bci='28'/>

Many pieces of output are prefixed by "bc" elements which indicate what the current bytecode in the parse is. "code" is the number of the bytecode itself and "bci" is the current bci.

<dependency type='unique_concrete_method' ctxk='586' x='604'/>

A "dependency" element indicates that class hierarchy analysis has indicated some interesting property of the classes that allows the compiler to optimistically assume things like there are no subclasses of a particular class or there is only one implementor of a particular method. The "type" attribute describes what kind of dependence this is. The "ctkx" attribute is a context class for the dependence. The "x" attribute is the method involved in the dependence.

<uncommon_trap method='589' bci='36' reason='unloaded' action='reinterpret' index='46' klass='608'/>

<uncommon_trap bci='28' reason='null_check' action='maybe_recompile'/>

"uncommon_trap" elements correspond to points in the bytecode parsing where the compiler decided to stop parsing the bytecodes and emit an uncommon trap. Uncommon traps are always emitted to handled unloaded classes by dropping out of the compiled code and back into the interpreter. They are also emitted in cases where some particular path or code pattern is considered uncommon and the compiler puts in the uncommon trap so that if that path becomes more common it can detect it so the method can be recompiled.

The uncommon trap element is used in 2 different ways in the output. It describes the emission of the trap during parse and also describes the event when the trap triggers during execution. The attributes in these 2 cases are different. All uncommon traps have "reason" and "action" attributes that drive their behaviour.

These are the reasons:

- array_check
- class_check
- div0_check
- intrinsic
- null_assert
- null_check
- range_check
- unhandled
- uninitialized
- unloaded

And these are the possible actions

- make_not_entrant
- maybe_recompile
- none
- reinterpret

They also have "bci" attributes to indicate where they occur in the current parse. Some have "index" attributes that refer back to the constant pool of containing method and indicate an entry which needs to resolved.

Sometimes "uncommon_trap" elements are used to handle exception paths so they will occur inline with before other elements. In this case they haven't terminated the main parse but only the exception path.

uncommon_trap elements may occur outside of the compilation_log portion of the output and these indicate that the uncommon trap in the generated code was executed. These include a "thread" attribute for the executing thread, the "compile_id", "compiler" and "level" of the nmethod that contained the trap.

<call method='596' count='-1' prof_factor='1'/>

"call" elements are emitted when processing call sites in the bytecodes. "method" refers to a method attribute describing the method to be invoked. "count" is the count of the number of times this site was execute according to the profile data. A "virtual" attribute indicates that we can't bind the method statically. The "inline" attribute indicates that inlining is allowed at this site. If the profile information includes a type profile of the possible receivers this will be included as "receiver", "receiver_count", "receiver2" and "receiver2_count".

The call element may be followed by several different attributes to indicate how it was handled. It may be followed by an "uncommon_trap" which indicates that something about the call was unhandled or it wasn't expected to be needed. If it's followed by "parse" then the method was inlined.

<inline_fail reason='already compiled into a medium method'/>
<direct_call bci='115'/>

A "direct_call" following the "call" means that we just emitted a normal call to the method. It may be preceded by an "inline_fall" attribute if we attempted to inlined but decided against it because of the inlining heuristics. It includes a "reason" attribute which is a textual description from the inlining logic.

<intrinsic id='_compareTo' nodes='27'/>

An "intrinsic" is a special internal implementation of a method. The "id" attribute is the internal name of the intrinsic. The actual method it applies to should be the one mentioned in the original "call" element.

Some call sites will be handled by using type profile information to emit type checks for common types and execute inlined versions for those cases. The output for these is somewhat complicated. "call" elements will be emitted for the expected methods followed by "predicted_call" elements.

<predicted_call bci='4' klass='600'/>

This indicates that a check for "klass" was emitted. These may also be followed by "uncommon_trap" elements. The sequence may be terminated by a class_check uncommon_trap with the comment "monomorphic vcall checkcast" to indicate that the predicted types were the only type we saw so any other type should cause an uncommon trap so the method can be recompiled. The whole sequence will then be followed by the code generation for each of the predicted methods which should be a "parse" phase for the inlining.