Understanding HotSpot in Plain English

Posted At : April 28, 2006 12:57 PM | Posted By : Steven Erat
Related Categories: Java, Quality Assurance, ColdFusion, Computer Technology

Not having a degree in Computer Science means that my view of web technology begins at a granular level and necessarily seeps from the top down into the foundations of how software works. For ColdFusion this course traverses the basics of HTTP and CSS, expands into the full range of CFML tags, functions, and architecture, then arcs into databases and SQL, and continues well into the depths of Java and J2EE technology and all that's contained therein. One low level topic that has often piqued my interest has been the question of what **exactly** is the Sun HotSpot technology for Java Virtual Machines? Sure many ColdFusion pundits share recommendations for JVM tuning parameters, but if you're like me you find it rather superficial and want to know more about this HotSpot black box.

Many resources will point to the Sun whitepaper on Java HotSpot Technology, but I've not been fully satisfied with the high level view and marketing speak there. Recently I've discovered several helpful explanations, and today I've read one of the more comprehensive articles that's written in plain english about what HotSpot and JIT really are, including a great sidebar that could be an article all by itself which explains exactly what a compiler, language interpreter, and bytecode interpreter are and how they work. The article was written back in 1998, preceding the formal release of Sun HotSpot, but the content definitely hit the spot for me. I think you'll find it very helpful in the context of ColdFusion MX applications.

> **HotSpot: A new breed of virtual machine**
> Sun's HotSpot technology promises to deliver interpreted bytecodes that run faster than a compiled program! Is such a thing possible? Apparently. HotSpot combines a Java virtual machine (JVM) and a compiler in a unique new way that up until now has existed only in universities. The result is a powerful new technology that threatens to blow the doors off language performance as we know it. Find out about the inner workings of Sun's dynamic compiler, and learn which kinds of applications are -- and are not -- more effective with HotSpot than with the fastest JIT. Plus: Primers compare HotSpot technology to its predecessors -- standard VMs, compilers, and just-in-time compilers. (4,000 words)
> By Eric Armstrong

> **Watch your HotSpot compiler go**
> Together we have seen that a given piece of Java code will execute at very different speeds during an application's lifetime. Initially, it might execute in interpreted mode; if it executes enough times, it becomes a candidate for compilation to native code.
>
> For a Java program with high uptime, such as an application server, most methods will translate into native code sooner or later. At this state of the JVM, profiling gives a realistic picture of the application's performance. Profiling data taken early in the application's lifetime should either be discarded or not collected at all (unless you are specifically profiling your application's startup behavior). Unfortunately, I suspect that most programmers do not take this into proper consideration. Sun Microsystems' server HotSpot JVM has a larger default method invocation threshold than the client JVM (10,000 instead of 1,500 (see Resources)), and it might take minutes, if not hours, to properly warm up the JVM.
> By Vladimir Roubtsov

> **HotSpot on Wikipedia**
> Its name derives from the fact that as it runs Java byte-code, it continually analyzes the program's performance for "hot spots" which are frequently or repeatedly executed. These are then targeted for optimization, leading to high performance execution with a minimum of overhead for less performance-critical code. HotSpot is widely acclaimed as providing the best performance in its class of JVM. In theory, though rarely in practice, it is possible for adaptive optimization of a JVM to exceed hand coded C++ or assembly language.

These articles help me better understand how the JVM behaves underneath a ColdFusion MX server and application, although they don't teach the art of JVM tuning through modulation of HotSpot VM Options. That is the next step.

From these I've learned that since ColdFusion MX ships with the -server switch enabled, performance observed during load tests may not be completely accurate until the application has been exercised for hours because the HotSpot threshold for compiling bytecode to native code is quite high in that configuration. During initial testing periods after startup, the JVM is largely operating in interpreted mode only.

This would also help explain why JVM crashes leaving behind HotSpot crash logs happen after hours or days of uptime... The demand on the application has risen to the level where HotSpot begins to do its thing, finally exacerbating any HotSpot flaws or weaknesses. One example that comes to mind is these errors which have been reported infrequently in ColdFusion apps but their impact is that the JVM crashes:

```
Exception in thread "CompilerThread0" java.lang.OutOfMemoryError: requested 141380 bytes for Chunk::new. Out of swap space?
```

```
Exception java.lang.OutOfMemoryError: requested 1024000 bytes for GrET* in D:/BUILD_AREA2/jdk1.4.2/hotspotsrcsharevmutilitiesgrowableArray.cpp. Out of swap space?
```

While these errors are found among various Sun bug reports, the only easy solution is an indirect one, to disable the HotSpot compiler entirely with the -Xint optional argument which forces it to only run in interpreted mode. The errors might be caused when HotSpot begins to optimized certain frequently run methods or tight blocks of code. When this happens the compiler will convert the interpreted bytecode to native code and tuck it away in a little piece of memory. The compiler probably needs a little chunk of contiguous memory to store the optimized native code and the OS was unable to provide it. Notice the errors usually indicate that attempts to allocate 1-2MB of memory failed, even though its often noted that neither the JVM heap nor the OS memory usage has maxed out at the time of crash.

A more careful approach to resolve the crashes would be to determine the method which crashed, sometimes identified in the hs_err_pidNNNNN.log files, and add them to a list of methods to exclude from optimization. To do this create a file called .hotspot_compiler (note the ' . ' there) in the cf_root/runtime/bin (or jrun_root/bin) directory and add to it individual lines for methods to exclude. Say the crash log identified coldfusion.runtime.CFPage::Duplicate, put this on one line in the .hotspot_compiler file:

```
exclude coldfusion/runtime/CFPage Duplicate
```

This approach may require a great deal of trial and error and could require a lot of time to get right. Disabling the HotSpot compiler with -Xint should be used as a good first pass since it may be possible that the applications run on the ColdFusion server don't normally take much advantage of the HotSpot compiler and so you might find that there is no clearly observable decline in performance. Its often said that forcing HotSpot to run in interpreted mode will always decrease performance, but I know first hand of several who have found it very satisfactory.

As a rule of thumb, the more homogenous the code, the more computationally intensive the application, then the more the application will benefit from HotSpot and is more likely to show a decline in performance when disabled. For a codebase that is built around a lot of I/O like file system operations or network/database calls, then HotSpot will be less utilized and you're more likely to not show a drop if HotSpot were disabled. This should be true for applications that are heterogeneous as well, such as on an ISP, where the code is diverse without much repetition in the codebase.