An Empirical Analysis of Java Performance Quality

Simon Chow

simonichow15@gmail.com

Abstract

Computer scientists have consistently searched for ways to optimize and improve Java performance utilizing a variety of methods. Methods including the just-in-time compilation, the garbage collection system, and adaptive optimizing have increased Java performance by reclaiming unneeded memory and optimizing areas of frequently executed code. Although many people have repeatedly improved the performance of their Java programs by modifying the code to make it more concise and efficient, Java performance can be upgraded without touching the original code at all. Adding additional RAM and computer cores can increase Java performance, along with reconfiguring VM and storage allocations, upgrading solid-state disks and using distinct disks for various types of files. This paper will cover how we can make Java performance even more efficient without changing the code of the Java program. After running these programs, we will experiment to find the most efficient methods and changes that can be made to the computer to maximize the Java performance. Without the need to change a single bit of code, we can optimize the Java performance in computers without going through every line of the Java program and figuring out which sections of the program should be modified to increase its efficiency. Thus, the user has the benefit of not having to understand how the program works to optimize the Java performance.

Biography

Simon Chow is a junior at Lincoln High School. He has participated in FIRST Robotics Challenges since 2007. Since 2014, he has participated in the high school branch of FIRST Robotics: FIRST Tech Challenge (FTC). In FTC, students program their Android phones using the Java language to navigate their self-designed robots around the FTC field, completing missions and attempting to outscore their opponents. Besides participating in FTC, Simon has also learned the fundamentals and main data structures of Java programming in Lincoln High School. Simon has also been part of a previous FTC team that has published the PNSQC paper "Brewing Quality in Android Robots" and presented it at the PNSQC conference in 2015.

Copies may not be made or distributed for commercial use

1 Introduction

Many people are familiar with improving Java programs or applications by changing the Java code directly. However, we are writing this paper to show how the Java performance can be improved by tuning Java benchmark parameters rather than modifying the code.

Related Work

So far, a lot of work has been done to creating SPECjvm benchmarks. Experiments haev also been conducted comparing the newer SPECjvm2008 version with other, older SPECjvm versions and observing the improvements over time. Ultimately, this paper is unique as it focuses on tuning Java parameters such as memory heap size and garbage collection to see its effect on improving Java performance.

2 SPECjvm2008 and Benchmarks

SPECjvm2008 is a benchmark suite that measures the performance of a computer system through its Java Runtime Environment (JRE). SPECjvm utilizes ten major benchmarks: Compiler, Compress, Crypto, Derby, MPEGaudio, Scimark, Serial, Startup, Sunflow and XML.

The compiler benchmark compiles a set of java files and can deal with memory using its own FileManager, rather than relying on a physical disk.

The compress benchmark compresses data using Lemep-Ziv-Welch (LZW) method, substituting common substrings with variable size code.

Crypto has 3 different subsections: Aes, rsa, and signverify. The Crypto benchmark encrypts and decrypts using their respective protocols.

The Derby benchmark focuses on BigDecimal computations and database logic.

The MPEGaudio benchmark is floating point heavy and tests mp3 decoding.

Another floating point benchmark is the Scimark benchmark. There are large dataset and small dataset versions of this test, with the large data set being connected to memory systems and the small dataset connected to the JVM.

The Serial benchmark transforms data structures into storable formats or vice versa. The Serial benchmark deals with primitives and objects.

The Startup benchmark start each benchmark's first operation.

The Sunflow benchmark tests graphics visualization using an open source rendering system.

The XML benchmark has 2 sub-benchmarks: transform and validation. Transform applies style sheets to XML documents. Validation processes style sheets.

When running a benchmark, there will be a warmup phase (120 seconds), followed by an iteration phase (240 seconds). The warmup period will allow the computer to perform some initial, brief testing of the benchmark, but the results from the tests do not go into the actual benchmark results. However, the iteration phase does produce the benchmark performance results[1].

Running SPECjvm2008

We used Windows Command Prompt to run the SPECjvm benchmarks that we downloaded. In order to run SPECjvm, we had to point the JAVA_HOME variable to where the Java Virtual Machine (JVM) is being held in our computer.

Tuning the Parameters

There are many parameters. Two of them are -Xmx (the maximum heap memory size) and -Xms (the minimum heap memory size). When experimenting with the parameters, we decided to test the maximum memory allocation with 1GB, 2GB and then finally with 3GB (the parameters would then be -Xmx1g, -Xmx2g and -XMx3g respectively). To test the effects of changing these parameters on the specJVM runs, we changed the maximum memory heap size (-Xmx#g) in our run-SPECJVM file. From here, we can observe how changing the allocation of memory can affect how many operations the computer can perform. For the experiments in this paper, we will be testing with the Serial, Derby and Sunflow benchmarks.

Java Garbage Collection

Garbage collection is a process in programming languages in which the computer identifies all the objects that are being used in the program as well as the ones that are not. After identifying the dead objects, the computer will locate the dead objects and remove them, and rearrange the “living” objects in order to free up space and allow the program to run more efficiently[2].

We will be tuning garbage collection parameters in these SPECjvm experiments as well.

In addition to tuning the memory heap size parameters, we also tuned several parameters related to garbage collecting. We specifically experimented with G1, MarkSweep and Parallel garbage collection systems[3]. Here are the additional parameters for each of the garbage collection types.

- set JAVA_OPTS="-XX:+UseG1GC"
- set JAVA_OPTS=”-XX:+UseConcMarkSweepGC”
- set JAVA_OPTS="-XX:+UseParallelGC"

3 Experiments

For our experiments, we chose to perform SPECjvm runs of the Serial, Derby and Sunflow benchmarks.

Copies may not be made or distributed for commercial use

Serial

	Performance (op/min)		
Java Parameter	Trial 1	Trial 2	Trial 3
None	86.51	87.37	94.46
"-Xmx1g"	85.82	98.74	79.45
"-Xmx2g	83.7	97.67	75.86
"-Xmx3g	86.06	96.94	95.67

Derby

	Performance (op/min)	
Java Parameter	Trial 1	Trial 2
None	267.99	257.27
"-Xmx1g"	230.48	265.54
"-Xmx2g	225.02	241.62
"-Xmx3g	210.65	260.13

Sunflow

	Performance (op/min)
Java Parameter	Trial 1
None	45.17
"-Xmx1g"	44.73
"-Xmx2g	46.28
"-Xmx3g	45.82

Garbage Collection Data

Serial	Performance (op/min)		
Java Parameter	Trial 1	Trial 2	Trial 3
None	94.31	96.20	91.49
G1	94.18	87.69	96.59
MarkSweep	90.00	96.28	94.73
Parallel	96.26	98.52	94.66

Derby	Performance (op/min)		
Java Parameter	Trial 1	Trial 2	Trial 3
None	282.95	299.14	284.41
G1	303.54	298.83	312.73
MarkSweep	285.29	259.12	275.20
Parallel	253.10	284.79	275.78

Sunflow	Performance (op/min)		
Java Parameter	Trial 1	Trial 2	Trial 3
None	46.76	47.73	46.93
G1	47.19	49.93	51.09
MarkSweep	50.05	46.99	43.16
Parallel	44.31	41.80	43.33

4 Analysis

Performance Monitor

In order to see why our SPECjvm runs all came out roughly the same despite the change in the maximum memory heap size, we the performance monitor application to detect any changes in our computer's processing during the SPECjvm Serial runs, especially focusing on the CPU performance[4].

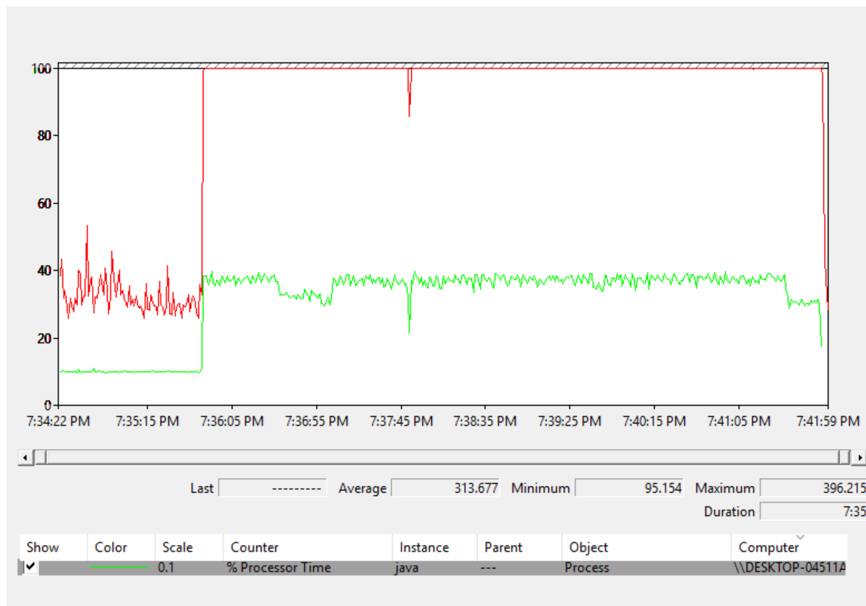


Figure 1: Graph of the data collected by Performance Monitor. The red line represents the total % processing time while the green line represents the Java % processing time.

While running Performance Monitor, we tested for both the total processing time percentage (red) and just the java processing (green). We found the green to peak at around 390 (as Figure 1 depicts the Java Percentage multiplied by 0.1), while red maxes out at 100. We believe the green goers above 100% as the computer has 4 logical processors, thus actually having a max at 400%, whereas the total processing (red) takes all 4 logical processors into account beforehand and thus caps at 100%.

Performing Data Analysis

We saved the data from Performance Monitor as an Excel .csv file. By saving the data as an Excel file, we were able to read, process and analyze the data using RStudio[4].

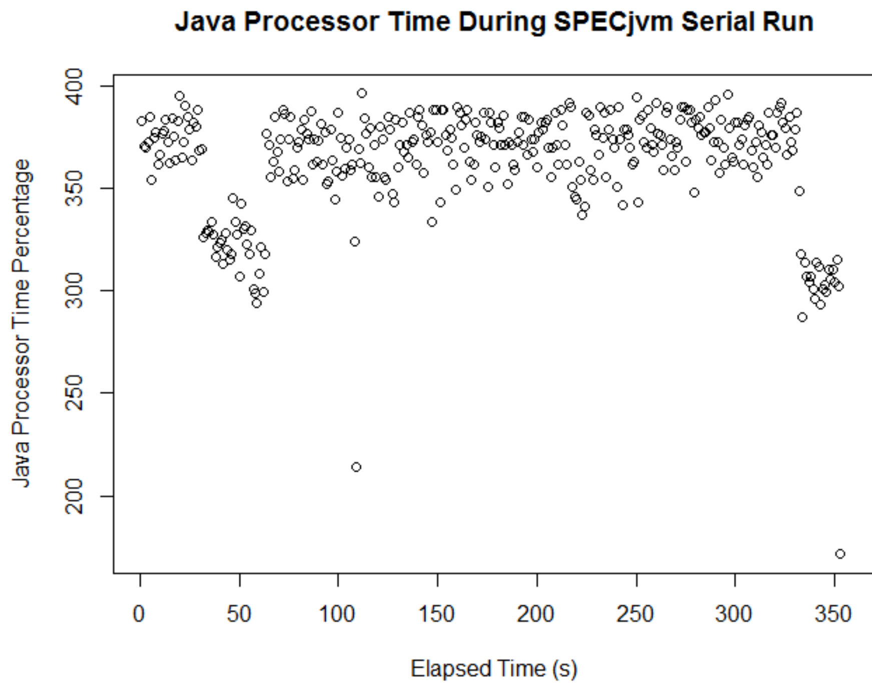
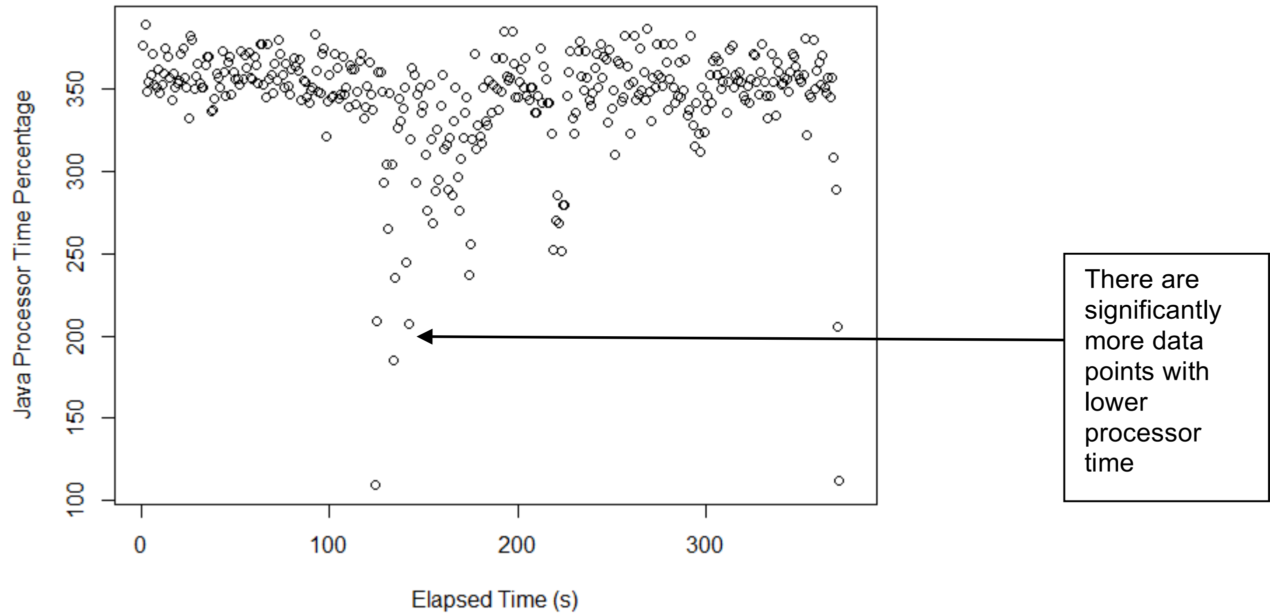


Figure 2: Java performance graph during a SPECjvm run of the Serial benchmark

When the Serial benchmark begins running, the Java % processing time increases up to nearly 400%. There is a lot of fluctuation between 350% and 400%. This fluctuation is normal as it means that each logical processor is on average using between 90% and 100% of their total processing power, due to the various applications running on the computer. Slightly after 100 seconds into the run, there is a significant drop as the Serial benchmark switches from its warm-up run to its iteration run. The significant drop at the end after 350 seconds signifies the end of the Serial benchmark run.



\

Figure 3: Java performance graph during a Serial benchmark run with modified memory heap size

This graph illustrates a Serial benchmark run, but with a modification to the memory heap size. By setting Java memory heap size parameter to -Xmx1g, the Java performance of the computer can be affected. When comparing this graph to the unmodified JAVA_OPTS graph, we notice that the data points on this graph are more scattered and on average lower, than the original graph.

Derby Graphs

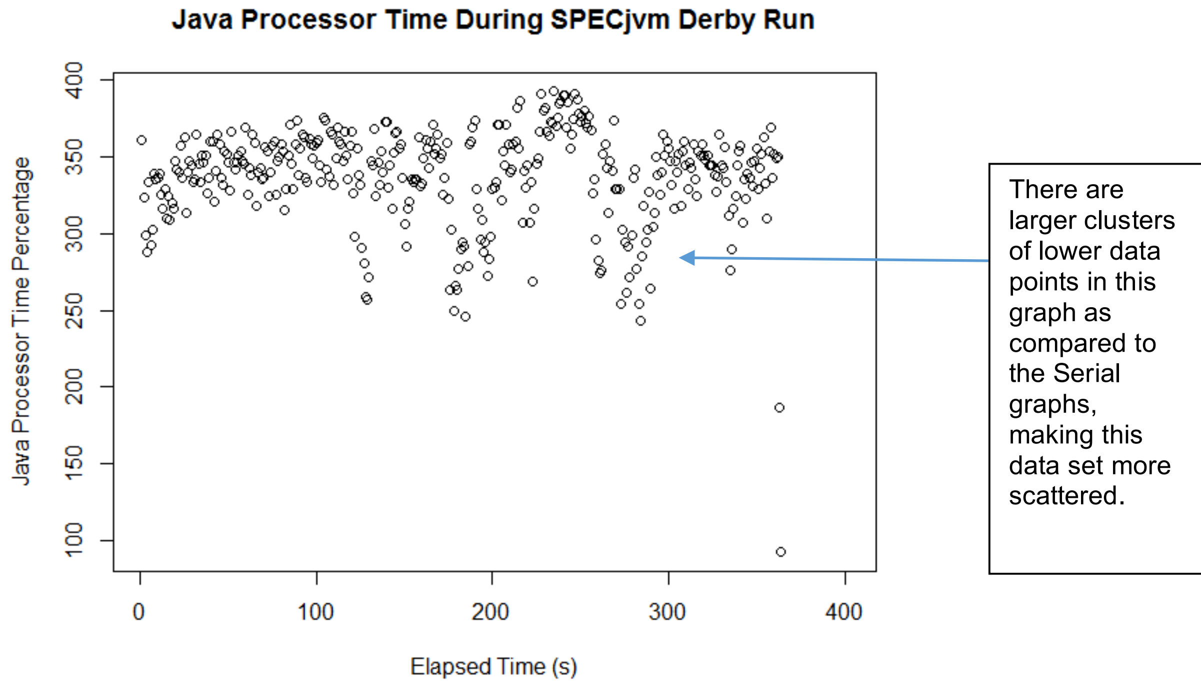


Figure 4: Java performance graph with Derby benchmark

The graph of the Derby benchmark appears to be more scattered than the respective Serial benchmark graph, fluctuating from 250% to 400%, instead of 350% to 400%. Consequently, most of the data points are near the 350% or 340% line whereas the original Serial benchmark graph clustered at the 375% line.

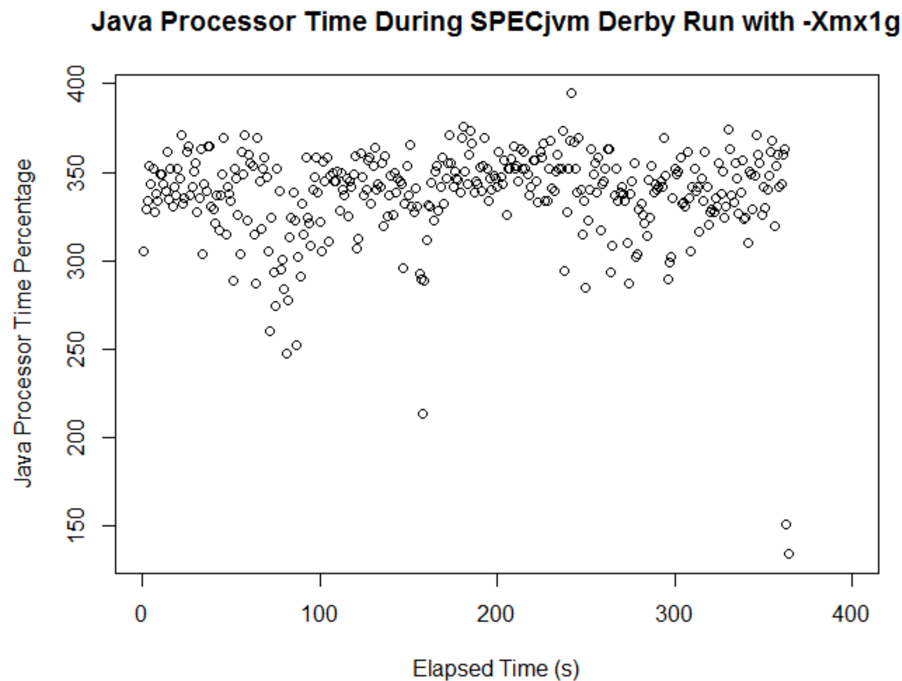


Figure 5: Java performance graph with Derby benchmark and modified memory heap size

After modifying the memory heap size, the data and graph appear roughly the same. It is possible that fluctuation caused this graph to look like the original Derby graph even though it should be different. Thus, the memory heap size does not seem to affect the performance, at least not significantly. A possible explanation of this is that the computer already has more than enough memory to run the SPECjvm experiments and therefore, additional memory will have little to no effect on the computer's performance or processing.

5 Conclusion

Modifying the memory heap size parameter seems to have little to no effect on the Java performance of our computer during the experiments. We believe the reasoning for this to be because the memory heap size is already more than sufficient at 1GB and thus adding in any more gigabytes of memory should have theoretically no effect on the performance. Thus, a sensible next step for these experiments is to try running the java benchmarks with lower maximum heap memory sizes, such as 500MB.

For the garbage collection options, it appears that the G1 garbage collection is faster in performance than the parallel and MarkSweep performance while it is unclear whether the parallel, or the MarkSweep garbage collection options is the slowest of the three.

Experimental Errors and Other Factors

The SPECjvm experiments were done on different days and thus different application on the computer were opened and running, varying the performance results. Another unexpected variable in our experiments was that we tested our SPECjvm runs on two different computers as our first computer broke partway through the tests. Thus, the slight differences in memory or processing across these computers

may have played a role in our SPECjvm results. To remedy these errors, only the applications necessary for the experiments should have been open and the same computer should have been used the whole time.

The contributions of this paper are:

- 1) Understand the trade-offs between Java configurable parameters and the performance of Java workloads
- (2) Design experiments for different Java programs, and also non-Java programs to understand performance trade-offs
- (3) Conduct performance analysis without modifying the code of the Java programs

Acknowledgments

The author wishes to express their gratitude to his reviewers Suresh Chandra Bose and Moss Drake, whose assistance and feedback have been invaluable.

References

[1] "Standard Performance Evaluation Corporation." *SPEC - Standard Performance Evaluation Corporation*, Standard Performance Evaluation Corporation, spec.org/.

[2] Morgan, Johnny. "An Overview of Garbage Collection in Java - DZone Java." *Dzone.com*, DZone, 27 June 2017, dzone.com/articles/an-overview-of-garbage-collection-in-java.

[3] Lee, Sangmin. "Understanding Java Garbage Collection | CUBRID Blog." *Open Source Database*, CUBRID, 31 May 2017, www.cubrid.org/blog/understanding-java-garbage-collection/.

[4] "Windows 7 Performance Monitoring Tools." *SearchITChannel*, TechTarget, searchitchannel.techtarget.com/feature/Windows-7-performance-monitoring-tools.

[5] *RStudio Team (2015). RStudio: Integrated Development for R. RStudio, Inc., Boston, MA URL http://www.rstudio.com/.*