Ultra Low Latency Trading Systems

A blog about techniques used to build www.SubMicroTrading.com

Saturday, 9 May 2015

Java JVM Tuning for Ultra Low Latency

There is no JVM arg that fits all applications, key is have a repeatable full test bed and run full scale benchmarks over hours not seconds. Rinse and repeat several times for EACH arg change. The args I focus on are the ones on SubMicroTrading which performs no GC and has almost no JIT post warmup.

Please note some of these flags are now on by option … sorry I havent checked, still worth bringing them to attention I think.

For standard java applications which do lots of GC with mainly short lived objects, I would recommend try the G1 collector … for market data I found it much better than concurrent mark sweep. I will blog about that another time … spent weeks tuning poorly designed apps (advice don't bother buy Zing).

Note each Java upgrade brings new options and tweaks existing performance, sometimes up, sometimes down, so re-benchmark each Java upgrade.

Treat micro benchmarks with care, discuss the Generics benchmark and explain how on PC was different to Linux

Avoid BiasedLocks … they incur regular millisecond latency in systems I have tested

JVM Args for Recording Jitter (JIT/GC)

-XX:+	PrintCompilation
-XX:+	CITime
-XX:+	UnlockDiagnosticVMOptions
-XX:+	PrintInlining
-XX:+	LogCompilation
-verbose:	gc
-XX:+	PrintGCTimeStamps
-XX:+	PrintGCDetails

Rather than regurgitate what I previously googled on understanding output from PrintCompilation :- http://blog.joda.org/2011/08/printcompilation-jvm-flag.html

For ultra low latency you want no GC and no JIT, so in SMT I preallocate pools and run warmup code then invoke System.gc(). I take note of the last compilation entry then while re-running controlled bench test look for new JIT output (generally recompilation). When this occurs I go back to the warmup code and find out why the warmup code had to be recompiled. This generally comes down to either the code not being warmed up, or the routine was too complicated for the compiler. Either add further warmup code or simplify the routine. Adding final everywhere really helps.

Writing warmup code is a pain, and I am gutted the java.lang.Compiler.disable() method is not implemented (or at least it wasn't in Open JDK1.6 … empty method doh!). Ideally I would invoke this when application is warm and have no recompilation due to the compiler thinking it can make further optimisations.

Java can recompile and recompile this only happens in my experience when method is too complex. Ofcause if a recompilation is due because java inlined a non final method and the optimisation was premature then the code needs to be corrected. What I want to avoid recompilation optimisations from edge cases that infrequently go into code branches.

Note you cannot guarantee no GC and no JIT under any situation in a complex system. What you can do is guarantee no JIT/GC for KEY specified scenarios that the business demand. If a trading system does 10 million trades a day, I would set a goal of no GC/JIT under NORMAL conditions with 30 million trades then check performance upto 100 million to see at which point jitter occurs. If for example the exchange disconnect you during the day, and that kicks in a few milliseconds of JIT its not important. You don't need pool every object … just the key ones that cause GC. More on that in future blog on SuperPools.

I remember speaking to Gil Tene from Azul, while working at Morgan Stanley and really tried to get across how much more JIT is of a pain than GC. Some exciting developments seem to have been made with Zing and I would have been very interested in benchtesting that … alas I just don't have time at present. Very impressed with Azul and Gil and how they respond to queries and enhance their product ….. so much better than Sun/Oracle were with Java.

http://www.azulsystems.com/products/zing/whatisit

SubMicroTrading JVM Arguments

The following are the arguments that SubMicroTrading run with, this includes the algo container, OMS, exchange sim and client sim.

-XX:+	BackgroundCompilation	Even with this on there is still latency in switching in newly compiled routines. I really wish that switch time was much much quicker !
-XX:	CompileThreshold=1000	If you don't want to benefit from fastest possible code given the runtime heuristics you can force initial compilation with -Xcomp … an option if you don't want to write warmup code. This may run 10% slower but sometimes much slower depending on the code.
-XX:+	TieredCompilation	So code is initially compiled with the C1 (GUI/client) compiler, then when its reached invocation limit is recompiled with the fully optimised C2 (server) compiler. The C1 compiler is much quicker to compile a routine than the C2 compiler and reduced some outlying latency in SMT for routines that were not compiled during warmup (eg for code paths not covered in warmup).
-XX:-	ClassUnloading	Disable class unloading, don't want any possible jitter from this. SMT doesn't use custom class loaders and tries to load all required classes during warmup.
-XX:+	UseCompilerSafepoints	I had hoped that disabling compiler safepoints would reduce JIT jitter but in SMT multithreaded system it brings instability so I ensure the safepoints are on ….. More jittter I don't want ho hum.
-XX:	CompileCommandFile= .hotspot_compiler"	The filename used to be picked up by default but now you have to use this command. This is really handy, if you have a small routine you cant simplify further which causes many recompilations then prevent it by adding a line to this file, example :- exclude sun/nio/ch/FileChannelImpl force This means the routine wont be compiled, you need to benchmark to determine if running routine as bytecode has noticeable impact.
-XX:+	UseCompressedOops	I kind of expected this to have a small performance overhead but in fact it has slightly improved performace … perhaps thru reduced object size and fitting more instances into cpu cache.
-X	noclassgc	Again all classes loaded during warmup and don't want any possible jitter from trying to free up / unload them
-XX:-	RelaxAccessControlCheck	To be honest no idea why I still have this in or even if its still required !
-D	java.net.preferIPv4Stack= true	If you upgraded java and spent hours working out why your socket code isnt working anymore, this could well be it … DOH !!!
-	server	Don't forget this if running benchmarks on PC
-XX:+	UseFastAccessorMethods	
-XX:+	UseFastJNIAccessors	
-XX:+	UseThreadPriorities	Not sure this is needed for SMT, I use JNI function to hwloc routines for thread core affinity
-XX:-	UseCodeCacheFlushing	
-XX:-	UseBiasedLocking	Disable biased locking, this causes horrendous jitter in ultra low latency systems with discrete threading models **Probably the single biggest cause of jitter from a jvm arg that I found.**
-XX:+	UseNUMA	Assuming you have a multi CPU system, this can have significant impact … google NUMA architecture

JVM Arguments to experiment with … didn't help SMT, but may help you

-XX:-	DoEscapeAnalysis	Try disable escape analysis and see what the impact is
-X	comp	Mentioned earlier, compile class when loaded as opposed to optimising based on runtime heuristics Avoids JIT jitter but code in general is slower than dynamically compiled code. Cant remember if it compiles all classes on startup or when each class is loaded, google failed to help me here !
-XX:+	UseCompressedStrings	Use byte arrays in Strings instead of char. SMT has its own ReusableString which uses byte arrays. Obviously a no go for systems that require multi byte char sets like Japanese Shift-JIS All IO is in bytes so avoid the constant translation between char and byte
-XX:-	UseCounterDecay	Experiment disabling / reenabling with recompilation decay timers. I believe the decay timers delay recompilation from happening within 30seconds. A real pain in warmup code. I run warmup code, pause 30seconds then rerun! Must be a better way. Wish decent documentation existed that wasnt hidden away !
-XX:	PerMethodRecompilationCutoff=1	Try setting maximum recompilation boundary … didn't help me much

I have tried many many other JVM args but none of those had any favourable impact on SMT performance.

Click here for my list of Ultra Low Latency Blogs and Future Topics

Posted by Unknown at 16:19

No comments:

Post a Comment

To leave a comment, click the button below to sign in with Google.

SIGN IN WITH GOOGLE

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)