



Bachelor's Thesis

Android Memory Management in Garbage Collection Using Code Optimization Techniques and Implementing GraalVM

Submitted by

Md. Mahedi Hasan

ID: 171-35-2052

Department of Software Engineering

Bachelor of Science in

Software Engineering

Supervisor

Md. Khaled Sohel

Assistant Professor

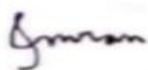
Department of Software Engineering
Daffodil International University

May 31, 2021

Approval

This thesis is being titled as “Android Memory Management in Garbage Collection Using Code Optimization Techniques and Implementing GraalVM” submitted by Md. Mahedi Hasan, 171-35-2052 to the Department of Software Engineering, Daffodil International University has been accepted as satisfactory for the partial fulfillment of the requirements for the degree of Bachelor of Science in Software Engineering and for approval as per its inner styles and contents inside.

BOARD OF EXAMINERS



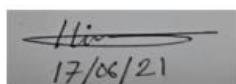
Dr. Imran Mahmud
Associate Professor and Head
Department of Software Engineering
Daffodil International University

Chairman



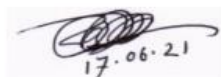
Md. Shohel Arman
Senior Lecturer
Department of Software Engineering
Daffodil International University

Internal Examiner 1



Farhan Anan Himu
Lecturer
Department of Software Engineering
Daffodil International University

Internal Examiner 2

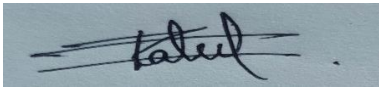


Professor Dr. Mohammad Abul Kashem
Department of Computer Science and Engineering
Dhaka University of Engineering and Technology

External Examiner

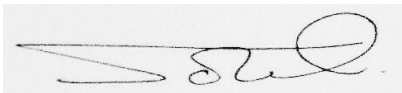
DECLARATION

It has been declared that this thesis, including all the scientific experimental works, has been completed by me under the supervision of Md. Khaled Sohel, Assistant Professor, Department of Software Engineering of Daffodil International University. I also declare that neither this thesis nor any part of this whole scientific experiment has been submitted elsewhere for the award of any degree.



Md. Mahedi Hasan
Student ID: 171-35-2052
Batch: 22
Department of Software Engineering, Faculty
of Science and Information Technology,
Daffodil International University.

Certified By



Md. Khaled Sohel
Assistant Professor,
Department of Software Engineering, Faculty
of Science and Information Technology,
Daffodil International University.

Acknowledgement

Thanks to Allah Almighty, Who gave me courage and patience to carry out this work.

I want to thank my supervisor, Mr. Md. Khaled Sohel sincerely, for his guidance, understanding, and warm spirit to finish this thesis. I always feel that I would not end the thesis without his wise instruction in this limited time.

I would also wish to express my gratitude to Dr. Imran Mahmud, Head of the Software Engineering Department, for inspiring us in all means. I am also thankful to all the lecturers, Department of Software Engineering for their unconditional support and encouragement.

My deepest gratitude goes to my beloved parents, Khaja Akram Ullah and Nasrin Akter, for their unceasing encouragement and prayers. Special and profound thanks to my brother Md. Kamrul Hasan for his unquestioning support year after year.

I offer my special thanks to all my friends for their kindness and moral support.

Table of Contents

1	INTRUDUCTION	9
1.1	Background	9
1.2	Motivation of the Research	9
1.3	Problem Statement	10
1.4	Research Question	10
1.5	Research Objectives	10
1.6	Research Scope	10
1.7	Thesis Organization	10
2	LITERATURE REVIEW	11
3	TOOLS & METHODS.....	16
3.1	GraalVM	16
3.1.1	A Different approach of using GraalVM instead of JVM.....	16
3.1.2	Memory Management at Image Run Time	17
3.1.3	Advantages of GraalVM:	18
3.2	Java Virtual Machine	21
3.3	Eclipse.....	21
3.4	IntelliJ	22
3.5	Java VisualVM	22
3.6	Memory Analyzer Tool	22
3.7	Linux	22
4	IMPLIMENTATIONS	23
4.1	Using GraalVM instead of JVM for better memory Management:.....	23
4.2	Controlling Garbage Collection	24
4.2.1	Control initial Heap Size using Flag	24
4.2.2	Analyzing Heap Dump.....	25
4.2.3	Memory Analyzer (MAT).....	26
4.3	Tuning the Garbage Collection.....	28

4.3.1	Parameter for New Ratio.....	30
4.3.2	Parameter of Survivor Ratio.....	30
4.3.3	Parameter of MaxTenuringThreshold	30
4.4	G1 Garbage Collector	31
4.5	Efficient Mark & Sweep Process	32
4.6	Using Profiler on Application	33
4.6.1	Overall Monitoring.....	33
4.6.2	CPU Profile	34
4.6.3	CPU Sample.....	34
4.6.4	Thread CPU Time	35
4.6.5	Memory Heap Histogram.....	35
4.6.6	Memory Per Thread Allocation	36
4.7	Java Flight Recorder	37
4.7.1	Flame Graph.....	37
4.7.2	Call Tree.....	38
4.7.3	MethodList.....	39
4.7.4	CPU Timeline	40
4.7.5	Java Flight Recorder Events.....	41
4.8	Better use of List in Java for better performance improvement	42
4.8.1	ArrayList	42
4.8.2	Using Vector instead of ArrayList	43
4.8.3	Using LinkedList.....	43
4.9	Better Coding Choice.....	45
4.9.1	Primitives Vs Object	45
4.9.2	Double Vs Big Decimal	46
4.9.3	String Concatenation vs String Builder.....	47
4.9.4	Loops vs Stream vs Parallel Stream.....	48
5	5 RESULTS & DISSCUSSION	49
5.1	Performance Improvement of GraalVM vs JVM	49

5.2	Controlling Heap Size (Section 4.1.1)	50
5.3	Better List Choice (Section 3.7)	50
5.4	Tuning Garbage Collection (Section 3.2):.....	51
5.5	Profiler Results (Section 3.5.1)	52
5.6	Better Coding Choice (Section 3.8)	54
6	CONCLUSIONS AND RECOMMENDATIONS.....	56
6.1	Findings and Conclusion.....	56
6.2	Limitations.....	56
6.3	Future Works.....	57
6.3.1	Automatic Reference Object	57
6.3.2	GraalVM implementation	57
6.4	Conclusion.....	57
7	Reference	58
8	Appendix 1	61
	Account Clearance.....	61
9	Appendix 2	62
	PLAGARISM CHECK.....	62

Abstract

Android OS is the most widely used smartphone OS in the world. Around 73% of the world smartphone users use Android as their smartphone. The percentage is about 96% in our country. But unfortunately, it has always lacked iOS due to poor memory management. Android is built on Java Virtual Machine. Android uses Garbage Collector as its memory management which is a memory management technique in JVM. Many memory management techniques have been proposed, such as Controlling Memory leaks, Managing GPU Buffers, Adaptive Background App, Micro-Optimization, Detecting and Fixing Memory Duplications, Partitioning Memory, Non-Blocking Garbage Collector Dynamic Caching, etc. All these techniques revolve around Android's current memory structure, which is Garbage Collector. But for a long time, other memory management techniques that are not based on Garbage Collector are not proposed. So, in this research, we will have found some way to improve the existing system of Garbage Collector, a new Virtual machine is proposed called GraalVM instead of Java Hotspot Virtual Machine. We have used those techniques we can get optimal memory management for our Android platform.

1 INTRODUCTION

1.1 Background

Android is the most used smartphone OS today because it is open source and can be easily integrated into your hardware by manufacturers, making Android devices cheaper than iOS competitors. Currently, there are over 2 million applications available on Google Play Store. Having the positives also has its negatives. One of the main problems facing Android users is unexpected crashes and slowing down of devices over time. These problems mainly occur when the device runs out of memory. Phone manufacturers continue to increase main memory to compensate, but this is not the solution.

Currently, the Android operating system runs in a memory structure known as a garbage collector. Garbage Collector is a tool of Java memory management, as Android is based on the JVM. Garbage Collector search and identifies dead objects and reclaims space when they are no longer needed. We can free up memory in two ways, first, by periodically checking for the presence of dead objects and freeing their memory. Secondly, immediately freeing memory when allocated to make it more significant than the free memory available.

1.2 Motivation of the Research

According to StatCounter, there are 2 billion active smartphone users, and 71.93% are Android users. Nowadays, smartphones have become a necessity for everyone. Most of us rely too much on smartphones to complete our daily tasks. In some respects, they have replaced our computers. Over time, smartphones are becoming much more powerful devices. Only increasing main memory cannot be a solution for Android devices. I was fortunate to be the evolution of the smartphone industry from the very beginning. So, I always observed that day-by-day and Android phones increase their main memory for a better user experience. Also, it becomes slow with time, drains more battery, becomes hot, crushes the app very often. But on the other hand, iOS devices are released with lower battery and Main Memory, but they perform very well and sometimes better than high configured Android Devices. So, I was curious about what makes the iOS devices faster and optimized than the same configured Android devices? Also, it did not have a slowing down or battery draining issue like android devices. I was curious about these things from the beginning of my bachelor's life, so I decided to work on this research in my final year thesis.

1.3 Problem Statement

We find that there are many memory management systems used in Android OS which is based on current Garbage Collection techniques. The garbage collection is based on JVM. But we want to use more modern memory management techniques like Automatic Reference Counting. We did not find many memory management techniques besides garbage collectors. So, we will try to implement some optimization techniques that will lead us to more optimized programs. There is an alternative VM instead of JVM is called GraalVM. But it is not that much easy to implement the whole Java to GraalVM instantly.

1.4 Research Question

- How our proposed GraalVM is better than JVM ?
- How can we improve current JVM?

1.5 Research Objectives

- Implementing different optimization techniques of Garbage Collection in Java (Android) for optimized memory management.
- Find a better Virtual machine than current Java Virtual Machine.
- Using the memory in more optimized way that devices can work faster and better even with less memory, so devices perform well over the time.

1.6 Research Scope

Our research is based on optimized memory management techniques which we want to use in the current JVM. Obviously, using an entirely new technique in such an old environment is not so easy. Because nowadays, there are built-in memory management systems in the programming language. Java was a modern language when Android was invented, but over time it did not get any good plug in that can use real-time memory management techniques while coding. If we can find a way to utilize the memory in a more optimal way, then companies can use fewer hardware resources like processors, batteries but get a very good performance over time. In this way, people do not need to change their phones every year; as a result, our environment will be less polluted. Also, we can have a long-term good experience with our existing android phones.

1.7 Thesis Organization

This thesis consists of 7 sections. Next sections include Literature Review in Chapter 02 based on different case studies, Tools & Methods including feature selection, exploratory data analysis, model development and deployment in Chapter 03. Then in Chapter 4 we implemented our proposed solution like GraalVm and other JVM optimization techniques. Then in Chapter 5 in result part we showed table wise results and improved result that we got from Chapter 4. After that we discusses Result in Chapter 05. And we finally put our findings,

limitations conclude with future works can be done from here in Chapter 06. Chapter 07 is including with reference part.

2 LITERATURE REVIEW

Throughout this research we found some paper related to our topic of memory management. Some of them are around GC and some of them are also unique. In this section will try to review each paper related to our research and find out their findings and gaps.

Author	Year	Paper	Method	Keywords	Findings
Kashif Tasneem, Ayesha Siddiqui, Anum Liaquat	2019, February	Android Memory Optimization	ARC	Main Memory, Operating System, OS, Android, iOS, Memory, Optimization, Cache, Pages, Paging, Garbage Collector, GC, ARC, Automatic Reference Counting	This research proposes a different memory management technique instead of traditional Garbage collection on Android OS.

Aponso, G. C. A. L.	2017	Effective Memory Management for Mobile operating Systems. American Journal of Engineering Research (AJER), 246.	Dalvik VM	Kernel layers, Power management, Memory management, Low memory killer, Android virtual machine.	Low Memory Killer(LMK) starts to act and kill Least Recent Used(LRU) app when device will go to low memory stage. Out of Memory Killer(OOMK) which kill app having low priority.
---------------------------	------	--	--------------	---	--

Linares- Vasquez, M., Vendome, C., Tufano, M., & Poshyvanyk, D	2017	How developers microoptimize Android apps. Journal of Systems and Software, 130, 123.	FindBugs, PMD and LINT	Optimizations, Mining software repositories, Empirical studies, Android, Measurement	Micro optimizations must be applied on smartphone apps because they are more prone to performance issues. Usually developer don't use micro optimization because it have to be maintain formerly life cycle, and it's costly for smaller development
--	------	--	------------------------------	---	--

Yovine, S., & Winniczuk, G.	2017, May.	CheckDroid: a tool for automated detection of bad practices in Android applications using taint analysis. In Proceedings of the 4th International Conference on Mobile Software Engineering and Systems (pp. 175-176). IEEE Press.	CheckDroid tool.	Android applications, taint analysis, programming practices, bad patterns, runtime errors, newbie Android programmers	Long running task should be divided into sub thread. Those sub thread should have low priority then the main thread.
-----------------------------	------------	--	------------------	---	--

Qian, J., & Zhou, D.	2016	Prioritizing Test Cases for Memory Leaks in Android Applications. Journal of Computer Science and Technology, 31(5), 869-882.	Memory leak diagnosing techniques	Android, memory leak, test case prioritization, test execution	This research proposed a technique which determines a natural sequence of GUI events such as app launch and close. Repetition of such events should not increase memory usage if there are no memory leaks. If there are leaks, memory will keep on increasing
----------------------	------	---	-----------------------------------	--	--

Kwon, S., Kim, S. H., Kim, J. S., & Jeong, J.	2015, October	Managing GPU buffers for caching more apps in mobile systems. In Proceedings of the 12th International Conference on Embedded Software (pp. 207216). IEEE Press.	GPU buffer	Graphics processing unit, Mobile communication, Memory management, Androids, Humanoid robots, Kernel, Memory architecture.	Managing GPU Buffer compress the GPU buffer memory when app is sent to background. But here compression and uncompressing will increase access time which makes the solution less optimized
---	---------------	--	------------	--	---

Lee, B., Kim, S. M., Park, E., & Han, D.	2015, July	Memscope: Analyzing memory duplication on android systems. In Proceedings of the 6th Asia-Pacific Workshop on Systems (p. 19). ACM	Memscope Design	Memory duplication, Android system, Operating system, Memory management, Contextual software domains.	For avoiding memory duplication Android used several mechanisms such as zRAM and Kernel Same-Page Merging (KSM) Both those mechanism reduce memory usage but consume greater number of CPU cycles and power.
--	------------	--	-----------------	---	--

Kim, H., Lim, H., Man-atunga, D., Kim, H., & Park, G. H.	2015	Accelerating Application Start-up with Nonvolatile Memory in Android Systems. IEEE Micro, 35(1), 15-25.	TDRAMP hybrid solution	Nonvolatile memory, Memory management, Phase change materials, Random access memory, Mobile communication, Accelerators	NonVolatile Memory(NVM) or Phase Change Memory (PCM)as backup of main memory is proposed. NVM Memory is popular because it consumes less power. PCM is fast, very energy efficient for reading operations but consumes a lot of power in write operations and is very slow
Gerlitz, T., Kalkov, I., Schommer, J. F., Franke, D., & Kowalewsk S.	2013, October .	Nonblocking garbage collection for real-time android. In Proceedings of the 11th International Workshop on Java Technologies for Realtime and Embedded Systems (pp. 108-117). ACM.	Nonblocking garbage collection.	Real time garbage collector, Heap compaction, Heap fragmentation, Java Processor	This GC will work incrementally with short blocking phase. The speed of GC should be in accordance to the garbage created by the OS in order to avoid Out of Memory situations.

Lim, G., Min, C., & Eom, Y. I.	2013, January.	Enhancing application performance by memory partitioning in Android platforms. In Consumer Electronics (ICCE), 2013 IEEE International Conference on (pp. 649-650). IEEE.	Virtual memory node	Memory partitioning scheme, process lifecycle enhancement, mobile devices, virtual memory nodes	If memory runs out of one node, memory of only that node will be freed up. Normally, unreliable apps take up a lot more memory than reliable apps. By following this methodology, memory can be saved somewhat from running out
---	-------------------	--	------------------------	--	--

Table 1: Literature Review

3 TOOLS & METHODS

In this section, we will discuss what software, tools, plug-ins are being used in our research. Now we will discuss some major tools that we used to implement to show the outcome of our research.

3.1 GraalVM

GraalVM is a high-performance polyglot runtime which we can use in dynamic, static, and native languages. GraalVM offers features like accelerating the performance of existing Java applications also for building microservices. It provides advanced optimizing compiler technology to provide a high-performance Just-In-Time compiler that can be used to accelerate the performance of any JVM-based application without making any change in code. The future of the Java Virtual Machine is the GraalVM, and it is a development project from Oracle. It's a project to deliver an alternative virtual machine to the current Java Virtual Machine, which will provide better performance.

3.1.1 A Different approach of using GraalVM instead of JVM

The future of the Java Virtual Machine is the GraalVM and it is a development project from Oracle. It is a project to deliver an alternative virtual machine to the current Java Virtual Machine which will provide better performance.

Mainly there are three aspects of GraalVM that can be potentially useful for us if we are concerned about application performance.

- I. GraalVM is an alternative to the Java Virtual Machine.
We can run Java byte code using the Graal virtual machine as opposed to the standard Java Virtual Machine, and it will likely run faster.
- II. Secondly GraalVM provides an alternative java compiler to the regular java compiler. It provides more performance byte code.
- III. GraalVM can natively compile Java code to software that will run natively on our computer. (No JVM needed)

So doing this means that the software will not need a Java Virtual Machine to run and therefore it should run more quickly than the traditional way of running Java code.

3.1.2 Memory Management at Image Run Time

The Java heap is turned out when the native image starts up and increases or decreases in size while the native image runs. When the heap becomes full, the garbage collection is triggered to reclaim the memory of objects that are no longer used.

To managing the Java heap, Native Image offers different garbage collector implementation:

- The *Serial Garbage Collection* is the default GC in GraalVM. It is optimized for lower memory footprint and small Java heap sizes.
- The *G1 GC* is a multi-threaded GC optimized to reduce stop-the-world pauses and, therefore, improve latency while achieving high throughput.
- The *Epsilon GC* is a no-op garbage collector that does not do any garbage collection and therefore never frees any allocated memory.

Serial GC: The Serial GC is optimized for a lower footprint and small Java heap size. If any other GC is specified, the Serial GC will be used implicitly as the default on both GraalVM Community and Enterprise Edition. The Serial GC is a simple stop-and-copy GC. It generally divides the Java heap into a young and an old generation. Each generation normally consists of a set of equally sized chunks, each a connected range of virtual memory. Those chunks are the Garbage collector-internal unit for memory allocation and memory reclamation.

G1 Garbage Collector: GraalVM also provides the Garbage-First (G1) garbage collector based on the G1 GC from the Java HotSpot VM. G1 is a generational, incremental, parallel, mostly concurrent, stop-the-world, and evacuating Garbage Collection. It aims to present the best balance between latency and throughput. The G1 GC is an adaptive garbage collector with defaults that allow it to work efficiently without adjustment. However, it can be attuned to the performance needs of a particular application.

3.1.3 Advantages of GraalVM:

In this section, the advantages of *GraalVM* have been discussed.

Ideal for Microservices and Cloud: Oracle GraalVM Enterprise Edition's Ahead-of-Time compiler, called Native Image, allows your Java and JVM-based applications to be compiled ahead of time into a binary that runs natively on the system, improving startup and memory footprint.

GraalVM Native Image can decrease startup times of micro-services up to 100x and decrease memory usage by approximately 5x (Figure 1).

The major application frameworks are all compatible with GraalVM Enterprise

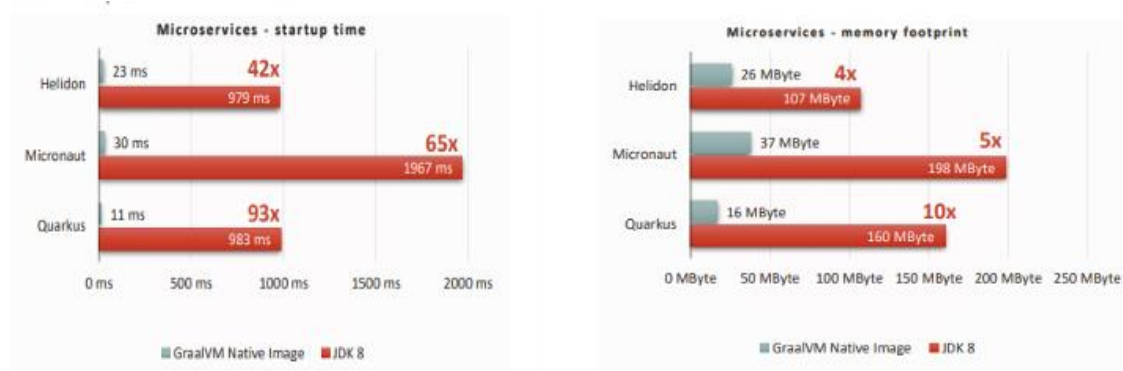


Figure 1: GraalVM Enterprise Native Image with GraalVM vs. JDK8

GraalVM Native Image provides instant and consistent throughput, as seen in Figure 2. This results in an initial transactions-per-second rate that is 1500% higher than a JIT compiled application, allowing microservices to perform at their peak, immediately. If the application is run for some time and the JIT compiler has time to warm up, it will achieve higher peak performance, e.g., 16% in the benchmark.

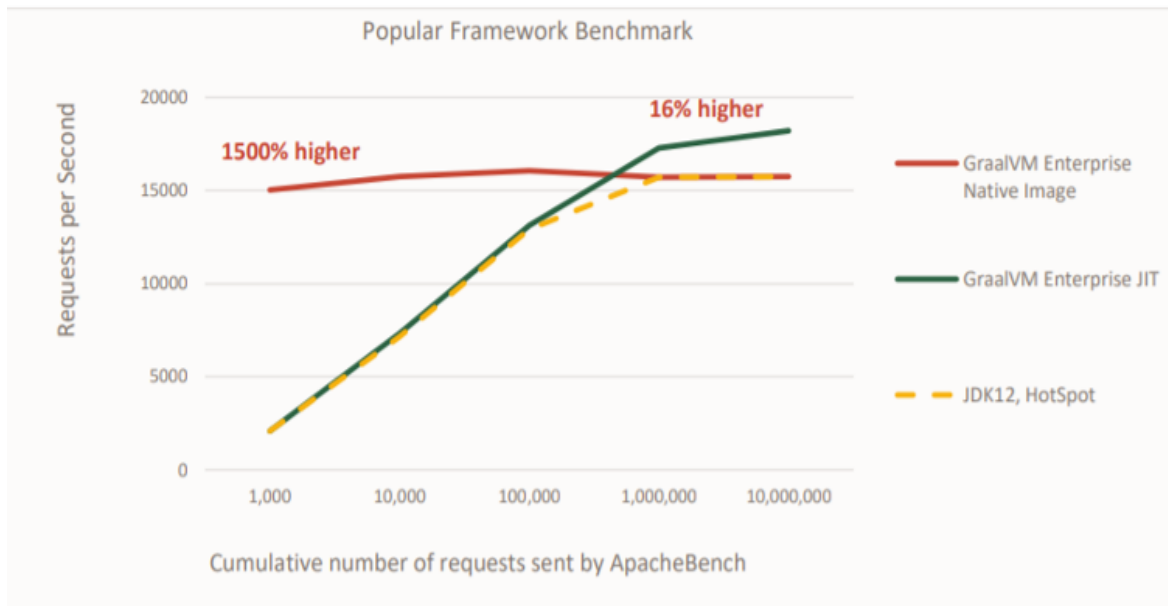


Figure 2: Throughput over time of GraalVM Enterprise using JIT compilation and Native Image (AOT) compilation vs JDK12

We do this first using Just-In-Time compilation with JDK8, the results of which are the chart on the left in Figure 3. Then, we repeat the process, but this time we use the GraalVM Enterprise Native Image (JDK8) compiled version of the same Micronaut application. The results of this run are depicted in the chart on the right in Figure 3.

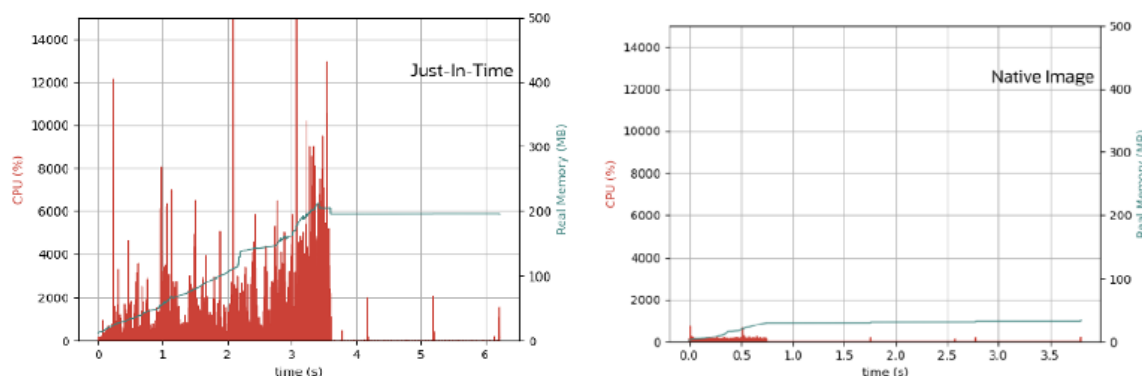


Figure 3: JIT vs. Native Image (AOT) starting up and serving two requests in the first ten seconds

JIT Version: The JIT version of the application takes approximately two seconds (2044ms) to start and return a valid response. During those three seconds, the process takes up as much as 15,000% of the CPU in spikes and averages about 3000%. It also climbs in memory usage until it plateaus at about 200MB.

Native: The Native Image compiled version of the application takes less than 1 second to startup (744ms) and return the first request that we use to confirm the process is ready. During this time, the service consumes approximately 50% CPU and plateaus at about 40MB of memory.

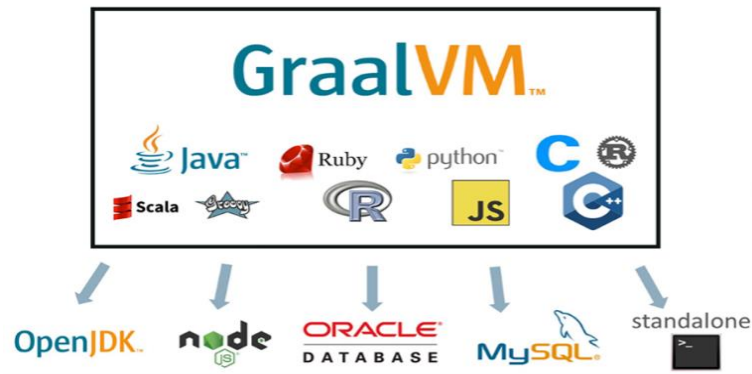


Figure 4: Language availability of Graal

i) Flexibility of working with many languages:

GraalVM allows developers the flexibility to build applications in different languages without the traditional overhead. Objects created in one language can be used directly in another language as if they are native to that language. This removes the traditional marshaling code required, simplifying the application, reducing memory and CPU usage, and getting the product to market more quickly.

Accelerating Application Performance: Without any code changes, GraalVM Enterprise can improve the performance of any Java application and any application that runs on the Java Virtual Machine. GraalVM aggressive in lining, polymorphic in lining, and partial escape analysis increase optimization opportunities, provide faster virtual method calls, and eliminate or delay object allocations. It provides for lower CPU used on the same code and fewer objects created, resulting in more miniature garbage collection and higher throughput. Faster application execution offers two benefits:

1. It reduces the response time for user requests, whether interactive or via RES. Applications are running on GraalVM exhibit lower latency, which is crucial when you remember that forty percent of consumers abandon web pages and shopping carts if the response time is over three seconds.
2. Applications that run faster free up CPU and memory sooner, allowing them to handle other requests or other applications running on the same server. In data centers with ever-increasing workloads, being able to service more requests with the same computing infrastructure reduces the need to purchase additional hardware. Thus, GraalVM Enterprise's reduction of required compute resources can lower capital cost expenditures on-premises and lower operating costs on the cloud.

ii) Real-Life Performance Improvement:

Twitter: Twitter approved the GraalVM JIT compiler for their Scala-based infrastructure and saw an 8-11% decrease in CPU consumption and a 20% increase in throughput. This resulted in a 512% decrease in the number of physical machines required for each service as it was moved to use GraalVM.

Oracle Cloud Infrastructure: Oracle Cloud Infrastructure moved to utilize GraalVM as the JIT compiler and runtime environment for its infrastructure. In doing so, it saw a 25% reduction in garbage collection time, a 10% increase in transactions/second, and has had 0 issues with 10s of millions of core hours of runtime since the migration.

3.2 Java Virtual Machine

The JVM has mainly two primary functions to allow

- I. Java programs can run on any device or OS (known as the “Write code once, run anywhere” principle)
- II. and to manage and optimize program memory.

Java virtual Machine is an engine that provides a runtime environment to run the Java Code or applications. It converts Java byte code into machine language. Java Virtual Machine is a part of the Java Run Environment. In other programming languages, the compiler generally produces machine code for a particular system. Java compiler produces the code version for a Virtual Machine known as Java Virtual Machine. So Java is the current language for Android OS, so we used JVM. We used three versions of Java from our implementation perspective: 1.8, 11, and the latest 16 version. When Java was released in the year 1995, on that time all computer programs were written to a specific OS, and program memory was managed by the software developer. So, the JVM was a revelation.

3.3 Eclipse

Eclipse is an essential tool for any Java developer, including a Java IDE, a CVS client, Git client, XML Editor, Maven integration, and WindowBuilder. We can easily combine multiple languages support and other features into any of our default packages. Eclipse Marketplace allows us for virtually unlimited customization and extension. It also has its own JVM package from version oldest to latest JVM version 16. So mostly in our research, we use Eclipse as our primary IDE. Also, we can change the version before running. We mostly used to change the default flag using runtime arguments.

3.4 IntelliJ

IntelliJ is the most modern and smart IDE where every aspect has been designed to maximize developer productivity. It has intelligent coding assistance and ergonomic design make development not only productive but also enjoyable. It has all support of Java, Kotlin, Groovy, Scala, Android, Maven, Git, SVN, Mercurial, Perforce, Debugger, Profiling tool, JavaScript, Database Tools, and SQL. Particularly in our implementation, I used IntelliJ for its Profiling tool. We used this tool to perform the Java Flight Recorder of our project and found what is wrong with the multi-threaded application.

3.5 Java VisualVM

Java VisualVM is a tool by Oracle that provides a visual/graphical interface for viewing detailed information about Java applications while running on a Java Virtual Machine. Java VisualVM organizes JVM data retrieved by the JDK tools and presents the information to enable us to view data on multiple Java applications and compare quickly. We can view data on local applications and applications that are running on remote hosts. We can also capture data about the JVM software, save the data to our local system, view the data later, or share the data with others. Now java VisualVM is bundled with JDK version 6 update 7 or greater. We used VisualVM to monitor our heap size in memory, CPU usage, classes, threads, performing garbage collection, force the GC, sampling, and profiling.

3.6 Memory Analyzer Tool

The Eclipse Memory Analyzer TOOL is a fast and feature-rich Java Heap Analyzer that helps us to find memory leaks and reduce memory consumption.

Memory Analyzer can analyze productive heap dumps with hundreds of millions of objects. It also quickly calculates the retained sizes of those objects and see who is preventing the Garbage Collector from collecting objects. Finally, it runs a report to extract leak suspects automatically.

So, in our research, we used MAT to find the leaking object in our project, that was leading us to memory leakage.

3.7 Linux

Linux is one of the greatest operating system for modification. We used Linux in our research for using GraalVM. Because in Windows version we do not have all the features on. So, we tried to run same java code in GraalVM & JVM to see the difference.

4 IMPLIMENTATIONS

4.1 Using GraalVM instead of JVM for better memory Management:

We have implemented a basic Java project of finding the largest prime number from a very big data set. So, at first we run the code into normal JVM and after that we implemented the same code in GraalVM.

```
mh@mh-VirtualBox: ~/Desktop$ java -version
openjdk version "11.0.11" 2021-04-20
OpenJDK Runtime Environment (build 11.0.11+9-Ubuntu)
OpenJDK 64-Bit Server VM (build 11.0.11+9-Ubuntu)
mh@mh-VirtualBox: ~/Desktop$ javac Main.java
mh@mh-VirtualBox: ~/Desktop$ java Main
UIDs generated
Integers generated
19800 primes found.
The largest prime was 999947881
time taken : 4757 ms.
mh@mh-VirtualBox: ~/Desktop$ java Main
UIDs generated
Integers generated
19979 primes found.
The largest prime was 999981049
time taken : 4811 ms.
mh@mh-VirtualBox: ~/Desktop$ java Main
UIDs generated
Integers generated
19964 primes found.
The largest prime was 999940477
time taken : 4849 ms.
mh@mh-VirtualBox: ~/Desktop$ java Main
UIDs generated
Integers generated
19942 primes found.
The largest prime was 999804749
time taken : 4865 ms.
```

Fig 1: OpenJDK & JVM

```
mh@mh-VirtualBox: ~/Desktop/graalvm-ce-java11-21.0.0.2/bin$ java Main
UIDs generated
Integers generated
19866 primes found.
The largest prime was 999891727
time taken : 4563 ms.
mh@mh-VirtualBox: ~/Desktop/graalvm-ce-java11-21.0.0.2/bin$ java Main
UIDs generated
Integers generated
20019 primes found.
The largest prime was 999929219
time taken : 4549 ms.
mh@mh-VirtualBox: ~/Desktop/graalvm-ce-java11-21.0.0.2/bin$ java Main
UIDs generated
Integers generated
19940 primes found.
The largest prime was 999925781
time taken : 4506 ms.
mh@mh-VirtualBox: ~/Desktop/graalvm-ce-java11-21.0.0.2/bin$ java Main
UIDs generated
Integers generated
20053 primes found.
The largest prime was 999979469
time taken : 4601 ms.
mh@mh-VirtualBox: ~/Desktop/graalvm-ce-java11-21.0.0.2/bin$ java Main
UIDs generated
Integers generated
19883 primes found.
The largest prime was 999982583
time taken : 4519 ms.
```

Fig2: OpenJDK & GraalVM

```
mh@mh-VirtualBox: ~/Desktop/graalvm-ce-java11-21.0.0.2/bin$ java Main
UIDs generated
Integers generated
19643 primes found.
The largest prime was 999997049
time taken : 4547 ms.
mh@mh-VirtualBox: ~/Desktop/graalvm-ce-java11-21.0.0.2/bin$ java Main
UIDs generated
Integers generated
20010 primes found.
The largest prime was 999978209
time taken : 4465 ms.
mh@mh-VirtualBox: ~/Desktop/graalvm-ce-java11-21.0.0.2/bin$ java Main
UIDs generated
Integers generated
20120 primes found.
The largest prime was 999964309
time taken : 4628 ms.
mh@mh-VirtualBox: ~/Desktop/graalvm-ce-java11-21.0.0.2/bin$ java Main
UIDs generated
Integers generated
19859 primes found.
The largest prime was 999940439
time taken : 4433 ms.
mh@mh-VirtualBox: ~/Desktop/graalvm-ce-java11-21.0.0.2/bin$ java Main
UIDs generated
Integers generated
19939 primes found.
The largest prime was 999950153
time taken : 4482 ms.
mh@mh-VirtualBox: ~/Desktop/graalvm-ce-java11-21.0.0.2/bin$ java Main
UIDs generated
Integers generated
20007 primes found.
The largest prime was 999994861
time taken : 4506 ms.
```

Fig 3: GraalVM & GraalVM

```
[main:5035] compile: 29,703.70 ms, 1.39 GB
[main:5035] image: 1,782.37 ms, 1.39 GB
[main:5035] write: 549.98 ms, 1.39 GB
[main:5035] [total]: 62,208.21 ms, 1.39 GB
mh@mh-VirtualBox: ~/Desktop/graalvm-ce-java11-21.0.0.2/bin$ ./main
UIDs generated
Integers generated
19823 primes found.
The largest prime was 999992639
time taken : 5422 ms.
mh@mh-VirtualBox: ~/Desktop/graalvm-ce-java11-21.0.0.2/bin$ ./main
UIDs generated
Integers generated
19974 primes found.
The largest prime was 999994549
time taken : 5390 ms.
mh@mh-VirtualBox: ~/Desktop/graalvm-ce-java11-21.0.0.2/bin$ ./main
UIDs generated
Integers generated
19897 primes found.
The largest prime was 999996149
time taken : 5424 ms.
mh@mh-VirtualBox: ~/Desktop/graalvm-ce-java11-21.0.0.2/bin$ ./main
UIDs generated
Integers generated
19603 primes found.
The largest prime was 999993143
time taken : 5379 ms.
mh@mh-VirtualBox: ~/Desktop/graalvm-ce-java11-21.0.0.2/bin$ ./main
UIDs generated
Integers generated
19819 primes found.
The largest prime was 999954149
time taken : 5438 ms.
```

Fig 4: GraalVM & Native-image

Figure 5: Average execution time for same code in GraalVM & JVM

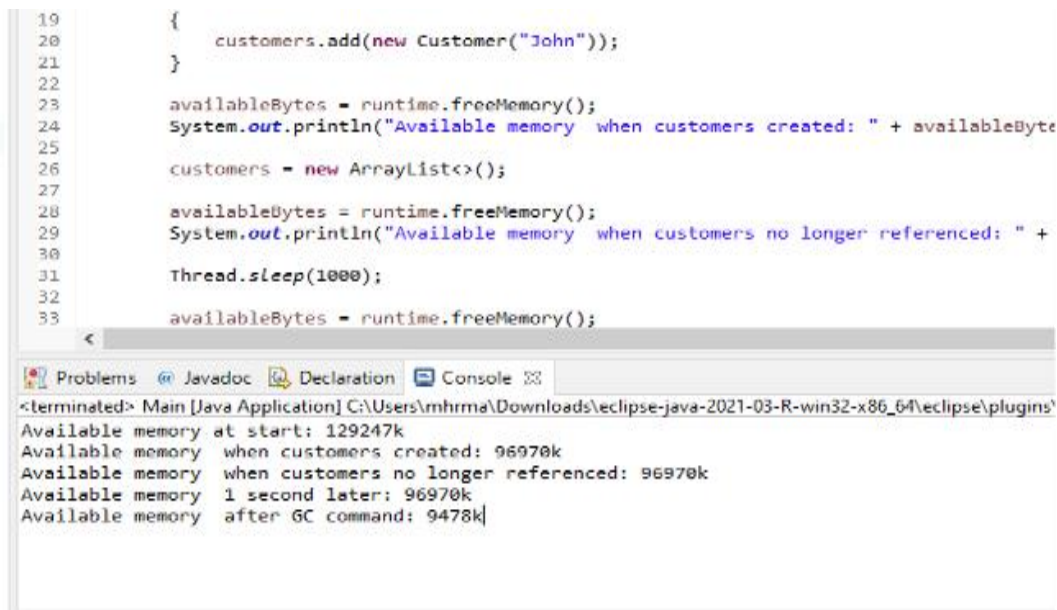
Result (Figure 5) is very satisfying that without making any change in code we got less execution time using GraalVM instead of JVM. It was 5.66% more faster than typical JVM. Also using GraalVM compiler and Virtual Machine together we get 7.26% improvement in execution time then we used to get in OpenJDK & Java Virtual Machine. So, we should also get better performance using native image features on GraalVM. But unfortunately, in our test Native image performed worst and it took 12.36% more execution time than normal JVM.

4.2 Controlling Garbage Collection

The idea of garbage collection is that programmers ask for objects to be allocated on the heap but they don't need to free them when they're finished with them. Instead, an automatic process will analyze the heap and it aims to work out which objects are no longer needed and any unneeded objects can be deleted and the memory that they occupy can be freed up. Java works out which objects are no longer needed using a very simple rule.

Any object on the heap which cannot be reached for a reference from the stack is eligible for garbage collection.

4.2.1 Control initial Heap Size using Flag



```
19     {
20         customers.add(new Customer("John"));
21     }
22
23     availableBytes = runtime.freeMemory();
24     System.out.println("Available memory when customers created: " + availableByte
25
26     customers = new ArrayList<>();
27
28     availableBytes = runtime.freeMemory();
29     System.out.println("Available memory when customers no longer referenced: " +
30
31     Thread.sleep(1000);
32
33     availableBytes = runtime.freeMemory();
```

<terminated> Main [Java Application] C:\Users\mhurma\Downloads\eclipse-java-2021-03-R-win32-x86_64\eclipse\plugins\

Available memory at start: 129247k
Available memory when customers created: 96970k
Available memory when customers no longer referenced: 96970k
Available memory 1 second later: 96970k
Available memory after GC command: 9478k

Figure 6: Default Heap Size and available free Heap

We start with 129 0000 kilobytes of memory. It goes down to 96970 kilobytes after creating 1 million customers in the program. And then after the GC command it has gone down hugely to just 9478 kilobytes of memory. One thing this tells us that certainly calling the GC method meant that the garbage collection process ran in this instance. However, is the amount of available memory so much less before creating the instance in Java 11. It is because of an optimization in the Java 11 garbage collection process which didn't exist in Java 8 or before. From a performance point of view there is an impact of this enhancement if we recall one of the things that every time virtual machine needs to go to the operating system after running GC and give back the extra memory that it does not use.

By using a flag to tell the virtual machine when we start to request a particular value to be the initial heap size and even in Java 11 if we use that flag the virtual machine will never let the amount of memory, we have reserved go below that initial heap size.

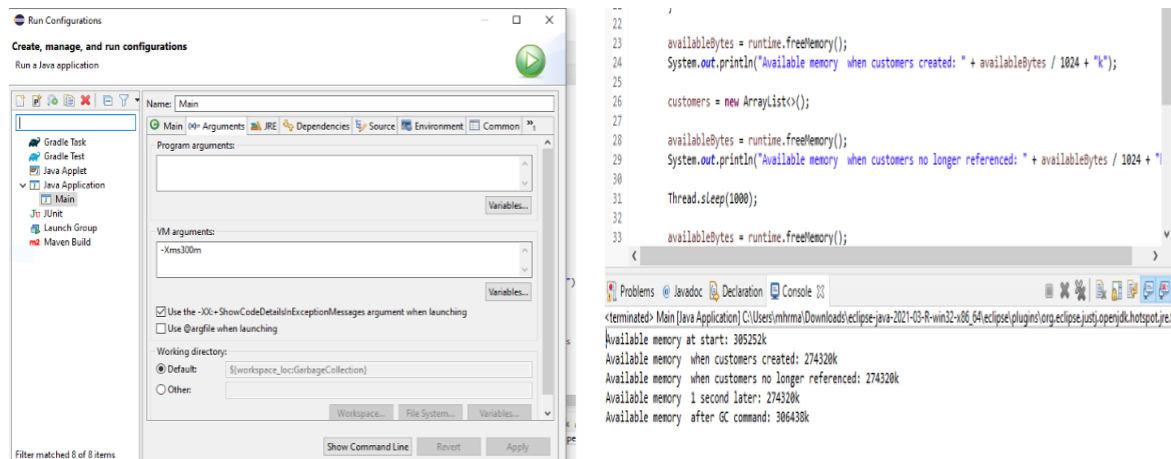


Figure 7: Tuning Heap Size and available free Heap

So, this time we started with 300 megabytes, and it went down to 274 megabyte after creating customer. But after the garbage collection (`System.gc()`) ran we're back up and actually it's similar to what we saw that result is again result is slightly more than 300 megabytes. So, we have fixed the issue of losing initial heap size.

4.2.2 Analyzing Heap Dump

If we find in a situation where the garbage collector is not working, and it is unable to free up enough space to be able to run your applications without them crashing with that out of memory error. We can run a program which have a memory leak so if we run the application it is going to run out of memory and crash. So, we will go to the Visual VM and see what object on the heap is causing the problem and generate a file of it.

Visual VM: We can go the Visual VM and see what object on the heap is causing the problem and generate a file of it. We opened Visual VM connect to our customer harness, and we'll have a look at the heap. We found the point where the heap is plateauing. We can just click on **HEAP DUMP** button, and it is going to generate the file for us.

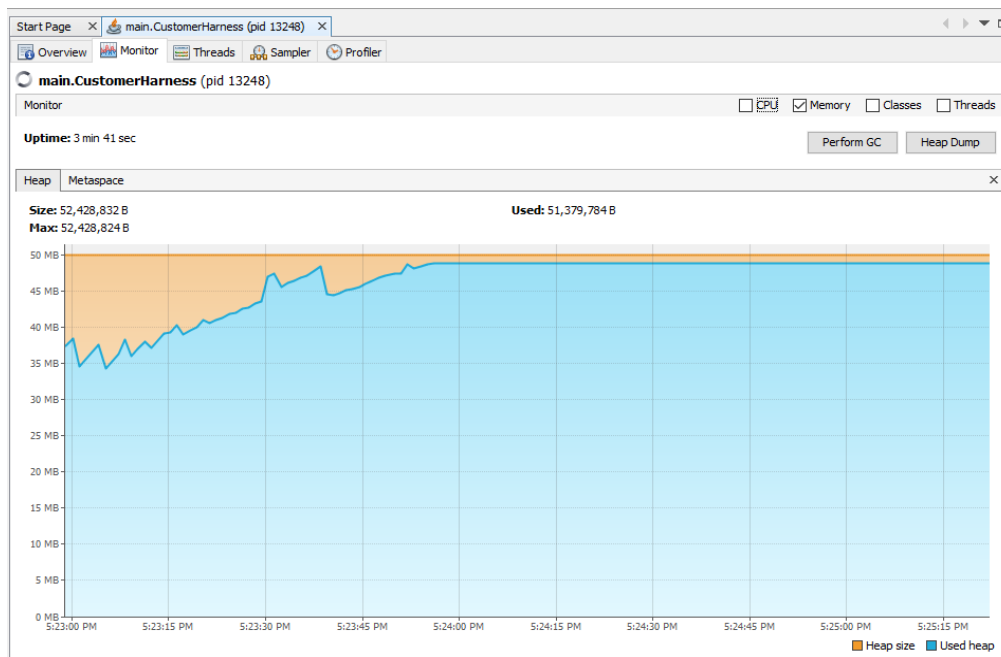


Figure 8: Heap monitor of Garbage Collection

4.2.3 Memory Analyzer (MAT)

Leak Suspect Report: This tells us what objects are kept alive on the heap and why they have not been garbage collected.

Component Report: Component report tells us as you can see things like if there were duplicate strings, empty collections, other things.

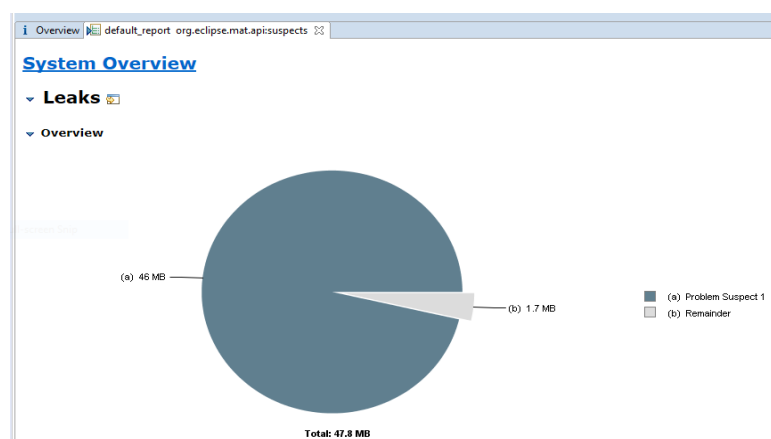


Figure 9: Leak Suspect Report

The pie chart here is saying that eclipse memory analyzes it all suspects there is a problem. In total of 50 megabyte there is an object on the heat which is taking up 46 megabytes and

everything else on the heap is just 1.7 megabyte. So that certainly looks like what eclipse think is a suspected memory leak or a leak suspect in their terminology.

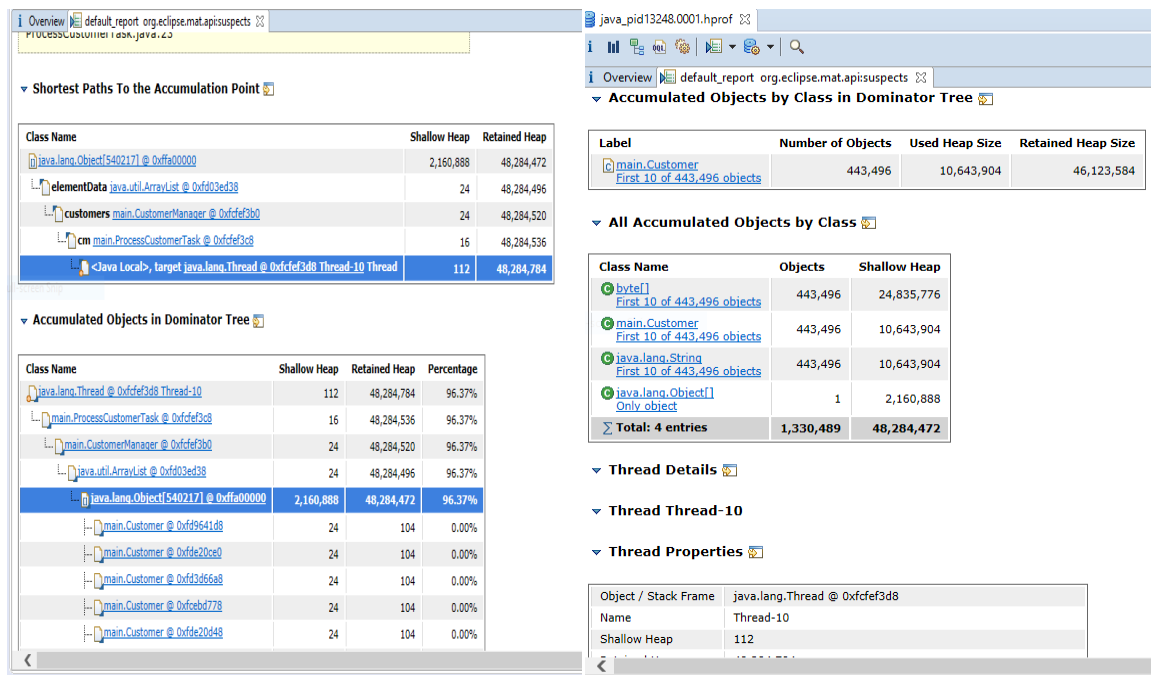


Figure 10: Leak Description

After seeing the details: Here in this dataset, we can see that we have an object called customers which is of type customer manager and it's using up almost all of that heap. That is the retained heap it's the amount of heat that could not be garbage collected.

The second table we can see in the manager class there is an array list, and the array list is containing an object. That is probably the underlying array that is used for the array list. And that object is containing all these different customer objects so we can see here clearly that what we have got is an array list containing lots of customer objects and that is not good for your application.

That gives us a clue as to why this application has run out of memory. The memory analyzer has told us is that our Heap has been taken up by all these different objects. But clearly there is an array list that is using a huge amount of retained heap. that allows us to go back to our application to find the problem and hopefully fix it. We now know that there is a particular Array list which is growing out of control and that clearly should not be happening. So Memory Analyzer (MAT) is something that's very useful for us to know about in your own applications.

4.3 Tuning the Garbage Collection

We can make a program that we will adding customers into a wait list 10000 customers and remove 5000. So, we are going to be freeing up a larger number of customers all in one go. Here we have created a structure that means that most customers will survive for a little while but not really for that long. Every time a customer is created it is going to survive until at least another 5000 customers have created and then at some point shortly after that it will be available for garbage collection. We are going to run this with a smallish heap in this case 20 megabytes. Now in Visual VM we can see Visual Representation of Garbage Collection.

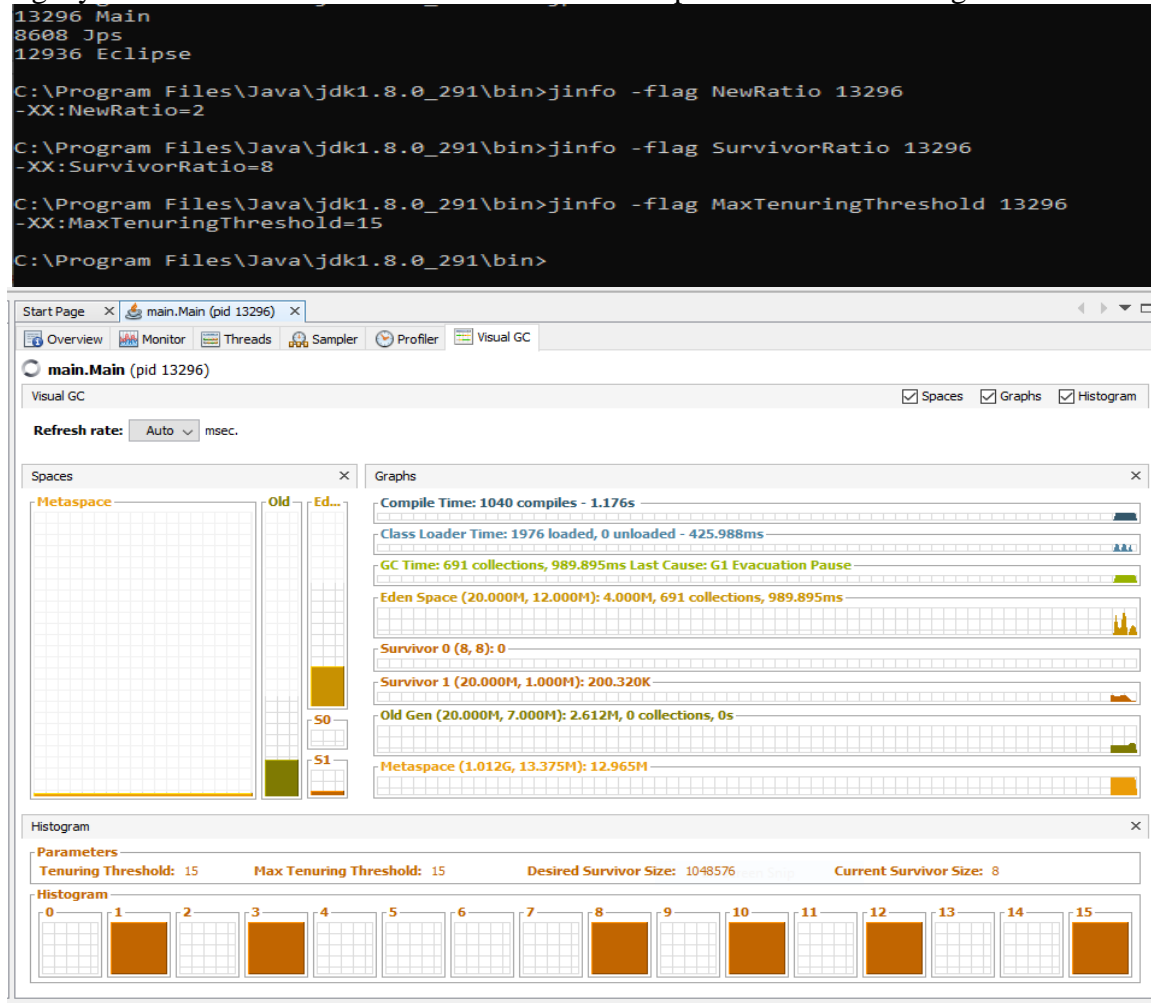


Figure 11: Default NewRatio(2), SurvivorRatio(8), MaxTenuringThreshold(15)

Garbage collection that takes place in the young generation are better for performance than garbage collections on the old generation. Young generation garbage collections are quick and efficient. We will not notice any real impact on our application's performance certainly compared to an old generation garbage collection. One thing we will certainly want to do is minimize the number of full garbage collections.

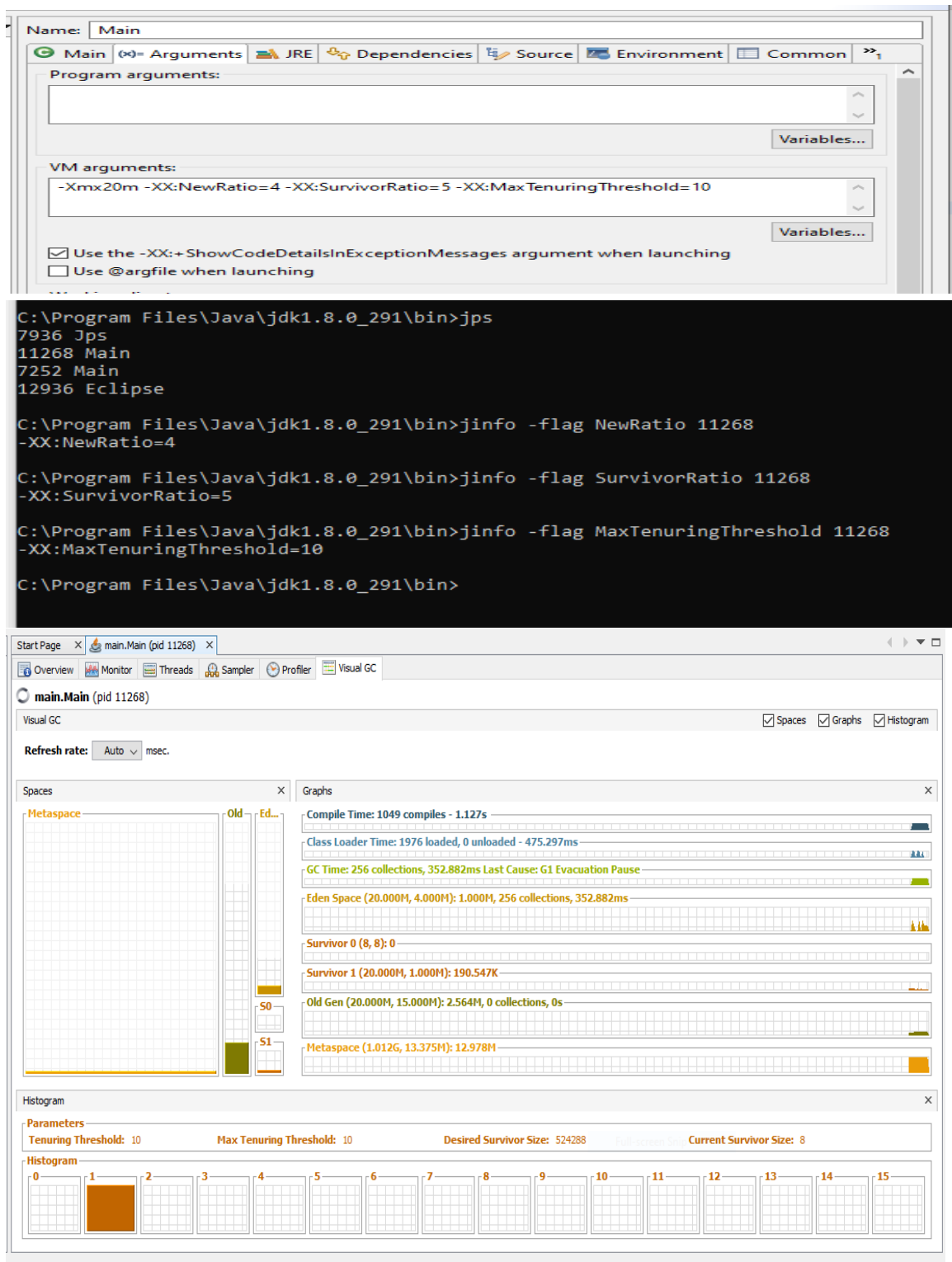


Figure 12: After tuning the NewRatio(4), SurvivorRatio(5), MaxTenuringThreshold(10)

4.3.1 Parameter for New Ratio

-XX:NewRatio = n

By tuning NewRatio we can resize the different parts of the heap. It means how many times bigger should the old generation be compared to the young generation. By default, we say this value is 2 then it means Old Generation will be twice of the Young Generation. So, if we want to increase the size of young generation and reduce size of Old Generation we can set value to 1.

4.3.2 Parameter of Survivor Ratio

-XX:SurvivorRatio = n

This flag means how much of the young generation should be taken up by the survivors' spaces S0 and S1. In our machine the default value is 8. So, this means that S0 and S1 should each be 1/8 the of the young generation and in the Eden space will be 6/8 so the Eden space is going to be three quarters of the young generation and our survivor spaces are each going to be half of the remaining quarter.

Now we could change this value and if we reduce it that makes these survivor spaces bigger. If we said for example survival ratio is 5 that means that each of the survivor spaces is going to be a fifth of the young generation which is bigger than the 8th that they are in default mode.

4.3.3 Parameter of MaxTenuringThreshold

-XX:MaxTenuringThreshold = n

MaxTenuringThreshold determines how many generations should an object survive before it becomes part of the old generation. So, we normally want this is to be high as possible because we want our objects to live in the young generation for as long as possible. So, in our case the value we check is 15 and this is the max value. If we want our objects to be promoted to the old generation sooner only then we can reduce the number from 15 but that will not of course be good optimized.

4.4 G1 Garbage Collector

Normally the heap is split into regions and by default there's 2048 of them. Initially some of these regions are allocated to the different parts of the heap. So, some will be allocated to Eden to S0 to S1 or to the old generation. Not all of them regions do need to be allocated initially.

When Java decides that a garbage collection is needed in the young generation it looks at those regions allocated to the young generation. It does the garbage collection process and then it can reallocate the number of regions allocated to each part of the young generation to give it what it thinks will be optimal performance. So, each time a young generation garbage collection happens the different parts of the heap could be changed. Garbage collector might decide some of the regions that have been previously allocated to the survivor spaces should now be allocated to the Eden space and it could decide to add some of the unallocated regions for example to the Eden space. So that happens every time a minor collection is needed.

When a full garbage collection takes place the garbage collector will work out for each region in the old generation which regions are mostly garbage, and it will collect the garbage from those regions first. That is why it is called G1 the Garbage First Collector. If a region contains only unreferenced objects that is a completely garbage region it can be fully cleared and made available. In fact, the G1 garbage collector does not actually therefore need to look at the entirety of the old generation. If it can clear a few regions that might be enough so that a real full garbage collection on the entire old generation is not actually necessary. It can still do a full garbage collection on the entire old generation if it is really needed but hopefully that will be a rare occurrence.

So, the idea is that the performance of the G1 garbage collector should generally be better than the other types of garbage collector that we've been looking at because when it needs to do a major collection it can often do just part of a major collection to free up enough memory and it has the ability to resize and reallocate different areas of the heap to different parts of the young and the old generation again to maximize performance.

Tuning the G1 GC if needed

Flag 1. `-XX:ConcGCThreads = n`

We can this flag to specify the number of concurrent threads available for smaller regional collections. It is only useful if we want to limit the number of threads so that we do not impact performance of other applications.

Flag 2. `-XX:InitiatingHeapOccupancyPercent = n`

By default, the G1 process starts when the entire heap reaches 45% of fullness. We can change this figure from the 45 percent default by using this flag. We can try changing this flag and see what number really work optimally based on our specific scenario.

Flag 3. `-XX:UseStringDeDuplication`

If we use G1 garbage collector then we can turn on String De-Duplication for creating more available space on the heap. If we enable this feature, it allows the garbage collector to make more space if it finds duplicate strings in the heap. When string de duplication feature turned on what the garbage collector will do is it will compare each of the strings and if it finds two strings that have the same value it will make the second object available for garbage collection.

4.5 Efficient Mark & Sweep Process

The general principle rather than searching for all the objects to remove instead the garbage collector looks for all the objects that need to be retained and it rescues them. The general algorithm that garbage collectors use is called Mark and sweep, and this is a two-stage process. The first stage is the marking, and the second stage is the sweeping in the marking stage. The garbage collector does not merely collect any garbage it actually collects the objects which are not eligible for garbage collection, and it saves them. This means that the garbage collection process is faster the more garbage there is. If everything on the heap was garbage well then, the garbage collection process would be pretty much instantaneous.

In Generational garbage collection so the heap is divided into two sections. One is called the young generation and one is called the old generation.

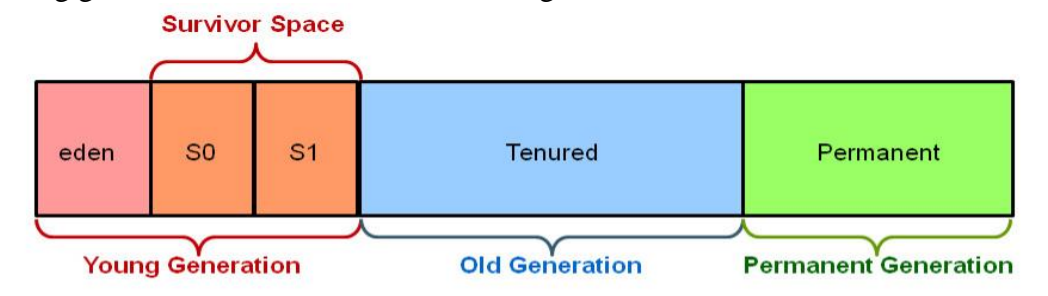


Figure 13: Generational Garbage Collection

The young generation which is full of new objects is probably mostly garbage so the process to garbage collect the young generation should be very quick. The garbage collection of the young generation is known as a minor collection so as our application runs there will be lots of minor garbage collections taking place and they will be pretty much instantaneous. Because the young generation will be quite small and mostly full of garbage. The young generation is split then into three sections called Eden S0 and S1 and they are called The Survivor spaces. The object is determined to be a long surviving object and will be moved from S0 or S1 into the old generation. This is all about then is a minor garbage collection.

A major garbage collection will take place which is a much slower process but in real world performing application these major garbage collections should be rare.

4.6 Using Profiler on Application

When we use profiling a Java application, we can monitor the Java Virtual Machine (JVM) and obtain data about application performance, including method timing, object allocation and garbage collection. We can use this data to locate potential areas in our code that can be optimized to improve performance.

Here we will use one project that is not optimized, then we will compare some code with some upgrade in threads and observe the difference.

4.6.1 Overall Monitoring

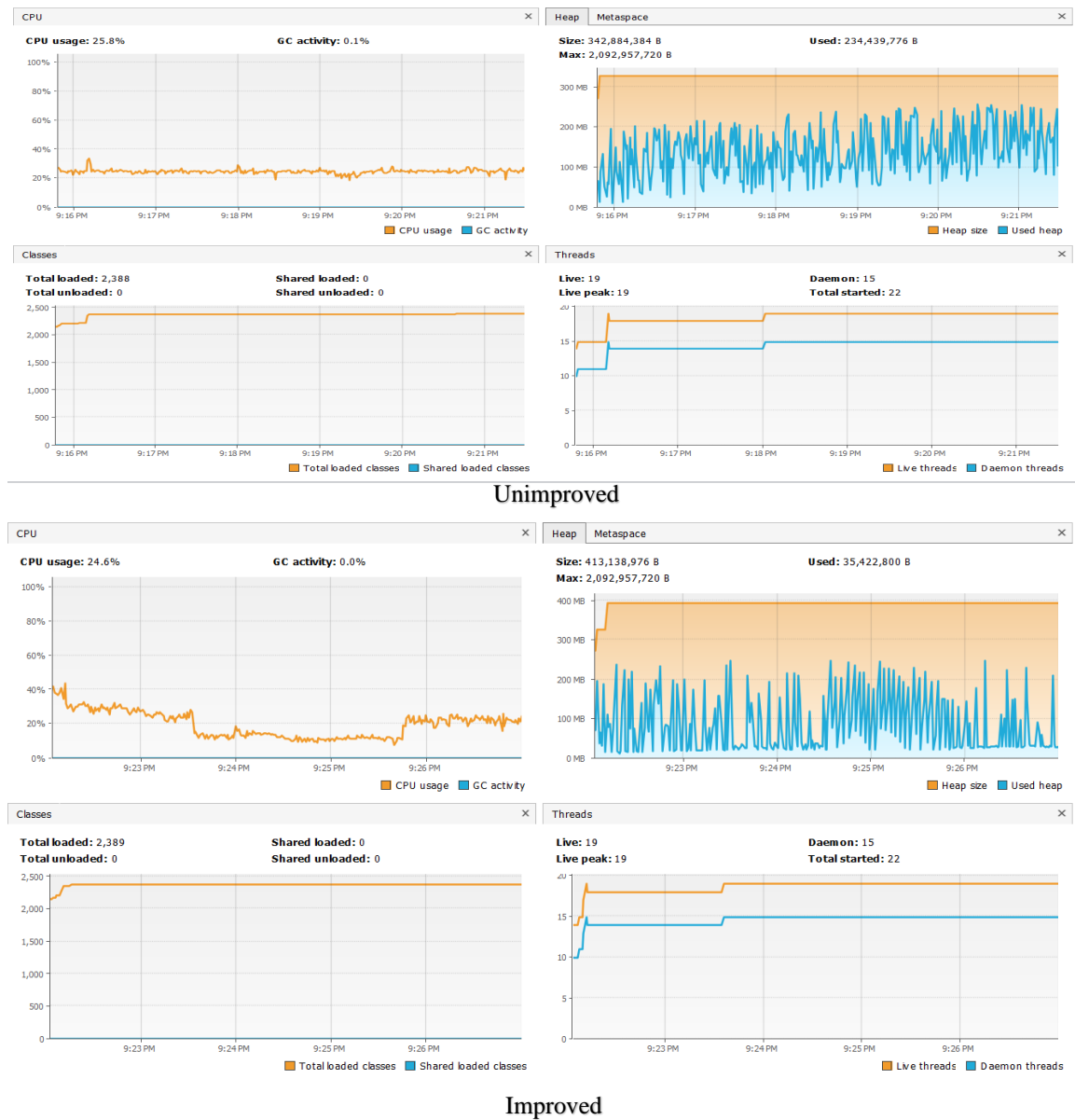


Figure 14: Overall Monitoring

4.6.2 CPU Profile

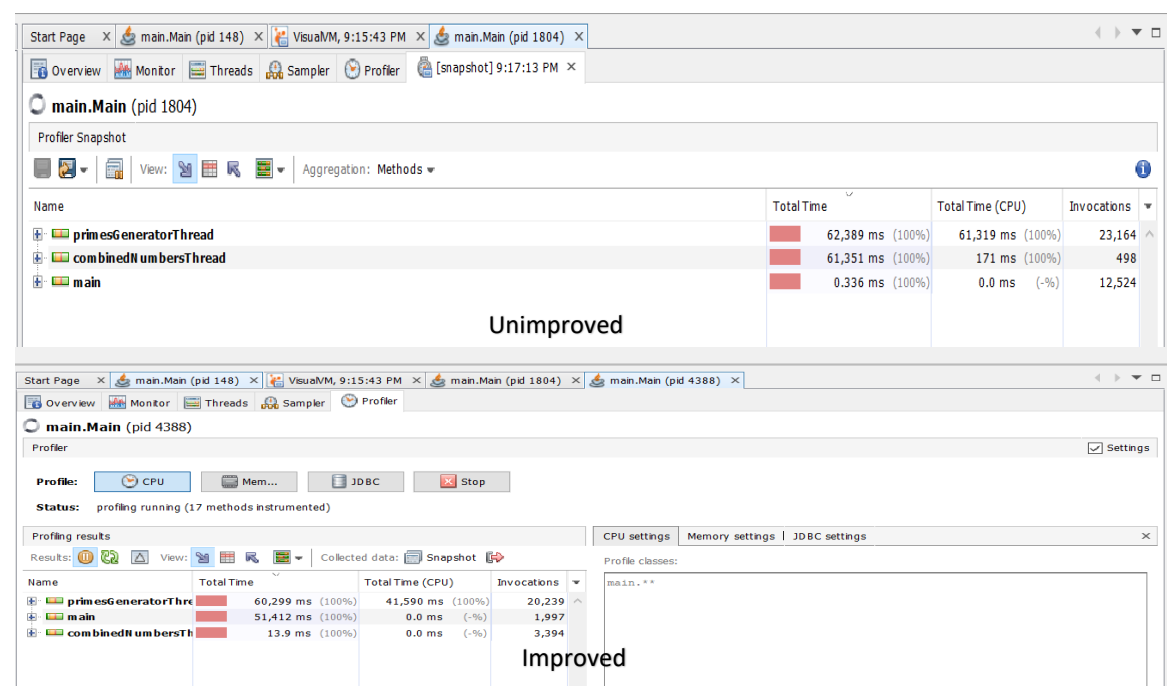


Figure 15: CPU Profiling

4.6.3 CPU Sample

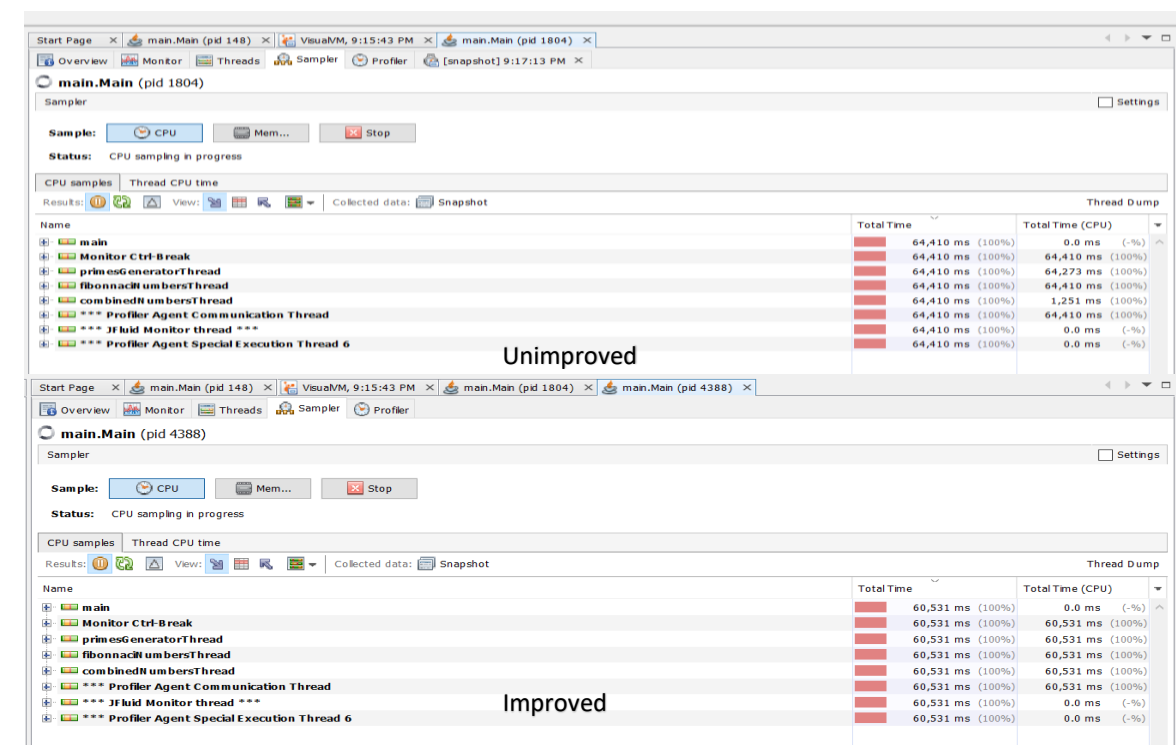


Figure 16: CPU Sampling

4.6.4 Thread CPU Time

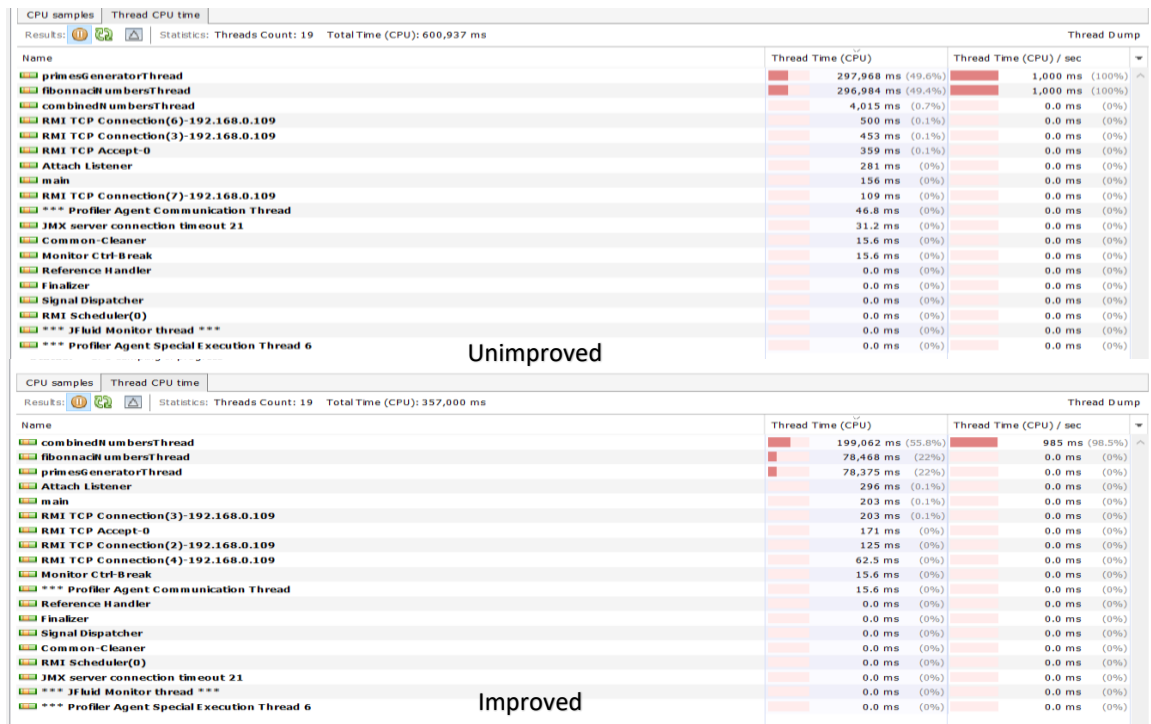


Figure 17: Thread CPU time

4.6.5 Memory Heap Histogram

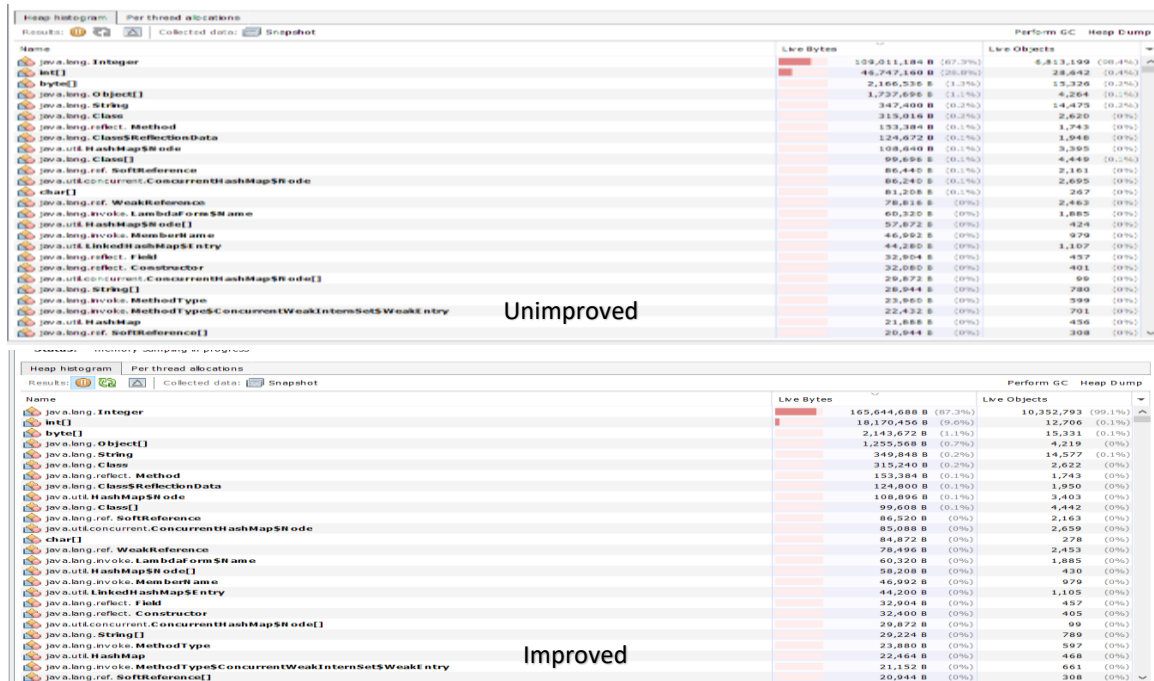


Figure 18: Memory Heap Histogram

4.6.6 Memory Per Thread Allocation

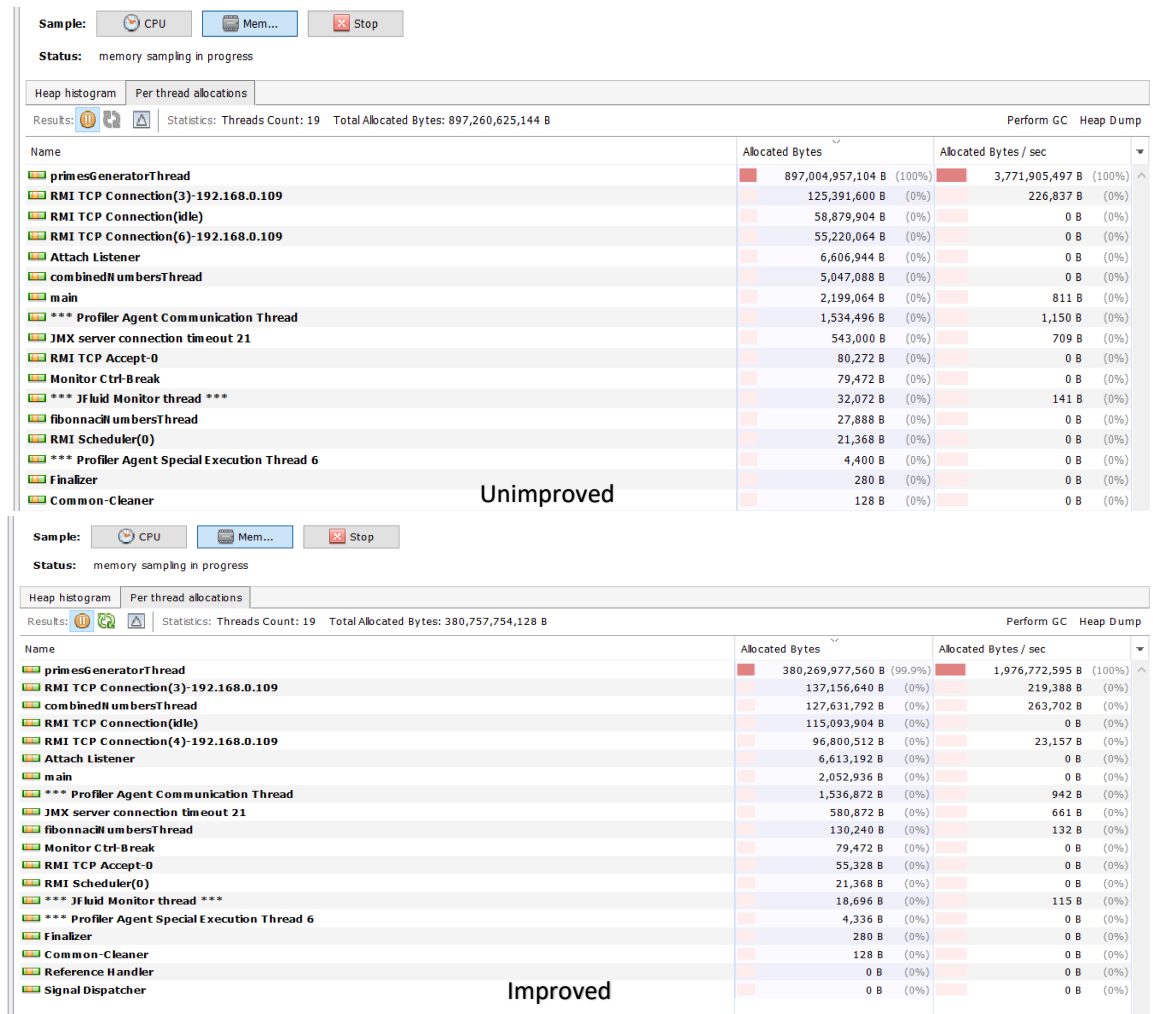


Figure 19: Memory Per Thread Allocation

So, we found in unimproved version what is going wrong, which thread is behaving weird. we found those things and then upgrade our code. We can see better result in the improved version.

4.7 Java Flight Recorder

In this implementation part, we used the same Fibonacci Prime Number project and see the difference in results between them.

4.7.1 Flame Graph

The flame graph visualizes the application call tree with the rectangles that stand for frames of the call stack, ordered by width. Methods that consume more CPU time and memory resources are wider than the others. The blue color of the blocks stands for native calls, yellow stands for Java calls.

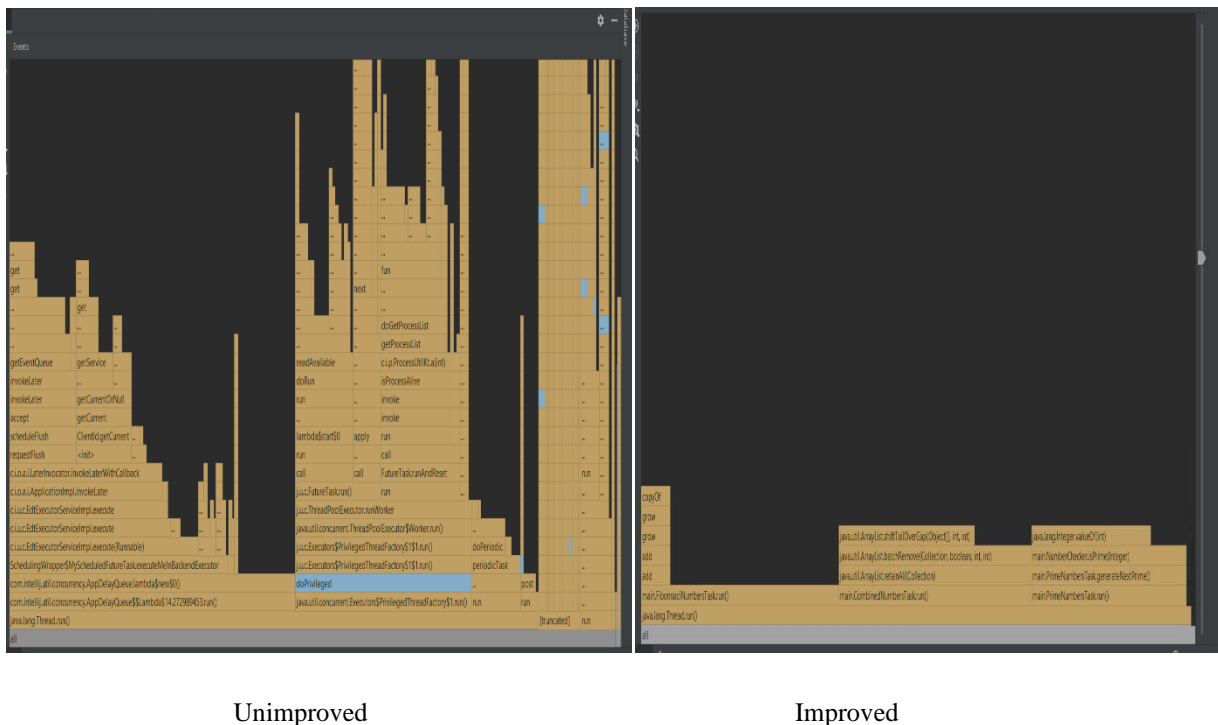


Figure 20: Flame Graph (Improved)

4.7.2 Call Tree

The Call Tree tab represents information about a program's call stacks that were sampled during profiling. The top-level All threads merged option shows all threads merged into a single tree. There is also a top-down call tree for each thread.

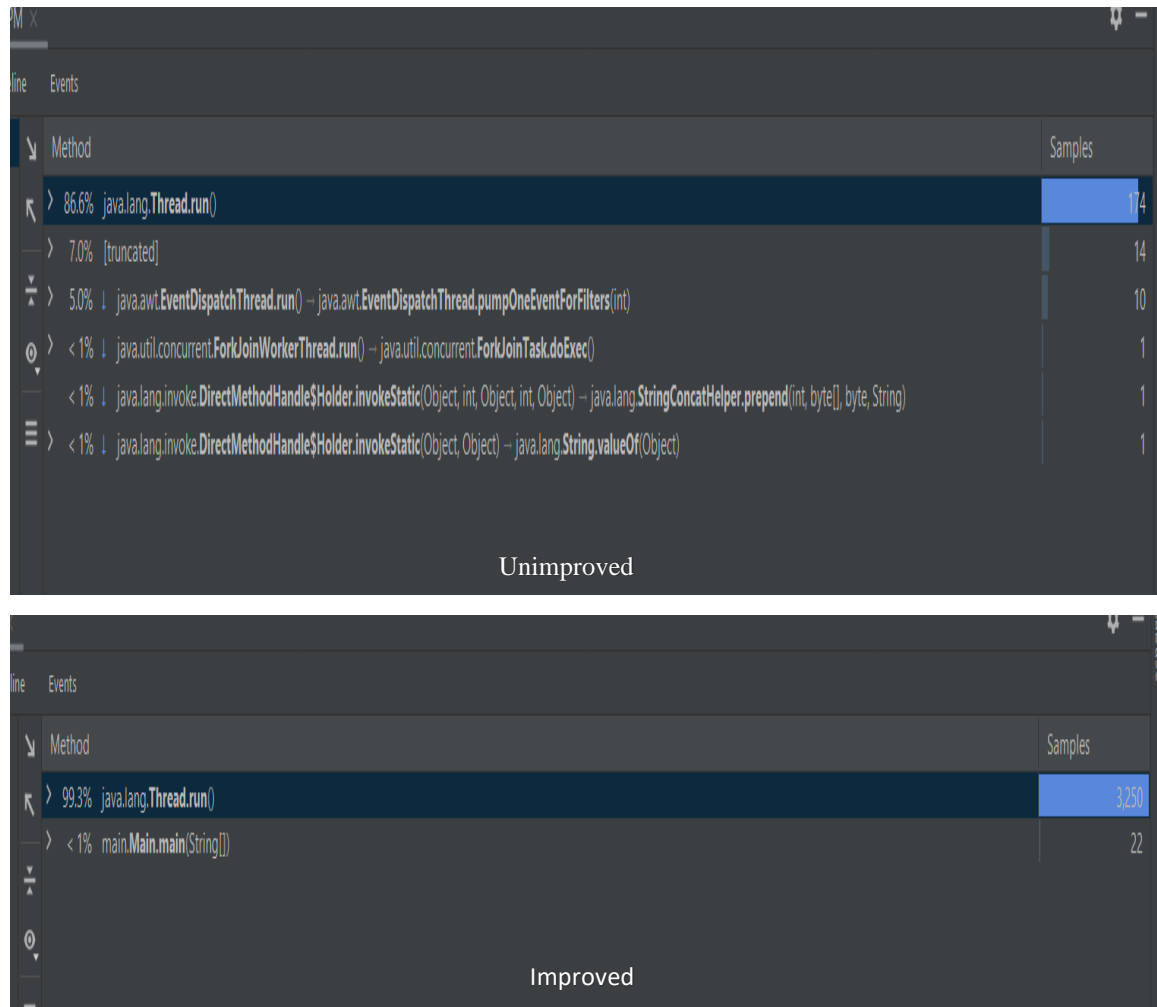


Figure 21: Call Tree

4.7.3 MethodList

The Method List collects all methods in the profiled data and sorts them by cumulative sample time. Every item in this list has several views.

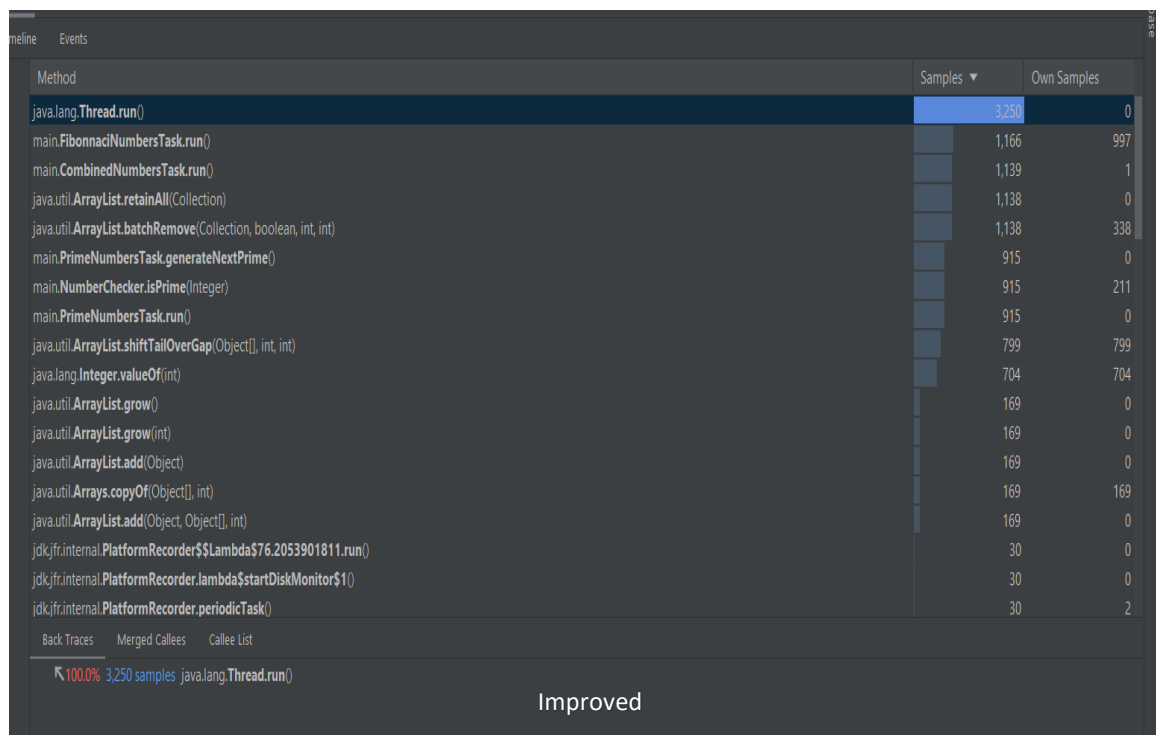
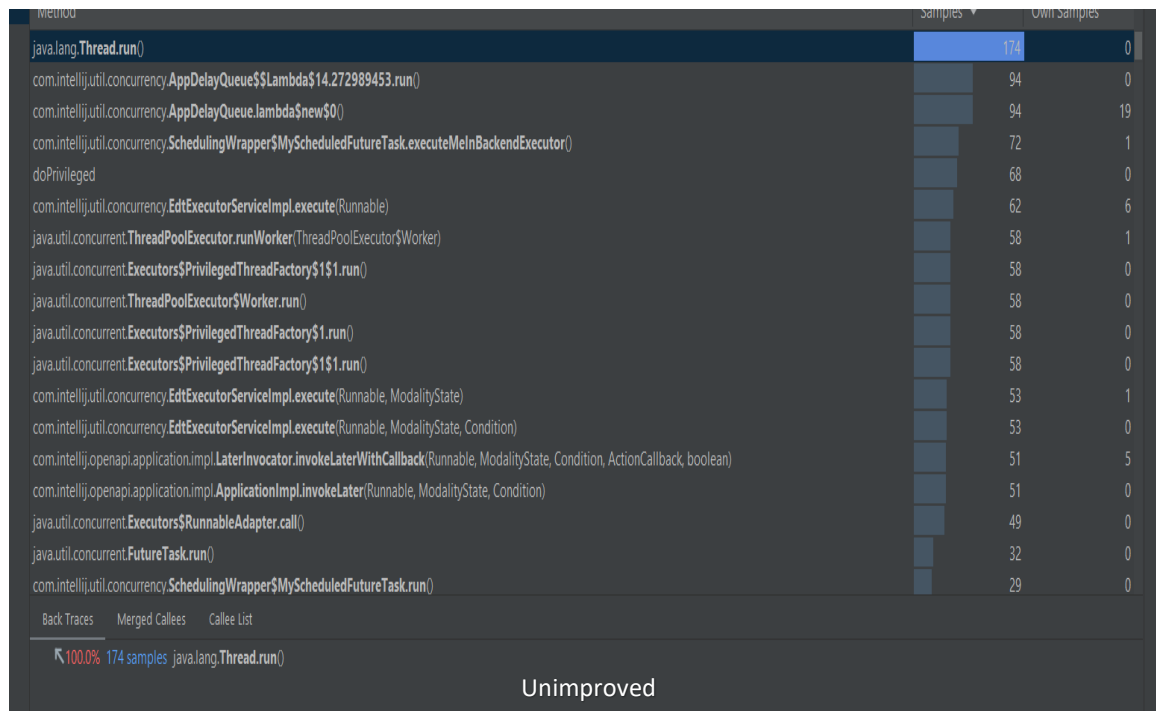


Figure 22: Method List (Unimproved)

4.7.4 CPU Timeline

The timeline helps you get full control over multi-threaded applications. It visualizes CPU consumption in your application as well in other running processes so that you can analyze the data in comparison.

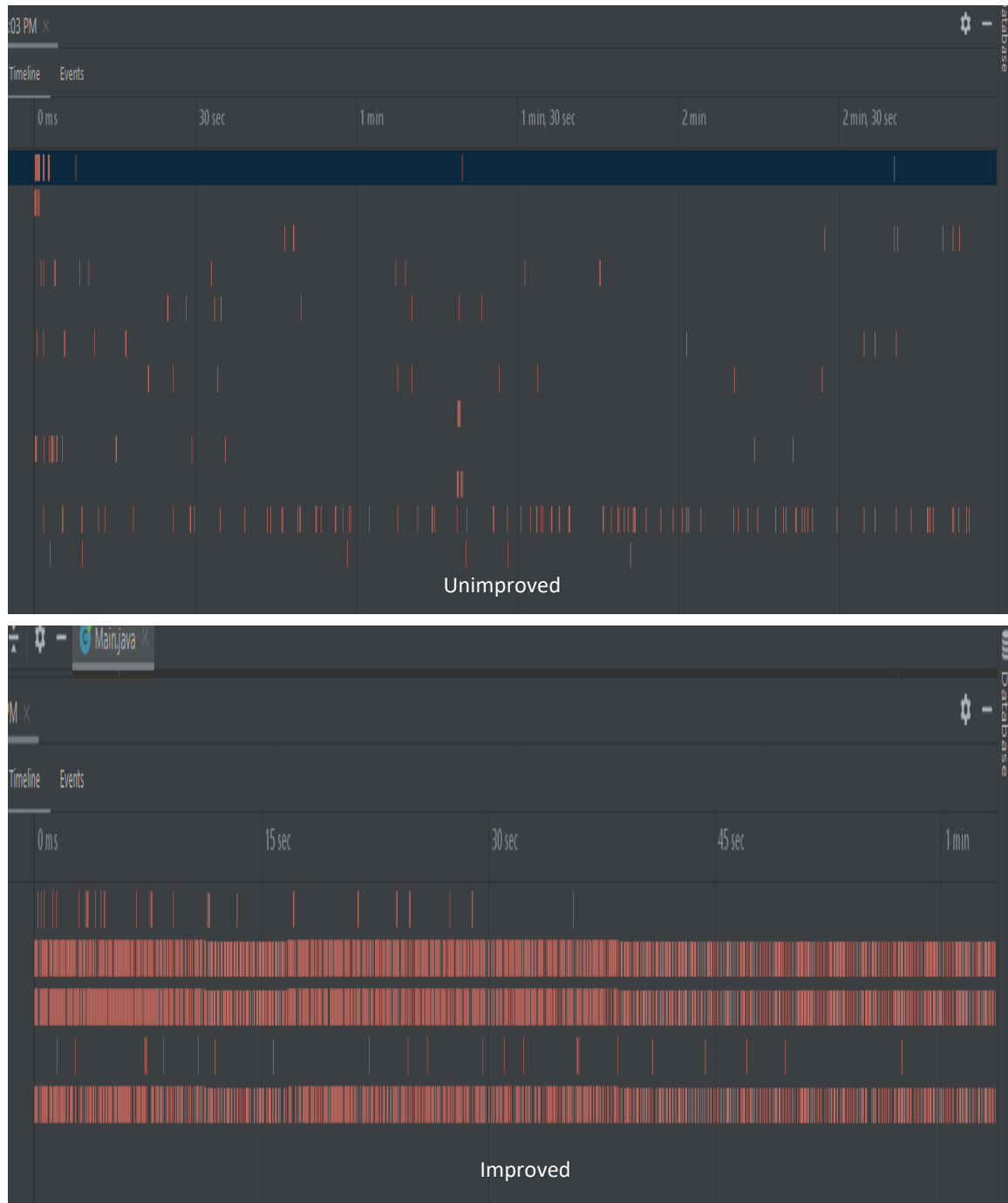


Figure 23: CPU Timeline

4.7.5 Java Flight Recorder Events

Java Flight Recorder collects data about events. Events occur in the JVM at a specific point in time. Each has a name, a timestamp, and an optional payload.

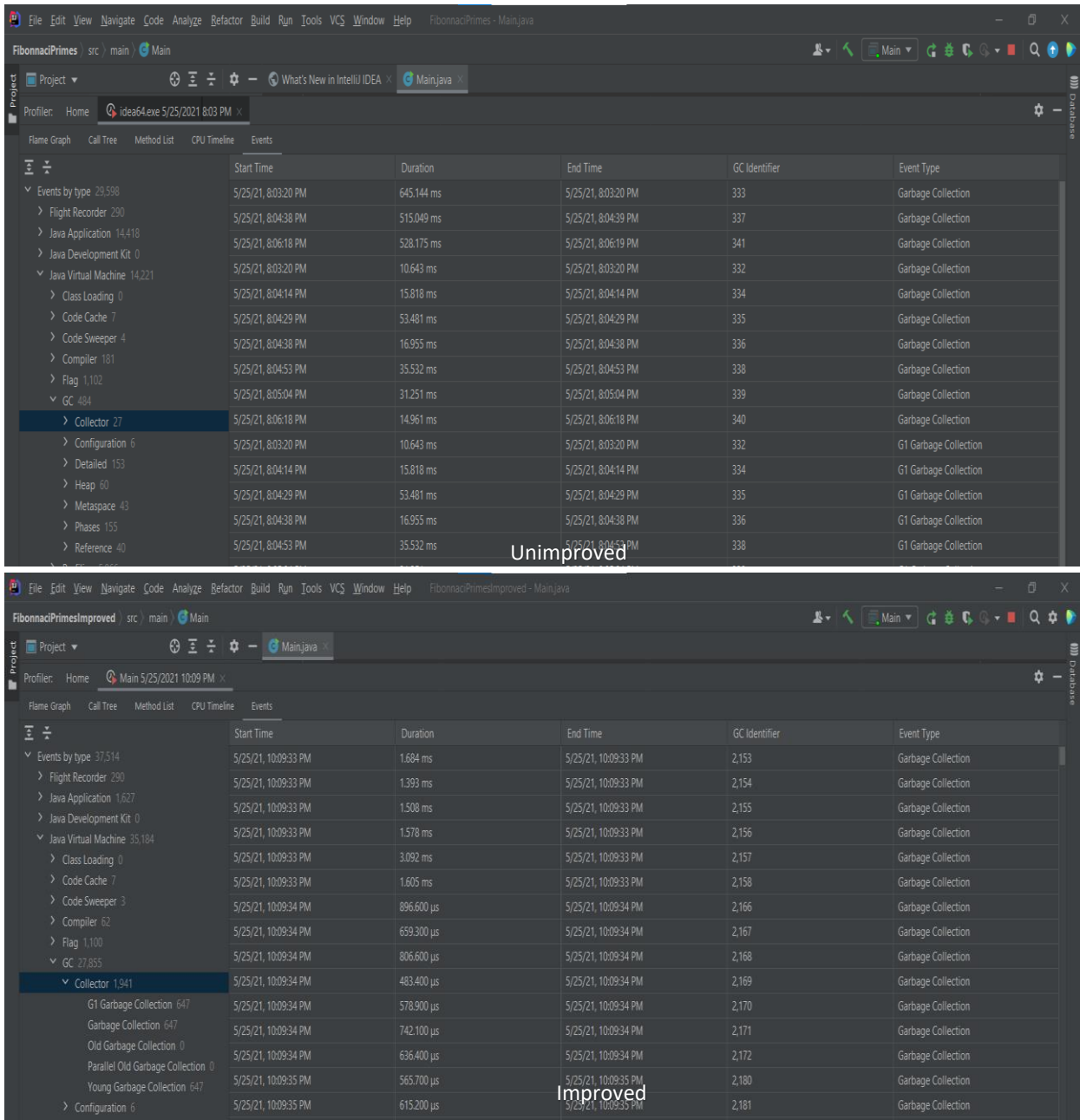


Figure 24: Garbage Collector Event

4.8 Better use of List in Java for better performance improvement

4.8.1 ArrayList

We know that as we add more or more items to an array list it is going to be list sizing itself over time. Because default value of the ArrayList is 10 and if there are more object than 10 then the new capacity of the array will be

$$\text{New Capacity} = \text{Old Capacity} + (\text{Old Capacity} / 2)$$

And that process of resizing could impact the performance of our application multiple times. We will have to allocate new space in memory for the new array and we will need to copy the data across and then the original object would be destroyed. So, if we have a list where we will be adding a very large number of objects this will lead to a lot of these resize operations and the time needed to do this resizing could be inefficient. Obviously, we will be creating objects that will be garbage collected the arrays that are discarded but this will create a less than optimal situation.

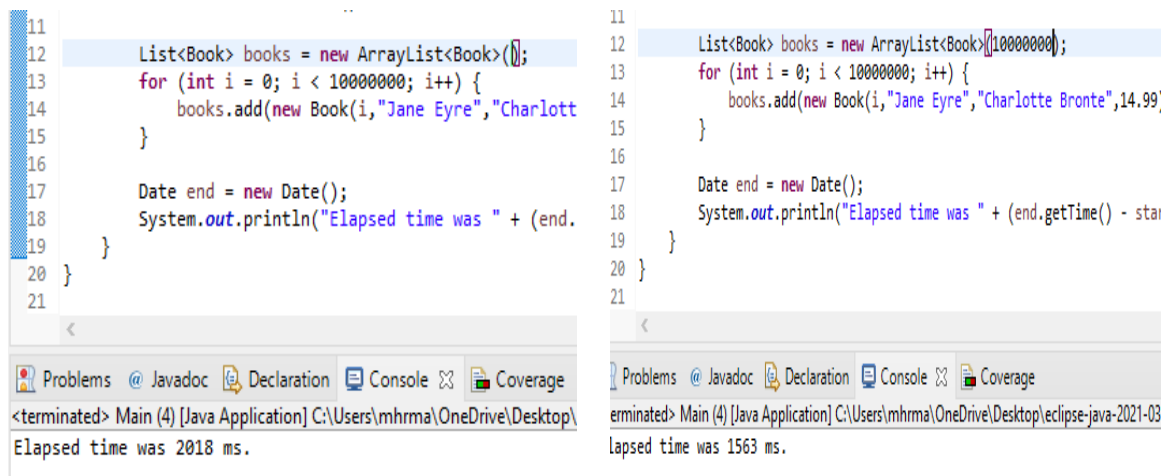


Figure 25: Execution time Default Array Size(2018ms) vs Fixed ArraySize(1563ms)

Without fixing ArrayList: Here we are creating an array list of books and adding 10 million books into this array list. And at this point we are using the empty constructor for the array lists. So, as we add more and more of these books that array this is going to need to be resized. It took 2018 milliseconds.

With Fixing ArrayList: While coding we know there is going to be 10 million. So, we want our initial array size to be 10 million and see that it only took 1563 milliseconds. Its $\frac{1}{4}$ of the time has been reduced. Because memory do not need to reallocate for this array multiple times, so it'll be faster. But this was not scientific. You could of course use something like GMH if you really wanted to. If we know it is going to be storing a lot of objects puts a number in here that we think will be big enough to store those objects that will certainly save us having to do a lot of resizing and it can impact the performance of our application as we have seen. Obviously, we do not want to put in too big a number here because that is going to be reserving

memory which may not be used and that could be wasteful. So, we must think carefully about the number to put in the constructor for your Array lists.

4.8.2 Using Vector instead of ArrayList

Certainly, if we ask which list should we use between Array and Vector list? Most programmers, they will simply say the vector is an old list type and we should absolutely be using the array list now instead. But there is an important difference between the vector and the array list and that is that the “Vector is thread safe”. So, if we are working in a multi-threaded application and we need a thread safe list then we might want to consider the copy on writer Array list. Well, here is an alternative. The vector is a thread safe array list. Now being thread safe comes at a performance cost. So, the Java docs advise us not to use it if we don’t need to be thread safe but of course if we do well it would absolutely be well worth testing.

4.8.3 Using LinkedList

What Happen when we want to add a new item Start of the list?

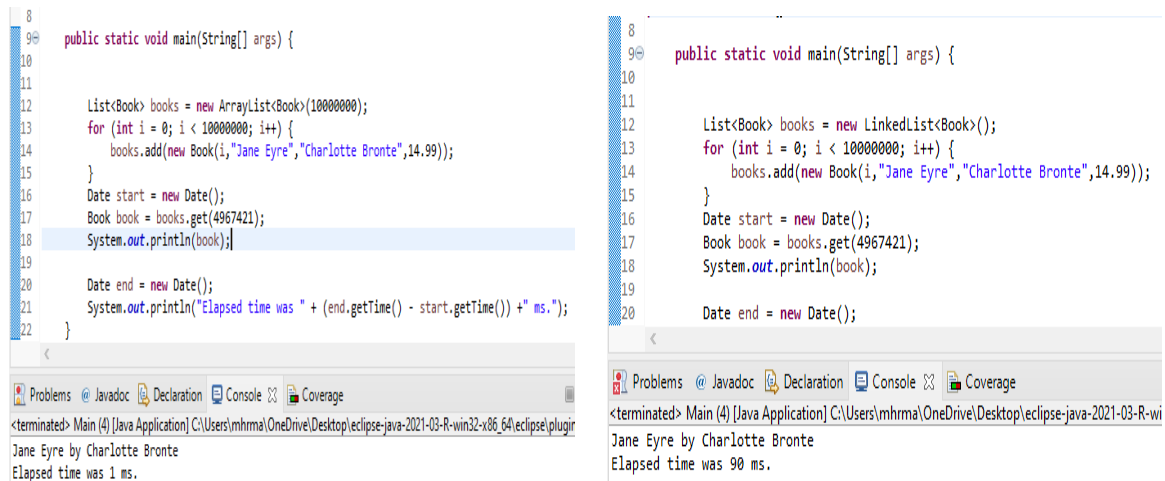
Adding an item to the start of a linked list will always also be quick. It is a simple operation we just need again to update a pointer. But adding an item to the beginning of an array list is a bit more complicated because all the existing items in the array this need to be shifted along one space in the array before the new item can be added at the start for a bigger way. That of course can be an expensive operation. So, to add items at the start of the list we will always get better performance using a linked list.

What Happen when we want to add a new item end of the list?

We know already that for the array list this will normally be quick because it is just simply putting a pointer into a vacant lot of the array although there is a potential performance impact if the list is full and needs to be re sized. When this happens, we know that the virtual machine needs to allocate a new array of memory it is to copy the data across from the old array and then at some point that’s older arraylist will be garbage collected and we have seen that there is absolutely a performance impact of this resizing when it comes to the linked list. In linked list there are no performance impact to add an item to the end of the list. We simply are adding a pointer so adding an item to the end of a linked list will always be quick. Java maintains which is the last item in the linked list so it can go straight to it. So, adding to the end of a list will always be optimal with a linked list and will normally be optimal with an array list unless of course there is a resizing operation to do so what about the start of the list.

Normally adding a item in starting point is faster in LinkedList and adding an item in last is also better in most case in LinkedList.

But the main drawback is this that when we remove an item from an array list, we can go straight to the item to move because we always know its position in the list. But with a linked list to find the item to remove will have to iterate through all the other items first. And this tends to be a slow operation. In an array list the underlying array object has the reference to every item in the list but in the linked list the references held in the object are to the first and the last notes only. And in fact this is probably the most important difference when it comes to the performance of an array list and a linked list.



The image shows two side-by-side screenshots of the Eclipse IDE, comparing the execution time of retrieving data from an ArrayList and a LinkedList. Both screenshots show a Java class with a `main` method that creates a list of 10,000,000 `Book` objects, each with the title "Jane Eyre" by "Charlotte Bronte" and a price of 14.99. The code then retrieves the book at index 4,967,421 and prints its details. The execution time is measured using `Date` and `System.out.println`.

Left Screenshot (ArrayList):

```
8 public static void main(String[] args) {
9
10
11
12 List<Book> books = new ArrayList<Book>(10000000);
13 for (int i = 0; i < 10000000; i++) {
14     books.add(new Book(i, "Jane Eyre", "Charlotte Bronte", 14.99));
15 }
16 Date start = new Date();
17 Book book = books.get(4967421);
18 System.out.println(book);
19
20 Date end = new Date();
21 System.out.println("Elapsed time was " + (end.getTime() - start.getTime()) + " ms.");
22 }
```

Console output:

```
<terminated> Main (4) [Java Application] C:\Users\mhrrma\OneDrive\Desktop\eclipse-java-2021-03-R-win32-x86_64\eclipse\plugin
Jane Eyre by Charlotte Bronte
Elapsed time was 1 ms.
```

Right Screenshot (LinkedList):

```
8 public static void main(String[] args) {
9
10
11
12 List<Book> books = new LinkedList<Book>();
13 for (int i = 0; i < 10000000; i++) {
14     books.add(new Book(i, "Jane Eyre", "Charlotte Bronte", 14.99));
15 }
16 Date start = new Date();
17 Book book = books.get(4967421);
18 System.out.println(book);
19
20 Date end = new Date();
```

Console output:

```
<terminated> Main (4) [Java Application] C:\Users\mhrrma\OneDrive\Desktop\eclipse-java-2021-03-R-win32-x86_64\eclipse\plugin
Jane Eyre by Charlotte Bronte
Elapsed time was 90 ms.
```

Figure 26: Execution time Retrieving data of Array(1ms) vs LinkedList(90ms)

The array list will normally give better performance overall, but we want to retrieve a particular item knowing its position in the list and a LinkedList will generally give better performance when we want to be adding items. Particularly if we are adding somewhere in the middle or near the start of the list and ask for removing items. Removing the first item in the list we will certainly get better performance from a linked list. But removing an item in the middle we might get better performance from an array list because we do not have to do the navigation to find that item first. So, while the array list is normally the list of choice there are plenty of times when a linked list will give better performance.

4.9 Better Coding Choice

4.9.1 Primitives Vs Object

If we will add up a lot of numbers, does it make a difference performance if those numbers are primitives or objects. So, what we are doing in this experiment is we are going to add 1 million longs together and see how long it takes to add those numbers. If we look at the two different implementations so these two versions of code are identical but for the fact that in the first version, we are using primitive longs in the second version we are using object longs.

```
14 public void run() {
15
16     System.out.println("warm up period starting");
17     addNumbers1(10001);
18
19     System.out.println("warm up period done");
20     try {
21         Thread.sleep(1000);
22     } catch (InterruptedException e) {
23     }
24     System.out.println("measurement period starting");
25
26     long start = System.currentTimeMillis();
27     addNumbers1(10000001);
28     long end = System.currentTimeMillis();
29     System.out.println("measurement period done");
30     System.out.println("time taken to add 1,000,000 longs: " + (end - start));
31 }
32 }
```

```
22 public void run() {
23
24     System.out.println("warm up period starting");
25     addNumbers2(10001);
26
27     System.out.println("warm up period done");
28     try {
29         Thread.sleep(1000);
30     } catch (InterruptedException e) {
31     }
32     System.out.println("measurement period starting");
33
34     long start = System.currentTimeMillis();
35     addNumbers2(10000001);
36     long end = System.currentTimeMillis();
37     System.out.println("measurement period done");
38     System.out.println("time taken to add 1,000,000 longs: " + (end - start));
39 }
40 }
```

Problems @ Javadoc Declaration Console Coverage

<terminated> Main (5) [Java Application] C:\Users\mhrma\OneDrive\Desktop\eclipse-java-2021-0

warm up period done
measurement period starting
measurement period done
time taken to add 1,000,000 longs: 7 milliseconds

Problems @ Javadoc Declaration Console Coverage

<terminated> Main (5) [Java Application] C:\Users\mhrma\OneDrive\Desktop\eclipse-java-2021-0

warm up period starting
warm up period done
measurement period starting
measurement period done
time taken to add 1,000,000 longs: 42 milliseconds

Figure 27: Execution time of Primitive(7ms) vs Object(42ms)

Finally, we can say if we use primitives then we are going to get better performance generally than if we use that object counterparts.

4.9.2 Double Vs Big Decimal

Big decimal might well be slower than the double because we have the overhead of the precision part of the work that's extra processing which would not be needed with a double. So, knowing nothing else about the implementation of the big decimal class we would expect it to perform more poorly than with doubles.

The image shows two side-by-side screenshots of the Eclipse IDE, comparing the execution time of Double and BigDecimal operations. Both screenshots show a Java class with a `run()` method. The left screenshot is for Double, and the right is for BigDecimal. Both methods perform a warm-up period (printing 'warm up period starting' and 'warm up period done', sleeping 1000ms, and printing 'measurement period starting') followed by a measurement period (printing 'measurement period done' and 'time taken to add 1,000,000 numbers:'). The Double version uses `addNumbers2(10000001)` and reports 38 milliseconds. The BigDecimal version uses `addNumbers1(10000001)` and reports 113 milliseconds. The console output at the bottom of each window confirms these results.

```
23
24 public void run() {
25
26     System.out.println("warm up period starting");
27     addNumbers2(10001);
28
29     System.out.println("warm up period done");
30     try {
31         Thread.sleep(1000);
32     } catch (InterruptedException e) {
33     }
34     System.out.println("measurement period starting");
35
36     long start = System.currentTimeMillis();
37     addNumbers2(10000001);
38     long end = System.currentTimeMillis();
39     System.out.println("measurement period done");
40     System.out.println("time taken to add 1,000,000 numbers:");
41 }
42
```

Problems @ Javadoc Declaration Console Coverage
<terminated> Main (5) [Java Application] C:\Users\mhurma\OneDrive\Desktop\eclipse-java-
warm up period done
measurement period starting
measurement period done
time taken to add 1,000,000 numbers: 38 milliseconds

```
24
25 public void run() {
26
27     System.out.println("warm up period starting");
28     addNumbers1(10001);
29
30     System.out.println("warm up period done");
31     try {
32         Thread.sleep(1000);
33     } catch (InterruptedException e) {
34     }
35     System.out.println("measurement period starting");
36
37     long start = System.currentTimeMillis();
38     addNumbers1(10000001);
39     long end = System.currentTimeMillis();
40     System.out.println("measurement period done");
41     System.out.println("time taken to add 1,000,000 num");
42 }
43
```

Problems @ Javadoc Declaration Console Coverage
<terminated> Main (5) [Java Application] C:\Users\mhurma\OneDrive\Desktop\eclipt
warm up period done
measurement period starting
measurement period done
time taken to add 1,000,000 numbers: 113 milliseconds

Figure 28: Execution time of Double(38ms) vs Big Decimal(113ms)

The result is as we expected earlier. Big Decimal is slower than Double. Because the additional requirements of having the precision part of a big decimal does make it perform worse than a double.

4.9.3 String Concatenation vs String Builder

One is by concatenating strings, and one is by using a string builder and its append method. So, let us understand how these two different versions perform and what we have got here is a process to generate 500,000 random names.

The first version is using version one of us generate names that using the string concatenation.

The figure displays two side-by-side screenshots of the Eclipse IDE, comparing the execution time of two Java programs. Both programs generate 500,000 random names and measure the time taken.

Left Screenshot (String Concatenation):

```
29
30 public void run() {
31
32     System.out.println("warm up period starting");
33     for (int i=1; i < 500000; i++)
34         generateNames1();
35
36     System.out.println("warm up period done");
37     try {
38         Thread.sleep(1000);
39     } catch (InterruptedException e) {}
40
41     System.out.println("measurement period starting");
42
43     long start = System.currentTimeMillis();
44     for (int i=1; i < 500000; i++)
45         generateNames1();
46     long end = System.currentTimeMillis();
47     System.out.println("measurement period done");
48     System.out.println("time taken to generate 500,000 names: ");
49 }
50
51
```

Console Output:

```
<terminated> Main (5) [Java Application] C:\Users\mhrma\OneDrive\Desktop\eclipse-java-
warm up period done
measurement period starting
measurement period done
time taken to generate 500,000 names: 441 milliseconds
```

Right Screenshot (String Builder):

```
44
45 public void run() {
46
47     System.out.println("warm up period starting");
48     for (int i=1; i < 500000; i++)
49         generateNames2();
50
51     System.out.println("warm up period done");
52     try {
53         Thread.sleep(1000);
54     } catch (InterruptedException e) {}
55
56     System.out.println("measurement period starting");
57
58     long start = System.currentTimeMillis();
59     for (int i=1; i < 500000; i++)
60         generateNames2();
61     long end = System.currentTimeMillis();
62     System.out.println("measurement period done");
63     System.out.println("time taken to generate 500,000 names: ");
64 }
65
66
```

Console Output:

```
<terminated> Main (5) [Java Application] C:\Users\mhrma\OneDrive\Desktop\eclipse-java-2021-
warm up period done
measurement period starting
measurement period done
time taken to generate 500,000 names: 189 milliseconds
```

Figure 29: Execution time of String Concatenation(441ms) vs String Builder(189ms)

Here we can see the String Builder need less execution time. String Concatenation need 57.14% less execution time than String Builder.

4.9.4 Loops vs Stream vs Parallel Stream

The figure consists of three screenshots of the Eclipse IDE, each showing a different Java implementation for processing a list of 5,000,000 strings. Each screenshot includes the source code and the console output.

Top Left Screenshot (Traditional Loop): The code uses a standard `for` loop to iterate through the list and calculate the length of each string. The console output shows a measurement period starting at 86254883, ending at 86254883, and taking 56 milliseconds to complete.

```
public void run() {
    List<String> names = new ArrayList<String>();
    for (int i=1; i < 5000000; i++)
        names.add(generateName());

    System.out.println("warm up period starting");
    calculateLength1(names);

    System.out.println("warm up period done");
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {}

    System.out.println("measurement period starting");
    long start = System.currentTimeMillis();
    calculateLength1(names);
    long end = System.currentTimeMillis();
    System.out.println("measurement period done");
    System.out.println("time taken to loop through 5,000,000 : ");
}
```

Top Right Screenshot (Stream): The code uses a stream to process the list. The console output shows a measurement period starting at 86254741, ending at 86254741, and taking 41 milliseconds to complete.

```
public void run() {
    List<String> names = new ArrayList<String>();
    for (int i=1; i < 5000000; i++)
        names.add(generateName());

    System.out.println("warm up period starting");
    calculateLength2(names);

    System.out.println("warm up period done");
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {}

    System.out.println("measurement period starting");
    long start = System.currentTimeMillis();
    calculateLength2(names);
    long end = System.currentTimeMillis();
    System.out.println("measurement period done");
    System.out.println("time taken to loop through 5,000,000 : ");
}
```

Bottom Screenshot (Parallel Stream): The code uses a parallel stream to process the list. The console output shows a measurement period starting at 86253775, ending at 86253775, and taking 27 milliseconds to complete.

```
public void run() {
    List<String> names = new ArrayList<String>();
    for (int i=1; i < 5000000; i++)
        names.add(generateName());

    System.out.println("warm up period starting");
    calculateLength3(names);

    System.out.println("warm up period done");
    try {
        Thread.sleep(1000);
    } catch (InterruptedException e) {}

    System.out.println("measurement period starting");
    long start = System.currentTimeMillis();
    calculateLength3(names);
    long end = System.currentTimeMillis();
    System.out.println("measurement period done");
    System.out.println("time taken to loop through 5,000,000 strings: " + (end - start));
}
```

Figure 30: Execution time of Traditional Loop (56) vs Stream (41) vs Parallel Stream (27)

The traditional Java loops takes 56 milliseconds, Stream took 41 milliseconds and finally Parallel Stream took only 27 milliseconds. By far the Parallel Stream is the most optimized scenario for our use case. Parallel streams will always give us better performance when we can use them assuming of our available CPU resources to process the multiple threads.

5 5 RESULTS & DISSCUSSION

Here in this research, we have done lots of implementation to see performance advantages or better optimization. Now we will represent the differences more proper way so anyone can understand the impact of our research.

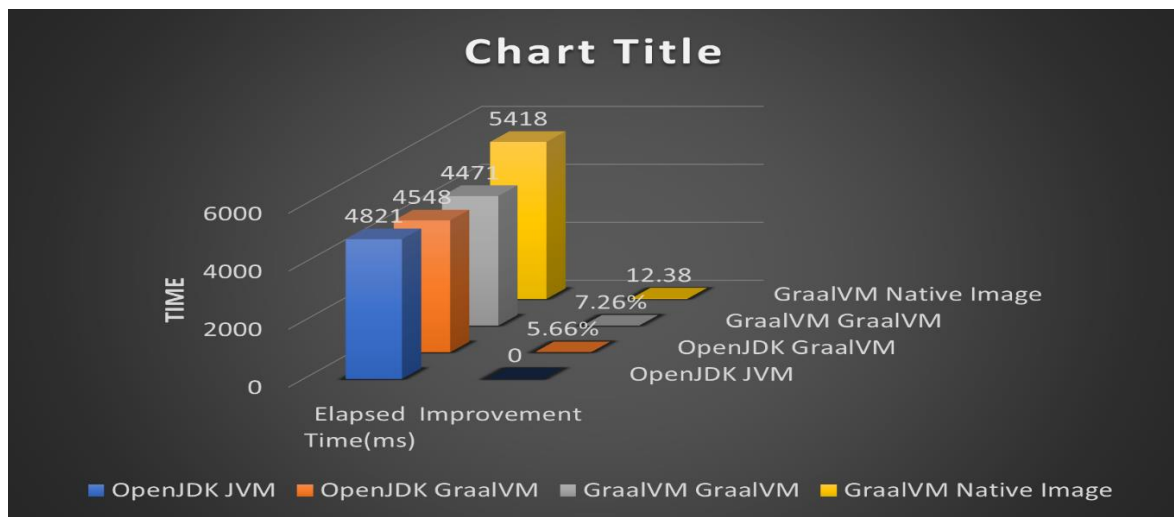
Now we will see the performance boost in the execution time we got from our research part.

5.1 Performance Improvement of GraalVM vs JVM

We can run some basic code of counting prime numbers. We implemented same Java code in both JVM and GraalVM of generating UUID.

Compiler	Virtual Machine	Elapsed Time(ms)	Improvement
OpenJDK	JVM	4821	Default
OpenJDK	GraalVM	4548	5.66%
GraalVM	GraalVM	4471	7.26%
GraalVM	Native Image	5418	12.38

Table: GraalVM vs JVM



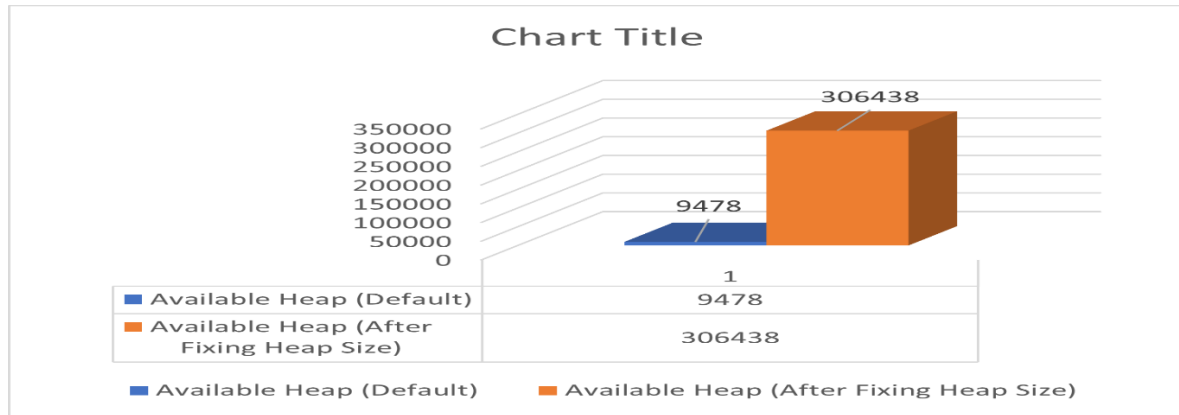
Graph: GraalVM vs JVM

Result is surprising that without making any change in code, we got 12.27% improvement in a basic task. So, it is quite great in term of performance improvements. Execution time may be not the most scientific measurement, but it can summery overall performance enhancement.

5.2 Controlling Heap Size (Section 4.1.1)

Test Name	Argument	Available Heap (Default)	Available Heap (After Fixing Heap Size)	Improvement in Percentage
Available Heap Size	-Xms300m	9478kb	306438kb	32.33 Times

Table: Available heap size comparison



Graph: Heap size

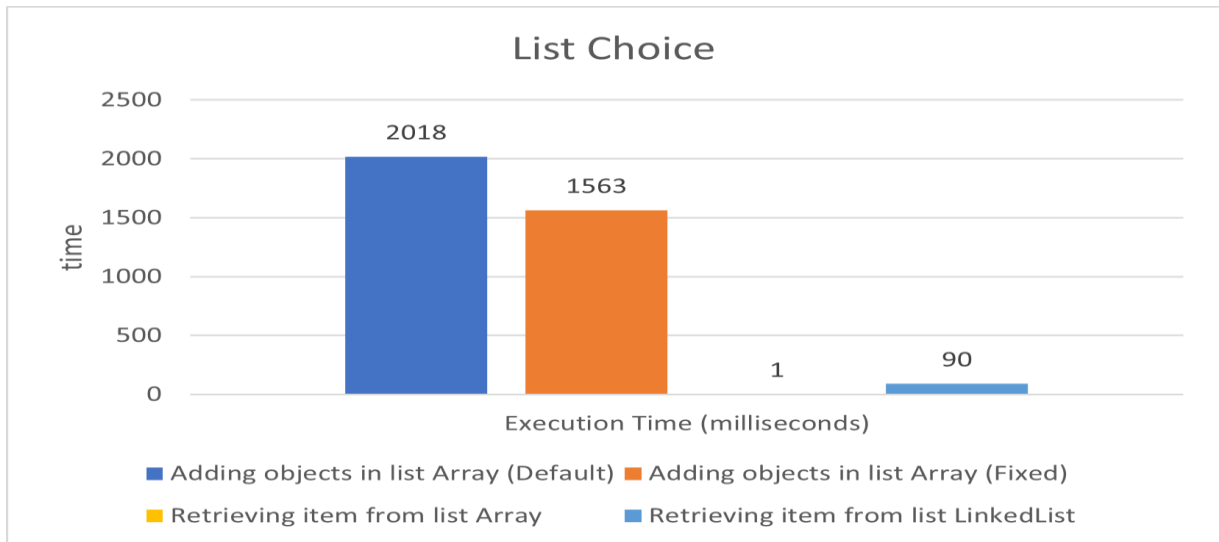
Here we can see after fixing the heap size, we get more available heap size then default.

5.3 Better List Choice (Section 3.7)

If we choose appropriate list type, then we can get a better performance from same code with different. Here we will see some comparison table.

Operation	List Name	Execution Time (milliseconds)	Execution Time (After Improvement)
Adding objects In list (4.7.1)	Array (Default)	2018 ms	22.54%
	Array (Fixed)	1563 ms	
Retrieving item from list (4.7.2)	Array	1 ms	98.89%
	LinkedList	90 ms	

Table: List comparison results



Graph: List comparison graph

Here we can see some difference how different code choice can affect our project. Normally fixing array is a better idea then keeping it default. In retrieving using Array is better than LinkedList.

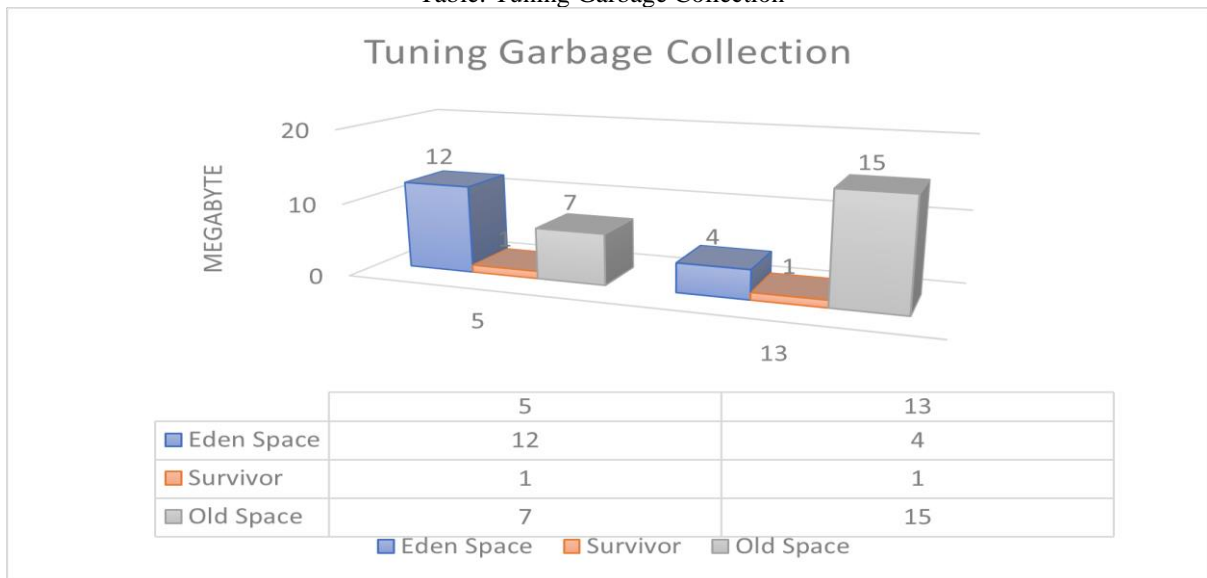
5.4 Tuning Garbage Collection (Section 3.2):

We can tune things show Garbage Collector will work and we can change some default value like young and old generation allocation space, what will be the ration of S0, S1 and Eden space, how many times it will be change between S0 and S1 survivor space.

Argument	Value	Eden Space	Survivor	Old Space
Default				
-XX:NewRatio	= 2			
-XX:SurvivorRatio	= 8	12MB	1MB	7MB
XX:MaxTenuring Thershold	= 15			

Improved/Tuned -XX: NewRatio	= 4			
-XX:SurvivorRatio	= 5	4MB	1MB	15MB
XX:MaxTenuring Thershold	= 10			
Difference		Eden Space Reduced 8MB 300% ↓	Survivor Space Same	Old Space Increased 8MB 214% ↑

Table: Tuning Garbage Collection



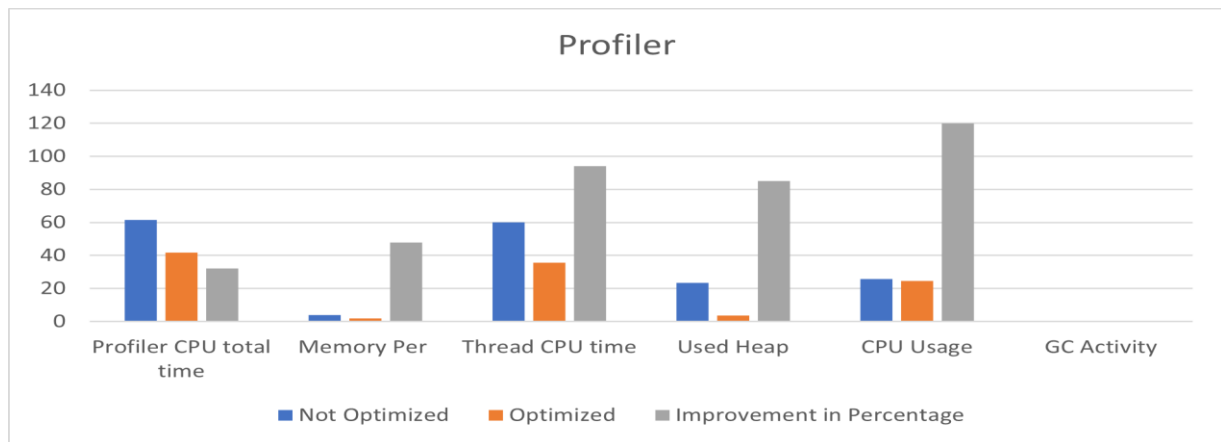
Graph: Tuning Garbage Collection

5.5 Profiler Results (Section 3.5.1)

We have tested a project where it will calculate all Fibonacci prime number. But we have taken a project that is not optimized then we profiled those data on Profiler and improved the code. Then we can compare the results. We have run both project for approximately 60 second and got those results.

Test Name	Not Optimized (1804)	Optimized (4388)	Improvement in Percentage
Profiler CPU total time	61389 ms	41590 ms	32.25%
Memory Per Thread Allocations	3.77 Gb/sec	1.97Gb/sec	47.74%
Thread CPU time	600973 ms	35700 ms	94.06
Used Heap	234.44 Mb	35 Mb	85%
CPU Usage	25.8%	24.6%	1.2%
GC Activity	0.1%	0.0%	-

Table: Profiler results



Graph: Profiler results

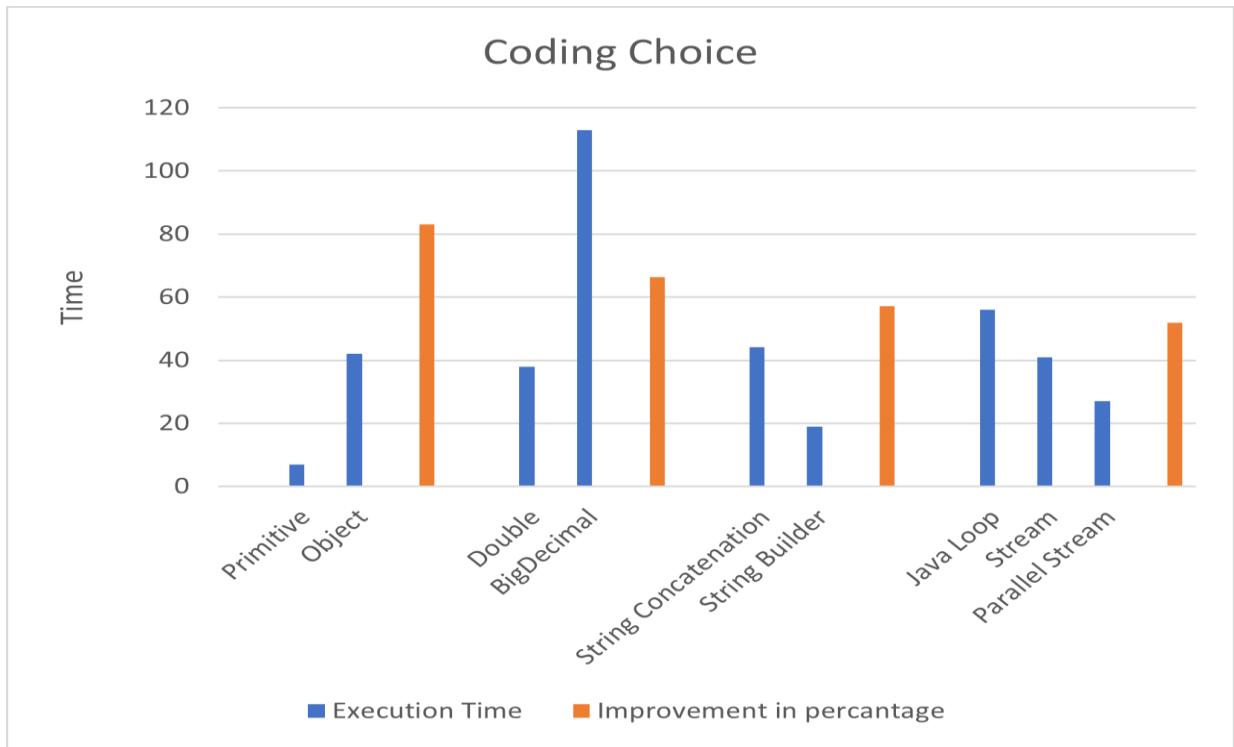
So, after running both project in Profiler for 60 seconds we found lots of difference. From the data of the Profiler of the unoptimized version we can find what is going wrong with our project and easily we can detect and fix those problems. In various aspect we can see the improvement after optimizing the code.

5.6 Better Coding Choice (Section 3.8)

We can improve our memory management by choosing the appropriate type of variables. It helps a lot and take less time to execute

Operation	Name	Execution Time (milliseconds)	Execution Time (After Improvement)
Adding big list of numbers (4.8.1)	Primitive	7	83%
	Object	42	
Adding big list of numbers (4.8.1)	Double	38	66.37%
	BigDecimal	113	
Generate 50000 names (4.8.3)	String Concatenation	441	57.14%
	String Builder	189	
Generate 50000 names (4.8.4)	Java Loop	56	51.78%
	Stream	41	
	Parallel Stream	27	

Table: Better Code choice comparison



Graph: Coding Choice Comparison

May be execution time is not the finest way of analyzing performance improvement, but it shows the overall scenario. Results numbers may be varying on different circumstance of project and can be change different real-life project when we will implement those techniques.

6 CONCLUSIONS AND RECOMMENDATIONS

6.1 Findings and Conclusion

Our goal was to find better memory management for the android operating system. As garbage collection is the current solution for android memory management, we tried to see how optimally we can code. We tried to find some way that Garbage Collector does not need to run frequently. Then we tuned some Java property to control heap and memory leak. After that, we proposed to use some arguments that can help to produce a more optimized application. We compared the execution time to see which process works fast and optimally. Also, we run an unoptimized project and saw its performance in various tools like Profiler, Flight Box, and Memory Analyzer to find the problem in those projects. After finding out the issues, we made some changes in the project, compare the improved project data with the unimproved project. Lastly, we proposed another GraalVM instead of JVM because memory management in GraalVM is much more optimized than JVM. Also, it has some latest garbage collection techniques that lead us to the excellent performance of our existing process. We are delighted with our implementation. We showed the results that our proposed strategies are working better. It will be beneficial for Java programmers to code following those strategies to find more optimized memory management.

6.2 Limitations

Java is not a modern language, so it does not have any modern memory management techniques like Swift or other programming languages. Android is so big that it is pretty impossible moving into a better programming language with better memory management by default. GraalVM can indeed handle memory management in a more optimized way. Also, changing from JVM to GraalVM is not as simple as it sounds. Personally, due to Covid 19 pandemic situation, I missed valuable instruction from our excellent faculties who are working for many years and expert in this field.

6.3 Future Works

Using GraalVM is an excellent approach to handle memory efficiently and should be adopted by Google for Android. It's true that java has become quite an old language and is not keeping up with the latest trends and features of modern development languages. Java was the best choice of language when Android OS was launched, but it looks obsolete after all these years. A virtual machine needs to adopt advanced features that help better code structure and contain micro-optimizations within itself. Some modern languages now include built-in memory optimization tips reported by the compiler at run time, which can significantly optimize the code written and use less memory. Also, we can work on some more things like

6.3.1 Automatic Reference Object

Now, java needs a marking process to observe the object that is not used by any variable. So, we should work more on the object and find a solution that object can mark itself in java, so the machine should not do the marking process. So, if the object can automatically mark itself and count whenever it gets a call from a variable and destroy it when its reference is zero, then the JVM needs no memory for the Mark and sweep process.

6.3.2 GraalVM implementation

GraalVM is a new approach, so it is not an industry-ready solution yet. So, we don't see it being used in the industry very often. We plan to do a real-life java project in JVM and then transfer it to GraalVM to compare the improvement.

6.4 Conclusion

In this research, we try to show every possible way to improve memory management in garbage collection. We find out the way then implemented the strategy in different types of projects. We found our approach performing well when we compared the outcome of every result. Those strategies will help programmers to be code more optimally without having the tension of memory management. By following those approaches from the beginning of the project, they can have better performing optimized applications.

7 Reference

- [1] Kashif Tasneem, Ayesha Siddiqui, Anum Liaquat International Journal of Computer Applications (0975 – 8887) Volume 182 – No. 41.
- [2] Aponso, G. C. A. L. (2017). Effective Memory Management for Mobile operating Systems. American Journal of Engineering Research (AJER), 246.
- [3] Linares-Vásquez, M., Vendome, C., Tufano, M., & Poshyvanyk, D. (2017). How developers micro-optimize Android apps. Journal of Systems and Software, 130, 1-23.
- [4] Yovine, S., & Winniczuk, G. (2017, May). CheckDroid: a tool for automated detection of bad practices in Android applications using taint analysis. In Proceedings of the 4th International Conference on Mobile Software Engineering and Systems (pp. 175-176). IEEE Press.
- [5] Qian, J., & Zhou, D. (2016). Prioritizing Test Cases for Memory Leaks in Android Applications. Journal of Computer Science and Technology, 31(5), 869-882.
- [6] Kwon, S., Kim, S. H., Kim, J. S., & Jeong, J. (2015, October). Managing GPU buffers for caching more apps in mobile systems. In Proceedings of the 12th International Conference on Embedded Software (pp. 207-216). IEEE Press.
- [7] Lee, B., Kim, S. M., Park, E., & Han, D. (2015, July). Memscope: Analyzing memory duplication on android systems. In Proceedings of the 6th Asia-Pacific Workshop on Systems (p. 19). ACM.
- [8] Kim, S. H., Jeong, J., & Lee, J. (2014). Selective memory deduplication for cost efficiency in mobile smart devices. IEEE Transactions on Consumer Electronics, 60(2), 276-284.
- [9] Gerlitz, T., Kalkov, I., Schommer, J. F., Franke, D., & Kowalewski, S. (2013, October). Non-blocking garbage collection for real-time android. In Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems (pp. 108-117). ACM.
- [10] Lim, G., Min, C., & Eom, Y. I. (2013, January). Enhancing application performance by memory partitioning in Android platforms. In Consumer Electronics (ICCE), 2013 IEEE International Conference on (pp. 649-650). IEEE.
- [11] MA, J., LIU, S., YUE, S., TAO, X., & LU, J. LeakDAF: An Automated Tool for Detecting Leaked Activities and Fragments of Android Applications.
- [12] Zhang, H., Wu, H., & Rountev, A. (2016, May). Automated test generation for detection of leaks in Android applications. In Automation of Software Test (AST), 2016 IEEE/ACM 11th International Workshop in (pp. 64-70). IEEE.

- [13] Shahriar, H., North, S., & Mawangi, E. (2014, January). Testing of memory leak in Android applications. In High-Assurance Systems Engineering (HASE), 2014 IEEE 15th
- [14] Vimal, K., & Trivedi, A. (2015, December). A memory management scheme for enhancing performance of applications on Android. In Intelligent Computational Systems (RAICS), 2015 IEEE Recent Advances in (pp. 162-166).
- [15] Baik, K., & Huh, J. (2014, June). Balanced memory management for smartphones based on adaptive background app management. In Consumer Electronics (ISCE 2014),
- [16] Zhou, Bowen, and Rajkumar Buyya. "Augmentation Techniques for Mobile Cloud Computing: A Taxonomy, Survey, and Future Directions." *ACM Computing Surveys (CSUR)* 51.1 (2018): 13.
- [17] Erol-Kantarci, M., & Sukhmani, S. (2018). Caching and Computing at the Edge for Mobile Augmented Reality and Virtual Reality (AR/VR) in 5G. In *Ad Hoc Networks* (pp. 169-177). Springer, Cham.
- [18] Guerrero-Contreras, G., Garrido, J. L., Balderas-Diaz, S., & Rodríguez-Domínguez, C. (2017). A context-aware architecture supporting service availability in mobile cloud computing. *IEEE Transactions on Services Computing*, 10(6), 956-968.
- [19] Bahmani, K., Argyriou, A., Erol-Kantarci, M.: Backhaul relaxation through caching. In: Imran, M., Raza, S.A., Shakir, M.Z. (eds.) Access, Fronthaul and Backhaul for 5G Wireless Networks. IET (2017)
- [20] Tang, L., He, S., & Li, Q. (2017). Double-sided bidding mechanism for resource sharing in mobile cloud. *IEEE Transactions on Vehicular Technology*, 66(2), 1798-1809.
- [21] Chen, X., Jiao, L., Li, W., & Fu, X. (2016). Efficient multi-user computation offloading for mobile-edge cloud computing. *IEEE/ACM Transactions on Networking*, 24(5), 2795-2808.
- [22] Ali, F. A., Simoens, P., Verbelen, T., Demeester, P., & Dhoedt, B. (2016). Mobile device power models for energy efficient dynamic offloading at runtime. *Journal of Systems and Software*, 113, 173-187.
- [23] Mao, Y., Zhang, J., & Letaief, K. B. (2016). Dynamic computation offloading for mobile-edge computing with energy harvesting devices. *IEEE Journal on Selected Areas in Communications*, 34(12), 3590-3605.
- [24] Yang, L., Cao, J., Liang, G., & Han, X. (2016). Cost aware service placement and load dispatching in mobile cloud systems. *IEEE Transactions on Computers*, 65(5), 1440-1452.

- [25] T. Verbelen, P. Simoens, F. D. Turck, B. Dhoedt, Aiolos: Middleware for improving mobile application performance through cyber foraging, *Journal of Systems and Software* 85 (11) (2012) 2629-2639.
- [26] E. Miluzzo, R. Caceres, and Y. Chen. Vision: mclouds-computing on clouds of mobile devices. In *Proceedings of ACM Workshop on Mobile Cloud Computing and Services*, 2012.
- [27] F. Bonomi, R. Milito, J. Zhu, S. Addepalli. Fog computing and its role in the Internet of Things. In *Proc. ACM MCC*, pp.13-16, August 2012, Helsinki, Finland.
- [28] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, A. Patti, Clonecloud: elastic execution between mobile device and cloud, in: *Proceedings of the sixth conference on Computer systems*, EuroSys '11, ACM, New York, NY, USA, 2011, pp. 301-314.
- [29] S. Sudevalayam and P. Kulkarni, "Energy harvesting sensor nodes: Survey and implications," *IEEE Commun. Surveys Tuts.*, vol. 13, no. 3, pp. 443-461, Jul. 2011.
- [30] L. Zhang, B. Tiwana, R. Dick, Z. Qian, Z. Mao, Z. Wang, L. Yang, Accurate online power estimation and automatic battery behavior-based power model generation for smartphones, in: *Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2010 IEEE/ACM/IFIP International Conference on, 2010, pp. 105-114.
- [31] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, P. Bahl, Maui: making smartphones last longer with code offload, in: *Proceedings of the 8th international conference on Mobile systems, applications, and services*, MobiSys '10, ACM, New York, NY, USA, 2010, pp. 49-62.
- [32] A. Dou, V. Kalogeraki, D. Gunopulos, T. Mielikainen, V. H. Tuulos, Misco: A mapreduce framework for mobile systems, in: *Proceedings of the 3rd International Conference on Pervasive Technologies Related to Assistive Environments*, PETRA '10, ACM, New York, NY, USA, 2010, pp. 32:1-32:8.
- [33] A. Shye, B. Scholbrock, G. Memik, Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures, in: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, ACM, New York, NY, USA, 2009, pp. 168-178.
- [34] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for VM-based cloudlets in mobile computing. In *IEEE Pervasive Computing*, vol.8, no.4, pp.14-23, October 2009.

8 Appendix 1

IDE: Integrated Development Environment

JVM: Java Hotspot Virtual Machine

GC: Garbage Collection

ARC: Automatic Reference Counting

OS: Operating System

VM: Virtual Machine

MAT= Memory Analyzer Tools

Visual VM= Java Visual Virtual Machine

Account Clearance

Student Portal

Md. Mahedi Hasan (171-35-2052)

Logout

Registration/Exam Clearance

Semester	Registration	Mid Term Exam	Final Exam/Assessment
Spring, 2017	✓	✓	✓ (Final Exam)
Summer, 2017	✓	✓	✓ (Final Exam)
Fall, 2017	✓	✓	✓ (Final Exam)
Spring, 2018	✓	✓	✓ (Final Exam)
Summer, 2018	✓	✓	✓ (Final Exam)
Fall, 2018	✓	✓	✓ (Final Exam)
Spring, 2019	✓	✓	✓ (Final Exam)
Summer, 2019	✓	✓	✓ (Final Exam)
Fall, 2019	✓	✓	✓ (Final Exam)
Spring, 2020	✓	✓	✗ (Final Exam)
Fall, 2020	✓	✓	✓ (Final Exam)

Student Portal

Md. Mahedi Hasan (171-35-2052)

Logout

Student Dashboard

৳651,650.00

Total Payable

৳651,650.00

Total Paid

৳0.00

Total Due

৳12,450.00

Total Others

9 Appendix 2

PLAGARISM CHECK

Turnitin Originality Report

Processed on: 20-Jun-2021 14:43 +06

ID: 1609333580

Word Count: 11105

Submitted: 1

171-35-2052 By Md. Mahedi Hasan

Similarity Index

15%

Similarity by Source

Internet Sources: 14%
Publications: 4%
Student Papers: 7%

7% match (Internet from 30-Apr-2019)

<https://www.ijcaonline.org/archives/volume182/number41/tasneem-2019-ijca-918504.pdf>

1% match (Internet from 06-Jan-2020)

<http://dspace.daffodilvarsity.edu.bd:8080/bitstream/handle/123456789/3547/P13646%20%2824%25%29.pdf?isAllowed=y&sequence=1>

1% match (Internet from 15-Mar-2021)

<https://docs.oracle.com/en/graalvm/enterprise/20/docs/overview/>

1% match (Internet from 29-Jan-2020)

<https://www.jetbrains.com/help/idea/cpu-profiler.html>

1% match (student papers from 04-Feb-2015)

[Submitted to Institute of Management Technology on 2015-02-04](#)

1% match (student papers from 30-May-2020)

[Submitted to Troy University on 2020-05-30](#)

< 1% match (Internet from 15-Mar-2020)

<http://dspace.daffodilvarsity.edu.bd:8080/bitstream/handle/123456789/3553/P13659%20%2829%25%29.pdf?isAllowed=y&sequence=1>

< 1% match (Internet from 26-Mar-2021)

<http://dspace.daffodilvarsity.edu.bd:8080/bitstream/handle/123456789/2088/P13003%20%2821%25%29.pdf?isAllowed=y&sequence=1>

< 1% match (Internet from 28-Mar-2019)

<https://docs.oracle.com/javase/9/gctuning/garbage-first-garbage-collector.htm>

< 1% match (Internet from 26-Aug-2020)

<https://oregonsigmanu.com/online-watch-store/>

< 1% match (Internet from 02-Mar-2016)

<http://acm2009.cct.lsu.edu/localdoc/java/6-docs/technotes/guides/visualvm/intro.html>

< 1% match (Internet from 16-Jul-2020)

https://mafiadoc.com/netbeans_59c082f71723ddbea5dcffee.html

< 1% match (Internet from 30-Nov-2019)

<https://www.techquark.com/2009/05/visualvm-visual-tool-for-viewing.html>

< 1% match (publications)

[Rod Stephens. "Essential Algorithms", Wiley, 2019](#)

< 1% match (Internet from 20-May-2019)

<https://cadurosar.github.io/rapport.pdf>

< 1% match (student papers from 26-Apr-2021)

[Submitted to University of Greenwich on 2021-04-26](#)

< 1% match (Internet from 12-Oct-2020)

<https://www.mmu.ac.uk/isds/support/apps/software-download-centre/eclipse.php>

< 1% match (student papers from 03-May-2021)

[Submitted to Harare Institute of Technology on 2021-05-03](#)

< 1% match (student papers from 14-Oct-2016)

[Submitted to Softwarica College of IT & E-Commerce on 2016-10-14](#)

< 1% match (student papers from 27-Aug-2020)

[Submitted to University of Newcastle upon Tyne on 2020-08-27](#)

< 1% match (Internet from 01-Apr-2019)

<https://www.globalsoftwaresupport.com/garbage-collection/>

< 1% match (student papers from 09-May-2021)

[Submitted to National University of Ireland, Galway on 2021-05-09](#)

< 1% match (student papers from 24-Sep-2020)

[Submitted to UNIVERSITY OF LUSAKA on 2020-09-24](#)

< 1% match (Internet from 19-May-2020)

<https://biography.omicsonline.org/bangladesh/daffodil-international-university/mr-md-khaled-sohel-185330>

< 1% match (student papers from 28-Mar-2018)

Class: Article 2018

Assignment: Journal Article

Paper ID: [937505521](#)

< 1% match (Internet from 07-Apr-2018)

<https://csdl.computer.org/csdl/mags/mi/2015/01/mmi2015010015-abs.html>

Bachelor's Thesis Android Memory Management in Garbage Collection Using Code Optimization Techniques and Implementing GraalVM Submitted by Md. Mahedi Hasan ID: 171-35-2052 Department of Software Engineering Bachelor of Science in Software Engineering Supervisor [Md. Khaled Sohel Assistant Professor Department of Software Engineering Daffodil International University](#) May 31, 2021 [All right Reserved by Daffodil International University Approval This thesis](#) is being titled as "Android Memory Management in Garbage Collection Using Code Optimization Techniques and Implementing GraalVM" submitted by Md. Mahedi Hasan, 171-35-2052 [to the Department of Software Engineering, Daffodil International University has been accepted as satisfactory for the partial fulfillment of the requirements for the degree of Bachelor of Science in Software Engineering and](#) for [approval as](#) per [its](#) inner styles [and contents](#) inside. -----
----- [Dr. Imran Mahmud Associate Professor and Head Department of Software Engineering Faculty of Science and Information Technology Daffodil International University](#) Chairman ----- Md Fahad Bin Zamal Assistant Professor & Associate Head (In-Charge) [Department of Software Engineering Faculty of Science and Information Technology Daffodil International University Internal Examiner](#) ----- [Name of Internal Examiner 2 Designation Department of Software Engineering Faculty of Science and Information Technology Daffodil International University Internal Examiner 2](#) -----
----- [Name of External Examiner](#) External Examiner [Designation Name of the Department Name of the University](#) 2 DECLARATION It has been declared that this thesis, including all the scientific experimental works, [has been](#) completed [by me under the supervision of](#) Md. Khaled Sohel, [Assistant Professor, Department of Software Engineering of Daffodil International University](#). I [also declare that neither this thesis nor any part of this](#) whole scientific experiment [has been submitted elsewhere for the award of any degree](#). Md. Mahedi [Hasan](#) Student ID: 171-35-2052 Batch: 22 [Department of Software Engineering, Faculty of Science and Information Technology, Daffodil International University. Certified By Md. Khaled Sohel Assistant Professor, Department of Software Engineering, Faculty of Science and Information Technology, Daffodil International University.](#) 3 Acknowledgement Thanks to Allah Almighty, Who gave me courage and

patience to carry out this work. I want to thank my supervisor, Mr. Md. Khaled Sohel sincerely, for his guidance, understanding, and warm spirit to finish this thesis. I always feel that I would not end the thesis without his wise instruction in this limited time. I would also wish to express my gratitude to Dr. Imran Mahmud, Head of the Software Engineering Department, for inspiring us in all means. [I am also](#) thankful [to all the](#) lecturers, [Department of Software Engineering](#) for their unconditional support and encouragement. My deepest gratitude goes to my beloved parents, Khaja Akram Ullah and Nasrin Akter, for their unceasing encouragement and prayers. Special and profound thanks to my brother Md. Kamrul Hasan for his unquestioning support year after year. I offer my special thanks to all my friends for their kindness and moral support.

4 Table of Contents	1
1 INTRODUCTION	9
1.1 Background	1.1
1.2 Motivation of the Research	9
1.3 Problem Statement	10
1.4 Research Question	10
1.5 Research Objectives	10
1.6 Research Scope	10
1.7 Thesis Organization	10
LITERATURE REVIEW	11
METHODS	16
GaalVM	16
3.1.1 A Different approach of using GraalVM instead of JVM	16
3.1.2 Memory Management at Image Run Time	17
3.1.3 Advantages of GraalVM:	18
Java Virtual Machine	21
Eclipse	21
3.4 IntelliJ	22
3.5 Java VisualVM	22
3.6 Memory Analyzer Tool	22
Linux	22
22 4 IMPLIMENTATIONS	23
4.1 Using GraalVM instead of JVM for better memory Management:	23
4.2 Controlling Garbage Collection	24
4.2.1 Control initial Heap Size using Flag	24
4.2.2 Analyzing Heap Dump	25
Memory Analyzer (MAT)	26
Tuning the Garbage Collection	28
4.3.1 Parameter for New Ratio	30
4.3.2 Parameter of Survivor Ratio	30
4.3.3 Parameter of MaxTenuringThreshold	30
4.4 G1 Garbage Collector	31
Efficient Mark & Sweep Process	32
4.6 Using Profiler	32

on Application	33
4.6.1 Overall	
Monitoring.....	
33 4.6.2 CPU Profile	
.....	34
4.6.3 CPU Sample	
.....	34
4.6.4 Thread CPU Time	
.....	35
4.6.5	
Memory Heap	
Histogram.....	35
4.6.6 Memory Per Thread Allocation	
.....	36
4.7 Java Flight	
Recorder	
37 4.7.1 Flame	
Graph.....	
37 4.7.2 Call	
Tree.....	
38 4.7.3 MethodList	
.....	
39 4.7.4 CPU Timeline	
.....	40
4.7.5 Java Flight Recorder	
Events.....	41
4.8 Better	
use of List in Java for better performance improvement	
42 4.8.1 ArrayList	
.....	
42 4.8.2 Using Vector instead of ArrayList	
.....	43
4.8.3 Using	
LinkedList.....	
43 4.9 Better Coding	
Choice.....	45
4.9.1 Primitives Vs Object	
.....	45
4.9.2	
Double Vs Big Decimal	
.....	46
4.9.3 String	
Concatenation vs String Builder.....	
47 4.9.4 Loops vs Stream vs Parallel Stream	
.....	48
5 5 RESULTS &	
DISCUSSION	49
5.1	
Performance Improvement of GraalVM vs JVM	
.....	49
5.2 Controlling Heap Size (Section	
4.1.1)	50
5.3 Better List	
Choice (Section 3.7)	
50 5.4 Tuning Garbage Collection (Section	
3.2):.....	51
5.5 Profiler Results	
(Section 3.5.1)	52
5.6 Better Coding Choice (Section 3.8)	
.....	53
<u>6 CONCLUSIONS AND</u>	
<u>RECOMMENDATIONS</u>	<u>54</u>
<u>6.1 Findings and</u>	
<u>Conclusion.....</u>	<u>54</u>
<u>6.2</u>	
Limitations.....	
<u>54 6.3 Future</u>	
<u>Works.....</u>	
<u>55 6.3.1 Automatic Reference Object</u>	
.....	55
6.3.2 GraalVM	
implementation	
55 6.4 Conclusion	
.....	
55 7 Reference	
.....	56
8	
Appendix 1	
.....	57
9	
Appendix 2	
.....	58
10	

Account Clearance

..... 59 7 [Abstract](#)

[Android OS is the most widely used smartphone OS](#) in the world. Around 73% of the world smartphone users use Android as their smartphone. The percentage is about 96% in our country. But unfortunately, [it has always lacked iOS due to poor memory management](#). [Android is built on Java Virtual Machine](#). Android uses [Garbage Collector](#) as its memory management which is a memory management technique in JVM. [Many memory management techniques have been proposed, such as](#) Controlling Memory leaks, [Managing GPU Buffers](#), Adaptive Background App, Micro-Optimization, Detecting and Fixing Memory Duplications, Partitioning Memory, Non-Blocking Garbage Collector [Dynamic Caching, etc.](#) [All these techniques revolve around Android's current memory structure, which is Garbage Collector](#). But for a long time, other memory management techniques that are not based on Garbage Collector are not proposed. So, in this research, we will have found some way to improve the existing system of Garbage Collector, a new Virtual machine is proposed called GraalVM instead of Java Hotspot Virtual Machine. We have used those techniques we can get optimal memory management for our Android platform. 8 1 INTRODUCTION 1.1 Background [Android is the most used smartphone OS today because it is open source and can be easily integrated into your hardware](#) by manufacturers, [making Android devices cheaper than iOS](#) competitors. Currently, there are over 2 million applications available on Google Play Store. [Having the positives also has its negatives. One of the main problems facing Android users is unexpected crashes and slowing down of devices over time](#). These problems [mainly](#) occur [when](#) the [device runs out of memory](#). Phone manufacturers continue to increase [main memory to compensate, but this is not the solution](#). Currently, the [Android](#) operating system [runs](#) in a [memory structure known as a garbage collector](#). [Garbage Collector is a tool of Java memory management, as Android is based on the JVM](#). [Garbage Collector search and identifies dead objects and reclaims space when they are no longer needed](#). We [can free up memory in two ways, first, by periodically checking for the presence of dead objects and freeing their memory](#). Secondly, [immediately freeing memory when](#) allocated to make [it](#) more significant than the free memory available. 1.2 Motivation of the Research According to StatCounter, there are 2 billion active smartphone users, and 71.93% are Android users. Nowadays, [smartphones have become a necessity for everyone](#). [Most of us rely too much on smartphones to complete our daily tasks. In some respects, they have replaced our computers](#). Over [time, smartphones are becoming much more powerful devices](#). Only increasing main memory cannot be a solution for Android devices. I was fortunate to be the evolution of the smartphone industry from the very beginning. So, I always observed that day-by-day and Android phones increase their main memory for a better user experience. Also, it becomes slow with time, drains more battery, becomes hot, crushes the app very often. But on the other hand, iOS devices are released with lower battery and Main Memory, but they perform very well and sometimes better than high configured Android Devices. So, I was curious about what makes the iOS devices faster and optimized than the same configured Android devices? Also, it did not have a slowing down or battery draining issue like android devices. I was curious about these things from the beginning of my bachelor's life, so I decided to work on this research in my final year thesis. 9 1.3 Problem Statement We find that there are many memory management systems used in Android OS which is based on current Garbage Collection techniques. The garbage collection is based on JVM. But we want to use more modern memory management techniques like Automatic Reference Counting. We did not find many memory management techniques besides garbage collectors. So, we will try to implement some optimization techniques that will lead us to more optimized programs. There is an alternative VM instead of JVM is called GraalVM. But it is not that much easy to implement the whole Java to GraalVM instantly. 1.4 Research Question • How our proposed GraalVM is better than JVM ? • How can we improve current JVM? 1.5 Research Objectives • Implementing different optimization techniques of Garbage Collection in Java (Android) for optimized memory management. • Find a better Virtual machine than current Java Virtual Machine. • Using the memory in more optimized way that devices can work faster and better even with less memory, so devices perform well over the time. 1.6 Research Scope Our research is based on optimized memory management techniques which we want to use in the current JVM. Obviously, using an entirely new technique in such an old environment is not so easy. Because nowadays, there are built-in memory management systems in the programming language. Java was a modern language when Android was invented, but over time it did not get any good plug in that can use real-time memory management techniques while coding. If we can find a way to utilize the memory in a more optimal way, then companies can use fewer hardware resources like processors, batteries but get a very good performance over time. In this way, people do not need to change their phones every year; as a result, our environment will be less polluted. Also, we can have a long-term good experience with our existing android phones. 1.7 Thesis Organization This thesis consists of 7 sections. Next

sections include Literature Review in Chapter 02 based on different case studies, Tools & Methods including feature selection, exploratory data analysis, model development and deployment in Chapter 03. Then in Chapter 4 we implemented our proposed solution like GraalVm and other JVM optimization techniques. Then in Chapter 5 in result part we showed table wise results and improved result that we got from Chapter 4. After that we discuss Result in Chapter 05. And we finally put our findings, 10 limitations conclude with future works can be done from here in Chapter 06. Chapter 07 is including with reference part.

2 LITERATURE REVIEW Throughout this research we found some paper related to our topic of memory management. Some of them are around GC and some of them are also unique. In this section will try to review each paper related to our research and find out their findings and gaps.

Author Year Paper Method Keywords Findings Kashif Tasneem, Ayesha Siddiqui, Anum Liaquat 2019, February Android Memory Optimization ARC [Main Memory, Operating System, OS, Android, iOS, Memory, Optimization, Cache, Pages, Paging, Garbage Collector, GC, ARC, Automatic Reference Counting](#). This research proposes different memory management technique instead traditional Garbage collection Android OS. a of on 11

[Aponso, G. C. A. L. 2017 Effective Memory Management for Mobile operating Systems. American Journal of Engineering Research \(AJER\), 246](#). Dalvik VM Kernel layers, Power management, Memory management, Low memory killer, Android virtual machine. [Low Memory Killer\(LMK\) starts to act and kill Least Recent Used\(LRU\) app](#) when device will go to low memory stage. [Out of Memory Killer\(OOMK\) which kill app having low priority](#). [Linares-Vasquez, M., Vendome, C., Tufano, M., & Poshyanyk, D 2017 How developers microoptimize Android apps. Journal of Systems and Software, 130, 123](#). FindBugs, PMD and LINT Optimizations, Mining software repositories, Empirical studies, Android, Measurement Micro optimizations [must be applied on smartphone apps](#) because [they are more prone to performance issues](#). Usually developer don't use micro optimization because it have to be maintain formerly life cycle, and it's costly for smaller development 12 Yovine, S., & Win- niczuk, G. 2017, CheckDroid: [May, a tool for automated detection of bad practices in Android applications using taint analysis. In Proceedings of the 4th International Conference on Mobile Software Engineering and Systems \(pp. 175-176\). IEEE Press](#). CheckDroid Android tool. applications, taint analysis, programming practices, bad patterns, runtime errors, newbie Android programmers [Long running task should be divided into sub thread](#). Those [sub thread should have low priority](#) then the [main thread](#). [Qian, J., & Zhou, D. 2016 Prioritizing Test Cases for Memory Leaks in Android Applications. Journal of Computer Science and Technology, 31\(5\), 869-882](#). Memory leak di- agnosing techniques Android, This research memory proposed a leak, test case prioritization, test execution [technique which determines a natural sequence of GUI events such as app launch and close. Repetition of such events should not increase memory usage if there are no memory leaks. If there are leaks, memory will keep on increasing](#) 13 [Kwon, S., Kim, S. H., Kim, J. S., & Jeong, J. 2015, October Managing GPU buffers for caching more apps in mobile systems. In Proceedings of the 12th International Conference on Embedded Software \(pp. 207216\). IEEE Press](#). GPU buffer Graphics processing unit, Mobile communication, Memory management, Androids, Humanoid robots, Kernel, Memory architecture. Managing GPU Buffer [compress the GPU buffer memory when app is sent to background](#). But here compression [and](#) uncompressing [will increase access time which makes the solution less optimized](#) [Lee, B., Kim, S. M., Park, E., & Han, D. 2015, July Memscope: Analyzing memory duplication on android systems. In Proceedings of the 6th Asia-Pacific Workshop on Systems \(p. 19\). ACM](#) Memscope Design Memory duplication, Android system, Operating system, Memory management, Contextual software domains. For avoiding memory duplication Android used [several mechanisms such as zRAM and Kernel Same- Page Merging \(KSM\)](#) Both those mechanism [reduce memory usage but consume greater number of CPU cycles and power](#). 14 [Kim, H., Lim, H., Man- atunga, D., Kim, H., & Park, G. H. 2015 Accelerating Application Start-up with Nonvolatile Memory in Android Systems. IEEE Micro, 35\(1\), 15-25](#). TDRAMP CMNonvolatile memory, [Memory management, Phase change materials, Random access memory, Mobile communication, Accelerators](#) hybrid solution NonVolatile [Memory\(NVM\) or Phase Change Memory \(PCM\) as backup of main memory](#) is proposed. [NVM Memory is popular because it consumes less power. PCM is fast, very energy efficient for reading operations but consumes a lot of power in write operations and is very slow](#) Gerlitz, T., i 2013, [Kalkov, I., , October . Schommer, J. F., Franke, D., & Kowalewsk S. Nonblocking garbage collection for real-time android. In Proceedings of the 11th International Workshop on Java Technologies for Realtime and Embedded Systems \(pp. 108-117\). ACM](#). Nonblocking garbage collection. Real time garbage collector, Heap compaction, Heap fragmentation, Java Processor [This GC will work incrementally with short blocking phase. The speed of GC should be in accordance to the garbage created by the OS in order to avoid Out of Memory situations](#). 15 [Lim, G., Min, C., & Eom, Y. I. 2013, January. Enhancing application performance by memory partitioning in Android platforms. In Consumer Electronics \(ICCE\), 2013 IEEE International Conference on \(pp. 649-650\). IEEE](#).

Virtual memory node Memory If memory runs out partitioning of one node, scheme, process memory of only that lifecycle node will be freed enhancement, up. Normally, mobile virtual nodes devices, unreliable apps take memory up a lot more memory than reliable apps. By following this methodology, memory can be saved somewhat from running out

Table 1: Literature Review

3 TOOLS & METHODS

In this section, we will discuss what software, tools, plug-ins are being used in our research. Now we will discuss some major tools that we used to implement to show the outcome of our research.

3.1 GraalVM

GraalVM is a high-performance polyglot runtime which we can use in dynamic, static, and native languages. GraalVM offers features like accelerating the performance of existing Java applications also for building microservices. It provides advanced optimizing compiler technology to provide a high-performance Just-In-Time compiler that can be used to accelerate the performance of any JVM-based application without making any change in code! The future of the Java Virtual Machine is the GraalVM, and it is a development project from Oracle. It's a project to deliver an alternative virtual machine to the current Java Virtual Machine, which will provide better performance.

3.1.1 A Different approach of using GraalVM instead of JVM

The future of the Java Virtual Machine is the GraalVM and it is a development project from Oracle. It is a project to deliver an alternative virtual machine to the current Java Virtual Machine which will provide better performance.

16 Mainly there are three aspects of GraalVM that can be potentially useful for us if we are concerned about application performance.

- I. GraalVM is an alternative to the Java Virtual Machine. We can run Java by code using the global virtual machine as opposed to the standard Java Virtual Machine and it will likely run faster.
- II. Secondly GraalVM provides an alternative java compiler to the regular java compiler. It provides more performance byte code.
- III. GraalVM can natively compile Java code to software that will run natively on our computer. (No JVM needed) So doing this means that the software will not need a Java Virtual Machine to run and therefore it should run more quickly than the traditional way of running Java code.

3.1.2 Memory Management at Image Run Time

The Java heap is turn out when the native image starts up and increases or decreases in size while the native image runs. When the heap becomes full, the garbage collection is triggered to reclaim the memory of objects that are no longer used. To managing the Java heap, Native Image offers different garbage collector implementation:

- The Serial Garbage Collection is the default GC in GraalVM. It is optimized for lower memory footprint and small Java heap sizes.
- The G1 GC is a multi-threaded GC optimized to reduce stop-the-world pauses and, therefore, improve latency while achieving high throughput.
- The Epsilon GC is a no-op garbage collector that does not do any garbage collection and therefore never frees any allocated memory.

Serial GC: The Serial GC is optimized for a lower footprint and small Java heap size. If any other GC is specified, the Serial GC will be used implicitly as the default on both GraalVM Community and Enterprise Edition. The Serial GC is a simple stop-and-copy GC. It generally divides the Java heap into a young and an old generation. Each generation normally consists of a set of equally sized chunks, each a connected range of virtual memory. Those chunks are the Garbage collector-internal unit for memory allocation and memory reclamation.

G1 Garbage Collector: GraalVM also provides the Garbage-First (G1) garbage collector based on the G1 GC from the Java HotSpot VM. G1 is a generational, incremental, parallel, mostly concurrent, stop-the-world, and evacuating Garbage Collection. It aims to present the best balance between latency and throughput. The G1 GC is an adaptive garbage collector with defaults that allow it to work efficiently without adjustment. However, it can be attuned to the performance needs of a particular application.

17

3.1.3 Advantages of GraalVM:

In this section, the advantages of GraalVM have been discussed. Ideal for Microservices and Cloud: Oracle GraalVM Enterprise Edition's Ahead-of-Time compiler, called Native Image, allows your Java and JVM-based applications to be compiled ahead of time into a binary that runs natively on the system, improving startup and memory footprint. GraalVM Native Image can decrease startup times of micro-services up to 100x and decrease memory usage by approximately 5x (Figure 1). The major application frameworks are all compatible with GraalVM Enterprise

Figure 1: GraalVM Enterprise Native Image with GraalVM vs. JDK8

GraalVM Native Image provides instant and consistent throughput, as seen in Figure 2. This results in an initial transactions-per-second rate that is 1500% higher than a JIT compiled application, allowing microservices to perform at their peak, immediately. If the application is run for some time and the JIT compiler has time to warm up, it will achieve higher peak performance, e.g., 16% in the benchmark.

18

Figure 2: Throughput over time of GraalVM Enterprise using JIT compilation and Native Image (AOT) compilation vs JDK12

We do this first using Just-In-Time compilation with JDK8, the results of which are the chart on the left in Figure 3. Then, we repeat the process, but this time we use the GraalVM Enterprise Native Image (JDK8) compiled version of the same Micronaut application. The results of this run are depicted in the chart on the right in Figure 3.

Figure 3: JIT vs. Native Image (AOT) starting up and serving two requests in the first ten seconds

JIT Version: The JIT version of the application takes approximately two seconds (2044ms) to start and return a valid

response. During those three seconds, the process takes up as much as 15,000% of the CPU in spikes and averages about 3000%. It also climbs in memory usage until it plateaus at about 200MB. Native: The Native Image compiled version of the application takes less than 1 second to startup (744ms) and return the first request that we use to confirm the process is ready. During this time, the service consumes approximately 50% CPU and plateaus at about 40MB of memory. 19 Figure 4: Language availability of Graal i) Flexibility of working with many languages: GraalVM allows developers the flexibility to build applications in different languages without the traditional overhead. Objects created in one language can be used directly in another language as if they are native to that language. This removes the traditional marshaling code required, simplifying the application, reducing memory and CPU usage, and getting the product to market more quickly. Accelerating Application Performance: Without any code changes, GraalVM Enterprise can improve the performance of any Java application and any application that runs on the Java Virtual Machine. GraalVM aggressive in lining, polymorphic in lining, and partial escape analysis increase optimization opportunities, provide faster virtual method calls, and eliminate or delay object allocations. It provides for lower CPU used on the same code and fewer objects created, resulting in more miniature garbage collection and higher throughput. Faster application execution offers two benefits: 1. It reduces the response time for user requests, whether interactive or via RES. Applications are running on GraalVM exhibit lower latency, which is crucial when you remember that forty percent of consumers abandon web pages and shopping carts if the response time is over three seconds. 2. Applications that [run faster free up CPU and memory sooner, allowing them to handle other requests or other applications running on the same server. In data centers with ever-increasing workloads, being able to service more requests with the same computing infrastructure reduces the need to purchase additional hardware. Thus, GraalVM Enterprise's reduction of required compute resources can lower capital cost expenditures on-premises and lower operating costs on the cloud.](#) 20 ii) Real-Life Performance Improvement: Twitter: Twitter approved the GraalVM JIT compiler for their Scala-based infrastructure and saw an 8-11% decrease in CPU consumption and a 20% increase in throughput. This resulted in a 512% decrease in the number of physical machines required for each service as it was moved to use GraalVM. Oracle Cloud Infrastructure: Oracle Cloud Infrastructure moved to utilize GraalVM as the JIT compiler and runtime environment for its infrastructure. In doing so, it saw a 25% reduction in garbage collection time, a 10% increase in transactions/second, and has had 0 issues with 10s of millions of core hours of runtime since the migration. 3.2 [Java Virtual Machine](#) The JVM [has](#) mainly [two primary functions to allow](#) I. [Java programs can run on any device or OS \(known as the "Write code once, run anywhere" principle\)](#) II. [and to manage and optimize program memory. Java virtual Machine is an engine that provides a runtime environment to run the Java Code or applications. It converts Java byte code into machine language.](#) Java Virtual Machine [is a part of the Java Run Environment. In other programming languages, the compiler generally produces machine code for a particular system. Java compiler produces the code version for a Virtual Machine known as Java Virtual Machine.](#) So [Java](#) is [the](#) current language for Android OS, so we used JVM. We used three versions of Java from our implementation perspective: 1.8, 11, and the latest 16 version. When Java was released in the year 1995, on that time [all computer programs were written to a specific OS, and program memory was managed by the software developer. So, the JVM was a revelation.](#) 3.3 [Eclipse Eclipse is an essential tool for any Java developer, including a Java IDE, a CVS client, Git client, XML Editor, Maven integration, and WindowBuilder.](#) We [can easily combine multiple languages support and other features into any of our default packages. Eclipse Marketplace allows us for virtually unlimited customization and extension.](#) It also has its own JVM package from version oldest to latest JVM version 16. So mostly in our research, we use Eclipse as our primary IDE. Also, we can change the version before running. We mostly used changing the default flag using runtime arguments. 21 3.4 IntelliJ IntelliJ is the most modern and smart IDE where every aspect [has been designed to maximize developer productivity.](#) It has [intelligent coding assistance and ergonomic design make development not only productive but also enjoyable.](#) It has all support of Java, Kotlin, Groovy, Scala, Android, Maven, Git, SVN, Mercurial, Perforce, Debugger, Profiling tool, JavaScript, Database Tools, and SQL. Particularly in our implementation, I used IntelliJ for its Profiling tool. We used this tool to perform the Java Flight Recorder of our project and found what is wrong with the multi-threaded application. 3.5 [Java VisualVM Java VisualVM is a tool by Oracle that provides a visual/graphical interface for viewing detailed information about Java Java applications while running on a Java Virtual Machine. Java VisualVM organizes JVM data retrieved by the JDK tools and presents the information](#) to enable us to view data on multiple Java applications and compare quickly. We [can view data on local applications and applications that are running on remote hosts. We can also capture data about the JVM software, save the data to our local system, view the data later, or share the data with others.](#) Now Java VisualVM [is](#)

[bundled with JDK version 6 update 7 or greater](#). We used VisualVM to monitor our heap size in memory, CPU usage, classes, threads, performing garbage collection, force the GC, sampling, and profiling. 3.6 [Memory Analyzer Tool](#) [The Eclipse Memory Analyzer TOOL is a fast and feature-rich Java Heap Analyzer that helps](#) us to [find memory leaks and reduce memory consumption](#). [Memory Analyzer](#) can [analyze productive heap dumps with hundreds of millions of objects](#). It also [quickly](#) calculates [the retained sizes of those objects](#) and [see who is preventing the Garbage Collector from collecting objects](#). Finally, it runs [a report to extract leak suspects](#) automatically. So, [in](#) our research, we used MAT to find the leaking object in our project, that was leading us to memory leakage. 3.7 Linux Linux is one of the greatest operating system for modification. We used Linux in our research for using GraalVM. Because in Windows version we do not have all the features on. So, we tried to run same java code in GraalVM & JVM to see the difference. 22 4 IMPLIMENTATIONS 4.1 Using GraalVM instead of JVM for better memory Management: We have implemented a basic Java project of finding the largest prime number from a very big data set. So at first we run the code into normal JVM and after that we implemented the same code in GraalVM. Figure 5: Average execution time for same code in JVM(7507ms) vs GraalVM(6568ms) Result (Figure 5) is very satisfying that without making any change in code we got a great improvement of 12.27%. It clearly shows that GraalVM is a more optimized virtual machine than typical Java Hotspot Virtual Machine. 23 4.2 Controlling Garbage Collection The idea of garbage collection is that programmers ask for objects to be allocated on the heap but they don't need to free them when they're finished with them. Instead, an automatic process will analyses the heap and it aims to work out which objects are no longer needed and any unneeded objects can be deleted and the memory that they occupy can be freed up. Java works out which objects are no longer needed using a very simple rule. [Any object on the heap which cannot be reached for a reference from the stack is eligible for garbage collection](#). 4.2.1 Control initial Heap Size using Flag Figure 6: Default Heap Size and available free Heap We start with 129 0000 kilobytes of memory. It goes down to 96970 kilobytes after creating 1 million customers in the program. And then after the GC command it has gone down hugely to just 9478 kilobytes of memory. One thing this tells us that certainly calling the GC method meant that the garbage collection process ran in this instance. However, is the amount of available memory so much less before creating the instance in Java 11. It is because because of an optimization in the Java 11 garbage collection process which didn't exist in Java 8 or before. From a performance point of view there is an impact of this enhancement if we recall one of the things that every time virtual machine needs to go to the operating system after running GC and give back the extra memory that it does not use. 24 By using a flag to tell the virtual machine when we start to request a particular value to be the initial heap size and even in Java 11 if we use that flag the virtual machine will never let the amount of memory, we've reserved go below that initial heap size. Figure 7: Tuning Heap Size and available free Heap So, this time we started with 300 megabyte, and it went down to 274 megabyte after creating customer. But after the garbage collection (System.gc) ran we're back up and actually it's similar to what we saw that result is again result is slightly more than 300 megabytes. So, we have fixed the issue of losing initial heap size. 4.2.2 Analyzing Heap Dump If we find in a situation where the garbage collector is not working and it's unable to free up enough space to be able to run your applications without them crashing with that out of memory error. We can run a program which have a memory leak so if we run the application it is going to run out of memory and crush. So we will go to the Visual VM and see what object on the heap is causing the problem and generate a file of it. Visual VM: We can go the Visual VM and see what object on the heap is causing the problem and generate a file of it. We opened Visual VM connect to our customer harness, and we'll have a look at the heap. We found the point where the heap is plateauing. We can just click on HEAP DUMP button, and it is going to generate the file for us. 25 Figure 8: Heap monitor of Garbage Collection 4.2.3 Memory Analyzer (MAT) Leak Suspect Report: This tells us what objects are kept alive on the heap and why they have not been garbage collected. Component Report: Component report tells us as you can see things like if there were duplicate strings, empty collections, other things. Figure 9: Leak Suspect Report The pie chart here is saying that eclipse memory analyzes it all suspects there is a problem. In total of 50 megabyte there is an object on the heat which is taking up 46 megabytes and 26 everything else on the heap is just 1.7 megabyte. So that certainly looks like what eclipse think is a suspected memory leak or a leak suspect in their terminology. Figure 10: Leak Description After seeing the details: Here in this dataset, we can see that we have an object called customers which is of type customer manager and it's using up almost all of that heap. That is the retained heap it's the amount of heat that could not be garbage collected. The second table we can see in the manager class there is an array list, and the array list is containing an object. That is probably the underlying array that is used for the array list. And that object is containing all these different customer objects so we can see here clearly that what we have got is an array list containing lots of

customer objects and that is not good for your application. That gives us a clue as to why this application has run out of memory. The memory analyzer has told us is that our Heap has been taken up by all these different objects. But clearly there is an array list that is using a huge amount of retained heap. that allows us to go back to our application to find the problem and hopefully fix it. We now know that there is a particular Array list which is growing out of control and that clearly should not be happening. So Memory Analyzer (MAT) is something that's very useful for us to know about in your own applications. 27

4.3 Tuning the Garbage Collection

We can make a program that we will adding customers into a wait list 10000 customers and remove 5000. So, we are going to be freeing up a larger number of customers all in one go. Here we have created a structure that means that most customers will survive for a little while but not really for that long. Every time a customer is created it is going to survive until at least another 5000 customers have created and then at some point shortly after that it will be available for garbage collection. We are going to run this with a smallish heap in this case 20 megabytes. Now in Visual VM we can see Visual Representation of Garbage Collection. Figure 11: Default NewRatio(2), SurvivorRatio(8), MaxTenuringThreshold(15) Garbage collection that takes place in the young generation are better for performance than garbage collections on the old generation. Young generation garbage collections are quick and efficient. We will not notice any real impact on our application's performance certainly compared to an old generation garbage collection. One thing we will certainly want to do is minimize the number of full garbage collections. 28

Figure 12: After tuning the NewRatio(4), SurvivorRatio(5), MaxTenuringThreshold(10) 29

4.3.1 Parameter for New Ratio -XX:NewRatio = n

By tuning NewRatio we can resize the different parts of the heap. It means how many times bigger should the old generation be compared to the young generation. By default, we say this value is 2 then it means Old Generation will be twice of the Young Generation. So, if we want to increase the size of Young generation and reduce size of Old Generation we can set value to 1. 4.3.2 Parameter of Survivor Ratio -XX:SurvivorRatio = n

This flag means how much of the young generation should be taken up by the survivors' spaces S0 and S1. In our machine the default value is 8. So this means that S0 and S1 should each be 1/8 the of the young generation and in the Eden space will be 6/8 so the Eden space is going to be three quarters of the young generation and our survivor spaces are each going to be half of the remaining quarter. Now we could change this value and if we reduce it that makes these survivor spaces bigger. If we said for example survival ratio is 5 that means that each of the survivor spaces is going to be a fifth of the young generation which is bigger than the 8th that they are in default mode. 4.3.3 Parameter of MaxTenuringThreshold -XX:MaxTenuringThreshold = n

MaxTenuringThreshold determines how many generations should an object survive before it becomes part of the old generation. So we normally want this is to be high as possible because we want our objects to live in the young generation for as long as possible. So, in our case the value we check is 15 and this is the max value. If we want our want objects to be promoted to the old generation sooner only then we can reduce the number from 15 but that won't of course be good optimized. 30

4.4 G1 Garbage Collector

Normally the heap is split into regions and by default there's 2048 of them. Initially some of these regions are allocated to the different parts of the heap. So, some will be allocated to Eden to S0 to S1 or to the old generation. Not all of them regions do need to be allocated initially. When Java decides that a garbage collection is needed in the young generation it looks at those regions allocated to the young generation. It does the garbage collection process and then it can reallocate the number of regions allocated to each part of the young generation to give it what it thinks will be optimal performance. So each time a young generation garbage collection happens the different parts of the heap could be changed. Garbage collector might decide some of the regions that have been previously allocated to the survivor spaces should now be allocated to the Eden space and it could decide to add some of the unallocated regions for example to the Eden space. So that happens every time a minor collection is needed. When a full garbage collection takes place the garbage collector will work out for each region in the old generation which regions are mostly garbage, and it will collect the garbage from those regions first. That is why it is called G1 the Garbage First Collector. If a region contains only unreferenced objects that is a completely garbage region it can be fully cleared and made available. In fact, the G1 garbage collector does not actually therefore need to look at the entirety of the old generation. If it can clear a few regions that might be enough so that a real full garbage collection on the entire old generation is not actually necessary. It can still do a full garbage collection on the entire old generation if it is really needed but hopefully that will be a rare occurrence. So, the idea is that the performance of the G1 garbage collector should generally be better than the other types of garbage collector that we've been looking at because when it needs to do a major collection it can often do just part of a major collection to free up enough memory and it has the ability to resize and reallocate different areas of the heap to different parts of the young and the old generation again to maximize performance. Tuning the G1 GC if needed Flag 1.

-XX:ConcGCTThreads = n We can this flag to specify the number of concurrent threads available for smaller regional collections. It is only useful if we want to limit the number of threads so that we don't impact performance of other applications. Flag 2. -XX:InitiatingHeapOccupancyPercent = n By default, the G1 process starts when the entire heap reaches 45% of fullness. We can change this figure from the 45 percent default by using this flag. We can try changing this flag and see what number really work optimally based on our specific scenario. Flag 3. -XX:UseStringDeDuplication 31 If we use G1 garbage collector then we can turn on String De-Duplication for creating more available space on the heap. If we enable this feature, it allows the garbage collector to make more space if it finds duplicate strings in the heap. When string de duplication feature turned on what the garbage collector will do is it will compare each of the strings and if it finds two strings that have the same value it will make the second object available for garbage collection.

4.5 Efficient Mark & Sweep Process

The general principle rather than searching for all the objects to remove instead the garbage collector looks for all the objects that need to be retained and it rescues them. The general algorithm that garbage collectors use is called Mark and sweep, and this is a two-stage process. The first stage is the marking, and the second stage is the sweeping in the marking stage. The garbage collector doesn't merely collect any garbage it actually collects the objects which are not eligible for garbage collection, and it saves them. This means that the garbage collection process is faster the more garbage there is. If everything on the heap was garbage well then, the garbage collection process would be pretty much instantaneous. In Generational garbage collection so the heap is divided into two sections. One is called the young generation and one is called the old generation. Figure 13: Generational Garbage Collection The young generation which is full of new objects is probably mostly garbage so the process to garbage collects the young generation should be very quick. The garbage collection of the young generation is known as a minor collection so as your application runs there will be lots of minor garbage collections taking place and they will be pretty much instantaneous. Because the young generation will be quite small and mostly full of garbage. The young generation is split then into three sections called Eden S0 and S1 and they are called The Survivor spaces. The object is determined to be a long surviving object and will be moved from S0 or S1 into the old generation. This is all about then is a minor garbage collection. A major garbage collection will take place which is a much slower process but in real world performing application these major garbage collections should be rare.

32 4.6 Using Profiler on Application

When we use [profiling a Java application](#), we [can monitor the Java Virtual Machine \(JVM\) and obtain data about application performance, including method timing, object allocation and garbage collection](#). We [can use this data to locate potential areas in our code that can be optimized to improve performance](#). Here we will [use](#) one project that is not optimized, then we will compare some code with some upgrade in threads and observe the difference.

4.6.1 Overall Monitoring

Unimproved Improved Figure 14: Overall Monitoring 33 4.6.2 CPU Profile Unimproved Improved Figure 15: CPU Profiling 4.6.3 CPU Sample Unimproved Improved Figure 16: CPU Sampling 34 4.6.4 Thread CPU Time Unimproved Improved Figure 17: Thread CPU time 4.6.5 Memory Heap Histogram Unimproved Improved Figure 18: Memory Heap Histogram 35 4.6.6 Memory Per Thread Allocation Unimproved Improved Figure 19: Memory Per Thread Allocation So, we found in unimproved version what is going wrong, which thread is behaving weird. we found those things and then upgrade our code. We can see better result in the improved version. 36 4.7 Java Flight Recorder In this implementation part, we used the same Fibonacci Prime Number project and see the difference in results between them.

4.7.1 Flame Graph

The flame graph visualizes the application [call tree](#) with [the rectangles that stand for frames of the call stack, ordered by width](#). Methods that consume more CPU time and memory resources are wider than the others. The blue color of the blocks stands for native calls, yellow stands for Java calls. Unimproved Improved Figure 20: Flame Graph (Improved) 37 4.7.2 [Call Tree The Call Tree tab represents information about a program's call stacks that were sampled during profiling. The top-level All threads merged option shows all threads merged into a single tree. There is also a top-down call tree for each thread.](#) Unimproved Improved Figure 21: Call Tree 38 4.7.3 MethodList The Method [List collects all methods in the profiled data and sorts them by cumulative sample time. Every item in this list has](#) several views. Unimproved Improved Figure 22: Method List (Unimproved) 39 4.7.4 CPU Timeline The timeline helps you get full control over multi-threaded applications. It visualizes CPU consumption in your application as well in other running processes so that you can analyze the data in comparison. Unimproved Improved Figure 23: CPU Timeline 40 4.7.5 [Java Flight Recorder Events Java Flight Recorder collects data about events. Events occur in the JVM at a specific point in time. Each has a name, a timestamp, and an optional payload.](#) Unimproved Improved Figure 24: Garbage Collector Event 41 4.8 Better use of List in Java for better performance improvement 4.8.1 ArrayList We know that as we add more or more items to an array list it's going to be list sizing itself over time. Because default value of the ArrayList is 10 and if there are more object than 10

then the new capacity of the array will be $\text{New Capacity} = \text{Old Capacity} + (\text{Old Capacity} / 2)$. And that process of resizing could impact the performance of our application multiple times. We will have to allocate new space in memory for the new array and we will need to copy the data across and then the original object would be destroyed. So, if we have a list where we will be adding a very large number of objects this will lead to a lot of these resize operations and the time needed to do this resizing could be inefficient. Obviously, we will be creating objects that will be garbage collected the arrays that are discarded but this will create a less than optimal situation. Figure 25: Execution time Default Array Size(2018ms) vs Fixed ArraySize(1563ms) Without fixing ArrayList: Here we are creating an array list of books and adding 10 million books into this array list. And at this point we are using the empty constructor for the array lists. So, as we add more and more of these books that array this is going to need to be resized. It took 2018 milliseconds. With Fixing ArrayList: While coding we know there is going to be 10 million. So, we want our initial array size to be 10 million and see that it only took 1563 milliseconds. Its of the time has been reduced. Because memory don't need to reallocate for this array multiple times, so it'll be faster. But this was not scientific. You could of course use something like GMH if you really wanted to. If we know it is going to be storing a lot of objects puts a number in here that we think will be big enough to store those objects that will certainly save us having to do a lot of resizing and it can impact the performance of our application as we have seen. Obviously, we do not want to put in too big a number here because that is going to be reserving 42 memory which may not be used and that could be wasteful. So, we must think carefully about the number to put in the constructor for your Array lists.

4.8.2 Using Vector instead of ArrayList

Certainly, if we ask which list should we use between Array and Vector list? Most programmers, they will simply say the vector is an old list type and you should absolutely be using the array list now instead. But there is an important difference between the vector and the array list and that is that the "Vector is thread safe". So, if we are working in a multi-threaded application and we need a thread safe list then we might want to consider the copy on writer Array list. Well, here is an alternative. The vector is a thread safe array list. Now being thread safe comes at a performance cost. So the Java docs advise us not to use it if we don't need to be thread safe but of course if we do well it would absolutely be well worth testing.

4.8.3 Using LinkedList

What Happen when we want to add a new item Start [of the list](#)? Adding [an item to the start of a linked list](#) will always also be quick. It is a simple operation we just need again to update a pointer. But [adding an item to the beginning of an array](#) list is a bit more complicated because all the existing items in the array this need to be shifted along one space in the array before the new item can be added at the start for a bigger way. That of course can be an expensive operation. So, to add items at the start of the list we will always get better performance using a linked list. What Happen when we want to add a new item end of the list? We know already that for the array list this will normally be quick because it's just simply putting a pointer into a vacant lot of the array although there is a potential performance impact if the list is full and needs to be re sized. When this happens, we know that the virtual machine needs to allocate a new array of memory it is to copy the data across from the old array and then at some point that's older arraylist will be garbage collected and we've seen that there is absolutely a performance impact of this resizing when it comes to the linked list. In linked list there are no performance impact to [add an item to the end of the list](#). We simply are adding a pointer so [adding an item to the end of a linked list](#) will always be quick. Java maintains which is the last item in the linked list so it can go straight to it. So, adding to the end of a list will always be optimal with a linked list and will normally be optimal with an array list unless of course there is a resizing operation to do so what about the start of the list. 43 Normally adding a item in starting point is faster in LinkedList and adding an item in last is also better in most case in LinkedList. But the main drawback is this that when we remove an item from an array list, we can go straight to the item to move because we always know its position in the list. But with a linked list to find the item to remove will have to iterate through all the other items first. And this tends to be a slow operation. In an array list the underlying array object has the reference to every item in the list but in the linked list the references held in the object are to the first and the last nodes only. And in fact this is probably the most important difference when it comes to the performance of an array list and a linked list. Figure 26: Execution time Retrieving data of Array(1ms) vs LinkedList(90ms) The array list will normally give better performance overall, but we want to retrieve a particular item knowing its position in the list and a LinkedList will generally give better performance when we want to be adding items. Particularly if we are adding somewhere in the middle or near the start of the list and ask for removing items. Removing the first item in the list we will certainly get better performance from a linked list. But removing an item in the middle we might get better performance from an array list because we do not have to do the navigation to find that item first. So, while the array list is normally the list of choice there are plenty of times when a linked list will give better performance. 44

4.9 Better Coding

Choice 4.9.1 Primitives Vs Object If we will add up a lot of numbers, does it make a difference performance if those numbers are primitives or objects. So, what we are doing in this experiment is we're going to add 1 million longs together and see how long it takes to add those numbers. If we look at the two different implementations so these two versions of code are identical but for the fact that in the first version, we're using primitive longs in the second version we are using object longs. Figure 27: Execution time of Primitive(7ms) vs Object(42ms) Finally, we can say if we use primitives then we're going to get better performance generally than if we use that object counterparts. 45

4.9.2 Double Vs Big Decimal Big decimal might well be slower than the double because we have the overhead of the precision part of the work that's extra processing which would not be needed with a double. So, knowing nothing else about the implementation of the big decimal class we would expect it to perform more poorly than with doubles. Figure 28: Execution time of Double(38ms) vs Big Decimal(113ms) The result is as we expected earlier. Big Decimal is slower than Double. Because the additional requirements of having the precision part of a big decimal does make it perform worse than a double. 46

4.9.3 String Concatenation vs String Builder One is by concatenating strings, and one is by using a string builder and it's appended method. So, let's understand how these two different versions perform and what we've got here is a process to generate 500000 random names. The first version is using version one of us generate names that using the string concatenation. Figure 29: Execution time of String Concatenation(441ms) vs String Builder(189ms) 47

4.9.4 Loops vs Stream vs Parallel Stream Figure 30: Execution time of Traditional Loop (56) vs Stream (41) vs Parallel Stream (27) The traditional Java loops takes 56 milliseconds, Stream took 41 milliseconds and finally Parallel Stream took only 27 milliseconds. By far the Parallel Stream is the most optimized scenario for our use case. Parallel streams will always give us better performance when we can use them assuming of our available CPU resources to process the multiple threads. 48

5 RESULTS & DISCUSSION

Here in this research, we have done lots of implementation to see performance advantages or better optimization. Now we will represent the differences more proper way so anyone can understand the impact of our research. Now we will see the performance boost in the execution time we got from our research part.

5.1 Performance Improvement of GraalVM vs JVM

We can run some basic code of counting prime numbers. We we implemented same Java code in both JVM and GraalVM. Compiler Virtual Machine Elapsed Time Improvement OpenJDK JVM 7507 Default GraalVM GraalVM 6586 12.27% Result is surprising that without making any change in code, we got 12.27% improvement in a basic task. So, it is quite great in term of performance improvements. Execution time may be not the most scientific measurement, but it can summery overall performance enhancement. 49

5.2 Controlling Heap Size (Section 4.1.1)

Test Name Argument Available Heap (Default) Available Heap (After Fixing Heap Size) Improvement in Percentage Available Heap Size -Xms300m 9478kb 306438kb 32.33 Times

Table: Available heap size comparison

5.3 Better List Choice (Section 3.7)

If we choose appropriate list type, then we can get a better performance from same code with different. Here we will see some comparison table.

Operation	List Name	Execution Time (milliseconds)
Adding objects	Array (Default)	2018 ms
In list	Array (Fixed)	1563 ms
Retrieving item from list	LinkedList	1 ms
Array	LinkedList	90 s
Table: List comparison results		50

5.4 Tuning Garbage Collection (Section 3.2):

We can tune things show Garbage Collector will work and we can change some default value like young and old generation allocation space, what will be the ration of S0, S1 and Eden space, how many times it will be change between S0 and S1 survivor space.

Argument	Value	Eden Space	Survivor Old Space	Default
-XX: NewRatio	-XX:SurvivorRatio	XX:MaxTenuring	Thershold = 2	= 8
15	12MB	1MB	7MB	Improved/Tuned
-XX: NewRatio	-XX:SurvivorRatio	XX:MaxTenuring	Thershold = 4	= 5
10	4MB	1MB	15MB	Difference
Eden Space	Reduced	8MB	300%	↓
Survivor Space	Same	Old Space	Increased	8MB
214%	↑	Table: Tuning Garbage Collection	51	

5.5 Profiler Results (Section 3.5.1)

We have tested a project where it will calculate all Fibonacci prime number. But we have taken a project that is not optimized then we profiled those data on Profiler and improved the code. Then we can compare the results. We have run both project for approximately 60 second and got those results.

Test Name	Not Optimized (1804)	Optimized (4388)	Improvement in Percentage
Profiler CPU total time	61389 ms	41590 ms	32.25%
Memory Per Thread Allocations	3.77 Gb/sec	1.97Gb/sec	47.74%
Thread CPU time	600973 ms	35700 ms	94.06
Used Heap	234.44 Mb	35 Mb	85%
CPU Usage	25.8%	24.6%	1.2%
GC Activity	0.1%	0.0%	-

Table: Analyzing Profiler Results

So, after running both project in Profiler for 60 seconds we found lots of difference. From the data of the Profiler of the unoptimized version we can find what is going wrong with our project and easily we can detect and fix those problems. In various aspect we can see the improvement after optimizing the code. 52

5.6 Better Coding Choice (Section 3.8)

We can improve our memory management by choosing the appropriate type of variables. It helps a lot and take less time to execute

Operation	Name	Execution Time (milliseconds)
Adding big list of numbers	Primitive	7
Object		42
Improvement		83%

Adding big list of numbers (3.8.1) Double BigDecimal 38 113 66.37% Generate 50000 names (4.8.3) String Concatenation String Builder 441 189 57.14% Generate 50000 names (4.8.4) Java Loop Stream Parallel Stream 56 41 27 51.78% Table: Better Code choice comparison May be execution time is not the finest way of analyzing performance improvement, but it shows the overall scenario. Results numbers may be varying on different circumstance of project and can be change different real-life project when we will implement those techniques. 53 6 CONCLUSIONS AND RECOMMENDATIONS 6.1 Findings and Conclusion Our goal was to find better memory management for the android operating system. As garbage collection is the current solution for android memory management, we tried to see how optimally we can code. We tried to find some way that Garbage Collector does not need to run frequently. Then we tuned some Java property to control heap and memory leak. After that, we proposed to use some arguments that can help to produce a more optimized application. We compared the execution time to see which process works fast and optimally. Also, we run an unoptimized project and saw its performance in various tools like Profiler, Flight Box, and Memory Analyzer to find the problem in those projects. After finding out the issues, we made some changes in the project, compare the improved project data with the unimproved project. Lastly, we proposed another GraalVM instead of JVM because memory management in GraalVM is much more optimized than JVM. Also, it has some latest garbage collection techniques that lead us to the excellent performance of our existing process. We are delighted with our implementation. We showed the results that our proposed strategies are working better. It will be beneficial for Java programmers to code following those strategies to find more optimized memory management. 6.2 Limitations Java is not a modern language, so it does not have any modern memory management techniques like Swift or other programming languages. Android is so big that it's pretty impossible moving into a better programming language with better memory management by default. GraalVM can indeed handle memory management in a more optimized way. Also, changing from JVM to GraalVM is not as simple as it sounds. Personally, due to Covid 19 pandemic situation, I missed valuable instruction from our excellent faculties who are working for many years and expert in this field. 54 6.3 Future Works Using GraalVM is an excellent approach to handle memory efficiently and should be adopted by Google for Android. It's true that java has become quite an old language and is not keeping up with the latest trends and features of modern development languages. Java was the best choice of language when Android OS was launched, but it looks obsolete after all these years. A virtual machine needs to adopt advanced features that help better code structure and contain micro-optimizations within itself. Some modern languages now include built-in memory optimization tips reported by the compiler at run time, which can significantly optimize the code written and use less memory. Also, we can work on some more things like 6.3.1 Automatic Reference Object Now, java needs a marking process to observe the object that is not used by any variable. So we should work more on the object and find a solution that object can mark itself in java, so the machine should not do the marking process. So if the object can automatically mark itself and count whenever it gets a call from a variable and destroy it when its reference is zero, then the JVM needs no memory for the Mark and sweep process. 6.3.2 GraalVM implementation GraalVM is a new approach, so it is not an industry-ready solution yet. So we don't see it being used in the industry very often. We plan to do a real-life java project in JVM and then transfer it to GraalVM to compare the improvement. 6.4 Conclusion In this research, we try to show every possible way to improve memory management in garbage collection. We find out the way then implemented the strategy in different types of projects. We found our approach performing well when we compared the outcome of every result. Those strategies will help programmers to be code more optimally without having the tension of memory management. By following those approaches from the beginning of the project, they can have better performing optimized applications. 55 7 Reference [1] Kashif Tasneem, Ayesha Siddiqui, Anum Liaquat International Journal of Computer Applications (0975 – 8887) Volume 182 – No. 41. [2] Aponso, G. C. A. L. (2017). Effective Memory Management for Mobile operating Systems. American Journal of Engineering Research (AJER), 246. [3] Linares-Vásquez, M., Vendome, C., Tufano, M., & Poshvanyk, D. (2017). How developers micro-optimize Android apps. Journal of Systems and Software, 130, 1-23. [4] Yovine, S., & Winniczuk, G. (2017, May). CheckDroid: a tool for automated detection of bad practices in Android applications using taint analysis. In Proceedings of the 4th International Conference on Mobile Software Engineering and Systems (pp. 175-176). IEEE Press. [5] Qian, J., & Zhou, D. (2016). Prioritizing Test Cases for Memory Leaks in Android Applications. Journal of Computer Science and Technology, 31(5), 869-882. [6] Kwon, S., Kim, S. H., Kim, J. S., & Jeong, J. (2015, October). Managing GPU buffers for caching more apps in mobile systems. In Proceedings of the 12th International Conference on Embedded Software (pp. 207-216). IEEE Press. [7] Lee, B., Kim, S. M., Park, E., & Han, D. (2015, July). Memscope: Analyzing memory duplication on android systems. In Proceedings of the 6th Asia-Pacific Workshop on

Systems (p. 19). ACM. [8] Kim, S. H., Jeong, J., & Lee, J. (2014). Selective memory deduplication for cost efficiency in mobile smart devices. *IEEE Transactions on Consumer Electronics*, 60(2), 276-284. [9] Gerlitz, T., Kalkov, I., Schommer, J. F., Franke, D., & Kowalewski, S. (2013, October). Non-blocking garbage collection for real-time android. In *Proceedings of the 11th International Workshop on Java Technologies for Real-time and Embedded Systems* (pp. 108- 117). ACM. [10] Lim, G., Min, C., & Eom, Y. I. (2013, January). Enhancing application performance by memory partitioning in Android platforms. In *Consumer Electronics (ICCE), 2013 IEEE International Conference on* (pp. 649-650). IEEE. [11] MA, J., LIU, S., YUE, S., TAO, X., & LU, J. LeakDAF: An Automated Tool for Detecting Leaked Activities and Fragments of Android Applications. [12] Zhang, H., Wu, H., & Rountev, A. (2016, May). Automated test generation for detection of leaks in Android applications. In *Automation of Software Test (AST), 2016 IEEE/ACM 11th International Workshop in* (pp. 64-70). IEEE. 56 [13] Shahriar, H., North, S., & Mawangi, E. (2014, January). Testing of memory leak in Android applications. In *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th* [14] Vimal, K., & Trivedi, A. (2015, December). A memory management scheme for enhancing performance of applications on Android. In *Intelligent Computational Systems (RAICS), 2015 IEEE Recent Advances in* (pp. 162-166). [15] Baik, K., & Huh, J. (2014, June). Balanced memory management for smartphones based on adaptive background app management. In *Consumer Electronics (ISCE 2014), The 8*

Appendix 1 IDE: Integrated Development Environment JVM: Java Hotspot Virtual Machine GC: Garbage Collection ARC: Automatic Reference Counting OS: Operating System VM: Virtual Machine MAT= Memory Analyzer Tools Visual VM= Java Visual Virtual Machine 57 9

Appendix 2 PLAGARISM CHECK 58 10 Account Clearance 59