MicroBenchmarks

So You Want to Write a Micro-Benchmark

Here are some rules, in priority order, to know about when you write micro-benchmarks for Hotspot.

Rule 0: Read a reputable paper on JVMs and micro-benchmarking. A good one is Brian Goetz, 2005. Do not expect too much from micro-benchmarks; they measure only a limited range of JVM performance characteristics.

Rule 1: Always include a warmup phase which runs your test kernel all the way through, enough to trigger all initializations and compilations before timing phase(s). (Fewer iterations is OK on the warmup phase. The rule of thumb is several tens of thousands of inner loop iterations.)

Rule 2: Always run with `-XX:+PrintCompilation`, `-verbose:gc`, etc., so you can verify that the compiler and other parts of the JVM are not doing unexpected work during your timing phase.

Rule 2.1: Print messages at the beginning and end of timing and warmup phases, so you can verify that there is no output from Rule 2 during the timing phase.

Rule 3: Be aware of the difference between -client and -server, and OSR and regular compilations. The `-XX:+PrintCompilation` flag reports OSR compilations with an at-sign to denote the non-initial entry point, for example: `Trouble$1::run @ 2 (41 bytes)`. Prefer server to client, and regular to OSR, if you are after best performance. Also be aware of the effects of `-XX:+TieredCompilation`, which mixes client and server modes together.

Rule 4: Be aware of initialization effects. Do not print for the first time during your timing phase, since printing loads and initializes classes. Do not load new classes outside of the warmup phase (or final reporting phase), unless you are testing class loading specifically (and in that case load only the test classes). Rule 2 is your first line of defense against such effects.

Rule 5: Be aware of deoptimization and recompilation effects. Do not take any code path for the first time in the timing phase, because the compiler may junk and recompile the code, based on an earlier optimistic assumption that the path was not going to be used at all. Rule 2 is your first line of defense against such effects.

Rule 6: Use appropriate tools to read the compiler's mind, and expect to be surprised by the code it produces. Inspect the code yourself before forming theories about what makes something faster or slower.

Rule 7: Reduce noise in your measurements. Run your benchmark on a quiet machine, and run it several times, discarding outliers. Use `-Xbatch` to serialize the compiler with the application, and consider setting `-XX:CICompilerCount=1` to prevent the compiler from running in parallel with itself.

Relevant links

- http://www.ibm.com/developerworks/java/library/j-jtp02225/index.html (Brian Goetz article)
- http://www.azulsystems.com/events/javaone_2002/microbenchmarks.pdf (Cliff Click JavaOne talk)
- http://code.google.com/p/caliper/wiki/JavaMicrobenchmarks (Google micro-benchmark tool)
- http://stackoverflow.com/questions/504103/how-do-i-write-a-correct-micro-benchmark-in-java (crowd-sourced goodness)