# **What I Learned This Month**: IBM Java 7 vs Java 6

Scott Chapman
American Electric Power

IBM has been talking up the enhancements that they've put into their Java 7 JVM on z/OS for a while now and I've been eager to see if I'd have similar results for some of my workloads. The workload examples that they show only look really impressive for what appears to be fairly significant multithreaded applications running in WebSphere Application Server (WAS). We have one significant WAS workload, a few smaller WAS applications, and various other Java batch workloads and a few Java started tasks.

Any hope that Java 7 would be some sort of magic bullet was dispelled when I ran a trivial test batch job (not much more complicated than "Hello World!") under both Java 6 and Java 7. The Java 7 "HelloWorld" seemed to take significantly more zAAP CPU time than the Java 6 one. Of course "significant" is relative: around half a second more. But a half second increase is pretty significant when the original execution was a quarter of a second!

After that original test, I didn't have time to investigate much further. I did, however, have an opportunity to mention my results to one of the IBM guys who knows about such things. He indicated he wasn't surprised by that result. The Java 7 JVM apparently works a little harder during initilialization, preparing to optimize things. He suggested, as the published benchmarks suggest, that I wouldn't see the real Java 7 benefits until I started running more significant workloads.

I finally had time to circle back to this and run a few more significant tests. The workloads I chose were ones that were repeatable and somewhat important to me. But neither was a WAS application and they will be different than your workloads, so take my findings with a grain of salt. While my conclusions may be at least partially true for some other workloads, it's also likely that others will behave entirely differently.

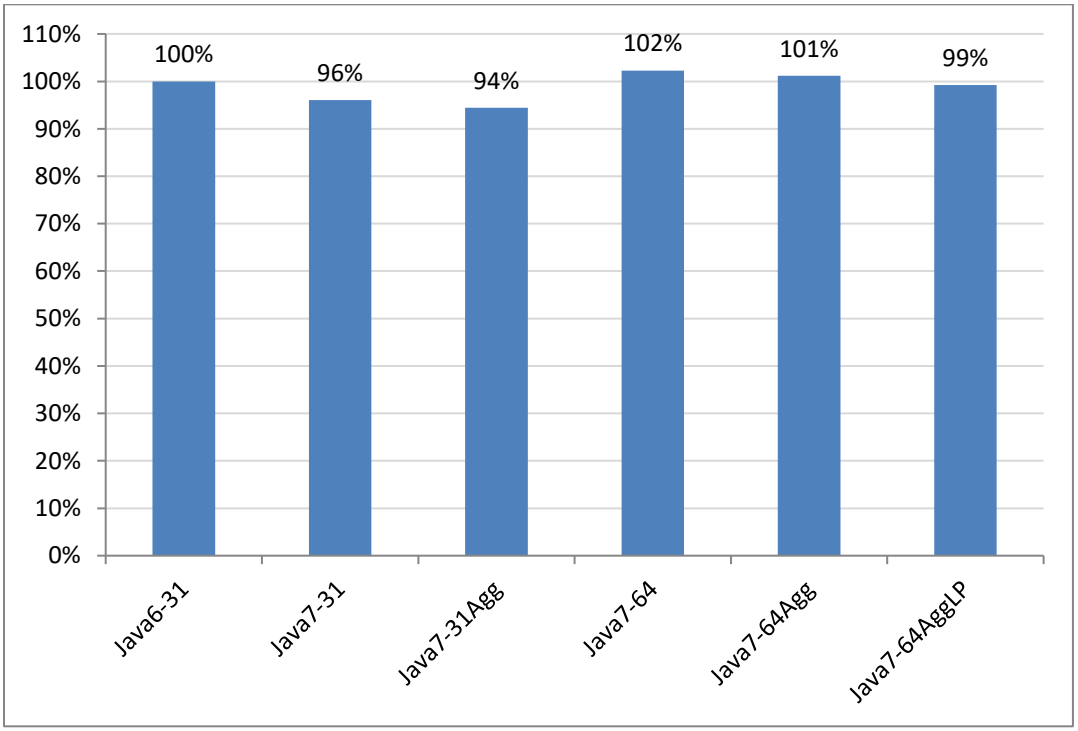
The first test was a single Java batch job. This job is really Data Processing 101: read some data, write some data. In this case, it reads some data out of a ZFS directory, transforms it, and writes output data to some files in another directory. Using a Java 6 JVM, this took 160 zAAP CPU seconds on one of our zEC12s. That was an average of three runs. Three runs under Java 7 averaged out to 172 seconds. That 7.5% increase is just enough over what I'm seeing for normal variation in zAAP CPU times to consider it a real increase, not normal variation. It's not significant to me, but it's interesting.

I also have a continuously running Java started task that periodically fetches data from the RMF distributed data server, runs some scripts, and records interesting performance data for me. If Java 7 needs more time to "warm up" to optimize its execution, this seems like a good test case.

This task is fairly consistent in what it does but there is some small variation based on the activity of the work that it's monitoring. Therefore, I ran it with different options for several days to average out any short-term variations and then compared the average utilizations for similar time periods. I've been running this task as a 31-bit Java 6 JVM, so that formed my baseline. I compared that to 31 and 64 bit Java 7. I then added the –Xaggressive option which enables additional optimizations.

While all of my 64-bit test cases used the "compressed references" option, I had intended to test support for large (1 MB) pages at the same time as the aggressive option. Unfortunately, while I enabled the –Xlp option, I didn't realize until later that the JVM was still utilizing 4KB pages. It turns out that I had under-defined LFAREA in SYS1.PARMLIB(IEASYSxx), so the system was not creating any large pages at IPL time. Once I corrected that problem, I discovered that we had never permitted non-authorized programs (such as my JVM for this started task) to use large pages. The lesson here is not to turn on –Xlp and just assume your JVM is using large pages. Instead, after turning on –Xlp, check the verbose garbage collection trace to see whether or not your JVM is actually using large pages. If not, large pages are probably not available for some reason and you'll need to check that you've done all the pre-requisite work to allow them to be used.

I ended up with 5 test cases to compare to the baseline, including a final 64-bit Java 7 execution with large pages correctly enabled. The results are shown below.




The variation is pretty small but I can tell you that the results were fairly consistent over time. So given time to run for an extended duration, it seems that

the Java 7 JVM is a bit more efficient than the Java 6 JVM. That's definitely good news. Even better is that with compressed references, the aggressive tuning options, and large page support, 64-bit Java 7 can potentially be even slightly more efficient than 31-bit Java 6. As expected, the 31-bit JVM is slightly more efficient than the 64-bit JVM, so if you are running a relatively small heap, sticking to 31-bit may still be your best option.

My guess is that a JVM with a larger and more active heap might see a more significant improvement from the use of large pages. I took a quick look at the SMF 113 data[1] for before and after running the JVM with large pages. There was a distinct drop off in the calculated percentage of CPU time taken for TLB[2] misses on the zAAP, from just under 8% to just under 6%. The zAAP activity on this test LPAR is dominated by the workload that I was testing, so it's gratifying that the average zAAP usage for the workload also showed a drop of about 2% as well. It sure is nice when all my numbers add up and tell a consistent story!

So it does seem that IBM has improved the performance of their Java 7 JVM vs. Java 6 for long-running JVMs. However, short-lived JVMs may experience a relatively minor degradation. There are also some tuning options (such as –Xquickstart) that may improve performance in those scenarios, but I didn't explore them.

Finally, when enabling large page support for your JVMs, be sure that you've done all the pre-requisite system work to enable large page support. Checking the verbose garbage collection trace to confirm large pages are in use is a good idea—even if you thought you did all the system work correctly.

As always, if you have questions or comments, you can reach me via email at ~~sachapman@aep.com~~ scott.chapman@epstrategies.com.

---

[1] The SMF 113 records record detailed processor measurements on z10 or above processors. There is no significant overhead to collecting these records in counter mode and the resulting data can be useful for validating changes like this as well as being an important input for capacity planning when moving between machine types.

[2] Translation Lookaside Buffer. Wikipedia has a succinct description: http://en.wikipedia.org/wiki/Translation_lookaside_buffer