Performance Considerations for Running ColdFusion 8 in 64-bit Mode

Posted At : August 29, 2007 1:47 PM | Posted By : Steven Erat
Related Categories: Java, Adobe, Linux, Mac OS X, ColdFusion, Travel, Computer Technology, Books

In yesterday's post about configuration nuances of using a 64-bit webserver and 64-bit JVM with ColdFusion 8 on the 64-bit Sun Solaris OS, Damon Gentry posted a comment that is, frankly, way above my head.

> I'm curious about if there are any performance gains by running CF8 with a 64-bit JVM. More specifically, given the CPU architecture differences between Intel/AMD, and Sparc (speed vs. cores), does it make since to stick with Solaris? I know that the Sparc T1 can support 32 cores, albeit at 1.2 GHz, whereas the Intel CPU can support 4 cores @ 3.6GHz. [*more*]

The short answer is, "I don't know".

64-bit Basics
Ok, so I'm not a computer scientist. I don't even have a computer science degree. However, I do have Google. And Wikipedia. And the rest of the Web. So, I've filtered through a variety of articles and selected the following to help inform me on the topic:

- 64-bit
  Wikipedia
- Frequently Asked Questions About the Java HotSpot VM
  Sun
- Tuning Garbage Collection with the Java 5 JVM
  Sun [see Types of Collectors]
- ColdFusion :: 64-Bit and What It Means To You
  Andrew Powell's Blog
- The 64-Bit Advantage
  PC Magazine

All of these articles are quite long, and I encourage you to read them if this subject interests you. Since I cannot precisely answer Damon's question, I'll try to summarize relevant information that I have gleaned from them about running a Java-based web application on a 64-bit JVM/OS. If you want details about any inferences, you should read the above articles, although I may end up quoting liberally here.

32-bit Applications
Any 32-bit application, not just web applications on a JVM, can theoretically use a maximum of 4 GB memory, per process, when running on a 32-bit platform. However, on 32-bit platforms, Windows at least, the practical limit for JVM-based applications is known to be closer to 1.5 GB. This is because part of the process address space is used for other OS-level purposes not specific to the application.

In contrast, when running an application on a 32-bit JVM on 64-bit Solaris, the application's process can use up to full 4 GB limit. Solaris/Sparc/UltraSparc configurations out there have been 64-bit for many years now, so when running any recent version of ColdFusion (CFMX 6 and up) on a 32-bit JVM (the default in CFMX 6 to CF8) on a recent version of Solaris (at least Solaris 9 and 10), the ColdFusion server's JVM heap should be able to utilize up to 4 GB. I would expect the same to be true about running ColdFusion in 32-bit mode on other 64-bit platforms as well.

> **From Wikipedia**:
> Some operating systems reserve portions of process address space for OS use, effectively reducing the total address space available for mapping memory for user programs. For instance, Windows XP DLLs and userland OS components are mapped into each process's address space, leaving only 2 to 3.8 GB (depending on the settings) address space available, even if the computer has 4 GiB of RAM. This restriction is not present in 64-bit Windows.

> **From Sun**:
> The maximum theoretical heap limit for the 32-bit JVM is 4G. Due to various additional constraints such as available swap, kernel address space usage, memory fragmentation, and VM overhead, in practice the limit can be much lower. On most modern 32-bit Windows systems the maximum heap size will range from 1.4G to 1.6G. On 32-bit Solaris kernels the address space is limited to 2G. On 64-bit operating systems running the 32-bit VM, the max heap size can be higher, approaching 4G on many Solaris systems. ... If your application requires a very large heap you should use a 64-bit VM on a version of the operating system that supports 64-bit applications.

64-bit Applications and Memory
So far the articles indicate that increased application memory limit as the primary advantage to running in 64-bit mode. In fact, theoretical memory limit on a 64-bit application is reported to be 16 EB where EB is exabytes. In more familiar terms this is 16 billion Gigabytes. In practical terms this is effectively an unlimited amount of memory per process, so the limit becomes a question of how much memory can an operating system support. The **Wikipedia article** indicates that MacBook Pro can support up to 16 GB, and recent 2.6 versions of the Linux kernel can support up to 64 GB. This **Sunfire comparison matrix** by Sun indicates support for up to 384 GB memory across 24 CPUs running Solaris 10 (I don't know if its a fair metric, but that works out to 16 GB memory / CPU).

64-bit Applications and Performance
Both the **Wikipedia article** and the **PC Magazine article** indicate that 64-bit applications should perform CPU intensive operations (floating point calculations),

multi-tasking, and large data set operations faster than 32-bit platforms.

Based on this I speculate that a ColdFusion application that relied heavily upon database queries returning "normal" size record sets and did a lot of disk or network I/O (cffile, cfimage, cfftp, etc) probably would *not* benefit from 64-bit. But an application that routinely worked with very large cached record sets, generated lots of charts, graphs, and PDFs, and extremely large amounts of data in the application and session scopes very likely *would* benefit from running in 64-bit mode.

The Java HotSpot FAQ notes, however, that since native pointers simply take up more space in 64-bit platforms (8 bytes rather than 4 bytes), performance may degrade depending on how many pointers are loaded during execution (I assume this translates to how many objects are referenced in JVM terms). Sun notes some interesting hardware differences regarding this:

> Java HotSpot FAQ:
> Generally, the benefits of being able to address larger amounts of memory come with a small performance loss in 64-bit VMs versus running the same application on a 32-bit VM. This is due to the fact that every native pointer in the system takes up 8 bytes instead of 4. The loading of this extra data has an impact on memory usage which translates to slightly slower execution depending on how many pointers get loaded during the execution of your Java program. The good news is that with AMD64 and EM64T platforms running in 64-bit mode, the Java VM gets some additional registers which it can use to generate more efficient native instruction sequences. These extra registers increase performance to the point where there is often no performance loss at all when comparing 32 to 64-bit execution speed. The performance difference comparing an application running on a 64-bit platform versus a 32-bit platform on SPARC is on the order of 10-20% degradation when you move to a 64-bit VM. On AMD64 and EM64T platforms this difference ranges from 0-15% depending on the amount of pointer accessing your application performs.

Damon's question regarding differing performance of 64-bit mode across varying CPU architecture is still one I cannot answer. This citation from Sun seems to indicate that an AMD64 chip (supported by Red Hat or Suse Linux) would perform better than Sparc. This question is better suited for a Sun expert than an Adobe one.

64-bit Java Applications and Garbage Collection
In 64-bit mode a Java application will be able to hold references to many more Java objects when the Java heap is increased accordingly. If you're still reading this then you probably know a thing or two about Garbage Collection. There are (at least) 4 GC algorithms available, Serial, Throughput, Low Pause, and Intermittent. The former is the default in a JVM if not otherwise specified and has the most notable interruptions while the application halts completely as GC is performed. The latter is deprecated and no longer recommended for use. This leaves the Throughput collector (a.k.a Parallel collector) and the Low Pause collector (a.k.a. Concurrent collector). The Java HotSpot FAQ recommends either of these for very large heaps.

For applications running in 32-bit mode on a single CPU, the Serial collector appears the most beneficial based on the HotSpot FAQ (Serial collector will be used if no other collection JVM option is passed in). In 64-bit mode with large heaps, the Parallel or Concurrent collectors are recommended, but which one suits your application and environment? For this, of course, you should read **full article**, especially the sections on "When to Use the Concurrent Low Pause Collector" and " When to Use the Throughput (Parallel) Collector".

The collector types are very similar, and it requires multiple, slow reads of their descriptions before the differences begin to become clear.

I understand the Throughput / Parallel collector to perform better when you have many CPUs to take advantage of and when you don't have many long lived objects (i.e. cached queries or application scope / session scope objects). This is a Young generation collector that has multiple threads (default 1 per cpu) that perform parallel collections during pauses. This collector is enabled with the -XX:+UseParallelGC JVM argument, and it is the default collector when ColdFusion 8 is installed in Server or Multiserver Configuration.

The Low Pause / Concurrent collector is described to have shorter pauses, and to be a Tenured generation collector. The application pauses briefly when GC begins, and pauses again when GC is about halfway done where the second pause is longer because multiple GC threads run during this pause. After the second pause the application resumes, and there is another GC thread that runs parallel to application execution which completes the GC task. This collector is enabled with the -XX:+UseConcMarkSweepGC JVM option. This is not the default for ColdFusion.

Once you have evaluated which collector is best for your application and environment, you may be satisfied with the default Throughput collector for ColdFusion, or you may wish to change it to Low Pause Concurrent collector by editing the java.args in the jvm.config file.

Anything Else?
Some people have indicated on **Andrew Powell's blog** that they are running ColdFusion MX (6 or 7?) on 64-bit JVMs on 64-bit platforms. While much of the core ColdFusion server may work, it is not supported in that configuration primarily because there are binary libraries used to provide certain functionality which are themselves compiled as 32-bit, and cannot be loaded by the 64-bit JVM. Among the changes required to provide support for ColdFusion 8 on 64-bit JVMs on Solaris include compiling those binaries in both 32-bit and 64-bit and adding switches to load the correct version accordingly.

This brings me to C++ CFX custom tags. If you will be upgrading from earlier versions of ColdFusion on Solaris to ColdFusion 8 in 64-bit mode on Solaris, then you must recompile those CFX tags as 64-bit.

Finally
If you intend to run ColdFusion in 64-bit mode on Solaris, that you will probably run with at least 2 or 4 CPUs and will probably have at least 4 - 16 GB of memory to utilize. A ColdFusion application that takes maximum advantage of full 64-bit mode would be one that utilizes the application and session scope heavily, caches very large

record sets, and performs high volumes of CPU intensive tasks such as charting/graphing, document generation, and image manipulation. Based on characterization of collector types, I believe that the Low Pause Concurrent collector type would provide best performance for this type of ColdFusion application.

Even if you cannot run ColdFusion on a 64-bit JVM on Solaris, you should at least be able to take advantage of the higher 4 GB memory ceiling by running ColdFusion in 32-bit mode on a 64-bit platform, including at least Solaris, Linux and possibly including Mac or Windows, using the default Throughput collector, which is a configuration supported by Adobe.