Java Performance Mysteries

Pranita Maldikar, San-Hong Li and Kingsum Chow

Pranita.maldikar@intel.com, {sanhong.lsh, kingsum.kc}@alibaba-inc.com

Abstract. While assessing software performance quality in the cloud, we noticed some significant performance variation of several Java applications. At a first glance, they looked like mysteries. To isolate the variation due to cloud, system and software configurations, we designed a set of experiments and collected set of software performance data. We analyzed the data to identify the sources of Java performance variation. Our experience in measuring Java performance may help attendees in selecting the trade-offs in software configurations and load testing tool configurations to obtain the software quality measurements they need.
The contributions of this paper are (1) Observing Java performance mysteries in the cloud, (2) Identifying the sources of performance mysteries, and (3) Obtaining optimal and reproducible performance data.

1 Introduction

Java is both a programming language and a platform. Java as a programming language follows object-oriented programming model and it requires Java Virtual Machine (Java Interpreter) to run Java programs. Java as a platform is an environment on which we can deploy Java applications. Java Enterprise Edition (Java EE) [1]is a platform which is used to run enterprise class workloads and applications. This platform provides API and runtime environment for developing and running multi-tiered, scalable, reliable enterprise applications. An Enterprise application is a business application that is complex, scalable, distributed and multi-tiered.

Java applications can be single tieredor multi-tiered. Single tier Java applications are self-contained applications that can perform certain operations. SPECjvm2008 [3]is an example of single tier Java benchmark. SPECjvm2008 benchmark comprises of dozens of Java applications. Some of these Java applications work on exercising encryption software like crypto.aes, crypto.rsa, crypto.signverify workloads while some of them are scientific mathematical algorithms like FFT, sparse matrix computation etc. that works with both large and small data set.

Multi-tier Java workload or enterprise Java application is a software application distributed over multiple machines. Such applications can be separated into multiple tiers like, client tier, middle-tier and data tier. For testing a Java EE workload theclient tier would be a load generator machine using load generator software like Faban [2]. This client tier will consist of users that will send the request to the application server or the middle tier.The middle tier contains the entire business logic for the enterprise application. Business logic is nothing but the code that gives functionality to the application. The data-tier is the data source for the application. It consists ofdatabase or the legacy data sources like mainframes.

2 Java Performance Monitoring

There are a number of factors that could affect the performance of Java program. Among them, garbage collection (GC) and just-in-time (JIT) compilation probably stand out as they are not present in native program performance monitoring.

Garbage collection is implemented by the Java Virtual Machine (JVM) for automatic memory management. Java HotSpotTM VM [8] (originally implemented by Sun, acquired by Oracle) supports several different garbage collection algorithms which were designed for serving different pause time and throughput requirements.For the multi-tier Java workloads running HotSpotTM in cloud, typically, we

consider choosing one of these three collectors:throughput collector, CMS collector and G1 collector, the best choice dependson workload characteristics.

The parallel old collector (enabled by -XX:+UseParallelOldGC) is designed for maximizing the total throughput of an application, but some individual business transaction might be subjected to a long pause.

The CMS collector (enabled by -XX:+UseConcMarkSweepGC) can concurrently collect the objects in old generation while application threads are running. The CMS algorithm is designed to eliminate long pauses.It has two quick stop-the-world (STW) pauses per old generation collection. Compared with throughput collector, the CMS may increase the CPU usage due to its background processing.

The G1 collector (enabled by -XX:+UseG1GC) is designed for processing large heap with minimal pauses. The G1 divides the entire heap into equally-sized region and 'smartly' collects the regions with the least live data first (garbage first) [9]. Similar with CMS, the collection for old generation in G1 is also done in concurrent way.

We have the different GC collector choices as there is no one right algorithm for every workload, different GC collector optimizes for different user scenarios. In general, the parallel old collector is optimized for throughput. The CMS collector is optimized for pause time. The G1 collector is optimized for reducing pause time on large heaps more than 4GB.

Selecting the right GC algorithm can significantly help the application performance.As a rule of thumb: if the GC overhead is > 10%, the wrong GC algorithmmight be probably used.

The first important tuning for GC is the size of the application's heap. Smaller heap size means more frequent GC and not enough time for performing business logic. Larger heap size means less frequent GC, but the duration of each of GC pauses will be increased, as the time spent in GC pauses is generally proportional to the size of space being collected. Choosing proper heap size also depends on workloads. Heap occupancy is one important measure for heap size setting, as a rule of thumb: size the heap so that it is 30% occupied after a full GC[10].

The Just-In-Time compiler component in JVM is used to turn bytecode into much faster native code for accelerating Java programming language performance. There are a couple of typical tuning practices for JIT which can help the improvements in the performance of application.

The size of code cache is the first basic thing which we should pay more attention to. Generally, just make sure that the code cache is large enough otherwise the application might mysteriously slow down if it is running out of code cache space.

Inlining is one of important optimizations made by JIT to eliminate the overhead of method call. Thus small and frequently called methodsmight be good candidates for inlining.

To identify and resolve Java performance issues, the following methods might be considered. (1) Monitor the logs, e.g. GC or JIT log, (2) Use the diagnostics JVM parameters which will provide more useful information for your debug, and (3) Use profiling tools, likes VisualVM, JMC(shipped with OracleJDK as commercial feature set).

Logs are usually thekey to find possible performance bottleneck; there are some open source tools to visualize these logs, e.g. gcviewer[5] is a good tool for gc log analytics and jitwatch[6] provides visual insights into JIT compilation.

Diagnostics JVM parameters [7] is another good way to explore the performance related issues, e.g. enable inlining diagnostics output by adding -XX:+PrintInlining parameter and then check which method is failed to inline.

There are a lot of profiling tools for measuring application performance, technically speaking, the profiling can be made by either sampling or runtime instrumentation. Both instrumentation-based and sampling-based tools have different characteristics with their own advantages and disadvantages.Profiling does not come for free, we should pay close attention to the overhead introduced by profiling tool otherwise the results collected by them couldbe misleading for final conclusion.

3 Java Performance Mysteries

In a datacenter of a cloud service provider, before upgrading the hardware systems to a newer version it is very crucial to evaluate the goodness of the new hardware compared to the older ones. For evaluation we use multiple workloads/benchmarks to assess the performance [4]benefit it offers over the previous generation. In this paper we are evaluating two platforms called Platform A and Platform B using an Enterprise Javaworkload. We are using a two tier Java EE Web Application workload. The workload consists of a client tier which generates load using faban driver. Based on the injection rate the faban driver will start client users that send http request to the application server. The application server will process the request and send a response back to the client with appropriate web page. The web page is computed dynamically after processing the request from the client. When a user gets a response back from the application server it will immediately send another request. Advantage of this is, we can saturate the application server with very small number of clients to evaluate its maximum performance capability. This

workload is useful in the test environment where we want to get quick performance numbers with high CPU saturation. We are running the workload on two platforms called Platform A and Platform B with fixed number of users. For this workload we are not sure of the optimal heap size and GC algorithm. So we are evaluating both the platforms with small and large heap sizes and two different GC algorithms called GC-1 and GC-2. The table below summarizes our findings on both the Platforms.

Table 1. Performance of Both Platforms at Higher Number of Users

config	web101.5W Platform A Small Heap GC-1	web101.5O Platform B Small Heap GC-1	web101.6F Platform A Large Heap GC-1	web101.6G Platform B Large Heap GC-1	web101.6B Platform A Small Heap GC-2	web101.6C Platform B Small Heap GC-2	web101.6I Platform A Large Heap GC-2	web101.6L Platform B Large Heap GC-2
Normalized Throughput	8.01	9.94	9.01	10.92	11.07	11.60	11.75	12.20
Performance Improvement		1.24		1.21		1.05		1.04
Network Utilization (Mbps)	6,344	7,839	7,108	8,609	8,730	9,089	9,263	9,611

3.1. Impact of GC-1 Garbage Collection Algorithm on Platform A and b

Based on Run ID web101.5W vs web101.6F and web101.5Ovs web101.6G we can infer that larger heap gives better throughput. Large heap results in less frequent GC. The JVM spends more time running the application and less time collecting garbage.For GC-1 garbage collection algorithm we can also say that Platform B produced 24% higher throughput than Platform A with small heap size. This performance delta is reduced to 21% with large heap.

3.2. Impact of GC-2 Garbage Collection Algorithm on Platform A and B

With runs web101.6B vs web101.6I and web101.6C vs web101.6L we see similar performance improvement with heap size. But if we look at platform comparison then we see only 5% performance improvement with Platform B over Platform A with small heap size and 4% performance improvement with large heap. After close inspection we realized that the performance impact with this GC algorithm is misleading because we are very close to saturate the network capacity. Hence these performance numbers should not be used to compare the two platforms.

3.3. Impact of Different Garbage Collection Algorithms on Platform A

For Platform A with small heap (web101.5W vs web101.6B) GC-2 algorithm provides 38% performance benefit over GC-1 algorithm and with large heap (web101.6F vs web101.6I)the performance delta reduced to 30%.

3.4. Impact of Different Garbage Collection Algorithms on Platform B

For Platform B with small heap GC-2 algorithm provides only 16% benefit over GC-1 algorithm and with larger heap this delta is decreased to 11%.

To characterize the behaviour of this Web Application workload the system level performance data is insufficient. We need information about the JVM behaviour, and the methods compiled on both the Platforms. We will have better understanding of the workload performance after looking at the JIT code and see how many resources were spent in the hot methods and GC methods for both the garbage collection algorithm on both the Platforms. For that we need to profile the application and collect JVM data.

3.5. Java Profiling

Data collected for performance evaluation does not come for free. It always comes at a cost of performance. Java profiling tools usually run in background, but every time a method is called or compiled the information gets logged in the tool. This has additional overhead. You should be very cautious when using such tools for performance testing.

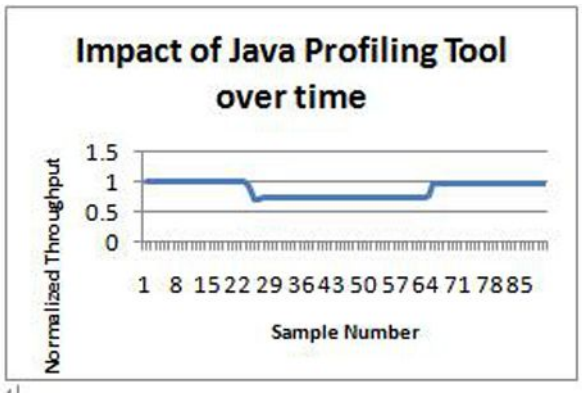

Figure 1. Performance Impact of a Java Profiling Tool

The overhead associated with the tools could be quite high. With one of the Java Profiling tool we saw 20% performance overhead. The throughput dropped down by 20% when the tool was collecting JVM information. Whenever such tools are used they might alter JVM behaviour and interfere with the workload performance.

Hence for performance testing one should always be careful when looking at the summary statistics for the workload. We should always look into the detail workload performance data. The overall summary data could be misleading. Figure 1demonstrates the impact of one such profiling tool used. We saw 20% performance dip when the JVM profiling tool was running. The workload summary reported the average throughput overrun time. It did not notify of thedrop in performance. Only after looking at the detail data we could see the performance impact of the tool.

Table 2. Performance of Both Platforms at Lower Number of Users

Config	web101.7R Platform A Small Heap GC-1	web101.7C Platform B Small Heap GC-1	web101.7Q Platform A Large Heap GC-1	web101.7B Platform B Large Heap GC-1	web101.7V Platform A Small Heap GC-2	web101.7W Platform B Small Heap GC-2	web101.7U Platform A Large Heap GC-2	web101.7X Platform B Large Heap GC-2
Normalizd Throughput	87.324	85.735	95.28	98.585	92.694	106.352	98.374	117.598
Performance Improvement		0.98		1.03		1.15		1.20
Network Utilization (Mbps)	6,896	6,764	7,522	7,766	7,326	8,391	7,769	9,275

With a low overhead tool we collected detailed data about the JIT code and we observed something really weird. There wasno change in performance when we moved to large heap on the same platform with same configuration.

When we looked into the Java JIT code we realized that between these two configurations the compiled code was different. In one of the runthe print methods got inlined. Whereas in the other run we saw lot of function calls to the print method.Just by method inlining we can observe significant performance difference.JVM is designed to make run time compilation and optimization decisions. Most of the times it will do the right thing, but then occasionally we do see such anomalies that may cause us to make wrong inferences.

With the previous number of users we were close tonetwork saturation.Hence for thenext set of experiments we reduced the number of users.

Table 2summarizes the performance of both the platforms with lower number of users. The strange thing is now when you look at the performance delta between the two platforms the picture that we saw with Table 1completely reverses. With GC-1 algorithm we see Platform A being better than Platform B by 2% for smaller heap and platform B being better than platform A by 3% for larger heap. On the other hand GC-2 algorithm shows 15% performance benefit on Platform B with small heap and 20% performance benefit with large heap. This configuration with lower users tells a completely different story.

4 Summary and Discussion

We showed a couple of examples of Java performance mysteries and identified the sources of those performance mysteries. Understanding Java performance mysteries would enable us to obtaining optimal and reproducible performance data that would in turn help us to make the right decisions in the data center.

References

1. http://docs.oracle.com/javaee/6/firstcup/doc/gcrky.html
2. https://java.net/projects/faban/
3. https://www.spec.org/jvm2008/
4. Khun Ban, Robert Scott, Huijun Yan and Kingsum Chow, “Delivering Quality in Software Performance and Scalability Testing”, Pacific Northwest Software Quality Conference, October 2011, Portland, OR.

URL http://www.uploads.pnsqc.org/2011/papers/T-40_Ban_etal_paper.pdf

5. http://www.tagtraum.com/gcviewer.html
6. https://github.com/AdoptOpenJDK/jitwatch
7. https://docs.oracle.com/javase/7/docs/webnotes/tsg/TSG-VM/html/clopts.html
8. http://www.oracle.com/technetwork/java/whitepaper-135217.html
9. http://www.oracle.com/technetwork/articles/java/g1gc-1984535.html
10. Scott Oaks, "Java Performance: The Definitive Guide", Oreilly