Memory Optimization of Java Based Applications with the help of Garbage Collection Log

By

Bashar M. Faraneh

Supervised by:
Dr. Akram M. O. Al Mashayki

Amman Arab University

College of Computer Science and Information Technology
Department of Computer Science

December, 2013
Amman – Jordan

Authorization

I am Bashar Misbah Faraneh, authorize Amman Arab University the right to provide copies of the dissertation to libraries, institutes, agencies or individuals when necessary.

Name: Bashar Misbah Faraneh

Signature: [illegible]

Date: 03/05/2014

Resolution of the examining committee

This dissertation titled **"Memory Optimization of Java Based Applications with the help of Garbage Collection Log"**, has been defended and approved on 03-02-2014

Examining Committee	**Title**	**Signature**
Dr. Alaa Alhamami	President	Alahamami
Dr. Ahmad Al-Kayed	Member	[illegible]
Dr. Akram Al-Mashayki	Supervisor	[illegible]

Dedication

I dedicate this thesis to all those who have assisted me throughout my study for the Master’s degree.

Amongst those are my father, mother, wife, my kids and sisters.

My parents who encouraged me to take this step forward and continue the Master’s degree, at the beginning; I thought that my study for this degree won’t add up a value for me, especially that I work in the field of software development. Now since I came up with the idea of this thesis, I am very grateful that I accomplished this degree and I will be looking forward to continue with the PHD InshAllah.

As said; Many thanks to my beloved wife Lama who supported me by all means throughout my study.

I also dedicate this thesis to **Dr. Akram Al Mashayky** and all my friends.

Acknowledgement

First of all, all praise to Allah who helped me to come up with each thought within this thesis.

I would like also to thank my supervisor **Dr. Akram Al Mashayki** for his support, patience and guidance.
I would also like to thank the head of the department Dr. Alaa Al-Hamami for his patience.

Table of Contents

List of Figures

List of Abbreviations

DB:	Database.
GC:	Garbage Collection
I/O:	Input-Output
JVM:	Java Virtual Machine
KB:	Kilo Bytes.
MB:	Mega Bytes.
RAM:	Random Access Memory.

Audience of the Thesis

This work has been prepared for experience Java developers as well as project managers who need to get knowledge in the performance issues related to the memory of any Java web application in general present a systematic approach for finding and solving memory related issues with the help of the output of the garbage collection process.

Abstract

Nowadays, performance of web applications tends to be one of the most important topics when building applications. We build applications to serve different kinds of needs that also would vary in their importance. It would take tremendous efforts building such beneficial applications, thus maintaining such applications and making sure that they are highly scalable and available is even a harder job. Accordingly, this thesis aims to shift the way of approaching and tackling performance problems to a new era, which is being proactive rather than reactive. Memory related problems are one of the most popular problems amongst performance problems, according to the design and purpose of the application, we can predict or come up with a sufficient resources setup for the application, but still, this may still not be sufficient to maintain an available and scalable application. In fact; the resources setup may vary according to the usage of the application
Applications may not crash, or it may not be suffering from a noticeable slowness, this does not mean that the available memory for the application or the memory usage of the application is normal.

With the help of the Garbage Collection Log in Java, the researcher believes that we can learn many things about the memory usage if normal or not and even if the memory setup is correct or not, which as a result may enable us to tune Java based applications performance accordingly by tuning it’s memory usage.

There are many tools available in the market that generates readings from the garbage collection log, but none are used to really analyze this log and generate even more useful information that could easily tell us the source of the problem and how to act. We need to read the pattern of memory usage for the application which no other tool provides.

We cannot depend on Snapshots of the memory at a specific time (Heap dumps) to get the whole picture of the memory usage and pattern since generating a Heap dump at a specific time is a costly operation, thus, a constant generation of Heap dumps at each second of the course of the application sounds irrational.
So, we need other means to analyze the memory usage/consumption of the memory throughout the course of the application that is not costly and easy to get.

The problem is that we used to deal with garbage collection log as the output of the garbage collection operation, but as the researcher will prove in this work, the activities of the garbage collection not only show the result of each operation, but could also tell us more like if the current memory setup is correct or if there is a problem with the application implementation itself, in order to take action and fix issues accordingly.

Doing so, could even be very beneficial in determining problems that the customers did not notice yet, even that they do exist. If No one is complaining, this does not mean that everything is going fine behind the scenes.

The new approach helps discovering problems that with the right parameters and circumstances could explode to a severe performance issue. The theories and concepts that are theoretically presented by this work are proved and justified using a new tool that the researcher has developed to help better solve performance issues related to memory. This will be the first tool to tackle and present solutions for such issues rather than only some readings.

الخلاصـــــــــة

في الوقت الحاضر يكاد يكون أداء تطبيقات الويب من أهم العوامل التي تأخذ بعين الاعتبار عند البدء ببناء تطبيقات جديدة. الهدف من بناء تطبيقات جديدة هو تلبية الإحتياجات المتنوعة لسوق العمل بمستوياتها المتباينة من الأهمية ، وهكذا تطبيقات مفيدة تحتاج إلى مجهود كبير لا بل المحافظة عليها و ضمان استمرارية فعاليتها مع إزدياد عدد المستخدمين هو التحدي الأكبر
بناء على ما سبق ، يأتي دور هذه الفرضية التي تهدف إلى تغيير المسار المعتمد بحل المشاكل المتعلقة بالأداء بجعلها خطوة إستباقية بدلا من أن تكون خطوة تصحيحية. تعتبر المشاكل المتعلقة بالذاكرة واحدة من أكثر المشاكل التي تؤثر على أداء أي تطبيق ، يمكننا القول بأننا نستطيع التنبؤ بالمصادر والمعدات اللازمة لتشغيل التطبيق من خلال كيفية و حجم إستخدامه مع أن هذا الشيئ قد لا يكون حقيقة الأمر.

بإستخدام Garbage Collection Log الذي تم تطويره بلغة الجافا، يمكننا معرفة الكثير عن كيفية إستخدام الذاكرة، إن كان هناك من خطأ ما، وإن كانت طريقة إستعمال الذاكرة هي الطريقة الأمثل.

في الوقت الحالي تتوافر الكثير من الأدوات التي تقوم بقراءة Garbage Collection Log لإعطاء بعض القراءات التي يمكن الاستفاده منها، لكن ولا واحدة منهم تقوم بتحليل تفصيلي لتقديم معلومات إضافية قد تكون أكثر أهمية بالإشارة إلى مصدر المشكلة وكيفية معالجتها. لهذا نحن بأمس الحاجة للأداة التي تقوم بذلك.

لا يمكننا الإعتماد على الطرق المتوفرة حاليا مثل Heap dumps مثلا نظرا لكلفتها، فهي تعطي معلومات عن الذاكرة بوقت محدد فقط لذا يبدو من غير المنطقي أستعمالها بشكل مستمر. لذا نحن بحاجة لإستخدام أداة سهلة لتحليل كيفية إستخدام الذاكرة بشكل مستمر لمراقبة أداء التطبيق

مشكلات الأداء برؤية جديدة

المشكلة هي أننا إعتدنا على التعامل مع garbage collection log كنتيجة للعملية ذاتها لكنني هنا وكباحث مهتم بهذا الموضوع سوف أقوم بإثبات أن garbage collection هي ليست فقط لعرض نتائج كل عملية وإنما تزودنا بمعلومات إضافية على سبيل المثال إن كانت الذاكرة تستخدم بطريقة صحيحة أم لا، أو إن كانت هناك أي مشكلة بالتطبيق نفسه فهذا يساعد على تدارك المشكلة و حلها بشكل فوري. وبهذه الطريقة سيتم تحديد المشاكل التي تواجه المستخدمين ومعالجتها حتى قبل أن يتم ملاحظتها من قبلهم، فعدم إكتشاف المشاكل و التذمر منها لا يعني بالضرورة أن كل شئ يسير على ما يرام.

الطريقة الجديدة تساعد على إكتشاف المشاكل المتعلقة بالأداء والتي قد تنجم عن إستخدام التطبيق بالظروف الإعتيادية

تم إثبات النظريات والمفاهيم التي سبق ذكرها نظريا في هذا العمل من خلال الأداة الجديدة التي قام بتطويرها الباحث للمساعدة بحل مشاكل الأداء المرتبطة بالذاكرة.

ستكون هذه الأداة هي الأولى من نوعها التي تقدم حلول حقيقية عوضا عن تلك التي تقدم مجرد قراءات.

Chapter One – Introduction

1.1 Importance of Applications

Computer applications play a significant role nowadays in accomplishing complex tasks in no time. Key businesses cannot stand having their applications being unavailable even shortly. [1]

Customers buy applications to help them accomplishing their business processes efficiently and quickly. They even build strategic plans and decisions with the help of such applications.

For this reason, customers cannot tolerate applications that were supposed to fulfil and achieve their objectives being not available or even of low performance.

Another new term emerged lately, which is the “User Experience” in which the users/customers nowadays help setting the limits and standards for the tolerated and acceptable response time. As a result, many methodologies and tools were introduced to help measuring the application’s efficiency, productivity, and for sure performance.

Making the application more scalable imposes new challenges and even threats for the application’s owners to deal with. On the other hand, making applications more scalable threatens their availability.

Application’s availability as mentioned previously in this research means simply whether the application is online or offline, applications that are always available can be defined as highly available applications, otherwise, it is set to be poorly available.

Simply, application’s Performance can be described, as the raw speed of your application in terms of a single user.

Java Based Applications

According to recent studies of the programming community, Java is probably one of the most popular programming languages in the world. We are used to the presence of Java in all kind of servers, personal desktops, and more recently embedded systems. In fact the Java Standard Platform is being used in all types of embedded devices, ranging from routers, smart phones, and 3G telecommunication devices. [2]

The Java language has many advantages over other programming languages as follows:

- Platform Neutral
- Automatic Memory Management
- Object Oriented
- Easy to learn.

One of the important strengths of the Java language as stated previously is the automatic memory management, in which the management of the memory of a Java based application is left to the JVM itself, rather being the responsibility of the developers besides the role of designing/building applications.

Having such ability to automatically manage memory of a program leaves not much space for developers to tune the application's memory.

This is correct up to an extent, developers still could have an impact on the application's memory if for example they choose the wrong API's to accomplish the tasks, or if they create objects that are irrelevant to the task or even redundant.

Programs may even need different resources to accomplish tasks, like I/O operations, Database operations, or any other resources. It is extremely important to efficiently use such resources, because if not, it may extremely affect the application's memory which is a valuable resource that should be best utilized in able to maintain a scalable and available application.

As pointed previously, it is the responsibility of the Java to automatically manage the memory of the applications it hosts; still, there are many responsibilities on the developers of the applications, which if not done carefully could extremely negatively impact the memory aspect of any application.

1.2 1.2 Importance of Web Applications

As illustrated in the previous section, it is a critical job to best utilize the available resources of any application, especially the memory part.

The job becomes even harder when it comes web based applications. For instance, what is acceptable memory consumption within a stand-alone application may not be acceptable for a web based application.

Key businesses need to extend their services base as much as possible, web applications make this much easier to achieve.

Web Application Basic Structure

It starts when any client request a service from the web application, the server then tries to process the request and forward it to the correct handler to handle the request, it may require accessing the database, and finally, the response is sent back to the client with an answer to his request, as part of the response.

This end to end operation is called Request-Response model, starts from the client as a request and also ends there as a response. This operation may face some bottlenecks at any stage which may affect the response time and/or the ability for the server to serve more requests, which as a result lowers the application's performance.

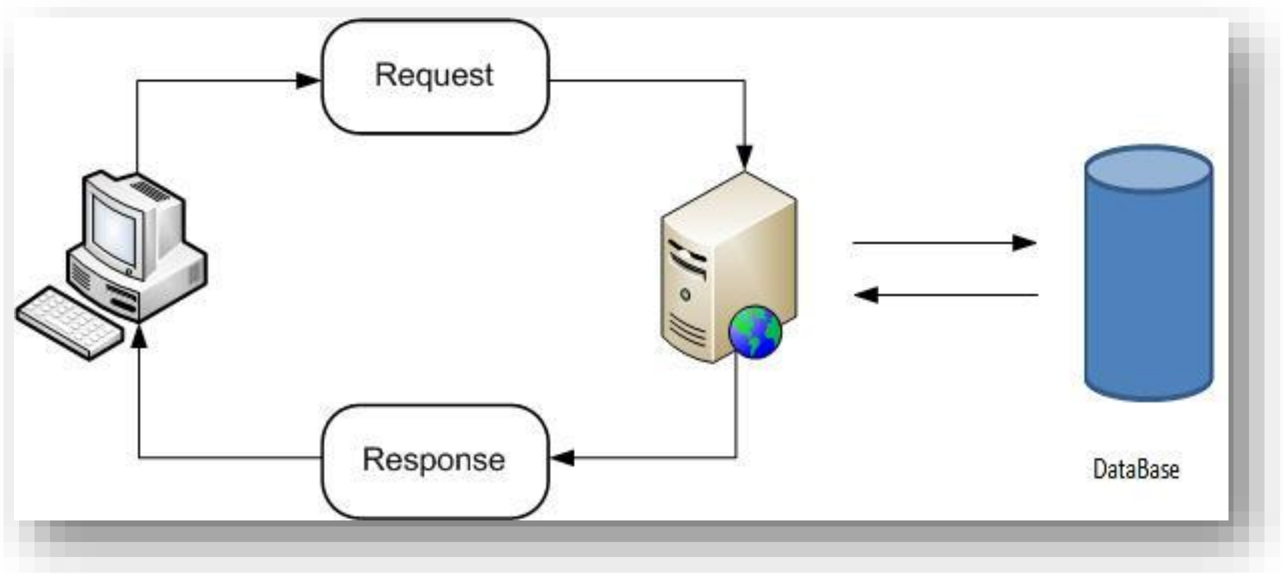


Figure 1: Basic HTTP Web Communication

The operation can be referred to as the transaction, which is simply; a single logical application behavior, that is; the client's request leads to a call to the appropriate application logic on the server.

1.3 1.3 Benchmarking & Performance measurement

Customers need to do as much transactions as possible to make the best of a profit.

In the world of web applications, what is the standard measurement for performance?

It can be different according to the perspective, for some of us, it might refer to scalability, the more the scalable, the better the performance, for others it might mean the availability, but for user's it's usually the response time.

Drilling down into the "Response time" aspect of the performance is somehow tricky, for instance, an operation within an application maybe taking much time because the resources are being inefficiently used by another operation within the application, which as a result could be misleading regarding our findings and analysis.

Take another example, observing memory consumption for a specific operation within a specified time interval could also be misleading; results that may seem normal could be not and vice versa. Memory could be highly used due to inefficient usage of the resource rather than it is being efficiently used.

For that reason and many others, getting out with reliable, reasonable benchmarks for performance is somehow tricky.

Who defines benchmarks?

According to the researcher's experience with web applications, setting up the benchmarks of the performance for the different business processes was only left to the applications owners.

With time, and the increasing demand on web applications, customers gained more experience with web applications and the depended more and more on these applications in a way that can determine what is acceptable and what is not.

1.4 1.4 What is Poor Performance?

Activities that create demands for system resources that cause response time to exceed users expected tolerances.

When it comes to poor performance, it could be related to a congested network that is currently unable to host new customers, it could be related to DB where it is loaded with many data, it could be related to IO operations where the requested number of operation is more than the servers could handle, it could be a CPU issue, where there are many clients to be served per milliseconds, or it could be related to memory issues where there is no enough memory to serve that amount of requests.

This work is going to target the memory aspect of the performance with respect to the garbage collection process, more specifically application's performance that is built on top of the Java technology.

Performance monitoring or measurement is not a new term; it has been always the main concern for application's owners.

In order to achieve acceptable response time, all of the components of the web application should be tuned to achieve a better application's performance.

Memory is one of the critical resources needed for any application, for this reason, it should be carefully used, otherwise applications would be inefficient, not scalable or event not available.

For instance, memory tuning and monitoring applications are divided into two approaches:

- Memory Profiling Tools.
- Detailed analysis of objects distributions and allocation sizes.

1.5 1.5 Performance Aspects

As already illustrated in the previous chapter, performance issues could be classified under many categories:

- Memory related issues.
- CPU related issues.
- I/O related issues.
- Network related issues.
- Database related issues.

These are considered critical resources that if not used efficiently and effectively could easily result in resource starvation which may affects the **response time** of an application, hence lowering the **user's experience**.

If we started with the Database tuning options, this topic have many researches and its own methodology that reached to a point that is mature enough to tune DB transactions or even DB structure related problems.

The same goes for network and files operations (I/O) related issues, many tools emerged that enables technical teams to monitor the traffic of the network and the operations on the file system to track source of problems.

The discussed aspects at the beginning of this section are easy and straight forward to track, tracking a specific path of the network, tracking traffic of specific nodes, tracking the source of I/O problems.

As for the other classifications: CPU and Memory, they have been always considered a tricky topic, it always bares more researches and investigations. Actually problems related to these two categories of a Java applications are the most common and needs more efforts to work on.

These aspects directly affect response time in which:

- A congested CPU could simply affect response time, hence the user's experience.

- An overloaded memory also could simply affect response time.

They are very important because with each request within a web application, CPU is needed to execute certain tasks (processes) and memory is needed to load and prepare the commands for the CPU to execute.

1.6 1.6 The Problem

Memory issues fall under the following two classifications:

- Insufficient Memory Available.
- In-Efficient Memory Usage.

Knowing the classification of a memory issue has a big impact on solving the memory related problems, without knowing the type of the problem; it would be impossible solving the issues.

Several methodologies currently exist to help solving memory issues, the problem here is not with the shortage of the methodologies/approaches or even tools, the problem is with discovering issues at the right time.

The second shortage of the current methodologies is the weakness to discover memory issues if no one is complaining or if no one noticed any performance issue. Current approaches would be very costly in fulfilling this need as this work is going to present.

Whenever memory related issues are spotted, the current approaches would provide an excellent way of analyzing the issue and the real cause of it. The challenge here is that most of the memory related issues do not appear out of a sudden [3]. This means that in order to analyze and spot memory issues (which as pointed already pointed, most of them do not appear out of a sudden) with the help of the current approaches would be very costly and not applicable.

1.7 1.7 Proposed solution

What we need is a way to know how efficient the memory usage or setup for our applications is, the approach is simply by depending on another source of profiling and dealing with it in a new way to know these very important points: The memory sufficiency for the application and the Efficiency of the memory usage for the application.

If we want to approach the memory issues and activities, we need the most efficient way, which is the “Garbage Collection Log”, where the Garbage Collection is the process responsible for freeing and managing the memory resources. From the activities of the Garbage Collection, it can tell us a lot about the health of an application from memory perspective.

The Garbage collection activities are represented by a “log” file which contains every single activity for the garbage collector whenever it needs to cleanup/manage memory for the application.

When we say “manage memory” it is referred to the “Allocation and De-allocation” processes, allocation refers to the process of allocating memory for objects and processes, while de-allocating refers to the process of reclaiming back the reserved memory for these objects/processes whenever they no longer need them.

Profiling the Garbage Collection Log is a not new thing, but it needs more attention to make it easier on application developers or even system administrators at the customer’s side to take a decision towards any memory issues if present, current tools provide numbers out of the Garbage collection log, which still needs analysis from the application developers. The Proposed approach in this work does the analysis and provides straight forward recommendations for either the application developers or again the system administrators.

It even presents a new approach for when to monitor the garbage collection log to resolve the issues related to memory in general or even related to memory leaks as discussed previously, in order to get early and constant feedback on the health of the memory rather than waiting for customers to complain about the performance of the application that is related to memory. It is very frustrating for customers to have their applications very slow or being unavailable for any reason, so by this we can know if any memory issues available before they turn out to be a show-stopper issues, hence react upon before it’s too late.

It is a not common case to have performance issues within applications, at the contrary, applications should be developed very efficiently and effectively not to produce any performance issues, still maybe according to the usage of the application or even to the number of users, things may change and in specific scenarios, some components of the application may appear to be weak whenever applications are heavily loaded. For this reason, we need to be act before memory issues become severe problems.

Depending on the two current approaches mentioned in section (1.5.1 Introduction) for memory monitoring, would be too late, because they are used either to check the current usage of the memory by the application, or even to analyze a snapshot of the memory at a specific time; usually this snapshot is taken whenever the application is about to crash or when it is extremely slow.

If No one is complaining, this does not mean that everything is fine.

The approach that the researcher has followed depends on the fact of discovering small or average performance issues related to memory before they turn out into big performance problems.

This research comes up with a systematic approach towards tuning the memory of **Java based applications** with the help of garbage collection log

The research propose theoretical solutions towards solving problems related to memory depending on the garbage collection log, then it even presents a practical solution by developing a new tool that facilitates the process of discovering problems and presenting a better way in solving them.

Implementation Details

The theories and solution provided by this work **only applies to web based Java applications**. The solution was developed using the latest technologies in the Java language.

Technical details:

- Java 5 & 6.
- Eclipse Kepler IDE
- PrimeFaces
- gcviewer

1.8 Structure of the Thesis

1.9 Chapter One - Introduction:

The aim of this chapter is to put the readers on track with the importance of Web applications in accomplishing nowadays tasks and according to their importance, the availability and scalability of applications varies.

It also discusses in brief the meaning of Performance and the means of measuring it for Web applications and the challenges that faces any performance tuning strategy of web applications.

It also simply defines what a poor performance is when it comes to the memory aspect of an application.

It also puts the readers on track for the importance of memory aspect of any web application and its role in maintaining stable application.

1.10 Chapter Two - Literature Review:

This chapter put the readers on track with the current approaches towards monitoring of the memory of an application and also the tools available to analyze garbage collection log.
It finally point out what the current approaches/tools are missing.

1.11 Chapter Three - Java Memory Structure and Possible Issues:

This chapter explains the structure of the memory for the JVM and its different divisions. It also explains the Object’s lifecycle within the JVM memory. This is very important in order to understand how to come up with possible solutions to memory related issues.

1.12 Chapter Four - The Proposed Solution:

This chapter illustrates the solution that this research presents to approach memory related issues that causes performance bottlenecks.

1.13 Chapter Five - The Tool:

This chapter illustrates the developed tool by the researcher, which is a new tool that the researcher developed to approach and present solutions to the performance issues that are related to the memory or garbage collections activities.

1.14 Chapter Six – Conclusions and Future Work:

This chapter briefly explains the conclusion of this work along with the future work plan.

Chapter Two – Literature Review

1.15 2.1 Chapter Introduction

As previously said, there are many available tools and researches that targets all the performance issues for applications, from DB monitoring, I/O, network, CPU to memory monitoring applications.

Within this chapter, the research will only focus on the best available memory monitoring tools developed so far, and then it will finally briefly explain the elaboration and the added value that this research will present.

Memory is considered one of the systems important resources, according to the nature of the application; the memory setup varies.

Applications could be small, medium or large/enterprise sized. In all cases, applications providers all over the world consider the memory an expensive and limited resource that should be used efficiently.

As mentioned previously; this paper will target the memory performance of Java based applications.

Before drilling down into the problem and the proposed solutions, we have to point out what is currently introduced in the market and what has been developed in this area.

1.16 2.2 Memory monitoring Methods

There are many options for monitoring memory of applications, mainly done with the help of tools nowadays.
There are many tools from different vendors today in the market; they mainly focus on monitoring the memory used by an application throughout the course of an application.
Such capabilities are very important, but not to a big extent. Meaning that, they are useful up to a limit only, they have much functionality that can be categorized in brief to:

- Instant monitoring: Instantly monitor the memory, used to check out the memory usage currently by applications.
- Recorded monitoring: For history tracking; used to check out the memory usage within a specified time-frame.

Why both options are needed?

The monitoring capabilities provided by such tools are mostly used by the applications developers to track how much memory a specific process consumes, this is beneficial in checking if the implementation of this process should be tuned or not, or even to decide if another implementation methodology should be chosen.
Also, it gives us a way to check the overall progress of the application in terms of memory usage for a specific business day within the customer's organization.

2.2.1 Memory readings

There are a number of tools available that helps monitoring the memory usage of an application at runtime, showing the available against free memory.

This is helpful for the applications owners to use if to monitor whether the application is performing well or not in general, or in a specific scenario.

It could be used to get:

- Initial Memory Used.
- Average Memory Used.
- Max Memory Used.
- Min Memory Used.

This is interesting and beneficial; it helps in getting to know how much memory applications consume according to the above classifications. The problem with this approach, let us call it, “Online Memory Monitoring” in reference to getting instant readings about the memory for an application, the problem here is that it does not give us much details and also it does not inform us when wrong things happen.

On the other hand, these readings could be misleading because the same application could behave differently on the customer’s side, many things could affect the application’s performance, from the way of usage, to peak hour’s usage and to the number of users using the application, these could be of big difference also, and the output even varies from customer to another.

These concerns should be covered in any monitoring strategy, which currently are not.

2.2.2 Memory Profiling

One of the most famous tools to monitor memory usage is the **VisualVM** which is an Oracle product (originally from Sun Microsystems).

It contains some features to monitor the memory usage as illustrated in Figure 2. As an example, it uses charts to represent instant consumption of the memory during the lifetime of the application.

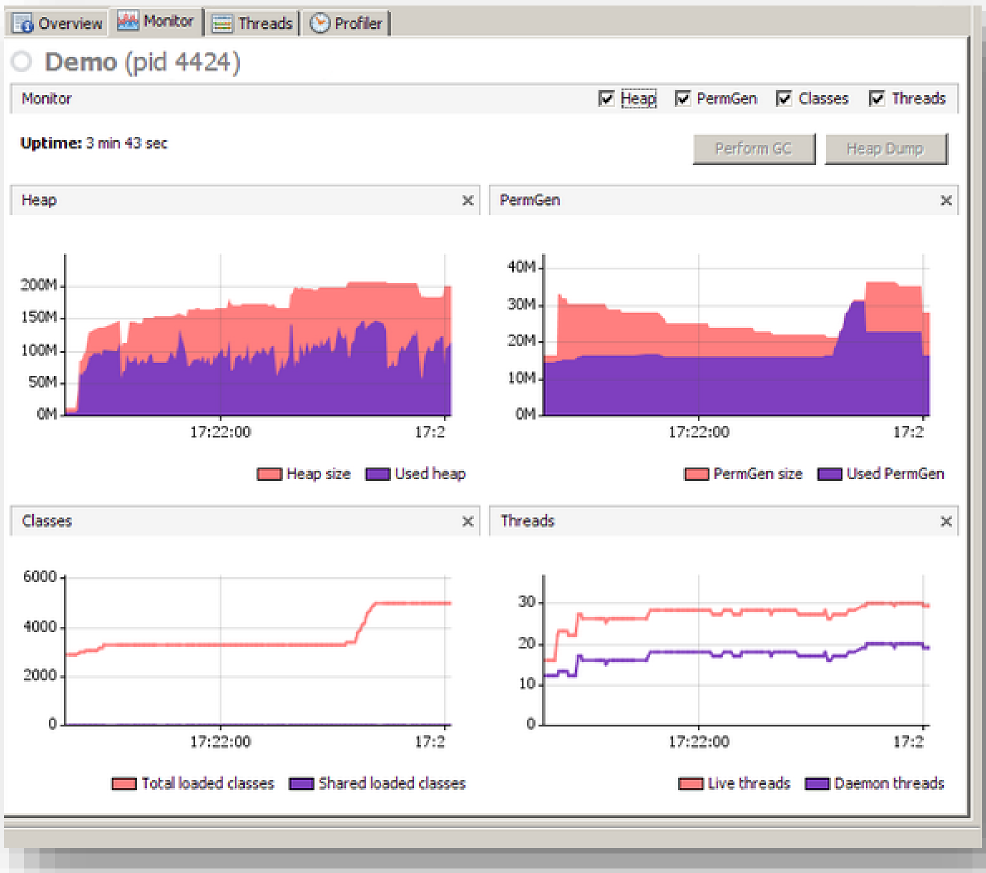


Figure 2: Visual VM - Memory Monitoring

2.2.3 Offline Memory Monitoring

Besides the online monitoring provided by such tools, it has an offline monitoring option, were it could show the usage in details the distribution of objects in memory. Meaning that; it shows for each type of an object how much memory it is using from the total amount of memory.

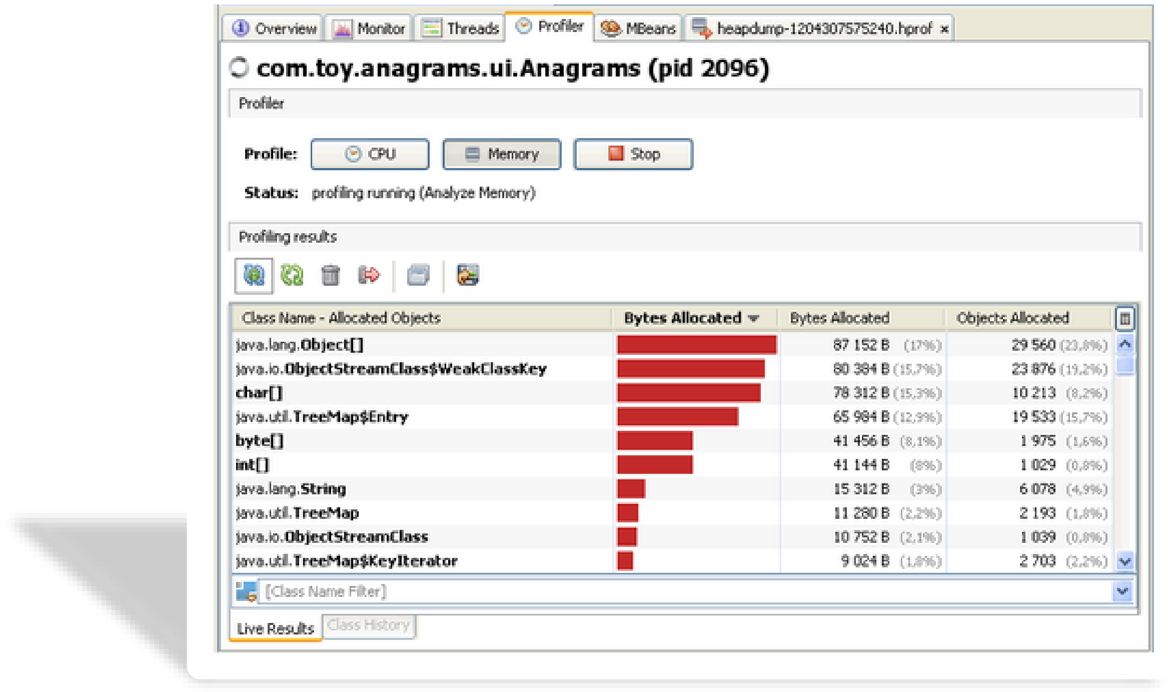
This is done through what is called a “memory dump” to get a snapshot of the memory consumption at a specific time.

Heap dump is a snapshot of the memory of a Java process [4].

This can be done in two ways:

- Automatic: An automatic heap dump could be generated whenever a threshold is met or an out of memory error is thrown
- Manual: A manual snapshot of memory could be generated at any time.

Heap dumps are very useful in showing which entities most consuming memory as shown in Figure 3.



com.toy.anagrams.ui.Anagrams (pid 2096)

Profiler

Profile: CPU | Memory | Stop

Status: profiling running (Analyze Memory)

Profiling results

Class Name - Allocated Objects	Bytes Allocated ▾	Bytes Allocated	Objects Allocated
java.lang.Object[]		87 152 B (17%)	29 560 (23,8%)
java.io.ObjectStreamClass$WeakClassKey		80 384 B (15,7%)	23 876 (19,2%)
char[]		78 312 B (15,3%)	10 213 (8,2%)
java.util.TreeMap$Entry		65 984 B (12,9%)	19 533 (15,7%)
byte[]		41 456 B (8,1%)	1 975 (1,6%)
int[]		41 144 B (8%)	1 029 (0,8%)
java.lang.String		15 312 B (3%)	6 078 (4,9%)
java.util.TreeMap		11 280 B (2,2%)	2 193 (1,8%)
java.io.ObjectStreamClass		10 752 B (2,1%)	1 039 (0,8%)
java.util.TreeMap$KeyIterator		9 024 B (1,8%)	2 703 (2,2%)

[Class Name Filter]

Live Results | Class History

Figure 3: Visual VM – Heap Dump Analysis

It shows the memory usage per object which is very useful in determining areas with highest memory consumption,

As we noticed, such tools give us indications of the application’s health at specific point of execution, which is after the fact an issue occurs.

So, using such tool, gives us instant indication of the application’s health which is not quite accurate in determining the overall progress of an application.

Heap Dumps Trade-off

As heap dumps are very useful, they are also costly, they actually stop the processes of the application from being executed until finished, and the time it takes to generate a heap dump varies depending on the amount of memory currently being used.

This means that, we can only know the memory details only at specific time, which as a result does not represent the memory consumption of an application.

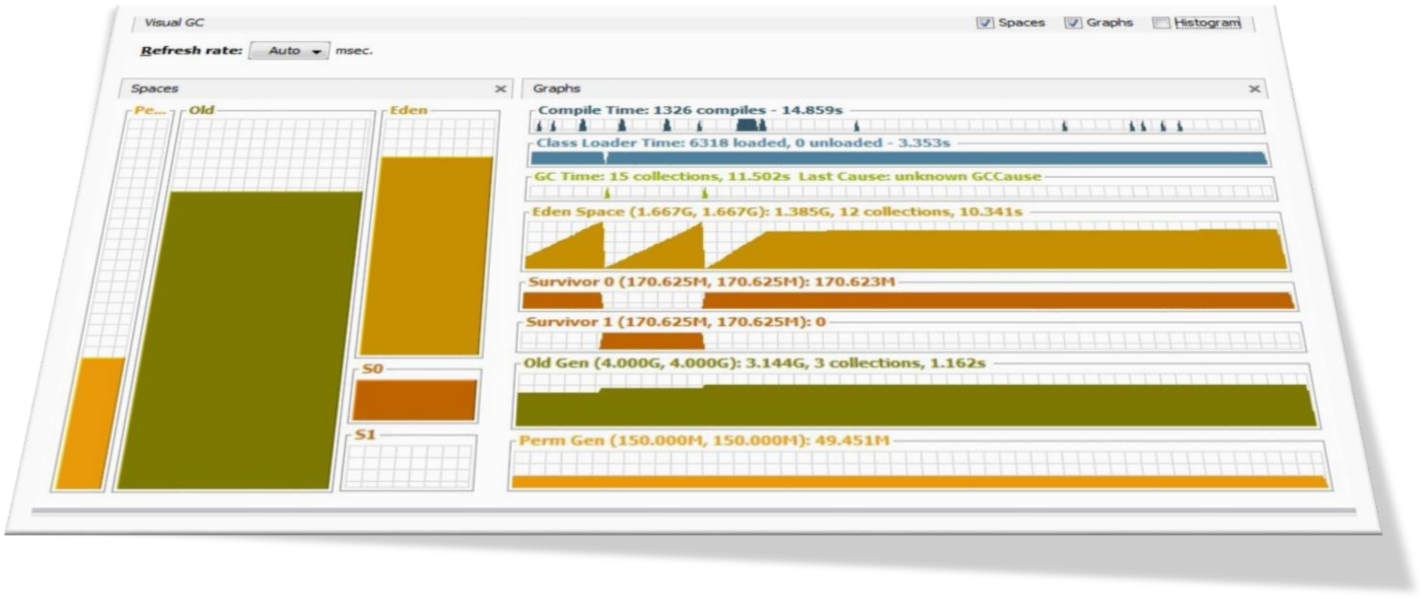
2.2.3 Garbage Collection Profiling

Garbage collection is a process that is responsible for freeing unused objects from memory, these activities runs in the course of the application.

There are also many tools used to monitor the activities of the garbage collector, again, many of them are categorized under the **“Online”** monitoring tools (Profiling tools).

Online garbage collection monitoring tools are used to monitor the activities of the garbage collector towards a specific process, hence knowing how the garbage collector behaves.
Again, this is not highly useful since, it is only used to monitor single process readings.

Here is how the Visual VM tool helps with the garbage collection log.



Compile Time: 1326 compiles - 14.859s
Class Loader Time: 6318 loaded, 0 unloaded - 3.353s
GC Time: 15 collections, 11.502s Last Cause: unknown GCCause
Eden Space (1.667G, 1.667G): 1.385G, 12 collections, 10.341s
Survivor 0 (170.625M, 170.625M): 170.623M
Survivor 1 (170.625M, 170.625M): 0
Old Gen (4.000G, 4.000G): 3.144G, 3 collections, 1.162s
Perm Gen (150.000M, 150.000M): 49.451M

Figure 4: Visual VM – Garbage Collection Monitoring

As it is shown in the Figure 4, the Visual GC that is part of the Visual VM tool, it shows the different areas of memory that the garbage collection is responsible for managing them.

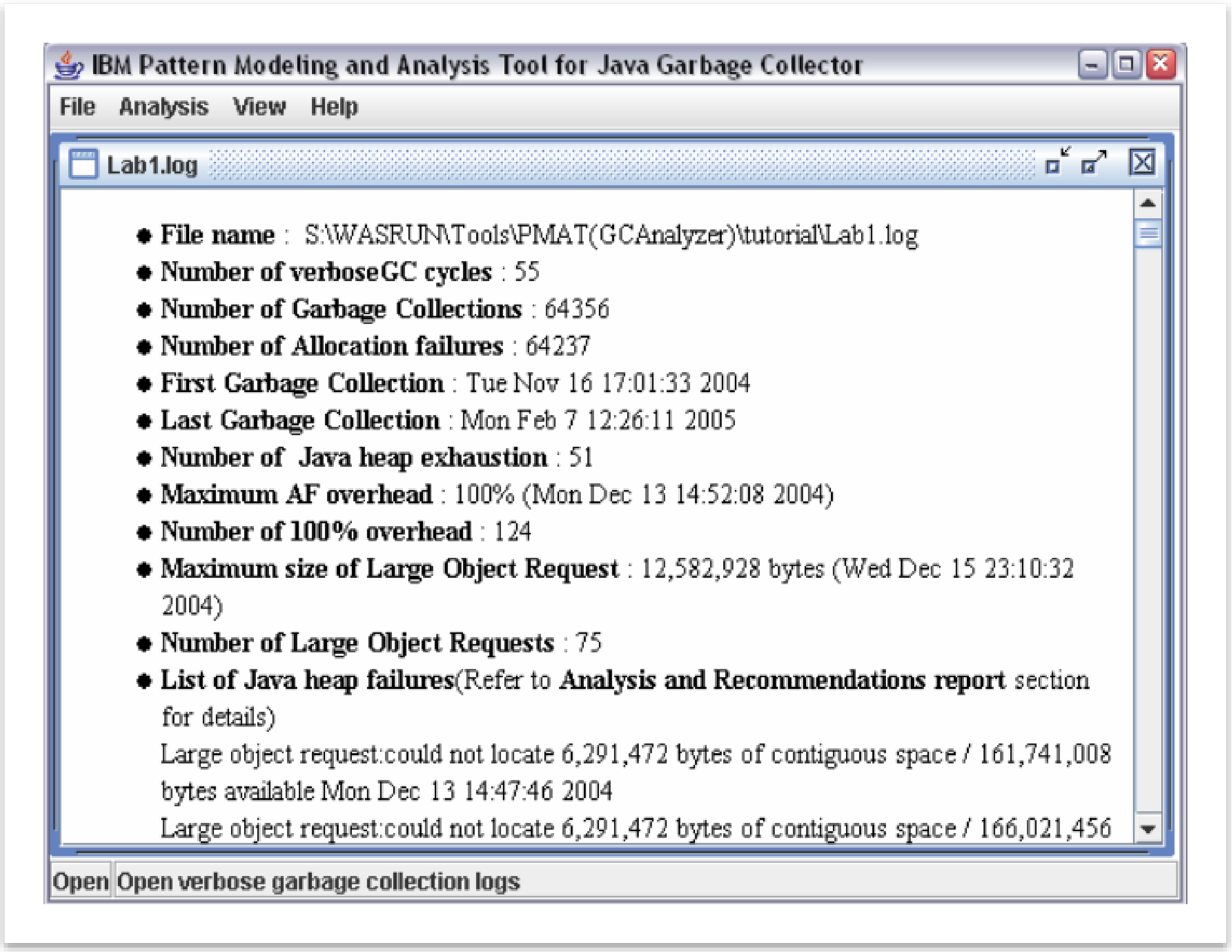
2.2.4 Offline Garbage collection monitoring

2.2.4.1 IBM PMAT for IBM Rational Developer

IBM PMAT is a good tool for analyzing garbage collection log and giving some statistics of the collection process.

The researcher thinks that it’s a good tool but it’s not sufficient because we need a tool that gives us the pattern between the different readings and drives us through the process of tuning with the help of garbage collection log.

Here is some screenshots from the tool:


IBM Pattern Modeling and Analysis Tool for Java Garbage Collector
File Analysis View Help
Lab1.log
Open Open verbose garbage collection logs

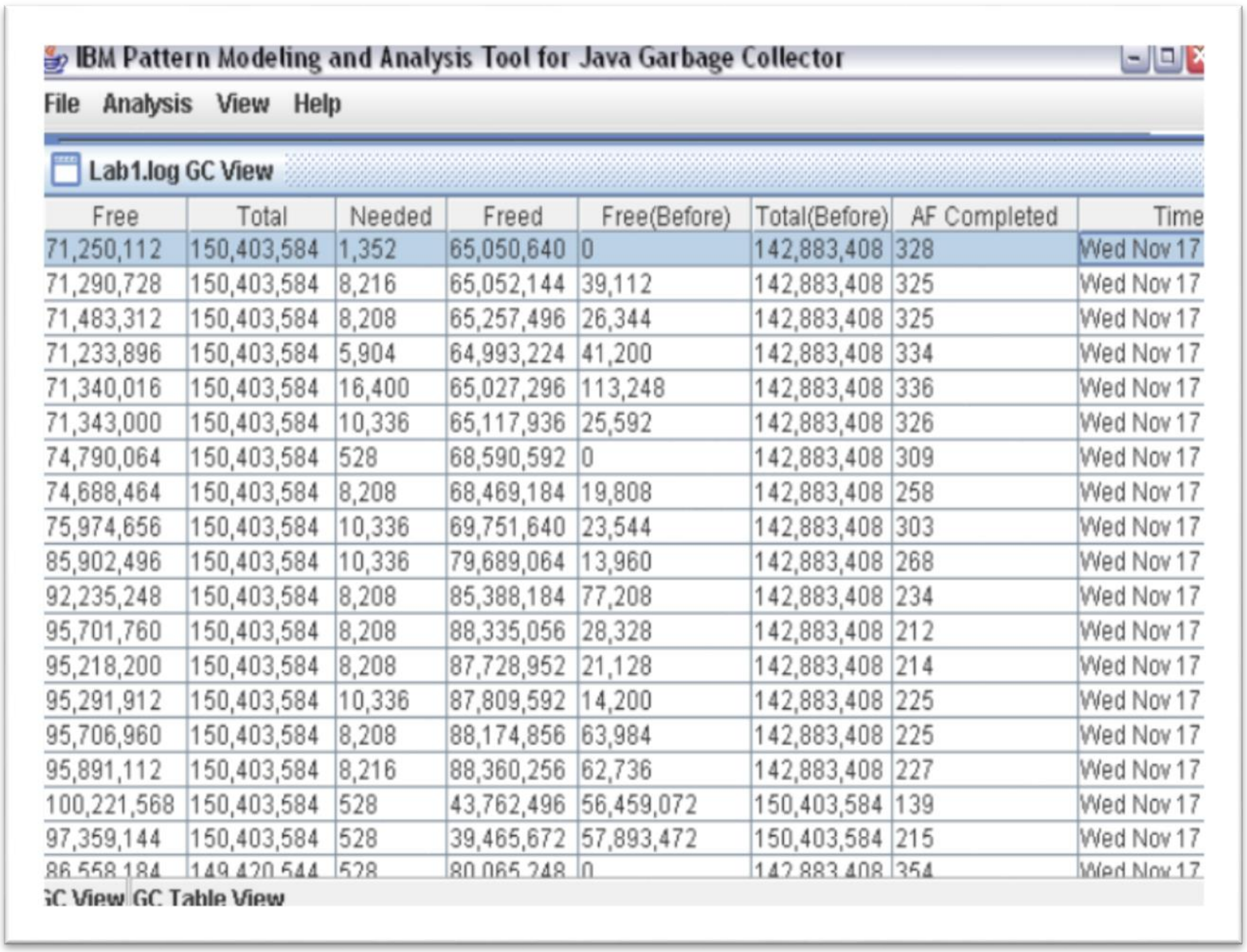
- **File name** : S:\WASRUN\Tools\PMAT(GCAnalyzer)\tutorial\Lab1.log
- **Number of verboseGC cycles** : 55
- **Number of Garbage Collections** : 64356
- **Number of Allocation failures** : 64237
- **First Garbage Collection** : Tue Nov 16 17:01:33 2004
- **Last Garbage Collection** : Mon Feb 7 12:26:11 2005
- **Number of Java heap exhaustion** : 51
- **Maximum AF overhead** : 100% (Mon Dec 13 14:52:08 2004)
- **Number of 100% overhead** : 124
- **Maximum size of Large Object Request** : 12,582,928 bytes (Wed Dec 15 23:10:32 2004)
- **Number of Large Object Requests** : 75
- **List of Java heap failures**(Refer to **Analysis and Recommendations report** section for details)
  Large object request:could not locate 6,291,472 bytes of contiguous space / 161,741,008 bytes available Mon Dec 13 14:47:46 2004
  Large object request:could not locate 6,291,472 bytes of contiguous space / 166,021,456

Figure 5: IBM PMAT – Garbage Collection Analysis

This tool is used to analyze garbage collection log offline, that still not efficient enough to put us on track with the status of the application and what can be done rather than only raw readings that do not tell us what should be done.

As shown in Figure 5, it illustrates some readings that are good, but not good enough, because we need to know the pattern between the application and these readings.

Figure 6 shows the analysis of each garbage collection event.



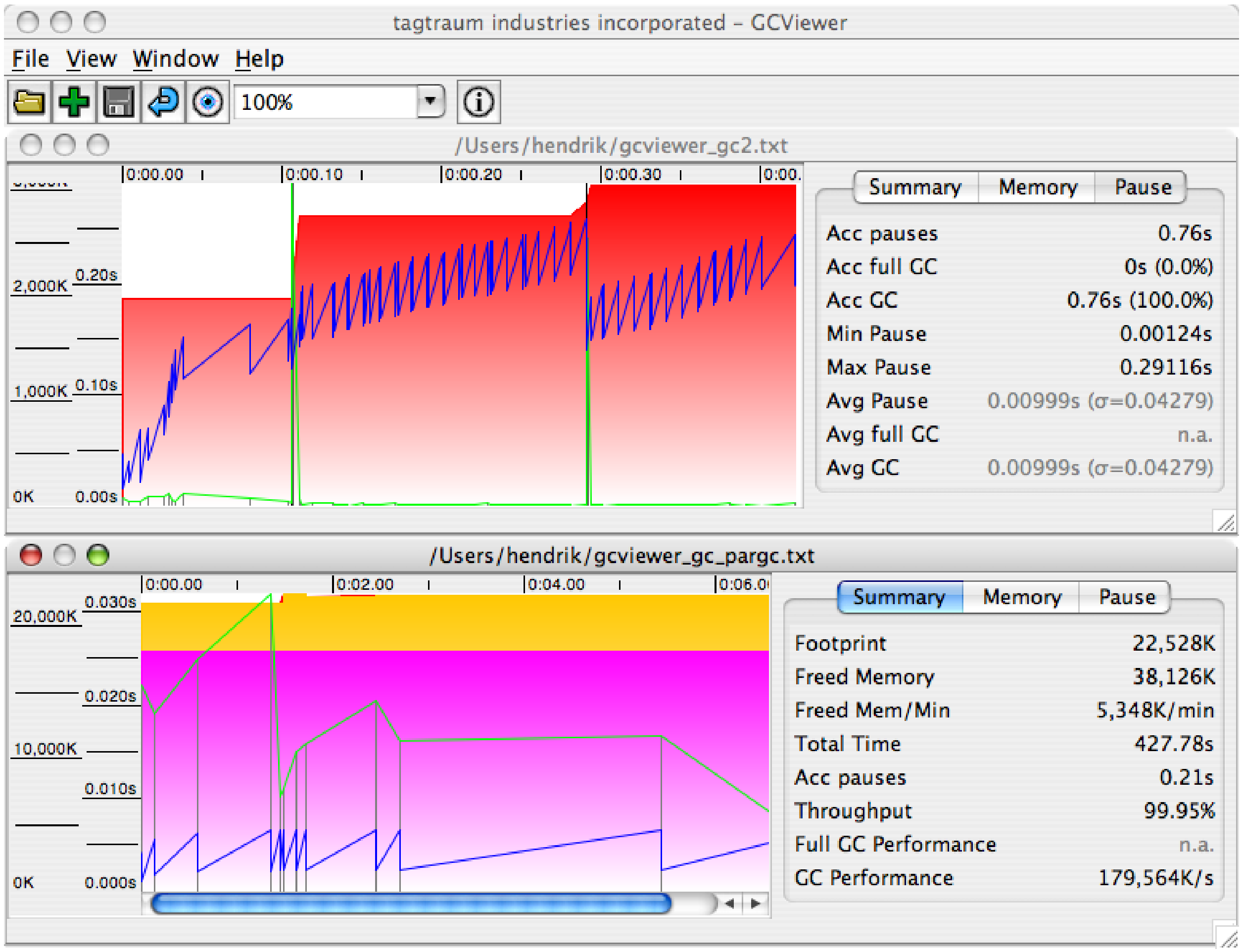
IBM Pattern Modeling and Analysis Tool for Java Garbage Collector
File Analysis View Help
Lab1.log GC View
GC View GC Table View

Free	Total	Needed	Freed	Free(Before)	Total(Before)	AF Completed	Time
71,250,112	150,403,584	1,352	65,050,640	0	142,883,408	328	Wed Nov 17
71,290,728	150,403,584	8,216	65,052,144	39,112	142,883,408	325	Wed Nov 17
71,483,312	150,403,584	8,208	65,257,496	26,344	142,883,408	325	Wed Nov 17
71,233,896	150,403,584	5,904	64,993,224	41,200	142,883,408	334	Wed Nov 17
71,340,016	150,403,584	16,400	65,027,296	113,248	142,883,408	336	Wed Nov 17
71,343,000	150,403,584	10,336	65,117,936	25,592	142,883,408	326	Wed Nov 17
74,790,064	150,403,584	528	68,590,592	0	142,883,408	309	Wed Nov 17
74,688,464	150,403,584	8,208	68,469,184	19,808	142,883,408	258	Wed Nov 17
75,974,656	150,403,584	10,336	69,751,640	23,544	142,883,408	303	Wed Nov 17
85,902,496	150,403,584	10,336	79,689,064	13,960	142,883,408	268	Wed Nov 17
92,235,248	150,403,584	8,208	85,388,184	77,208	142,883,408	234	Wed Nov 17
95,701,760	150,403,584	8,208	88,335,056	28,328	142,883,408	212	Wed Nov 17
95,218,200	150,403,584	8,208	87,728,952	21,128	142,883,408	214	Wed Nov 17
95,291,912	150,403,584	10,336	87,809,592	14,200	142,883,408	225	Wed Nov 17
95,706,960	150,403,584	8,208	88,174,856	63,984	142,883,408	225	Wed Nov 17
95,891,112	150,403,584	8,216	88,360,256	62,736	142,883,408	227	Wed Nov 17
100,221,568	150,403,584	528	43,762,496	56,459,072	150,403,584	139	Wed Nov 17
97,359,144	150,403,584	528	39,465,672	57,893,472	150,403,584	215	Wed Nov 17
86,558,184	149,420,544	528	80,065,248	0	142,883,408	354	Wed Nov 17

Figure 6: IBM PMAT – Garbage Collection Output

2.2.4.2 GCViewer 1.33

This is another tool used to analyze garbage collection log offline, that still not efficient enough to put us on track with the status of the application and what can be done rather than only raw readings that do not tell us what should be done as illustrated in Figure 7.


tagtraum industries incorporated - GCViewer
File View Window Help
100%
/Users/hendrik/gcviewer_gc2.txt
Summary Memory Pause
/Users/hendrik/gcviewer_gc_pargc.txt
Summary Memory Pause

Acc pauses	0.76s
Acc full GC	0s (0.0%)
Acc GC	0.76s (100.0%)
Min Pause	0.00124s
Max Pause	0.29116s
Avg Pause	0.00999s (σ=0.04279)
Avg full GC	n.a.
Avg GC	0.00999s (σ=0.04279)

Footprint	22,528K
Freed Memory	38,126K
Freed Mem/Min	5,348K/min
Total Time	427.78s
Acc pauses	0.21s
Throughput	99.95%
Full GC Performance	n.a.
GC Performance	179,564K/s

Figure 7: GCViewer Console

Before drilling into the concepts, algorithms and hypothesis that this research is going to present, it is needed first to point out and explain the memory and the garbage collection structure in Java.

Chapter Three – Java Memory Structure and Possible Issues

1.17 3.1 Chapter Introduction

This chapter will put us on track with the structure of the memory within any JVM. It is important to understand this structure in details in order to come up with a better understanding of the possible approaches towards memory tuning.

1.18 3.2 Java Objects Management

Java Enterprise Applications are Java based applications that are developed to serve enterprise needs. One of the most important features in the Java programming language is that the management of memory resources is left to Java itself.

The architecture of the Java language is designed to best serve the concept of automatic handling of the objects within an application. With time, objects maybe still referenced and still needed, other objects may not be needed anymore, so that's why within the Java language is a process called **“Garbage Collector”** that manages such operations.

Garbage collectors are of two types, thus, in order for the collector to operate efficiently, it was a must dividing the Java memory called “Heap” into generations, assigning each collector to each of these generations.

In order to understand how this work will help in determining and solving performance problems with the help of the “Garbage Collector” activities, it is needed first to explain the Java memory structure and how it is divided into different areas and what does each area hold.

It is very critical first to understand the basic features of the Java language regarding the collection process before presenting any solutions on how to resolve the known issues.

1.19 3.3 Memory Structure/Hierarchy

Any memory consists of the following types:

- Registers.
- RAM.
- Cache.

Memory in general is needed because it is known of its speed over traditional storage drives (Hard disk drives) for permanent storage, accessing data within memory area is much faster than loading and reading it from the permanent storage, that's why it is more expensive.

For this reason, we should carefully deal with such small/costly resource that is needed for any application.

1.20 3.4 The Java platform

3.4.1 Virtual Machine:

One of the most important concepts in the Java programming language is promoting it from being only a programming language to be an independent Virtual Machine that can manage itself.

As previously said, Java is not just a programming language only, it's a platform by which it can operates solely within the hosting machine if it grants the JVM some of the resources in order to operate.

Memory, CPU and storage are one of the common resources needed by the JVM.

This work will only target the memory aspect of an application because of its critical role in the availability and scalability of any application.

Any Java Virtual Machine (JVM) memory is divided into the following areas/segments:

- **Heap Memory:** The Heap memory is the runtime data area from which memory for all class instances and arrays are allocated. Heap memory is managed through an automatic process called "Garbage Collection"
- **The Non-Heap Memory:** Stores the runtime constant pool.

Being a platform itself, made it easy for programs to run on any platform, since the responsibility of making the same code runs on any platform is the responsibility of the Java platform.

At the startup of any Java program, the JVM gets some portion of the RAM memory from the operating system, as the shown in Figure 8; it is the general view of a JVM memory.

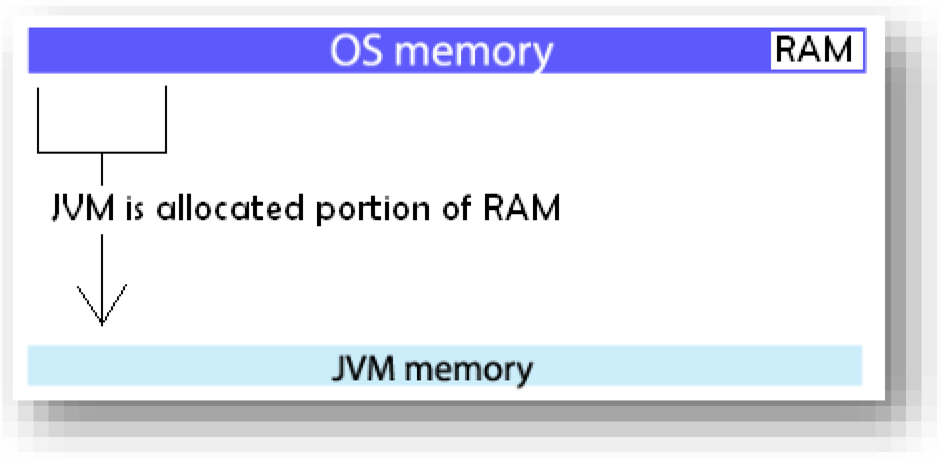


Figure 8: JVM Memory Allocation

The Heap memory is a pre-allocated memory from the RAM, at startup, the JVM is given a pre-allocated portion of RAM once, so it does not have to fetch free memory from the RAM

1.21 3.5 JVM Memory Structure

Since the Java language is not only a programming language, it's a platform that is self-contained and has the needed capabilities to manage its resources on its own; therefore it has its own memory structure.

3.5.1 JVM Memory Divisions

The JVM memory is divided into 2 main areas [5].

- Heap Memory.
    - ✓ Young Space.
    - ✓ Old Space.

- Non-Heap Memory.
    - ✓ Permanent Generation.
    - ✓ Code Cache.

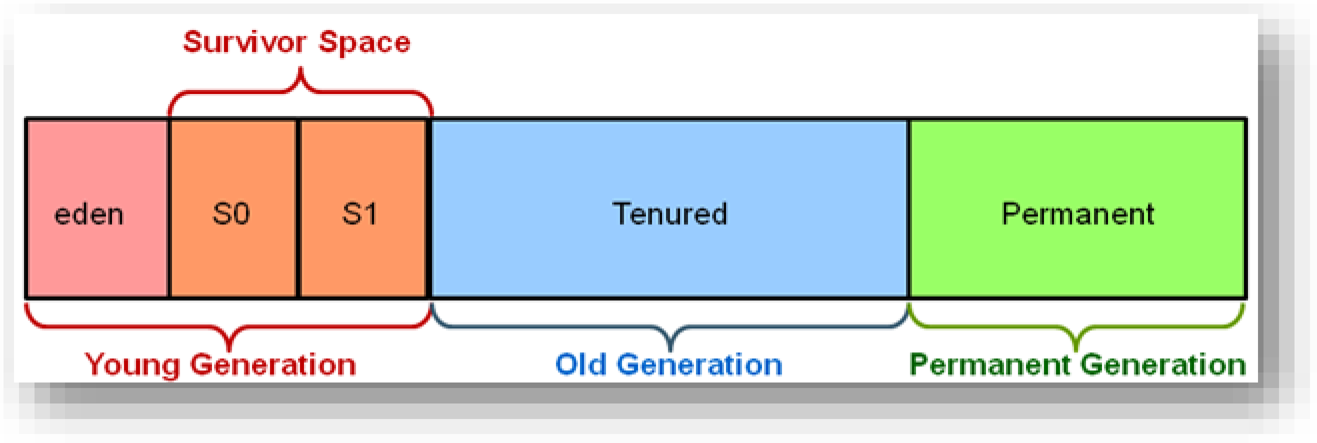


Figure 9: Heap Divisions

The Heap memory is where Java program data is stored. The Non-Heap space is where the meta-data required by the Java Virtual Machine is stored.

3.5.2 JVM Generations

A. The Young Space [5]

1. The Eden Space:

Any new object will be stored in the part of the Heap memory, the Eden space.

2. Survivor Spaces:

The objects that reside in this space are objects who survived through the process of garbage collection of the Eden Space.

Both the Eden Space and Survivor Space are called **“Young Generation”** space of the Heap.

The only difference is that Objects that age (Survive garbage collection are being moved from Eden to Survivor spaces)

B. The Old Space(Tenured Space) [5]

It is the pool that contains objects that existed for some time in the Survivor Space.

C. The Permanent Generation [5]

It is the area where the meta-data of an application is stored, whether this data was describing user data like his classes/methods or the standard library data.

Example: User’s Classes and Methods, even the Java library classes.

Someone could ask why the memory in Java has been divided into such spaces; they are divided in this way for two reasons:

1. Lifetime of Java objects.
2. The way in which the memory is being managed. (Algorithm or concept in which memory is managed or maintained).

The lifetime of any Java object is related to the garbage collector role in Java.

When we are talking about memory with the Java applications, it is needed to mention the important role of Garbage collection that is explained on different stages throughout this paper.

1.22 3.6 Heap Divisions

The most important part of the memory of a Java application is the Heap. It is the part that this work will focus on in terms of the performance tuning approaches.

Again, Heap is divided into 2 main parts:

1. Young Space.
2. Old Space.

The Young Space is where all new objects are allocated. It is called Young because only relatively new objects are stored in Young and kept there for a portion of time.

The Young as we saw is divided into two portions; Eden Space and Survivor Spaces.

Whenever a new object is created, it will be allocated to the Eden Space of the Young generation, which is the first stop of the newly created objects.

1.23 3.7 Garbage Collection

Java allows developers to create objects/variables without worrying about the cleanup of the resources that is; memory management. The process of allocation/de-allocation is handled by the Java platform itself.

Garbage Collection can be categorized under the concept of **Memory Management**.

Memory Management is the process of recognizing or identifying when the objects in memory are no longer needed. This process in the Java language is left to the Java platform itself, rather than being left to be handled by the programmer, since it may cause many issues such as memory leaks.

Garbage collector is responsible for:

- ✓ Allocating Memory
- ✓ Recover memory from objects that are no longer referenced.

Garbage collection solves many allocation/de-allocation problems, still there are some challenges that make this cleanup process a tough one, for example we can create objects indefinitely until there is no more memory available to handle this.

1.24 3.8 Memory Usage and Cleanup

With every trigger for a garbage collection, a log is generated accordingly. Before we take a closer look at how this log looks like or what does it contains, we need to know how we can benefit from it.

Garbage collection as this work pointed previously is the process of freeing up some chunks of the memory, whether it was of the Young or Old Space. With every trigger for a garbage collection, we can set a flag to log this operation.

From this log we can know very useful information that could tell us a lot about the health of the application, for instance, many Full or Minor GC events could mean a lot, it could mean that the garbage collection type is not efficient and should be changed, or it could mean to increase the allocated memory for the application or even it could mean to decrease the allocated memory for one of the Heap spaces.

Based on the researcher’s experience with many applications running on the Java platform, application owners tend to react to performance issues after the fact a problem rise. The application could be running and functioning well but maybe not up to the expectations, yet they would not know, with the correct circumstances, a performance bottleneck could easily show up eventually. The application could be suffering but a little, this is where we should focus on.

For instance, if an “Out of memory” error is thrown, the perfect solution is to have a “Heap dump” and/or “Threads dump” at the moment where there is an unbearable performance issues, or even; if an out of memory error is thrown.

But what about the overall behavior of the application, is everything is fine and doing well when no extreme issues rises up? An issue could be noticeable after hours or even days since the application's start up time, but it suddenly became a show stopper that should be solved and tends to be a severe issue that should be solved right away to be able to continue using the application.

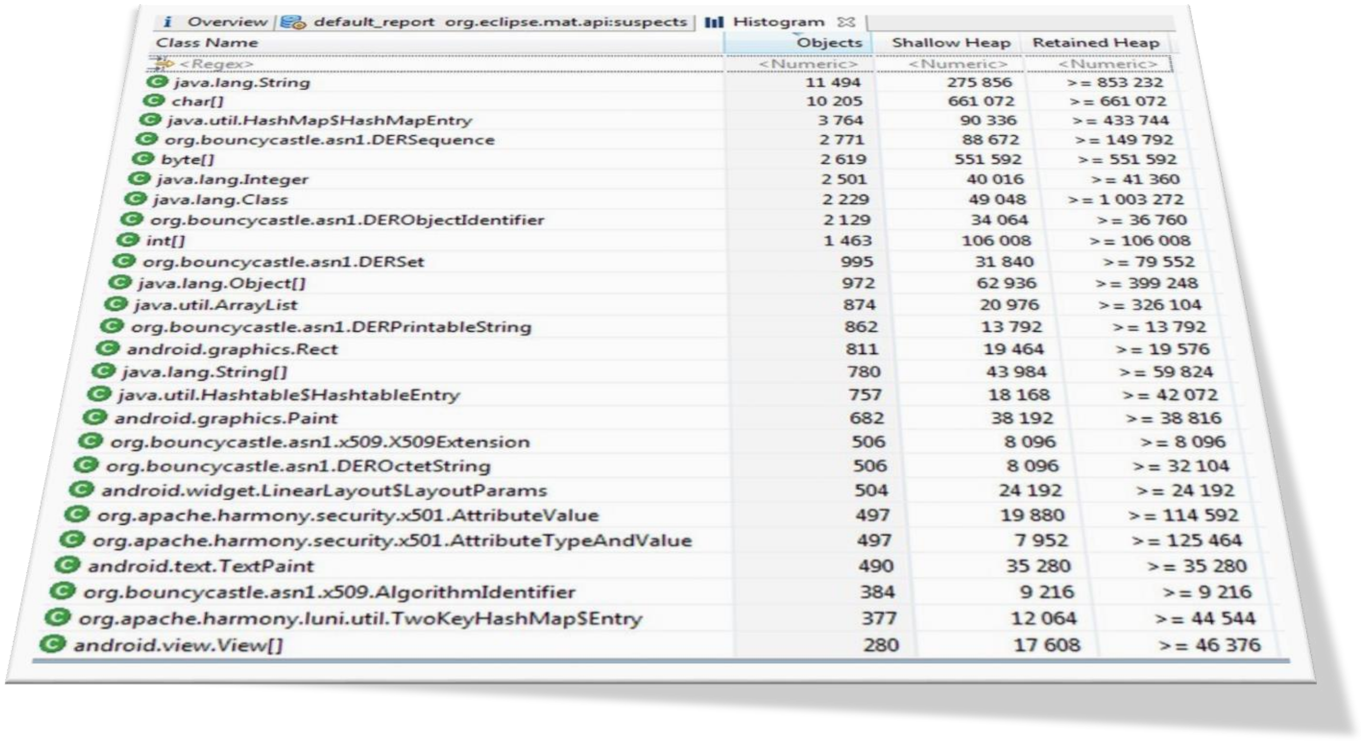
The researcher believes that we can do a lot and have useful information about the health of the application before a show-stopper issue rises. We can even predict what would happen with the current status of the application.

The researcher believes that such information that can be classified as "Corrective" and "Preventive" actions can be extracted from the Garbage collection log.

Preventive actions through which can give us an indication on how the application is going to behave soon or even later on. Corrective actions were when something starts to go wrong even if not yet emerged for the customer; we can act and solve it.

Figure 10 shows a sample Heap dump of an application, as illustrated, it shows the number of objects against each type and the Heap usage accordingly.

The Heap dump is a capture of the Heap usage at a specific time, so it only captures what objects were available at only certain point of execution, thus; it does not represent the overall status of the application, not even a close clue, it only shows at a certain point how the application is doing, it could be now going not bad, but we cannot know if there is something wrong about to happen



Overview | default_report org.eclipse.mat.api:suspects | Histogram

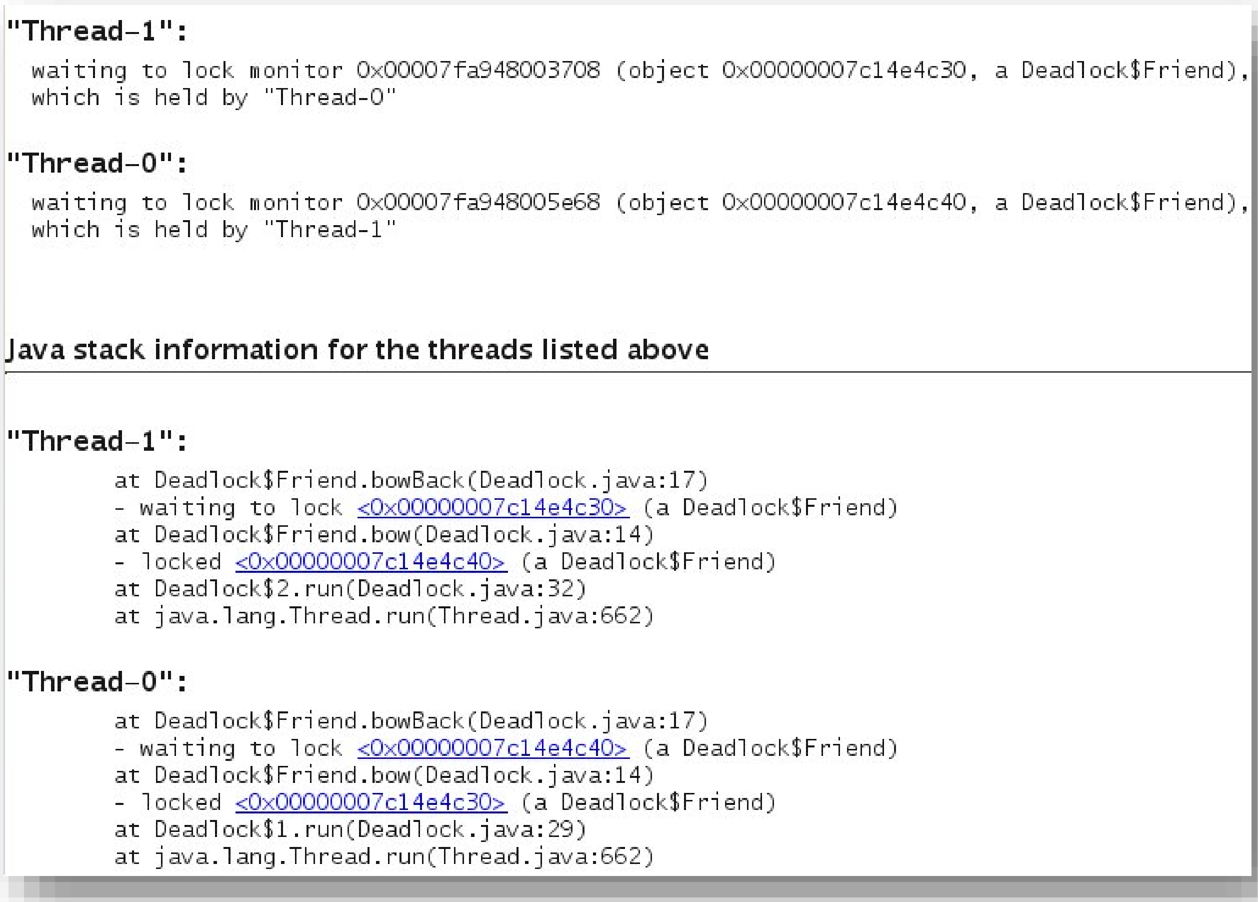
Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
java.lang.String	11 494	275 856	>= 853 232
char[]	10 205	661 072	>= 661 072
java.util.HashMap$HashMapEntry	3 764	90 336	>= 433 744
org.bouncycastle.asn1.DERSequence	2 771	88 672	>= 149 792
byte[]	2 619	551 592	>= 551 592
java.lang.Integer	2 501	40 016	>= 41 360
java.lang.Class	2 229	49 048	>= 1 003 272
org.bouncycastle.asn1.DERObjectIdentifier	2 129	34 064	>= 36 760
int[]	1 463	106 008	>= 106 008
org.bouncycastle.asn1.DERSet	995	31 840	>= 79 552
java.lang.Object[]	972	62 936	>= 399 248
java.util.ArrayList	874	20 976	>= 326 104
org.bouncycastle.asn1.DERPrintableString	862	13 792	>= 13 792
android.graphics.Rect	811	19 464	>= 19 576
java.lang.String[]	780	43 984	>= 59 824
java.util.Hashtable$HashtableEntry	757	18 168	>= 42 072
android.graphics.Paint	682	38 192	>= 38 816
org.bouncycastle.asn1.x509.X509Extension	506	8 096	>= 8 096
org.bouncycastle.asn1.DEROctetString	506	8 096	>= 32 104
android.widget.LinearLayout$LayoutParams	504	24 192	>= 24 192
org.apache.harmony.security.x501.AttributeValue	497	19 880	>= 114 592
org.apache.harmony.security.x501.AttributeTypeAndValue	497	7 952	>= 125 464
android.text.TextPaint	490	35 280	>= 35 280
org.bouncycastle.asn1.x509.AlgorithmIdentifier	384	9 216	>= 9 216
org.apache.harmony.luni.util.TwoKeyHashMap$Entry	377	12 064	>= 44 544
android.view.View[]	280	17 608	>= 46 376

Figure 10: Objects distribution

As said, this is helpful whenever an issue rises and must be fixed, the Heap dump is a consuming operation that cannot be done too frequent since it needs resources to operate.

THREADS DUMP

As shown in figure 11, Threads dump provide an image for the available running/blocked threads. From Threads dump, we can know instantly what threads are running. This technique is helpful to track performance issues that are not related to memory.



```
"Thread-1":
  waiting to lock monitor 0x00007fa948003708 (object 0x00000007c14e4c30, a Deadlock$Friend),
  which is held by "Thread-0"

"Thread-0":
  waiting to lock monitor 0x00007fa948005e68 (object 0x00000007c14e4c40, a Deadlock$Friend),
  which is held by "Thread-1"


Java stack information for the threads listed above
===================================================

"Thread-1":
        at Deadlock$Friend.bowBack(Deadlock.java:17)
        - waiting to lock <0x00000007c14e4c30> (a Deadlock$Friend)
        at Deadlock$Friend.bow(Deadlock.java:14)
        - locked <0x00000007c14e4c40> (a Deadlock$Friend)
        at Deadlock$2.run(Deadlock.java:32)
        at java.lang.Thread.run(Thread.java:662)

"Thread-0":
        at Deadlock$Friend.bowBack(Deadlock.java:17)
        - waiting to lock <0x00000007c14e4c40> (a Deadlock$Friend)
        at Deadlock$Friend.bow(Deadlock.java:14)
        - locked <0x00000007c14e4c30> (a Deadlock$Friend)
        at Deadlock$1.run(Deadlock.java:29)
        at java.lang.Thread.run(Thread.java:662)
```

Figure 11: Threads stack Trace

Now, if objects keep being promoted to the “Old Space” if finally full, garbage collection is triggered, this time of a different type, called “Major Garbage Collection” also known as “Full GC”. As the minor GC it is a “Stop the world” action, but since the size of the Old Space is much bigger than the Young Space, it is a costly operation, it will take longer than the Minor GC, thus a time consuming operation.

This research is going to present with a simple example, how could performance issues rise in the course of the application and are related to the code of the application or due to the configuration of the application itself.

This is important before starting to present any solution, in order to simplify the understanding of solutions afterwards.

It starts here...

Here are the following scenarios to put us all on track with how issues rise at the customer’s side and how do application owners (precisely development) reacts to such issues.

A customer let’s say X, has different users using the application, with time, users either start feeling that the application is less responsive and slower than usual. Users start complaining. With time, issues may become intolerable, so a complaint is sent to the customer’s support/IT department to check the issues out.

Customer’s support team receives the complaint and starts checking out the application, which not much to do about, so they start checking the server itself and its resources. This actually could give wrong solutions, for example; a CPU that is fully used could be interpreted that it is needed to increase the number of CPUs. A fully occupied memory may give wrong indication to increase the available memory.

Machine’s resources are limited, increasing the available resources is not always the solutions on the contrary, and doing that should have a stronger justified reason.

The work is going to explain how the Java platform manages its own resources.

1.25 3.9 JVM and Physical Machine Resources Sharing

Before knowing how to solve problems related to the resources, it is important to know how the JVM gets its resources and how they are consumed.

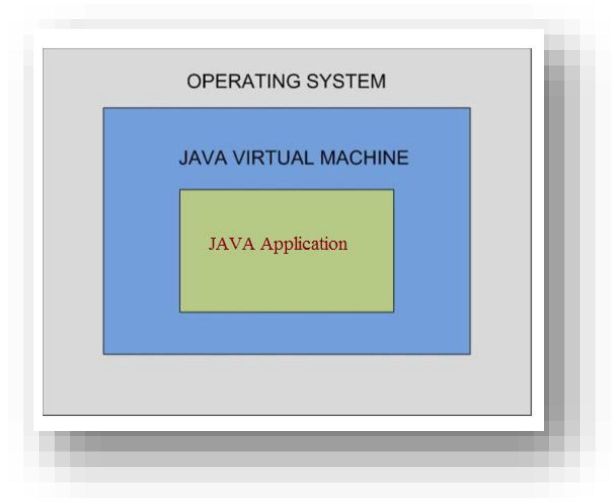


Figure 12 : Basic Java Application Architecture

Java applications are always hosted within Java Virtual machine (JVM), this gives the Java language superiority over other programming languages. This is illustrated in Figure 12.

Upon starting, the JVM request some resources from the “Operating System” in order for it to manage the application(s) it is going to host.

For instance, the virtual machine needs memory and CPU; let’s talk about the memory aspect.

Let’s take the following scenario as an example:

The Physical machine has a memory of about 3 GB of RAM. The application administrator chooses to set up the memory in accordance to the available memory, so if there are other programs or for example the database is also available on the same server, then approximately not more of 60 % of the available memory should be allocated to the application.

In brief, let us consider the application reserved about 1.5 GB of RAM; this amount is the maximum amount of memory to be reserved by the application. Now after the application starts, JVM reserve an initial portion of the memory to the application, let's say; 256 MB of RAM, with time it may need more and more, so the memory of the application keeps growing and growing.

If the application needs memory more than the maximum reserved, the OS will shut the JVM along with the application down because it is requesting more than what is allowed, if this is allowed, then it will affect other applications and may prevent other system processes from being able to run, that's why the Operating system chooses to shut it down.

But why after that someone done some estimation regarding the needed memory of the application, more memory is needed? Didn't the application owner recommend a recommended memory size?

What really caused this issue, what could be the causes of memory is not sufficient?

1.26 3.10 Memory Issues

As this work explained previously, if the application needs more memory than what it is allowed to reserve, then the application will be shut down, then someone could think that this is very likely to happen. This research will now explain that when the application is shut down because of the memory, it is not really as what it seems.

When the application's process gets terminated because of a memory related issue, it could be because of two reasons:

1. Insufficient Memory.
2. Inefficient usage of memory by the application.

These are two important concepts to classify performance issues related to memory

3.10.1 Insufficient Memory

Applications may crash (stop) because it needs more memory than the maximum configured/reserved memory for the applications.

Someone could simply say, well we can increase the memory, that could be true, but till when? We should know that memory (RAM) in any machine is a costly resource and it might not be feasible or even reasonable to ask the customer of the application to keep increasing the available memory for the application each time it needs more.

What really should happen is that the application's owners (developers) should define the scale or range of what this application needs, example; this application is classified that it needs memory up to 4 GB, 2 GB, 16 GB or whatever, what is important as said is to point out this information to the public or for each specific customer.

It is very important that this should be defined upfront, let's take another example, and suppose that the application is only supported on Linux operating system, would the application owners hide such important information? Or even that it could only run on a 64bit machines not on 32bit.

So as this information sounds very critical to be known by the customer prior to buying the application, it is also important to declare how much memory this application is expected to need.

So, someone could say now it's the role of the application's owner to specify this, the answer is yes, and also yes it would solve the problem, but not very precisely.

How could application's owners (developers) recommend the memory needed by the application?

It's not an easy job, the application developers could monitor how the application behaves in term of memory usage or even other resources usage which this work will not point out to, such as CPU usage, network usage, IO usage and DB usage.

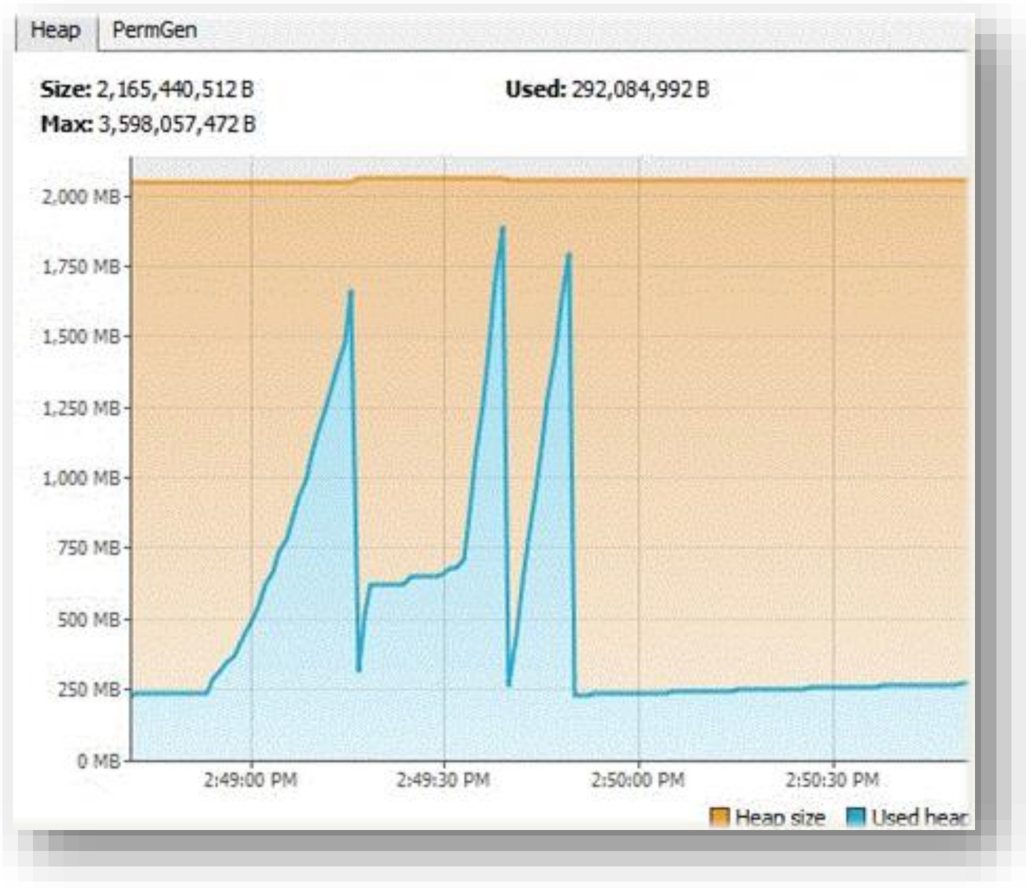


Figure 13: Heap memory normal usage.

For instance, Figure 13 shows the memory usage for an application, the developers could keep monitoring such readings about the memory while using the system normally.

This simulation of usage could give the developers an indication of how much the application needs memory, which is the output that they wish to recommend for their customers.

At this point, the application's owners are somehow comfortable with these recommendations, so they will provide the following necessary findings about the application:

- ✓ Work on Linux operating system.
- ✓ Work on 64 bit
- ✓ Recommended maximum memory size is for example: 2 GB.

This reliefs the application's owners from surprises later on, but does this mean that there will be no memory issues for the application? Does this mean that the customers will not suffer from the application being slow or down because of memory issues that are classified of **"Insufficient Memory"**

The answer to this question is yes but up to 70% - 80%, still not 100% coverage.

This is maybe due to the following reasons:

- ✓ Testing of the application may not have covered all of the scenarios of the business of the application.
- ✓ Customers may come up with scenarios that the developers & the application's owners could have missed.

- ✓ Load testing was not sufficient, the number of users that may uses the system may increase in a way that could increase the usage of the memory, hence, memory won’t be sufficient anymore.

As we noticed, it’s not an easy job at all. It requires tremendous efforts to come up with recommendations for the best memory setup.

So, for instance, if the customers deployed the application on the server and starts using it, if the recommended memory is found, an Out of Memory error means that it’s time to increase the memory resource for this application to keep functioning as expected.

3.10.2 Inefficient Memory Usage

This work explained in the previous section what is meant by an **“Out of Memory”** error and that it could be caused since no sufficient memory available or incorrect configuration of the application’s memory.

Now this work will explain the other possible cause for the **“Out of Memory”** error, which is the **“Inefficient Memory Usage”**.

Memory is available and it should be sufficient for the application, but suddenly it becomes not sufficient because it is being highly used but inefficiently by some of the application’s processes, giving a false indication regarding the need for an increase for the memory of the application.

Due to this, server’s administrators may choose to increase the memory available for the server and in this case, they are not solving the real cause of the issue, most probably when such issues occur, it is highly possible to recur.

Such misleading error “Out of memory” that is classified of “non-efficient” usage of the memory, the customer do not need to deal with, it should be sent for the application’s developers to do some investigation to know the real cause for this and solve it once and for all.

It is mainly related to either:

- ✓ Some parts of the applications may not be designed or developed efficiently.
- ✓ Correct memory setup, but not suitable memory cleaning algorithm.

Both reasons outlines inefficient memory usage, meaning that; memory usage is high in some cases or some areas while it shouldn’t be.

For example, a process which may be known that it shouldn’t consume more than let’s say 100MB, it is consuming up to 400MB, which as it sounds really bad, because there are other processes for different users that needs also memory usage.

Memory currently being used for updating bank accounts.

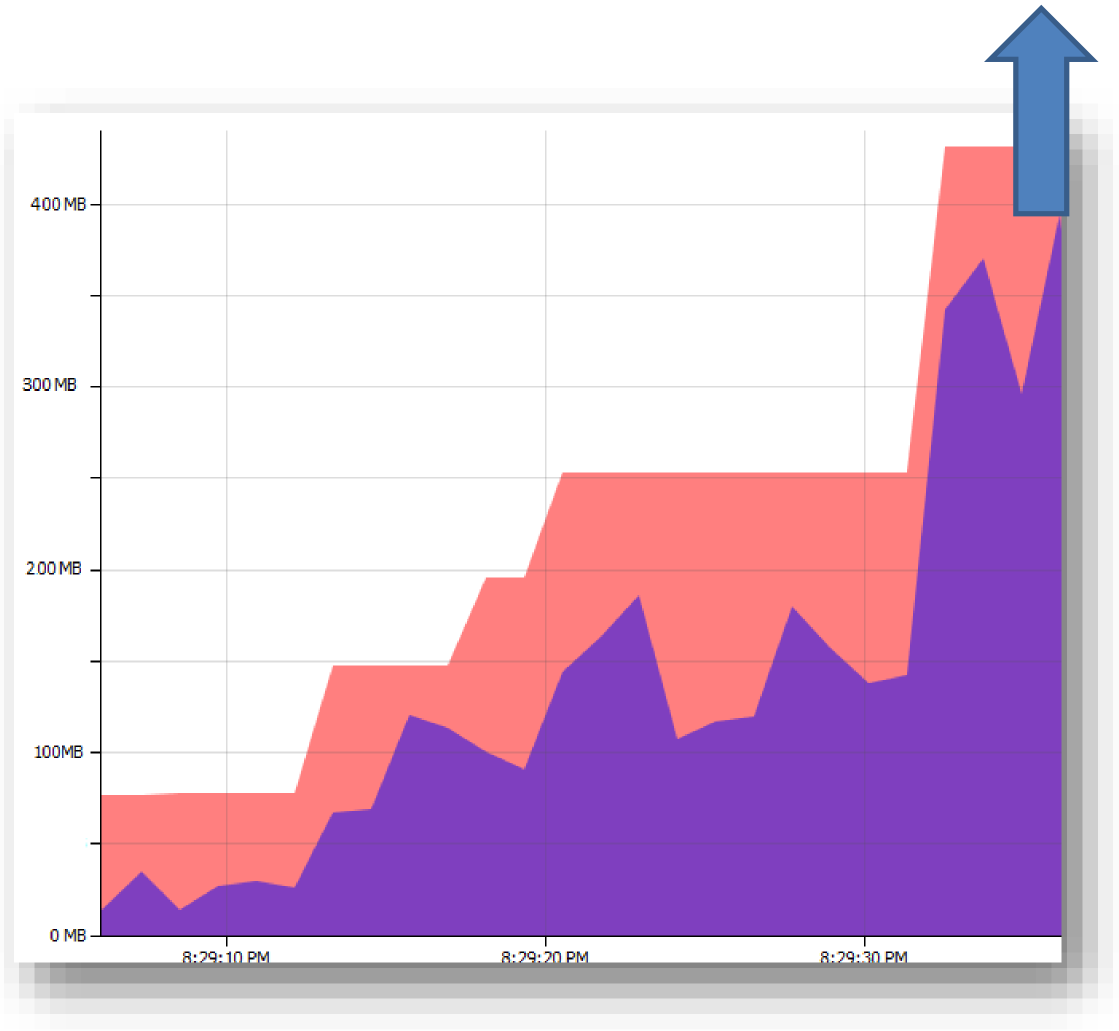


Figure 14: Big process memory consumption.

The application's owners or developers could choose to do some enhancements on the memory usage of an application, this process is called Tuning.

As Figure 14 illustrates, the memory usage for updating banks accounts nearly takes up to 400MB which the developers may consider it too much and think that they could enhance it so that it only uses up to 100MB of memory maximum in order to best utilize the memory resource.

The tuning this time is through enhancing the code written. This way, tuning will leave more memory to serve other areas of the application and even more users.

To summarize, the previous section was explaining memory issues related to bad design of parts in the code, which may lead to inefficient memory usage.

The research will point out to the second cause of such problem, which is:

> “Memory is setup well, but the chosen or the default cleaning algorithm is not suitable”

Before drilling down into this point, the work has to explain again in brief the cleaning process.

Each stage of the application execution and each process within the application need some resources in order to run. Since many processes could happen in the course of the application execution or even in seconds of time, allocation for memory for each process is done accordingly.

Memory allocated for each process within the application is being released within a process called “Garbage Collection”

In the following sections, the research will show examples of some programs producing different memory outputs.

Figure 15 illustrates an efficient usage of the memory by an application. As it is shown, the memory that is being used is shortly released to be used by another process or by another user.

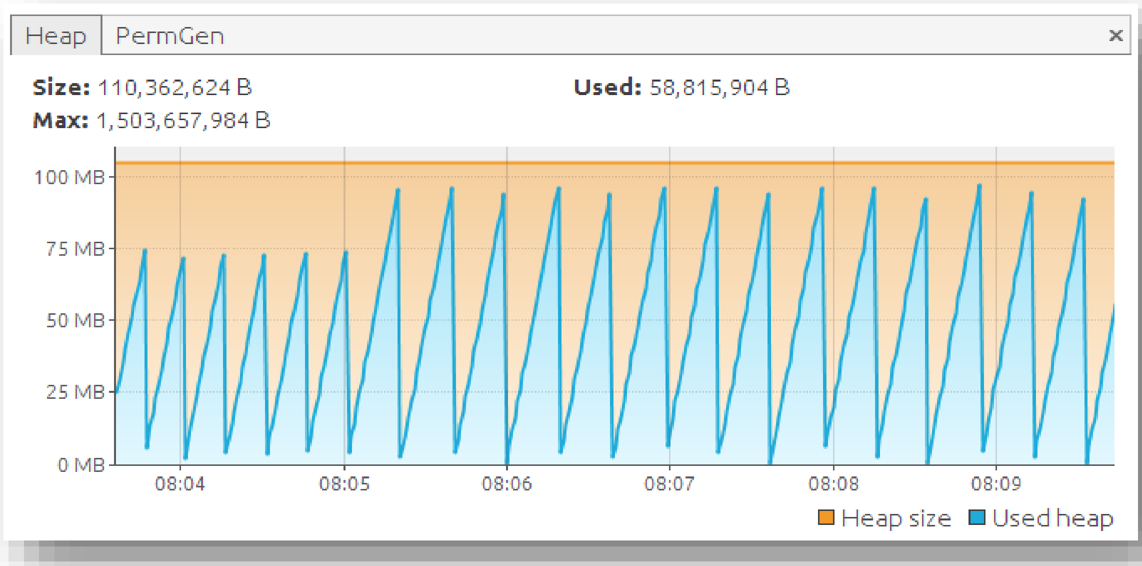
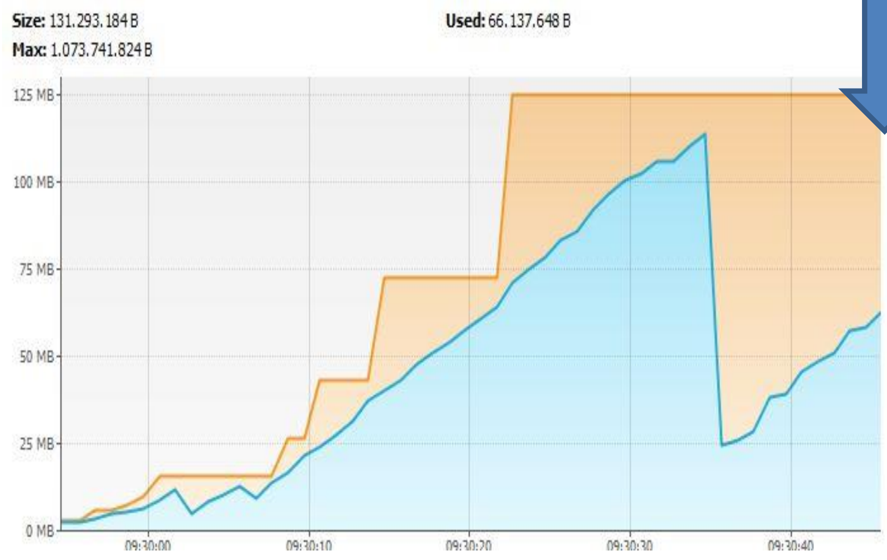


Figure 15: Efficient memory usage, no possible leak.

This is an efficient use of the memory, the spikes shows the usage of the memory, the spikes are followed with a decrease in memory usage, meaning that the used memory is now freed-up.

Memory is currently being used for the task.



Memory reserved for a process gets released afterwards. Meaning that: garbage

Figure 16: Efficient memory usage, no possible leak.

The process of which the memory usage decrease after the process is done is called garbage collection; it makes the usage decrease, so to show this more specifically:

Figure 17 illustrates an inefficient use of memory were the memory used is not getting released and so, it cannot be used by other process.

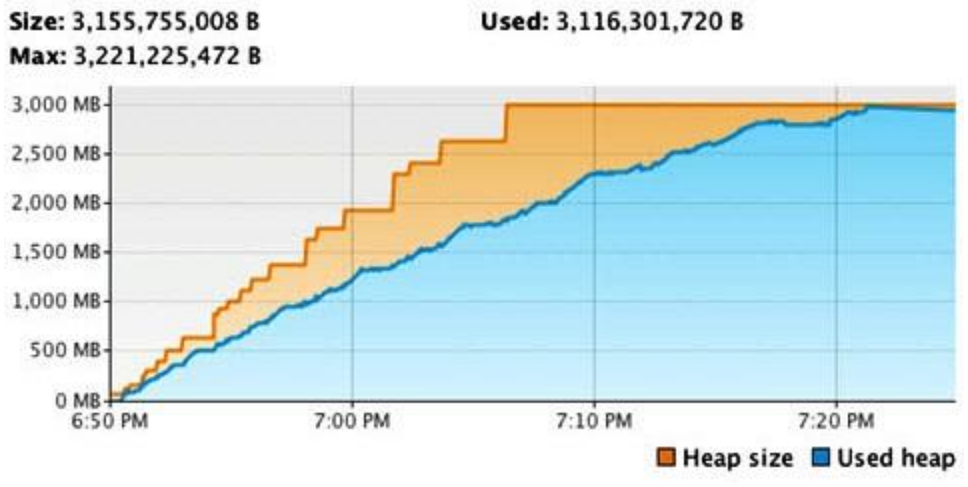


Figure 17: Non-efficient memory usage, possible leak.

The figure shows that memory is increasing over time without decreasing, which is very un-normal, but we should understand first when memory get released and when it’s not.

Let’s return to our previous example, updating bank accounts for users, suppose that this operation take 5 minutes to accomplish and uses about 300 MB of memory, what should happen here is that after the process is done, the 300MB should be released to be used again for another process. It should, meaning that this is the expected behavior, but is there a case where they don’t get released back?

Actually there is, it happens a lot and when happens, nothing you could do about it. This is referred to as **“Memory Leak”.**

This happens most of the time when code is not designed well, in which some of the Java objects keeps referencing the objects of the done process, when this happens, the garbage collector sees that there are other objects referring the objects of the process, at this point, the garbage collector is unable to reclaim back the memory of this process, hence memory won't be released, this is called a leak in the memory. Memory leaked won't be released until application is restarted.

But what is the harm from this, if a memory leak happens; does it mean that the application will not be able to run?

With time yes, so if the initial application memory needed was 100 MB, it is now 300 MB since memory reserved was not released. This means that less memory is now available for the application which means that with time, it is very likely that an **"Out of Memory"** error will happen.

As shown in Figure 18, it shows a memory leak for an application. It shows that after a process completes, memory does not return back to original Heap size, meaning that; the application lost over 1 GB as the figure shows and cannot be reclaimed back throughout the whole application.

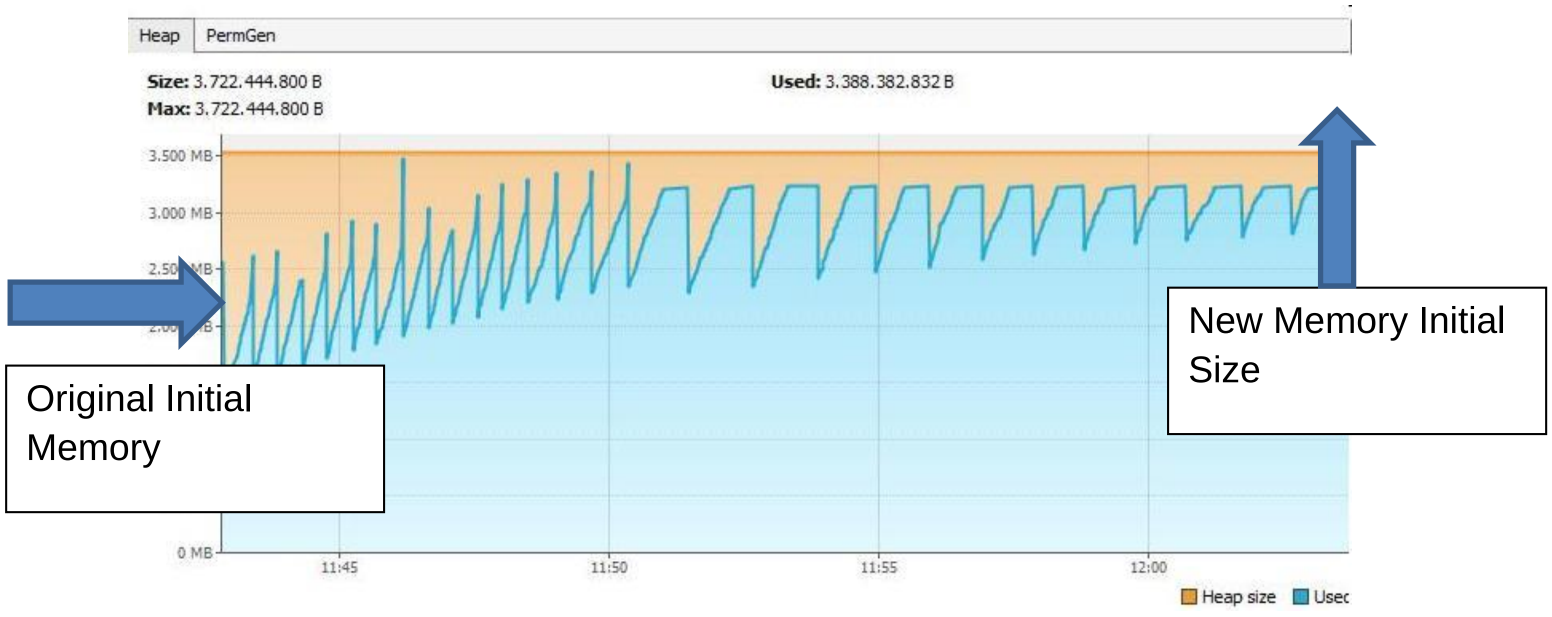


Figure 18: In-efficient memory usage, possible leak.

So in brief, an application not designed well could result in:

- ✓ Inefficient use of memory, as an example to remind us all: a process that is using 300MB while it should be only up to 100MB.
- ✓ Code not designed well that keeps referencing objects and not releasing them.

This is how the "Out of Memory" errors are classified, either:

- ✓ Insufficient Memory available.
- ✓ Inefficient Memory usage.

Now that we defined the categories, there should be no problem, but again there is.

When an application crashes (shuts down) because of a memory issue, the error thrown is:

"Out of Memory" error, there is no way to know if the memory really insufficient or because of inefficient use of memory.

In order to know the real reason behind this error, the developers have to do some thorough investigation, which is time consuming.

The researcher will present a way to know such detail and even more, with the help of a tool newly designed that is built on some algorithms that will be explained separately.

In order for the researcher to explain his theories and points, it is needed first to explain how the garbage collection process happens.

1.27 3.11 Object's Memory Allocation & Lifecycle

Garbage collection is a process about reclaiming objects that no longer needed, this process is triggered automatically by the JVM in a specific scenario.

Allocation is the process of allocating memory to certain process or task.

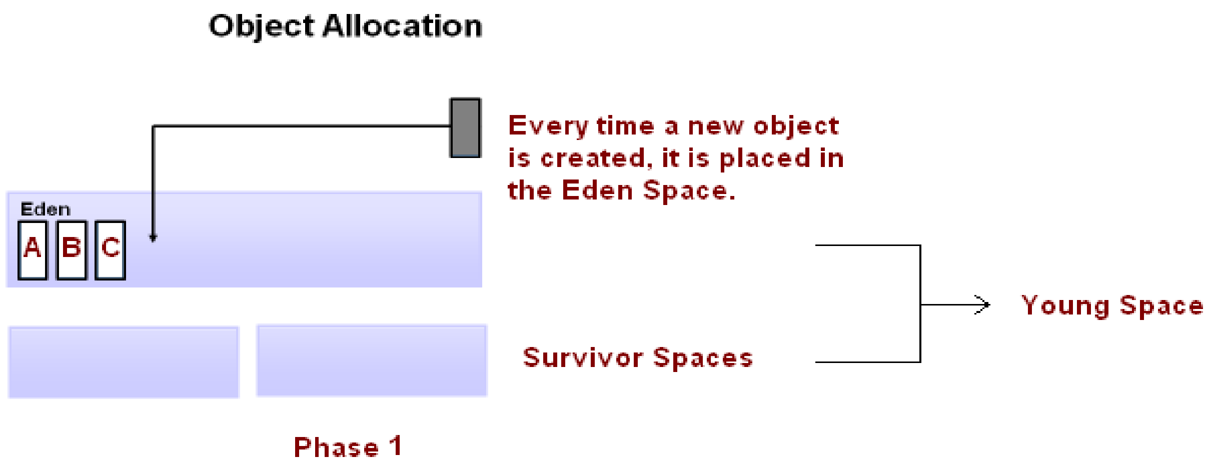
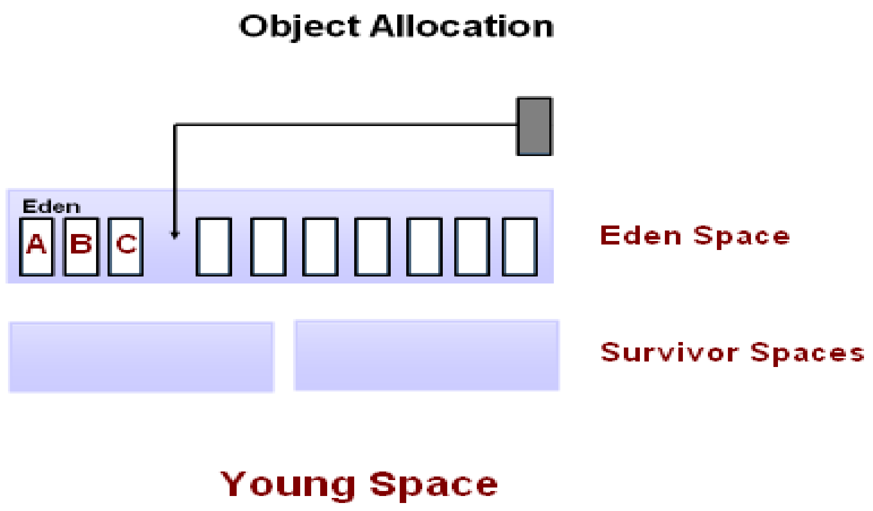

Object Allocation
Every time a new object is created, it is placed in the Eden Space.
Eden
A
B
C
Survivor Spaces
Young Space
Phase 1

Figure 19: Object Allocation within Heap memory – Phase 1

As shown in Figure 19, as soon as objects are created, they are placed in



the Eden space.

Figure 20: Object Allocation within Heap memory – Phase 2

With time, Eden Space gets full with objects and no more objects could be allocated into the eden space, so this space should be cleared out in a process of two stages:

1. Remove\Delete objects that are no longer used (Unreferenced Objects).
2. Move objects that are still used to the “Survivor Space” since they are still referenced.

This type of garbage collection is called Minor Garbage Collection.

When the minor garbage collection is triggered, it checks all objects in the Eden, if an object is still referenced, it won’t be deleted, which means it is currently in use. Referenced objects will be moved to another space called “Survivor Space” meaning; object survived a garbage collection.

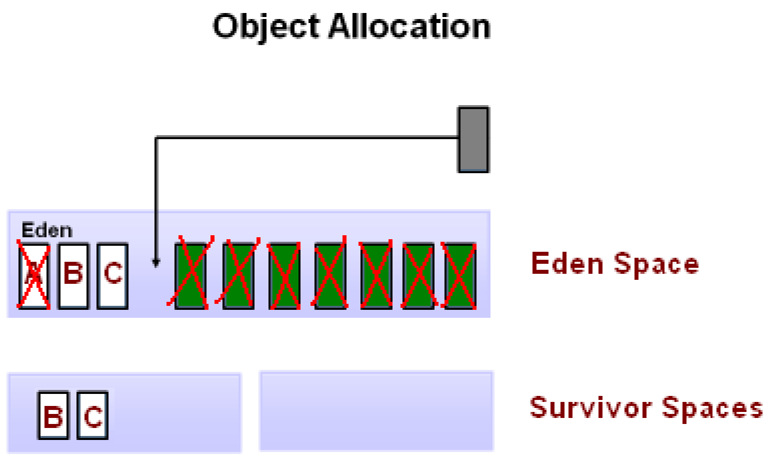


Figure 21: Objects allocation from Eden to Survivor Spaces

At certain point, after the minor gc is finished, Eden Space is emptied.

The process is repeated as long as the program is executing, newly created objects are kept assigned to the Eden Space, as soon as it is full; a minor garbage collection is triggered.

A normal program execution witnesses many and many Minor garbage collections.

At the second or later minor garbage collections, objects that were moved in a previous garbage collection to survivor spaces, will have their age incremented by one and will be moved to the next Survivor space, which we need to remember, still part of the Young space.

Meaning that, each time an object survive a minor garbage collection will move to the next Survivor Space. At the end of each garbage collection we can notice that Eden Space always cleared out and at least one Survivor space as well.

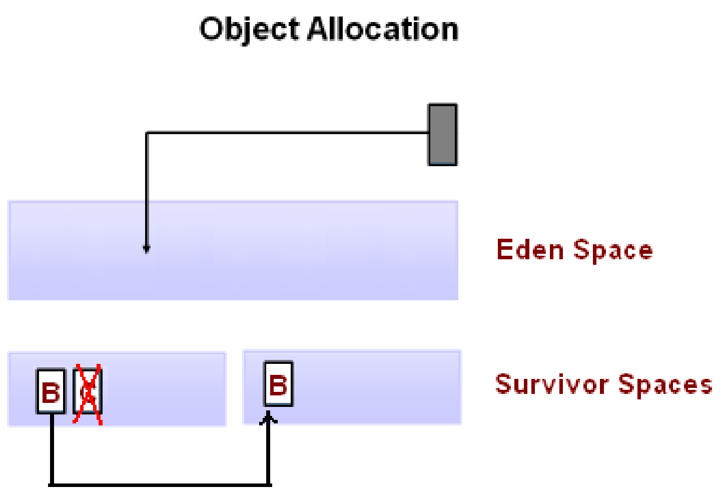


Figure 22: Object Allocation within Survivor spaces.

As shown in Figure 22, B is the only object in this example which survived as illustrated more than one minor gc, in this case; 2 garbage collections

An important thing to point out here is that when the garbage collection process is triggered, it means nothing else is running, program execution pauses during the garbage collection, which is unnoticeable.

If one of the still referenced objects exceeded the maximum age threshold**,** **“Tenuring Threshold”**, the objects will be moved to the “Old Space” or “Tenured Space” in a process called “Promotion”.

If the age threshold “Tenuring Threshold” was set to 15, then when an object age reaches 15 and is still referenced, it will be moved out of the Young Space to the Old Space.

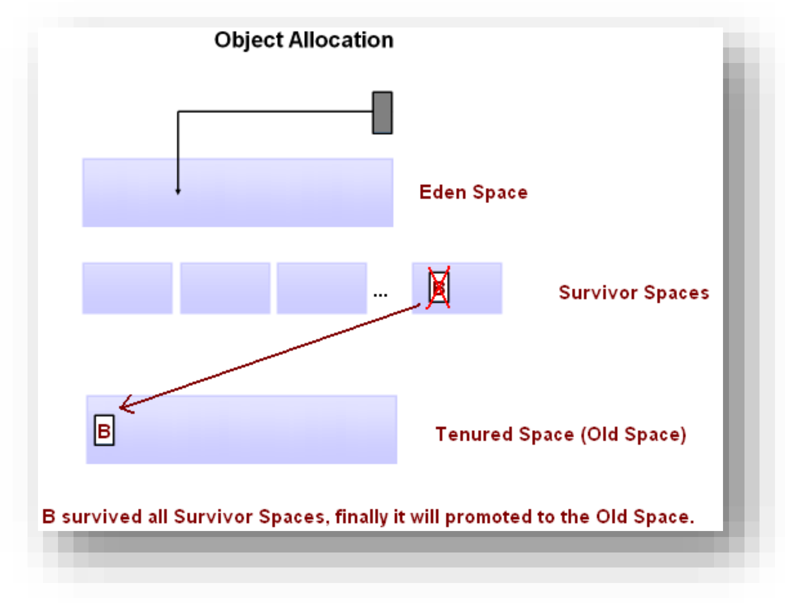

Object Allocation
Eden Space
...
Survivor Spaces
B
Tenured Space (Old Space)
B survived all Survivor Spaces, finally it will promoted to the Old Space.

Figure 23: Objects Promotion

So in brief, allocation is when allocating new objects into Young Space (specifically into Eden)

Promotion: Moving objects from Young Space to Old Space.

Allocation

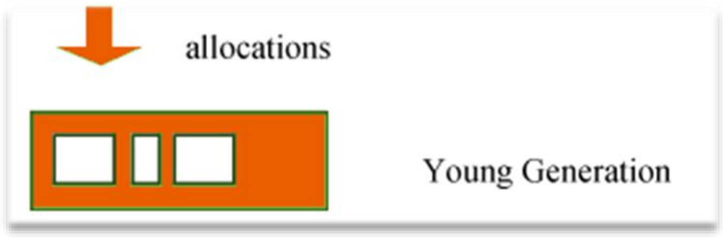


Figure 24: Objects Allocation

Allocation: When new objects are allocated memory in the Young Generation.

Promotion

Promotion: If objects survived a specific threshold of minor garbage collections, they are moved to the Old generation from the Young generation in a process called promotion.

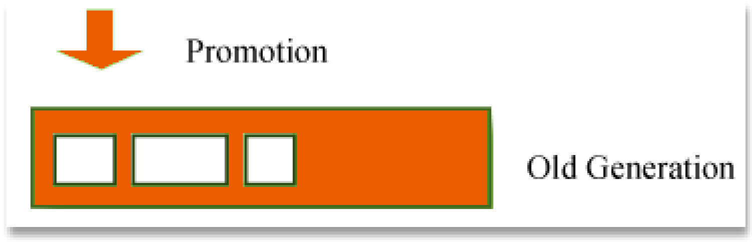


Figure 25: Objects Promotion

1.28 3.12 Garbage Collection Log

With each garbage collection activity, a log is generated accordingly. With each line of log, it tells us a lot, from the freed memory to the available to the different areas of the heap.

As an example, here is a sample line of the log file:

Sample GC Log Output

```
4.636: [GC [PSYoungGen: 230400K->19135KB(268800KB)] 230400K->19135KB(2058752KB), 0.0635710 secs]
```

If we zoom in on the important information in the GC Log:

230400KB->19135K(268800K)

268800KB >> is the Young generation size (256 MB)

230400KB >> Is the Young Generation size **BEFORE** garbage collection **(255 MB)**

19135KB >> is the amount of memory **FREED** after garbage collection **(18MB)**

The numbers before and after the arrow (e.g., 325407KB->83000KB from the first line) indicate the combined size of live objects before and after garbage collection, respectively.

1.29 3.13 Collection Types

Let’s in brief summarize what spaces the garbage collector is responsible of:

Young generation: Most of the newly created objects are located here. Since most objects soon become unreachable, many objects are created in the young generation, and then disappear. When objects disappear from this area, we say a "**minor GC**" has occurred.

Old generation: The objects that did not become unreachable and survived from the young generation are copied here. It is generally larger than the young generation. As it is bigger in size, the GC occurs less frequently than in the young generation. When objects disappear from the old generation, we say a "**major GC**" (or a "**full GC**") has occurred.

1.30 3.14 What Minor GC tells us?

Minor Collection is triggered when then JVM is unable to allocate space for new objects.

Young space is where the short living objects are placed.

It is called young because it is the place where objects which live for a short period of time are placed in.

Most of the objects are shortly living objects, which means that they live for a short period of time, for this reason, most of the garbage collections activities are of type **“Minor GC”.**

The Java virtual machine (JVM) will be suspended for the duration of the minor garbage collection; it’s a stop-the-world activity, even that they are considered fast garbage collection.

Now since minor garbage collections are too many and they are fast, they affect the overall response time of the application. Let’s always remember, it is fast because Young space is smaller than Old space, hence, it is quicker.

They are too small to cope with all the objects available, that’s why minor garbage collection is triggered very frequently.

Whenever the application is under heavy load, minor garbage collections could be very frequent; many frequent minor garbage collections could be as bad as a single long-lasting one.

So, it is not good to have frequent young garbage collections, the researcher will explain the possible root causes for the frequent young garbage collections:

1. Too small Young space considering the application’s load
2. Too many objects allocated very quickly.

3.

In both cases, young space fills up quickly, this triggers minor garbage collection

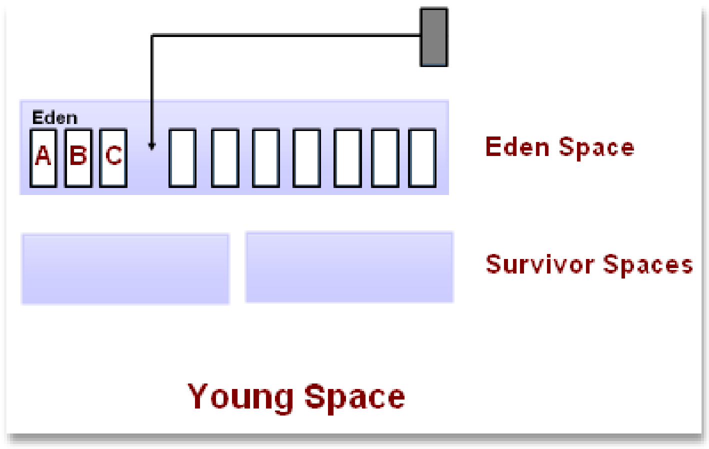


Figure 26: Minor garbage collection

Chapter Four: The proposed solution

1.31 4.1 Chapter Introduction - How this work helps out!

Again, it should be known that there is no other mechanism to depend on to know the overall progress of the application; it is very hard and costly to depend on other mechanisms as:

Heap Dump for available objects within the memory, it can be configured to be an automated process to generate a dump within a specific interval, but this is very consuming operation especially on a production environment.

Memory Profiling: Profiling is good but it is only efficient per process, to monitor the memory consumption of an operation.

CPU Profiling: same as the memory profiling.

Let us take an example on a performance problem and how things could be mistakenly interpreted.

Suppose that we have an application where the user is trying to submit his information to the application. The customers started to experience some delays in the response, as a result, the application developer's starts with an approach to record the duration of each process to know where the bottleneck is.

They seek the round trip time of the process, which is not costly, but it won't be a straightforward operation, meaning that; if a new business process is introduced, then there is no direct way to measure it without doing manipulation on the code.

While this approach gives an indication of the response time, it might be a false one, meaning that a high response time may not be caused by the operation itself but maybe caused by other operations causing the server to slow down, thus giving wrong indications and affecting corrective decisions.

As another example, the application could be performing well, but suddenly with the correct parameters, things could go wrong, it might be either because of wrong configurations for the memory or for the garbage collector.

So we need a way to catch the source of the issue, or at least, as in this case, we need a way to know how the application is behaving without or even before the customer starts complaining.

The theories and ideas within this work helps in solving the problem which faces the application developers in efficiently reading the gc log file trying to discover the patterns of the memory usage/setup and the garbage collector and finally trying to know if there is a problem or not.

1.32 4.2 Garbage Collection Log Analyzer to the rescue!

After many researches that were done by the research on the best way to monitor the performance of a Java based web application, many ideas cross the researcher’s mind.

If No one is complaining, this does not mean that everything is fine.

We were looking and the problems always from the wrong side, we were reactive to the problem rather than proactive.

This approach and way of thinking should be changed, we should be reactive before any problem rises, an application performance that **seems** good may not be actually good, customers who are fine with the performance of the application do not know what is really going on behind the scenes, thus, may give wrong indication to the software owners that things seems fine.

Problems maybe there, but, no one may know about it, and if we needed to do so and monitor each operation then this is unreasonable and undoable. Getting out with a feasible approach was very tricky and hard in a way that the application owners were comfortable with the fact that **if customers do not complain now then this means that everything is normal**

From the researcher's experience, this is not quite right, from his experience, he learned that the performance issues are like a "time bomb", if it's there and left, it would eventually explode, the only difference here is that, we did not know about if it's there, or we did not have the mechanism to know, until now. So, again there might be an indication to an issue, but with the right circumstances and parameters, it will appear eventually, then, it would be a severe issue that needs to be solved right away.

One of the major and most important activities within the life cycle of an application is garbage collection, where in brief; it's the process of cleaning objects no longer needed and maintaining and controlling old objects for the processes that still needs them.

The collection operation and its details can be documented and reserved for getting back to it later on; it shows the activities of the garbage collector and the available/consumed memory at each stage of the collection.

An efficient garbage collection process helps maintaining the application's performance by which it does not adds additional cost or complexity to the application, but also, if there are some issues with the application, it can be known from the behavior of the garbage collector.

But also, many times, the application suffers from performance issues from incorrect configuration of the application, meaning that the memory that is configured for the application maybe insufficient so it needs to be increased.

In other areas, memory that seems insufficient maybe because the memory usage is inefficient due to memory leak.

Customers or even technical developers may not bother to check for things that are not very critical, meaning that, as far as that the customer seems fine, and then nothing should be done.

The researcher supports his ideas and theories with a tool that helps in solving the problem that faces the application developers in efficiently reading the gc log file trying to discover the patterns of the memory and the garbage collector and finally trying to know if there is a problem or not.

1.33 4.3 Reading GC Log file

The problem here is that reading such file is hard; it is very technical and requires tremendous efforts to analyze and understand.

There are many approaches/tools designed currently, some of them are mentioned

From the researcher’s experience, we can depend on this file to check the overall progress of the application in which:

- The rate of collections.
- The pauses duration.
- Memory before collection.
- Memory after collection.
- Object’s life cycle (Its trip between the different areas of the Heap memory until its death)
- Memory allocation.

1.34 4.4 Yesterday’s weather

From the customers’ perspective, the application is considered performing well until users start experiencing a noticeable slowness within the application. This theory is correct only from the customer’s point of view, but when it comes to memory related issues, the same concept might not hold. Memory issues part of the application’s performance have different interpretations for the current situation.

Application may be suffering from memory issues but at this point, it is un-noticeable. Meaning that in many cases, memory problems (if exist) gradually increase with time until it reach a point where it starts affecting the performance of the application and finally become noticeable for the users.

Memory related issues may even be available within for long time within some applications and may not be noticeable until certain scenarios are available.

We need a new approach to catch up these issues before a low severity issues becomes a real severe issues, this work does not intend to predict memory issues, but it aims to report memory issues with a minimal cost with high level of correctness. **The available tools does not provide a way to analyze the file, it rather provide some discrete readings that still the developer need to do an analysis job which might be hard and time consuming.**

The only approach that can give us such readings is the Garbage Collection Log.

So, this is where the advantage of the garbage collection log lies in which, if there are misconfigurations for the memory, CPU, the garbage collector and its activities we can know about it and predict how it will behave with specific parameters.

This shifts the way of solving the performance issues from being corrective to preventive, from being reactive to proactive.

As said before, other means of performance monitoring and measurements shows the problem but not the root cause of the problem, making them not much of a help.

How the new approach helps out from both sides:

- **Corrective Actions:** With the help of the theories suggested by this work, the researcher designed a tool based on these theories to find performance issues if exist and give recommendations on how to solve them.
- **Preventive Actions:** It provides precautions in which if it finds anything that it seems may cause performance issues, it will report them and give recommendations on how to fix future issues.

Both are supported also with the use of visual charts to help give the bottom line of the status of the health of the application.

An important thing to remember here, the tools developed so far that reads the gc log are designed to illustrate the different outputs of the log visually, still it is required that a developer study it and analyze the log file, this work will come up with some theories based on current rules from Oracle and provide a tool that reads up these outputs and also suggest what should be done in an automatic way, rather than having developers analyze and study the log file.

The parameters to configure the settings of the application are many, they are very critical and important, if not setup correctly could lead to a struggling application. Because of these reasons, it is high probable that the application developers would mistake with the configurations of the application; therefore, we need a tool that is built on the standards to help figuring out the following:

1. Check if there are some missing parameters. (Section 4.4.1)
2. Check if there are some wrong parameters used. (Section 4.4.1)
3. Identify usage of parameters causing performance issues and suggest the alternatives. (Sections 4.4.2.1 and 4.4.2.2)
4. Identify the application's health and how it's performing. (Sections 4.4.2.1 and 4.4.2.2)

In order to for the tool that the researcher has developed to function well and to read the necessary garbage collection output.

The researcher helps out with his tool to discover if things are set-up right and the JVM parameters to produce garbage collection log is found.

4.4.1 Prerequisite Parameters

As said, one of the roles of the new tool is to discover if there are some missing or wrong parameters, let's start with the most necessary ones and explain the importance of their existence on the application's performance.

If necessary parameters are not found, the tool will issue a warning to turn it on using the below parameters.

Logging Parameters

The new tool cannot work at all without first making sure that there is a constant log file for the garbage collection process to build up the needed statistics and findings regarding the performance of the application.

"-Xloggc:c:\gclogFile.vgc -XX:+PrintGCDetails   -XX:+PrintGCTimeStamps"

The first parameter "**Xloggc:c:\gclogFile.vgc"** is used to

Used to allow logging (writing) the activities of the garbage collection process on a file so that the new tool could analyze this log.

It is highly recommended that in order for the new tool to give a high level of accuracy, the produced log file should be of a running application for at least 24 hours of time, this is recommended because the use of the application cannot be determined against a specific time of the day; most of the applications have a time where there is a peak usage of the application throughout the day.

Current approaches reads GC Log file only at application's crash which again is reactive rather than proactive, also as said, the available tools does not provide a way to analyze the file, it rather provide some discrete readings that still the developer need to do an analysis job which might be hard and time consuming.

The second one "**-XX:+PrintGCDetails"** is used to enable logging the details of the garbage collection process which includes the memory before and after, different areas of the heap and so on.

The third parameter "**-XX: +PrintGCTimeStamps"** is used to enable logging the time and the duration of the collection, this enables us to know the pausing duration of the collection.

These parameters and other parameters that this work will be explaining are all defined from within the Java language itself, each has a specific need.

This is a sample output for one of the garbage collection activities:

```
4.636: [GC [PSYoungGen: 230400KB->19135KB(268800KB)] 230400KB-
>19135KB(2058752KB), 0.0635710 secs] [Times: user=0.08 sys=0.01,
real=0.06 secs]
```

So, if we did not have this parameter "**Xloggc:c:\gclogFile.vgc"** there would be no such output.

If we did not have this parameter "**-XX:+PrintGCDetails",** then details of the collection.

If we did not have this parameter “**-XX:+PrintGCTimeStamps”,** there would be no timing information.

1. **Machine Classification & Category**

The type of the machine is very important for the application and how it performs over time.

For instance “server” class machines have hardware specifications and capabilities beyond the normal machines, which means; deploying the application on a server-class machine have a different effect than deploying it on a client-class machine.

As we know servers often have multiple processors and the processing architecture is very different than normal machines in which we gain the high computing, response and performance.

Why this is being pointed out? It is correct that the application will behave better when deployed on a server-class machine, but the researcher wants to point out that the garbage collection process is also affected by such an addition. The collection process itself will highly perform in general in this case; the researcher will explain each in details.

For this reason, the researcher highly recommends to make use of such a configuration that from his own experience with different applications and different customers, there are good percentage of them forget to make use of this parameter.

How this work helps out is by finding possible misconfiguration that could lead to bad performance and generate a hint message on how to fix such issue.

For instance, if the machine is a server-class machine and the used property is a “-client” then this will be caught by the tool in order to make use of the server capabilities.

4.4.2 Garbage Collections & Their Types

JVM periodically calls the garbage collector to remove objects that are no longer needed. It is triggered by the Young & Old spaces.

Whenever the Young space is filled up, a minor garbage collection is triggered to clean up the “Eden Space” as this work explained previously.

Whenever the Old (Tenured) space is filled up, a major garbage collection is triggered to clean up the “Old Space”, what also triggers a Full GC

- Calling System.gc()
- Permanent Generation is too low
- Running out of old gen

Let’s start with the minor garbage collection.

Throughout the researcher’s experience with different applications for different customers, he has found many issues that applications suffer from due to configurations of garbage collections or even due to miss-interpretation of the garbage collection log.

4.4.2.1 Minor Garbage Collection Activities to the Rescue

Before drilling into minor collection type, there is something important that should be known, which is: the **life-time of objects**.

Objects life-time plays a significant role in the allocation and clean-up process. For instance, most of the objects tend to have a short-life time since most of the processes do not span throughout the application’s life-time.

Here is an example about the objects distribution against their life-time.

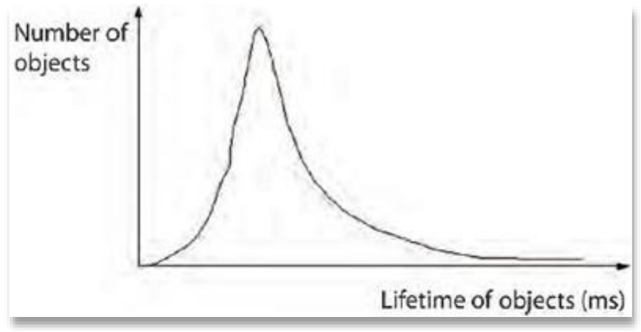


Figure 27: Objects distribution against time.

Newly created objects are intentionally moved to the Eden area of the Young space to act as a buffer zone for objects before moving them into the Old space, in other words, objects that are in the Young space and survives a specific threshold are then called “Old objects” and hence need to be moved to the **“Old Space”**.

As said previously, minor garbage collection occurs when the Eden space gets fully occupied, so in a process to clean up this area of heap, a minor garbage collection is triggered.

It is called minor because the Eden space is usually a small space and requiring cleaning it requires small amount of time.

Minor garbage collections are not very costly, but in the same time it is a “stop-the-world” operation which requires stopping the execution of all available threads of the application, only the thread responsible for the clean-up is left to execute, so it is a costly but a little.

Things could go bad when the number of minor garbage collections is very high which may result in intolerable **“stop-the-world”** events, which means bad performance.

So, what this work will offer is a way to know whether the number of minor garbage collection activities is considered normal or not.

The dilemma that faced this research with investigating and researching minor garbage collections is that it is normal to have many minor garbage collections, this is normal, so how could we know in different cases if this is really normal or not?

So the researcher reached out to the following theory:

GC tuning is not required if: [6]

- Minor GC is executed quickly (nearly within 50 ms).
- Minor GC is not too frequent (nearly not less than 10 seconds).

Such readings are not acceptable unless if the Young space is too small, other than that, this is a non-acceptable output.

So, using the tool, the researcher will report such findings with what to do with each

✓ **If Minor GC is taking much time**, then we should check the Young space, if it’s larger than expected (**should not exceed 40% of the Heap**) [7], then this is the explanation, hence we need to decrease the Young space.

If things are set up normally, if the size of the Young space is reasonable, should try to use the following collection type:

- -XX:+UseParallelGC [8]

This triggers parallel collection for the Young space to decrease the minor collection execution time. This is only possible if the machine is a **server-class** machine. It triggers more than one thread to clean-up the Young space.

- ✓ **If Minor GC is too frequent, (Not more than once within 10 seconds [6])** then we should check the **Young space**, if it's too small, then this is the explanation, hence we need to increase the Young space.

If things are set up normally (Young Space within normal range), then a Heap dump analysis should be done, there could be some objects are not getting freed-up which could highly mean memory leak could be present.

4.4.2.2 Major Garbage Collection Activities to the Rescue

As pointed out previously, objects that survived specific minor garbage collections are then moved to the Old Space.

When Old space fills up, full (Major) garbage collection occurs to free up the Old space from objects that are no longer referenced.

Tuning for Full GC is required if:

- Full GC is not processed quickly (within 1 second).
- Full GC is frequently executed (multiple times within 5 minutes).

So, if Full GC is too frequent, then, the Old Space should be increased.

- ✓ **If Full (Major) GC is too frequent (Once per 10 minutes [6]),** then we should check the **Old space**, if it's small, then this is the explanation, hence we need to increase the **Old space.**

If things are set up normally (Old Space within normal range), then a Heap dump analysis should be done, there could be some objects are not getting freed-up which could highly mean memory leak could be present.

- ✓ If Full (Major) GC is taking too much time (Not processed in more than 1 second [6]), then we should check the Old space, if it's too big, then this is the explanation, hence we need to decrease the Old space.

If things are set up normally, if the size of the **Old space** is reasonable, should try to use one of the following collection types:

- -XX:+UseParallelOldGC [8]
- -XX:+UseConcMarkSweepGC [8]

This triggers parallel collection for the **Old space** to decrease the major collection execution time. This is only possible if the machine is a **server-class** machine. It triggers more than one thread to clean-up the **Old space**.

There are two important concepts here to mention that would help application owners and developers to make up their decisions about the best collection type to choose.

1.35 4.5 Collection Types based on the need:

The types of collections are divided based on the usage need, classified as:

Throughput Need:

Meaning that, if we want to improve the performance of the application and there are many numbers of processors.

It focuses on maximizing the number of tasks done within a specific time, example:

- The number of transactions completed in a given time.
- The number of jobs that a batch program can complete in an hour.
- The number of database queries that can be completed in an hour.

It does not focus on quick response time.

In order to enable this, use this option alone:

-XX:-UseParallelOldGC [9]

It will enable Parallel in Old & Young.

If we wish to enable Parallel collection for only the Young generation, we can use this option:

-XX:-UseParallelGC   [9]

Beside either of the mentioned parameters, we can use the following option that triggers the number of threads to work in parallel to do the collection:

-XX:ParallelGCThreads=<desired number> [8]

Responsiveness Need (Low Pause Need):

This refers or focuses on how quickly the application responds to a specific request, examples:

- How quickly a desktop UI responds to an event
- How fast a website returns a page
- How fast a database query is returned

These types of applications do not tolerate large pause times. Usually this is used on machines with two or more processors in order **to share the resources (processor threads) with the garbage collector so that in order to omit pauses (stop-the-world).**

In another words, in order to clean the Old space with minimized pauses, it uses a separate garbage collector thread to run side by side with the applications threads, thus the application does not pause for garbage collection.

To enable this type of collection, use the following option:

-XX: +UseConcMarkSweepGC [9] And
-XX:ParallelCMSThreads=n [10] If this is used, be sure to use the following parameter as well:

-XX: +CMSClassUnloadingEnabled [11]

4.6 Object's Lifetime

Objects like anything in life, has a lifetime that imposes a challenge on the performance of the application.

An object is created when it is needed and destroyed when no longer needed. As we learned that in Java, the developer has nothing to do with destroying objects and cleaning them from memory. As stated before it's the role of the JVM to manage the instantiation and the cleanup processes for objects.

Tenuring/Promotion is the process for moving objects from the Young space to the Old space.

This is normal during the lifetime of the objects in memory, objects that survives multiple garbage collections are finally promoted/tenured from the Young space to the old space. What is not considered normal is moving these objects to the Old space earlier than expected, hence producing an issue.

More already stated, collecting objects from Old space is a more costly operation than collecting objects from the Young space, because almost always Old space is larger, hence reducing the Old space collection as much as possible means lowering pauses time, which as a result means, better application's performance.

How does the JVM do it?

Each object that survives a minor collection, meaning that; when an object in the Young space survives minor collection, its age is increased by one.

According to a specific threshold, when an object reaches the maximum age threshold, it is then promoted /tenured to the Old/Tenured space.

The age threshold is called **“Tenuring Threshold”** and the maximum value is called **“Maximum Tenuring Threshold”.**

In order to print out the Tenuring distribution for the available objects, the following parameter should be added.

-XX:+PrintTenuringDistribution

Let’s check a small part of the GC log file that demonstrates the objects distribution against their ages:

```
Desired survivor size 75497472 bytes, new threshold 1 (max 15)
       - age   1:   19321624 bytes,   19321624 total
       - age   2:      79376 bytes,   19401000 total
       - age   3:    2904256 bytes,   22305256 total
```

This means that the current tenuring threshold is 1, meaning first cycle of gc in the Young space.

It shows objects distribution against their age.

Desired survivor space is 75 MB.

The available objects size in this cycle of collection is nearly: 23 MB.

Each time the Survivor space is emptied, the threshold is reset.

For instance, the desired survivor space is 75MB and the current usage is 23MB, meaning that, the Survivor is not fully occupied, hence, promotion to the old space is not needed and still they can stay in Survivor space.

When the next minor gc is triggered, upon the Eden space being filled up, the minor gc scans over the Young space. If the objects in the Survivor space are still referenced, then they would stay in the Survivor space but this time increasing their age. So the next output would be:

```
Desired survivor size 75497472 bytes, new threshold 2 (max 15)
- age   1:   98376 bytes,   19321624 total
- age   2:   19321624 bytes,   19401000 total
- age   3:    79376 bytes,   22305256 total
        - age   4:    2904256 bytes,   22305256 total
```

As we can notice, this is the next round (next minor gc) as indicated by "new threshold 2" still the total size of all objects of all ages did not reach the desired survivor size, meaning that objects in the survivor space still won't be promoted to the Tenured space.

If the following appeared in the log directly after the previous lines, it means that objects are moved out of the Survivor space and that they were **prematurely promoted**, which is a bad thing.

Desired survivor size 75497472 bytes, new threshold 1 (max 15)

- age   1:   48376 bytes,   19321624 total

- age   2:   12321624 bytes,   19401000 total

As it is shown, a new cycle started, none of the objects found in the previous collection exist anymore, this means when objects are prematurely promoted, the Tenured space will be filling more rapidly, hence, more major garbage collection is going to happen.

Solution to this issue:

- Increase Young space.
- Resize Survivor spaces to allow more objects to survive in the young space.

If the Young space is already within normal range, then it is recommended increasing the Survivor space.

This is done by decreasing the survivor ratio.

Default value for **SurvivorRatio** is 8, lowering it would increase the Survivor space, hence, avoiding survivor space overflow which as a result lead to premature promotion.

Premature promotion should be minimized as possible since otherwise it could cause a Full garbage collection, which is a stop-the-world event which we are trying to minimize as possible.

The tool would catch premature promotion cases and recommend what should be done.

1.36 4.7 Memory Leaks

It is one of the most critical and toughest topics among all issues.

What is a memory leak?

It means that the JVM failed over many garbage collections to reclaim back memory used by objects that are no longer used or referenced.

Meaning that memory that was allocated for objects created to accomplish certain process that was done were not released because other objects are still mistakenly referencing these objects, so the garbage collector decides to skip these objects. [12]

How bad is that?

Leaks eat up memory very quickly resulting finally in an “Out of Memory” error which finally leads up to an application crash.

As said previously, crashes caused by an “Out of Memory” error are very general, especially when the specific error is Heap error, as said before could be either because:

- Insufficient Memory
- Inefficient use of memory (such as memory leak)

The Out of memory error caused by a memory leak is categorized under **“Inefficient use of memory”**, if this is the case, there is nothing we can do about it but to investigate the root cause and finally, fixing the code itself.

So, if there is no tuning that could be done for this, why the researcher is raising such an issue?

Simply because one of the toughest jobs is to find out if an out of memory error is because of a memory leak or not, this is where this work helps out, reading a graph to try to know if there is a memory leak or not, is a hard task to do, so the new approach as well as the tool will facilitate this decision on the developers.

How to discover?

This research came up with some simple theories to solve this dilemma.

A memory leak is a failed attempt for the garbage collector to free up the memory reserved over many collection processes.

What is expected from a full garbage collection process is to free up some memory that can be used for other processes, so if it failed to do that, then it’s a memory leak.

Actually memory leaks are the most challenging problems among memory issues, there are many patterns for memory leak. This study only aims to find the most common memory leak type. It is a very complex task to identify other types of memory leak from a garbage collection log alone.

As stated previously, the best approach to detect and track a memory leak is analyzing a Heap dump of the memory.

Why Memory Leak is related to “Old Space” of the Heap?

According to Oracle, most objects are shortly lived, meaning that they only stay a little time within the memory, only small portion of objects live a long time [13]. This is a normal or expected behavior for an application, most of the applications are operation based, meaning that, a customer who is requesting a specific service from a Forex-Exchange application needs for example to get the current “Gold” rate world-wide. This is an example of a single operation that when triggered will need some objects to be executed, once done, objects are no longer needed, this is an example of what is called a Stateless operation (Stateless objects).
On the other hand, a customer is using banking services from the Internet banking application, once logged in, customer may choose to do several operations within the same session that he is using originally, deposit operation, then maybe money transfer, followed by paying some bills online. These types of operations simulates the real need for state-full objects, were some objects that may represent as in this example the customer’s information, needs to live for the long-term.

From this example and from the reference below, the mostly common or typical object’s pattern is to have Young objects much more than Old Objects, i.e.: objects which lives shortly much greater than objects which lives long time.

What do this have to do with Memory Leak is that, since a leak is a continuous failed attempt to free-up some memory from the Old space of the Heap memory, and since Old space of the Heap only contains Old Objects (objects that have been around for long time) then for this reason, a leak is related only to old objects.

When objects are considered “OLD”?

According to Oracle, objects are considered old if they survived a certain number (threshold) of minor garbage collections, typically 15, meaning that objects which survived 15 minor garbage collections are then moved to the Old space and hence considered Old objects. [14]

This is why the main goal is to focus on Old space of the Heap to detect memory leaks and not to focus on minor garbage collections.

When minor garbage collection is triggered, the following occurs:

- Eden Space is emptied.
- One of the Survivor spaces is empty.
- With this, Young space always holds short lived objects.

When Full garbage collection is triggered:

- Old Objects within the Old space that are not referenced anymore are removed from the Old space.

Rule 1:

- Full Garbage collections should be minimized, since most of the objects are shortly lived, Young space is always smaller than the Old space. For this reason, collecting the Old space is more costly.

As a simple solution, in order to avoid Full garbage collections, we need to make sure that the Old space is always not fully utilized. One may say is to increase Heap memory and as a result, the Old space. This is a non-feasible/applicable solution, it only postpone the issue not solve it.

This takes us to the following question, why Full garbage collections are triggered?

- A process that requires much memory and there is no available space for this allocation process.
- An un-noticeable problem in the application's code that causes an un-noticeable leak in the Heap memory.

Memory leaks are not always related to Full Garbage Collection, a leak may happen without triggering any full gc. So the most appropriate way of detecting a leak is reporting the difference between the initial memory per day (at the start and end of each day).

Example of Application with Memory Leak:

Figure 28 illustrates the memory usage with time for an application, with code snippet attached below. As the code snippet shows, the application gets some resources without releasing them, with each resource, the reserved memory cannot for the resource cannot be released since the link to the resource is not dropped, therefore, memory for each resource will be reserved forever.

```
        public void processReports()  {

try {
     List reportsIds = getAllReportsIdsFromDB();
```

```
                    while (dao.next()) {
                      for(Iterator iter = reportsIds.iterator(); iter.hasNext();)
                            Integer reportId = (Integer) iter.next();
                                  Report report = new Report(reportId)
                                   report.open();
                                  // do things
                                 //   Did not Close Report
} catch (Exception e) {
        e.printStackTrace();
}}
```

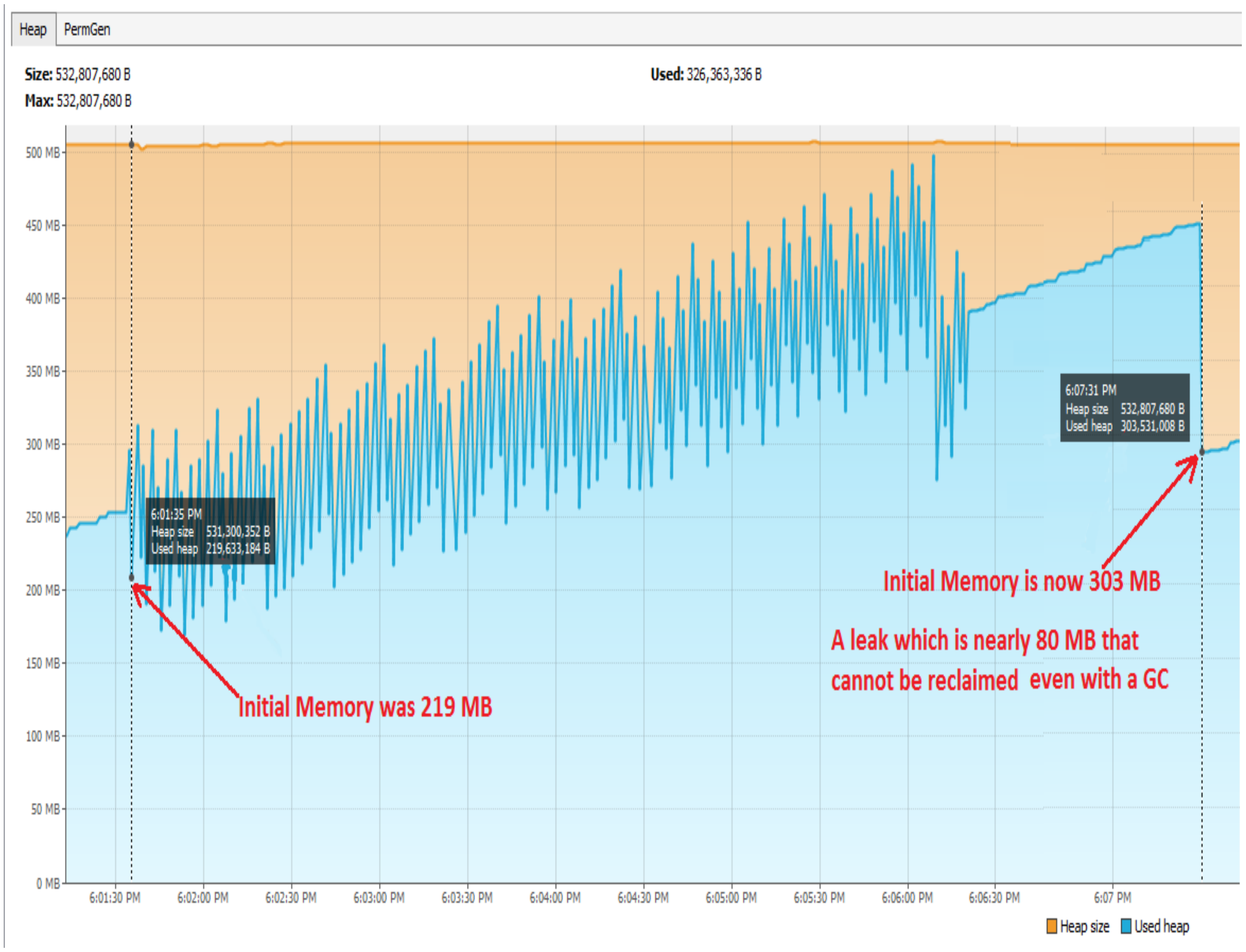


Figure 28 - Possible Heap Memory Leak

Example of Application with no Memory Leak:

On the other hand, the same code but this time with releasing the open resources.

```
public void processReports()  {      List reportsIds =
getAllReportsIdsFromDB();
                      while (dao.next()) {
                         for(Iterator iter = reportsIds.iterator(); iter.hasNext();)
                                 Integer reportId = (Integer) iter.next();
                                         Report report = new Report(reportId)
                                           report.open();
                                          // do things
                                         report.close();  // closed resources
} catch (Exception e) {
          e.printStackTrace();
```

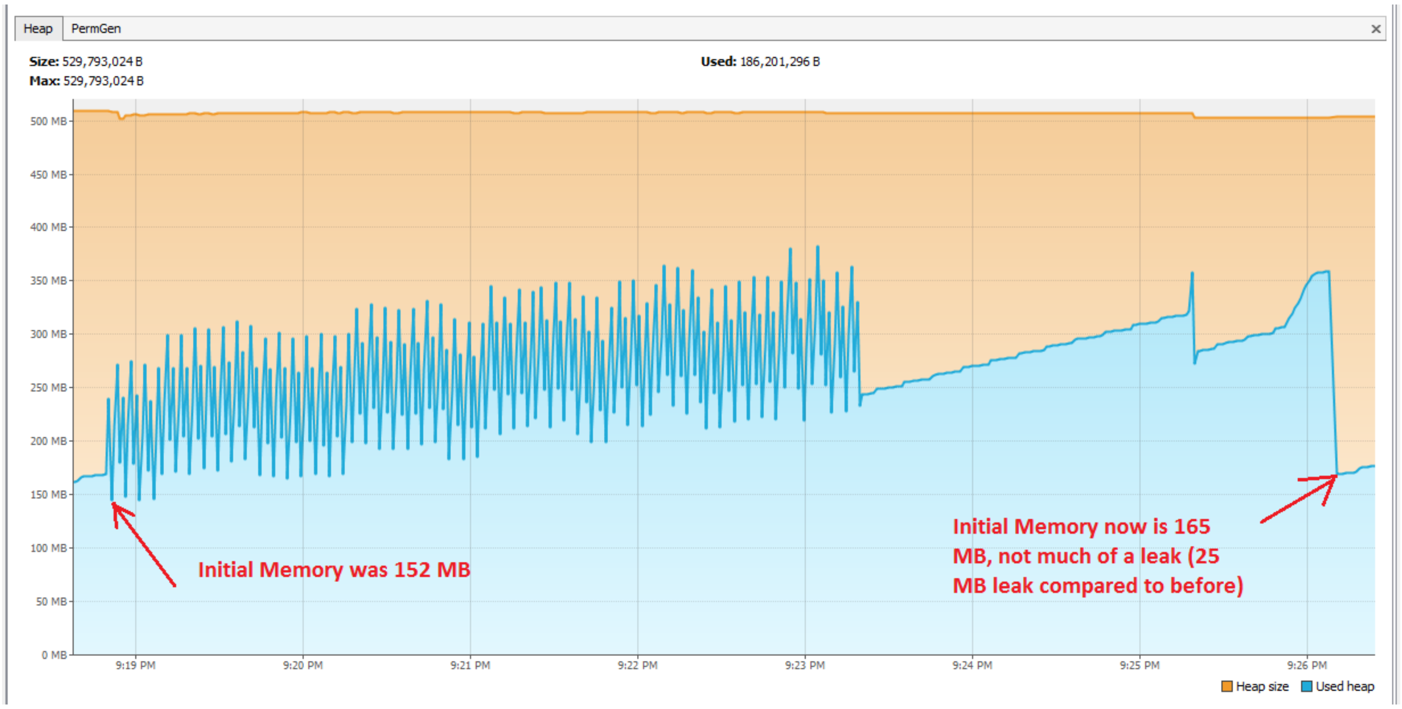


Figure 29 - Reclaimable Heap Memory

A detailed report (to be developed as a future work) will show more information regarding:

- How much memory (un-released) were used over a full day
- How long reclaiming memory did take.

From the definition of the garbage collection provided from Oracle itself, which is garbage collection is an automatic process for memory de-allocation of memory which is no longer used [15].

So, if memory before collection is the same or greater than memory after collection then actually the garbage collector failed to do the job, hence it's a memory leak.

Memory before GC >= Memory after GC

But this alone is not sufficient, because there could be some freeing occurred that are not relevant for the objects that are not referenced and are not the reason behind the leak.

So we can say now that a memory leak is present if:

Memory before GC >= Memory after GC

OR

Memory after GC is less than Memory after GC by a small portion.

In order to have a better analysis, memory should be analyzed more than once for multiple garbage collections, otherwise, the analysis could be misleading, meaning that, it could be a task that consumes large memory and it is going to release it after it is done, which mean no memory leak is present.

That is the very simple scenario for detecting again a very simple form of garbage collection, but actually this is not enough. Actually any heap memory reserved over time even after the application processes are done should be compared against the Heap memory reserved before these processes start. This amount of memory should be reported and analyzed with the correct tool (Heap Dump).

Report always if:

Initial Memory after GC > New Initial Memory after GC

This work votes for the second approach, reporting the difference between the Old Initial Heap sizes before the first trigger of the first garbage collection of any process compared to the new initial heap sizes after the last gc for the same process.

What to do?

As said, nothing much, if a leak is present then analyzing a Heap Dump of the memory is the only option to know which objects caused this leaking issue and fix the code causing this accordingly, which is also out of this research scope.

1.37 4.8 Heap Size

Very critical topic that is also tricky, a question that always application developers and administrators ask:

- Are we allocating enough Heap size for the application?

A tough question that was simplified with some simple theories after many investigations which is as follows:

The researcher will check the Average Heap size usage, taking an indication for the maximum amount of Heap.

This is the hypothesis:

If Heap usage is almost always full (meaning that almost always the Heap size is fully occupied) then it is a 100% that we need to increase the Heap size, i.e.: Heap is not sufficient for such application. Meaning that: we should increase the Heap in general, not specific areas of Heap.

This might be tricky, a Heap might be overused or the allocated Heap is almost always used because of a memory leak, that's why this theory is **only valid if there is no memory leak.**

This is made sure to be included in the tool, before giving users indication if an increase in Heap size is needed, will check first if there is a memory leak.

If there is a memory leak, then it's a 50% probability that we need to increase the Heap size with an indication that even the 50% is wrong, so solving the memory leak have a higher priority over increasing Heap size because a memory leak could lead to wrong indications, hence wrong decisions.

As always, this will be suggested all by the tool which reduces the complexity of the analysis & investigation, hence saving time.

Old & Young sizes

Old & Young sizes suggestions and recommendations are added previously as part of the major and minor garbage collection sections.

1.38 4.9 Permanent Generation Space

It is very important to maintain a sufficient space for the permanent generation because insufficient space could lead up to an Out of memory error, hence a crash of the application.

It is very beneficial to catch up such issues before happening to see if there could be a possibility of a performance issue or not.

If the amount of memory allocated for the Permanent space is almost always the same as the amount of memory used for the Permanent space, then it is time to increase the space of this generation.

Almost always means: either a full usage of the permanent space or almost full usage.

1.39 4.10 Implicit & Explicit GC

Garbage collection in Java is an implicit process, meaning that it is triggered by the JVM itself whenever it sees it appropriate.

The question here is why? Why would it be an automatic operation that the developer has nothing to do with it?

Simply because the cleaning process of objects no longer needed is not left to the developer, so it would make no sense to leave the triggering of the process in the hands of the developer.

Saying that, Java has provided a method (function) for the developer that it seems that it instructs the JVM to run the garbage collection process, at the same time, as it says in the documentation of this function, there is nothing guaranteed that the JVM will initiate the collection process accordingly. It only means that the JVM will take the request into consideration, but that is not certain.

Function is:System.gc()

This leads us to the following conclusion:

The collection process is implicit operation; developer has nothing to do with it.

But why this is topic is being brought up? That is because it appeared that this method has a very bad side-effect on the performance of the application, meaning that, if this method was called from the application’s code, it will result in tremendous performance issues, so it’s highly recommended that we prevent such invocations for this method.

The garbage collection log file will be used to check if there is any call to this function and give a hint to remove it from the code, or as a better quicker solution to disable such invocations if present.

The parameter (option) to disable such calls is:

-XX:+DisableExplicitGC

Throughput

It refers to the percentage of total time that the garbage collection process is not taking place, for instance, 70% throughput implies that the garbage collector consumes 30% of the JVM time.

1.40 4.11 Summary of Theories/Algorithms:

The below algorithms were extracted from different references (added below) in order to form up a clear analysis (algorithm) to track memory issues automatically. Other rules are basic rules extracted also from Oracle, it must be noted that the below algorithms were discrete data, this work adds up these rules in a systematic way (like a flow diagram) and then interpreted into a tool to detect memory issues.

Major rule:

Garbage Collection stops the execution of the application, Major garbage collection takes more time than the Minor garbage collection since Tenured (Old space) is bigger than the Young space as discussed previously in this chapter.

- Detects high Minor GC pause time:

1. If more than 50ms, the tool would trigger the need to tune to lower the collection time of the Minor garbage collection. The tool logs if the size of the Young space should be minimized accordingly to lower the collection time.

2. If size greater than 40% of the Heap size, a recommendation to keep the Young space below 40% of the Heap [7].

3. If the size matches point 2 criteria, then the following parameter should be used: -XX:+UseParallelGC [8]

- Detects High rate of Minor Garbage Collections:

Minor Collections should not be too frequent, once per 10 seconds [6].

1. If more than once within 10 seconds, the tool would trigger the need to tune the number of minor collections to minimize the application's pause time, hence, a better user experience (faster response)

- Detects High Frequency of Major GC:

1. If more than once 10 minutes, the tool would trigger the need to tune to lower the number of Major garbage collections. The tool then would recommend increasing the "Old space" size if it's lower than the recommended, below 60% of the Heap size. [7]

2. At this point, if things are set-up as expected, and still we have multiple Full garbage collections within 10 minutes, then a high possibility of Memory leak is available, hence, what should be done is to analyze the heap dump.

- Detects High Full Garbage Collection Time:

1. If more than one second, then the tool would trigger the need to tune to lower the collection time to be below 1 second. If the size of the Old space is more than 60% of the Heap size, then the tool would recommend lowering this size to be less than 60%.

2. If things are as expected in point 1, then the tool would trigger the need to use the following parameters:

```
-XX:+UseParallelOldGC   [8]
-XX:+UseConcMarkSweepGC [8]
```

If application seeks throughput need, then the tool would trigger the need to use the following flag:

```
-XX:-UseParallelOldGC [9]
```

And

-XX:-UseParallelOldGC [9] for the Young Space.

- Detects Premature Promotion:

1. If Premature Promotion is present, objects being promoted (moved) to the Old space before they reach the expected threshold, then the tool would trigger the need to tune by increasing the Eden space to prevent this or by increasing the tenuring threshold. Premature Promotion triggers Major garbage collection, which stops/freezes the application's threads. [16]

- Detects Missing Key Parameters:
The tool triggers an attention for the required parameters needed to best tune the application's performance as illustrated in section 5.2.

- Detects Explicit Garbage Collection:
The tool triggers the need to remove the explicit garbage collection calls [6] by adding the following parameter to ignore explicit GC calls:

-XX:+DisableExplicitGC

- Report the allocation against used Heap sizes for (Young, Old and Permanent generation space).

Chapter 5 – The Tool (GC Log Analyzer)

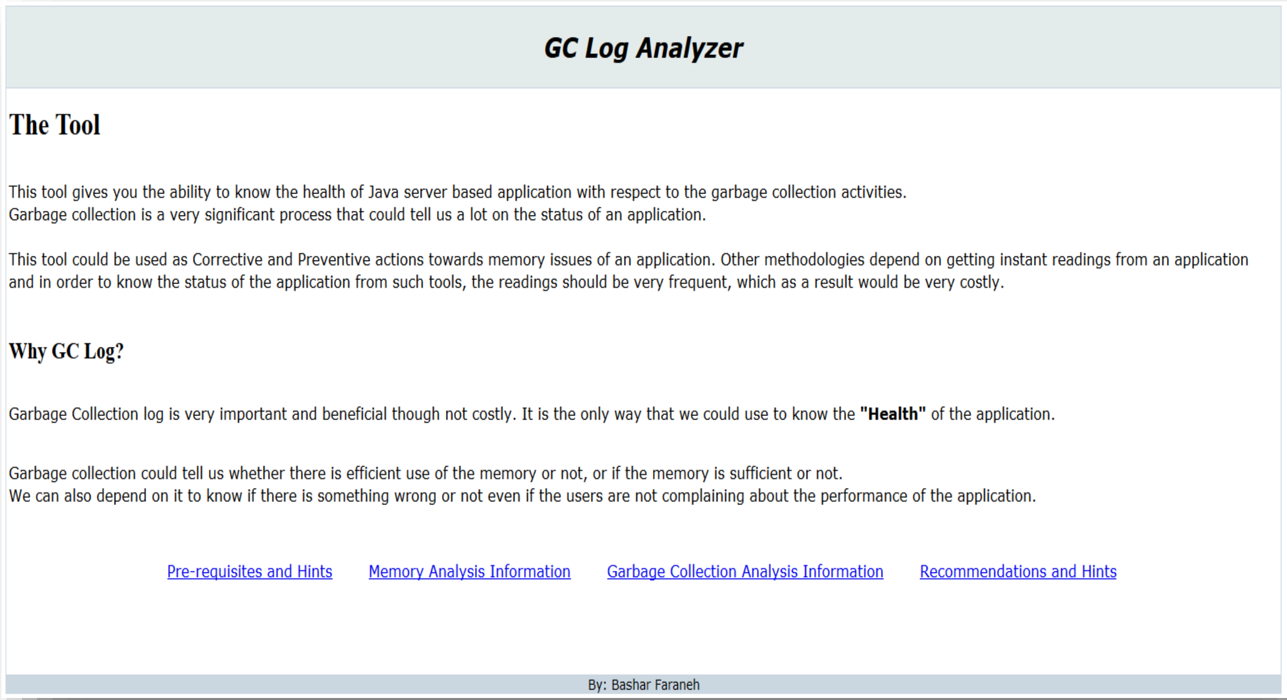
1.41 5.1 Chapter Introduction -About the Tool

The new tool is not just a software program, it's a new approach to enforces the theories and problems raised by the researcher in the thesis with a practical side that helps solving issues we have nowadays in the performance sector of applications

The gc log file has been neglected despite its importance because of the complexity of the output and because of the known approach for solving performance issues that depended on being reactive to any issue rather than being proactive.

It will extract the below information from the garbage collection log which is very beneficial in tuning the memory usage of any application:

- Free Virtual against Physical memory.
- Average usage against allocation.
- Young, Old and Permanent generations usage.
- Young, Old and Permanent generations allocation.
- Garbage Collection Distribution (Full, Minor and Explicit).
- Young GC Pause Time.
- Full GC Pause Time.
- Heap Usage in details.
- If memory leak available or not.
- Young space detailed information.
- Promotion Information.
- Garbage collections with different types in details.



GC Log Analyzer

The Tool

This tool gives you the ability to know the health of Java server based application with respect to the garbage collection activities.
Garbage collection is a very significant process that could tell us a lot on the status of an application.

This tool could be used as Corrective and Preventive actions towards memory issues of an application. Other methodologies depend on getting instant readings from an application and in order to know the status of the application from such tools, the readings should be very frequent, which as a result would be very costly.

Why GC Log?

Garbage Collection log is very important and beneficial though not costly. It is the only way that we could use to know the **"Health"** of the application.

Garbage collection could tell us whether there is efficient use of the memory or not, or if the memory is sufficient or not.
We can also depend on it to know if there is something wrong or not even if the users are not complaining about the performance of the application.

Pre-requisites and Hints   Memory Analysis Information   Garbage Collection Analysis Information   Recommendations and Hints

By: Bashar Faraneh

Figure 30 - GC Log Analyzer brief description

Figure 30 gives a brief introduction about the tool, what is the tool, on what platform it operates, the objectives of the tool and finally the need and the importance of it.

Depending on the activities of the “Garbage collector”, it could tell us a lot about the health of the application, from the memory availability before and after the collection, the number of collections, the type of collections, the lifecycle of objects between the different areas of the memory and finally whether that the issues occurred as a result of a bad code or design of the application.

Down the screen, we can find 4 links for this tool:

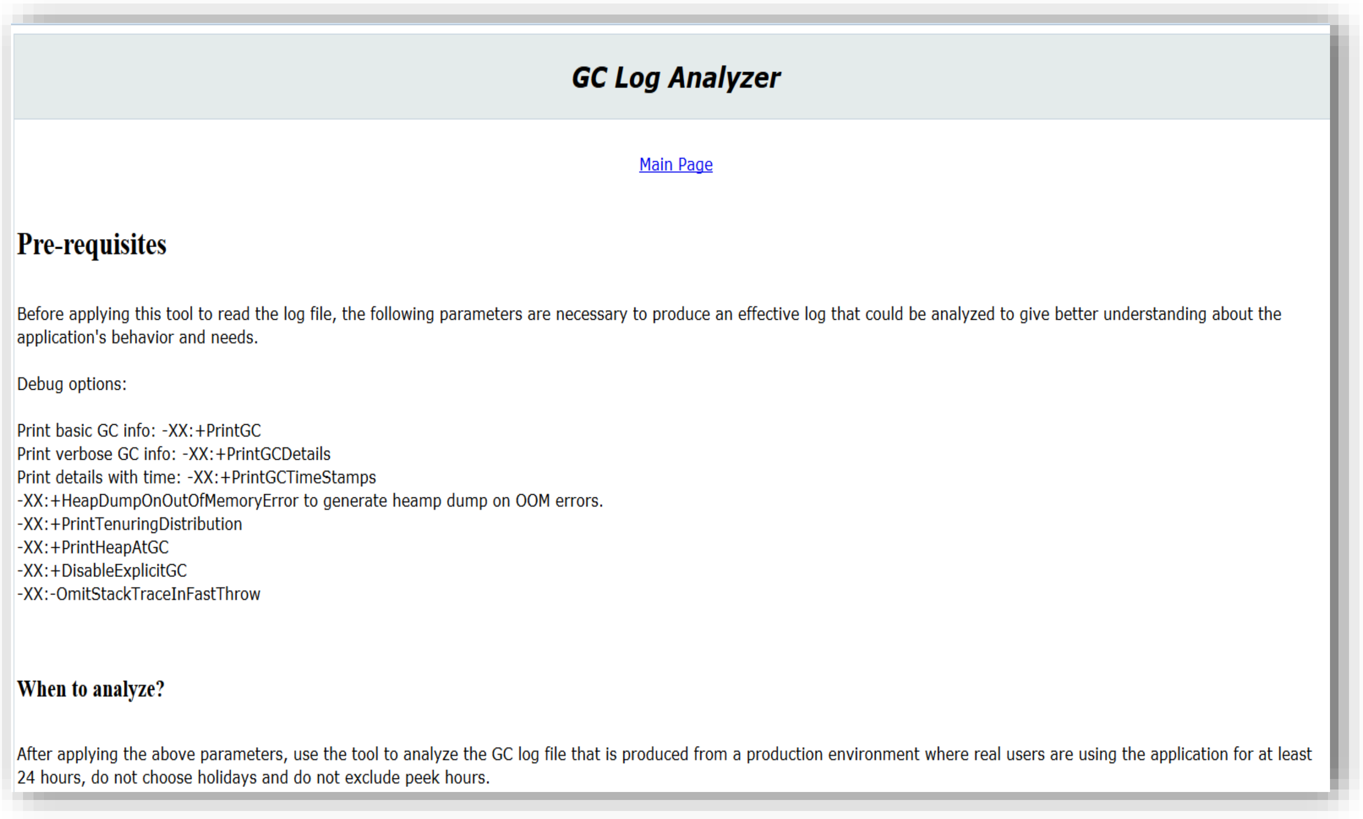
- **Prerequisites & Hints:** It explains in brief what setup or configuration is needed to produce a beneficial log file that the tool could depend on to give the most accurate and beneficial results that we can use to build up our decisions as a result.

- **Memory Analysis Information:** With the use of charts, this module shows the different usage of the memory by the application and the different segments of the memory against their setup and average usage.

- **Garbage Collection Analysis Information:** Again with the help of charts, it shows the activities of the garbage collector in general regarding the number of collections (how many pauses the application experienced) and for how long these pauses stopped the application from running (which maybe from milliseconds to multiple seconds).

- **Recommendations & Hints:** One of the most important aspects of the tool, which does not only represent information as charts, but also point to areas where the application seems to suffer and give recommendations and hints on how to tackle these issues.

1.42 5.2 Prerequisites & Hints


GC Log Analyzer

Main Page

Pre-requisites

Before applying this tool to read the log file, the following parameters are necessary to produce an effective log that could be analyzed to give better understanding about the application's behavior and needs.

Debug options:

```
Print basic GC info: -XX:+PrintGC
Print verbose GC info: -XX:+PrintGCDetails
Print details with time: -XX:+PrintGCTimeStamps
-XX:+HeapDumpOnOutOfMemoryError to generate heamp dump on OOM errors.
-XX:+PrintTenuringDistribution
-XX:+PrintHeapAtGC
-XX:+DisableExplicitGC
-XX:-OmitStackTraceInFastThrow
```

When to analyze?

After applying the above parameters, use the tool to analyze the GC log file that is produced from a production environment where real users are using the application for at least 24 hours, do not choose holidays and do not exclude peek hours.

Figure 31 : GC Log Usage Prerequisites

In Figure 31, the tool mentions the prerequisites of using such tool, the correct setup and parameters that it operates within and when to analyze the gc log.

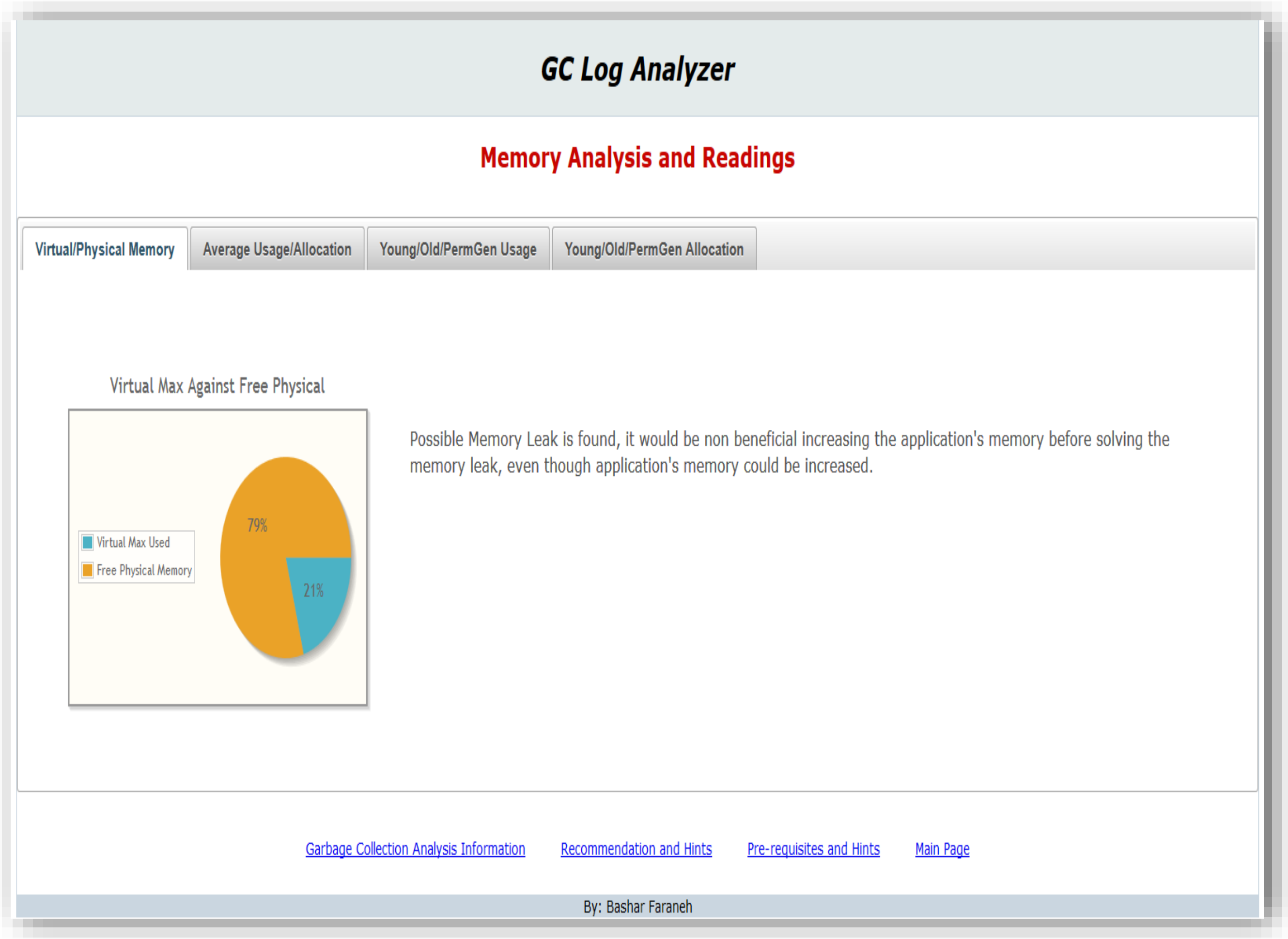
As stated in the figure, the tool is effective to analyze a GC log file that is produced from a production environment where real customers use the application with real scenarios.

This is important because, when application owners deliver the product/application to customers they test and verify it based on internal/local testing, which is not how things really go on at the customer's side.

It is also important that the file should be left to have as much log as possible of a working day at the customer side, not excluding peak hours.

1.43 5.3 Memory Analysis Information

5.3.1 Virtual Memory against Free Physical Memory



Possible Memory Leak is found, it would be non beneficial increasing the application's memory before solving the memory leak, even though application's memory could be increased.

Figure 32: GC Log – Memory Analysis – Virtual against Memory

Figure 32 shows the first reading from the GC log file after being parsed by the new tool.

The first tab in the figure shows what the application is using memory against what is really available as free physical memory within the machine.

This is an important information because administrators needs sometime to know about the status of their resources, meaning that, usually applications and their database reside either on a separate server or even at same server, but for a correct setup, they should not be on a server that is shared for other services/applications, so knowing how much the resources dedicated for the specific application could be extended is a useful information.

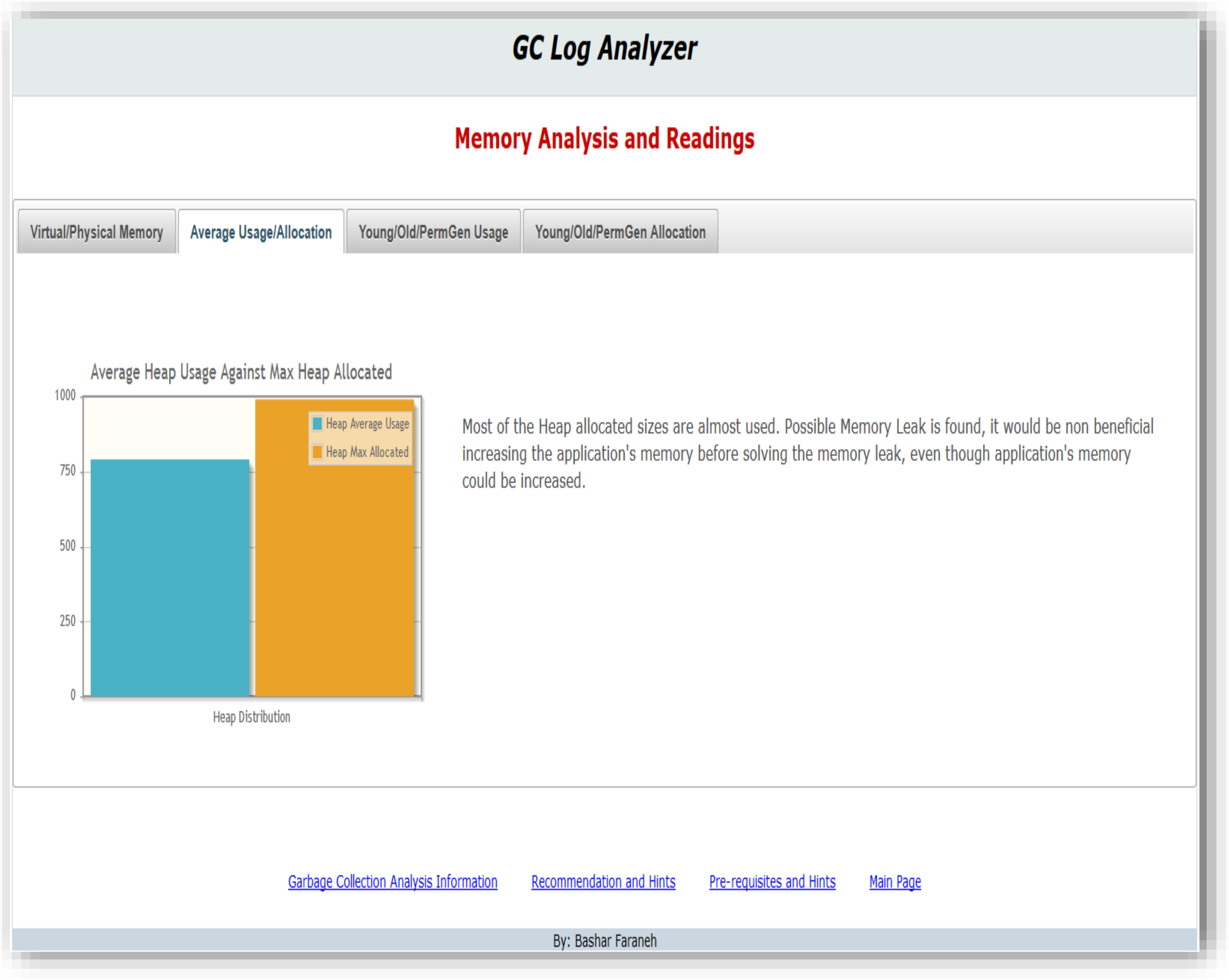
An important thing to notice here is that; having the readings ready as represented as a chart is a great thing, as it may seems, someone may falsely drive by this reading, that's why we need a hint here to inform the administrator that if there is a memory leak, then it should be solved first before taking any further decision, memory leak means that the memory resource is inefficiently used, thus even if there is a decent amount of free memory it would be a mistake to increase the application's memory before solving the leak first.

As the figure shows, the "Free Physical Memory" indicates the available "non-used" memory within the server and can be allocated for the application. If a memory leak is not fund, then it's an opportunity to increase the available memory for the application.

5.3.2 Heap Average against Allocation

Figure 33 represents the average Heap Usage in Megabytes against the maximum allocated Heap memory.

This gives us an indication if the allocated memory for the application is sufficient during the life cycle of the application.



Most of the Heap allocated sizes are almost used. Possible Memory Leak is found, it would be non beneficial increasing the application's memory before solving the memory leak, even though application's memory could be increased.

Figure 33: GC Log – Memory Analysis – Average Usage against allocation

A high average memory usage indicates that the allocated memory might not be sufficient for the application and sooner or later we need to increase it.

Again, if the average usage is not being affected by a memory leak, then it is as is seems by the chart, an increase for the available (allocate) heap memory is needed.

Allocation refers to the memory available for usage by the application but might not be fully used, a memory that is ready and dedicated for the application.

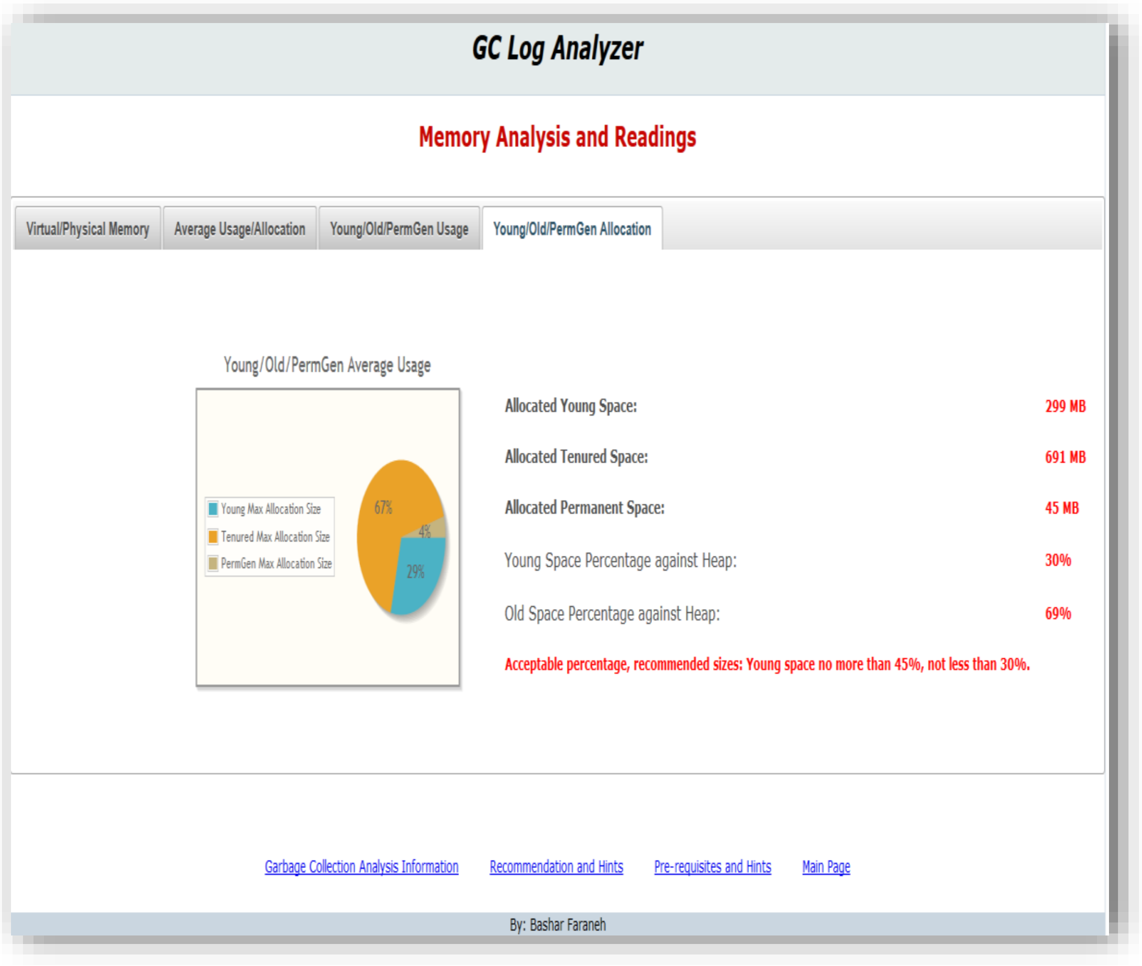
5.3.3 Young, Tenured and Permanent generations Allocations

As we learned from previous figure, we saw the average heap usage overall, here we have the ability to know how much memory each segment of the heap consumes.

The reading of the Young generation is not quite important because, almost always the Young space would be fully used since objects that are created are allocated to the Young space first and when fully occupied, they are moved to the Old space.

So it is a very common case that the Young space is almost always mostly used, as for the Old space; it is not always the case.

Permanent Generation is where classes' objects are stored, they are considered permanent since they live throughout the application life-time.


GC Log Analyzer
Memory Analysis and Readings
Virtual/Physical Memory
Average Usage/Allocation
Young/Old/PermGen Usage
Young/Old/PermGen Allocation
Young/Old/PermGen Average Usage
Young Max Allocation Size
Tenured Max Allocation Size
PermGen Max Allocation Size
67%
4%
29%
Garbage Collection Analysis Information
Recommendation and Hints
Pre-requisites and Hints
Main Page
By: Bashar Faraneh

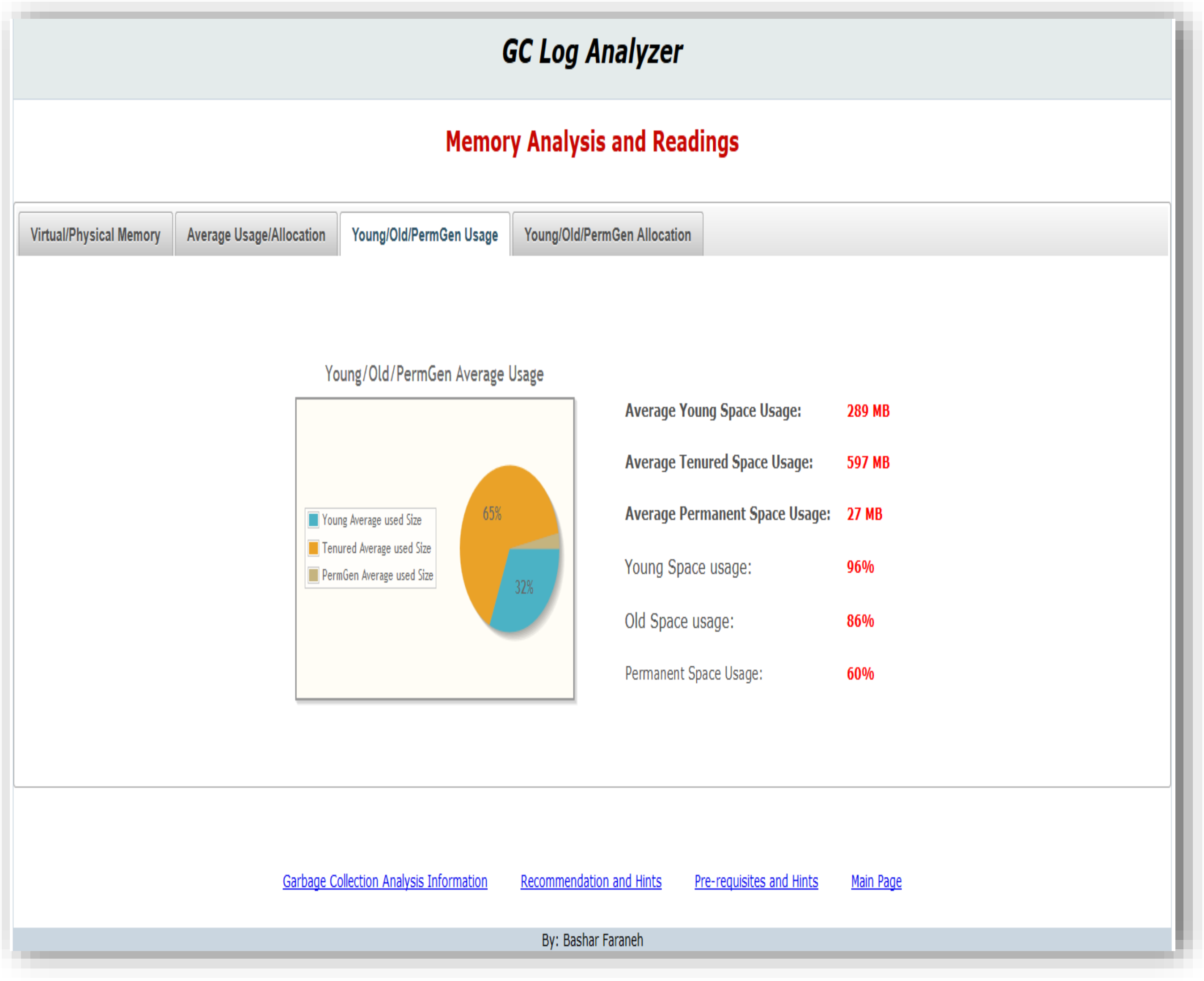
Allocated Young Space:	299 MB
Allocated Tenured Space:	691 MB
Allocated Permanent Space:	45 MB
Young Space Percentage against Heap:	30%
Old Space Percentage against Heap:	69%

Acceptable percentage, recommended sizes: Young space no more than 45%, not less than 30%.

Figure 34: GC Log – Memory Analysis – Young against Old against Permanent generation

From these readings, we can know the percentage of the Young against Old, if within acceptable range or not.

5.3.4 Young/Old & Permanent Generations usage


GC Log Analyzer

Memory Analysis and Readings

Virtual/Physical Memory
Average Usage/Allocation
Young/Old/PermGen Usage
Young/Old/PermGen Allocation
Young/Old/PermGen Average Usage
Young Average used Size
Tenured Average used Size
PermGen Average used Size
65%
32%

Average Young Space Usage:	289 MB
Average Tenured Space Usage:	597 MB
Average Permanent Space Usage:	27 MB
Young Space usage:	96%
Old Space usage:	86%
Permanent Space Usage:	60%

Figure 35: GC Log – Memory Analysis – Young against Old against Permanent generation usage.

Figure 35 shows a chart that represents the different areas of the Java Heap memory against their allocation. As stated before, the Java Heap is divided into 3 main areas:

- Young Space.
- Old Space.
- Permanent Generation Space.

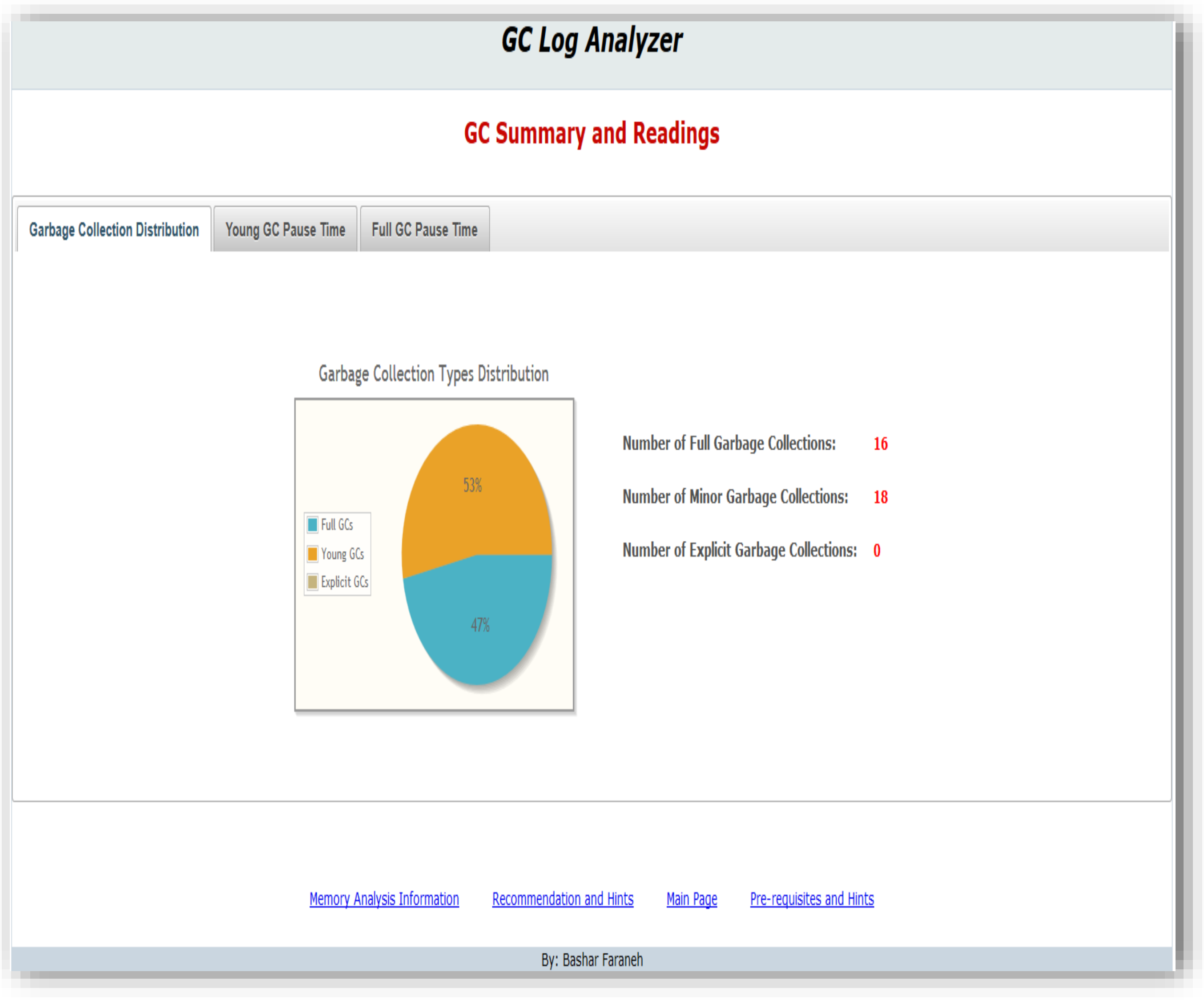
From these readings, we can know which among other areas reserves more memory.

The final figure of the Memory analysis shows the allocated memory for the different parts of the Heap memory, comparing to the previous chart, we can know the allocation for each area in the Heap memory against the average usage.

1.44 5.4 Garbage Collection Analysis Information

5.4.1 Garbage Collection Distribution

Figure 36 shows the distribution of the different types of garbage collections:



Number of Full Garbage Collections: 16

Number of Minor Garbage Collections: 18

Number of Explicit Garbage Collections: 0

Figure 36: GC Log – Garbage Collection Analysis – Collection Distribution

Garbage Collectors can be divided into 2 types:

- Full GC: A collector that works on the Old space area of the heap.
- Minor GC: A collector that works on the Young space area of the heap.

It also report garbage collections as a result of being explicitly called from the code which has a bad impact on the performance.

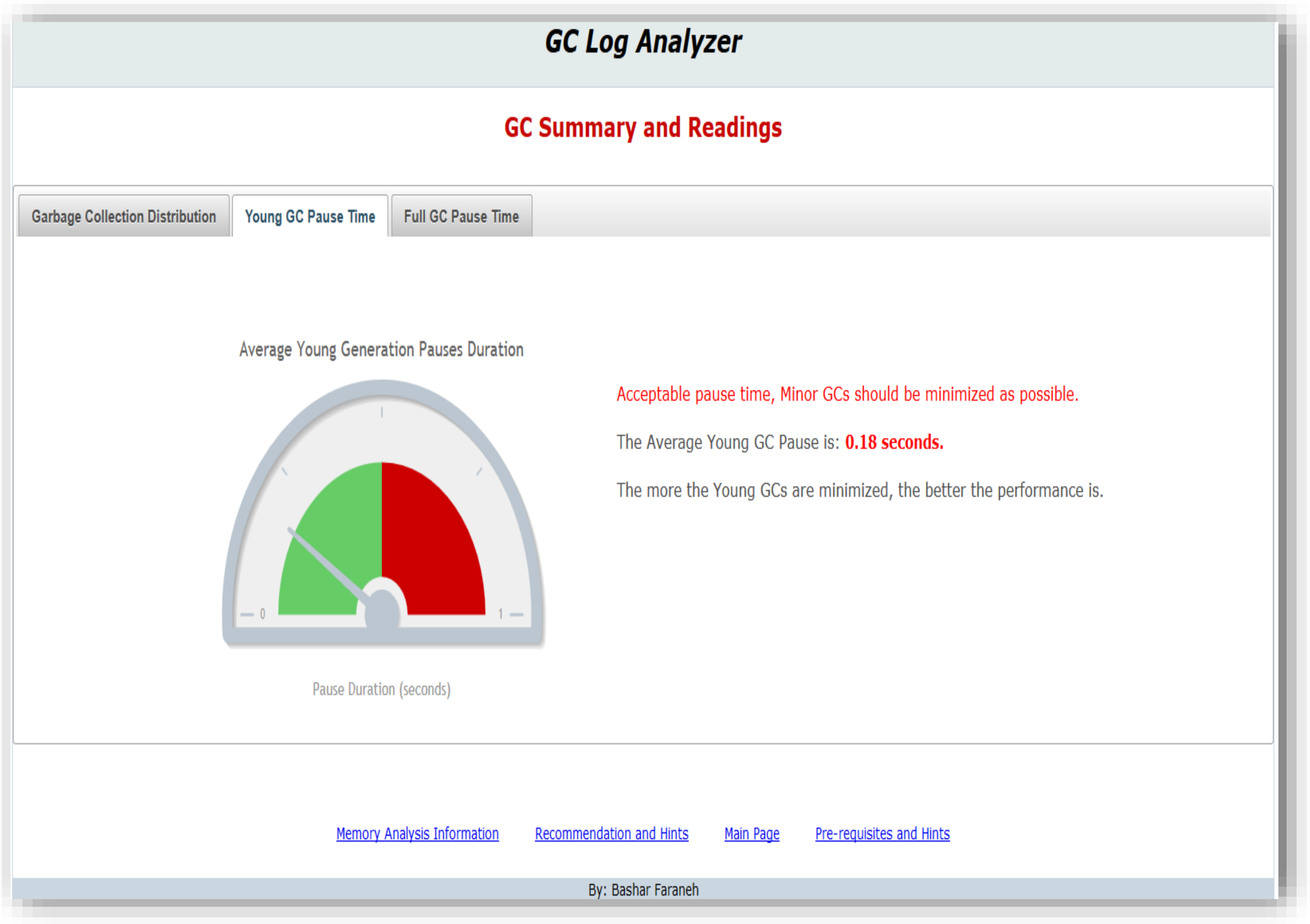
It would be non-beneficial making use of these numbers because we need to know another detail which is; the frequency of these collections of both types. Meaning that even if the number of collections is let’s say 100, but in respect to what? They would be a bottleneck only if they are too frequent; this detail is found under the **“Recommendations and hints”** section.

5.4.2 Young GC Pause Time

The chart within Figure 37 illustrates the pause duration in average caused by the minor garbage collector.

Minor garbage collectors are not expected to be costly collectors since they work on a small area of the heap (the Young) which should be usually smaller than the Old space.

The chart shows the average pause caused by the minor gc, anything below 0.5 seconds is acceptable.



GC Log Analyzer

GC Summary and Readings

Garbage Collection Distribution
Young GC Pause Time
Full GC Pause Time
Average Young Generation Pauses Duration
0
1
Pause Duration (seconds)

Acceptable pause time, Minor GCs should be minimized as possible.

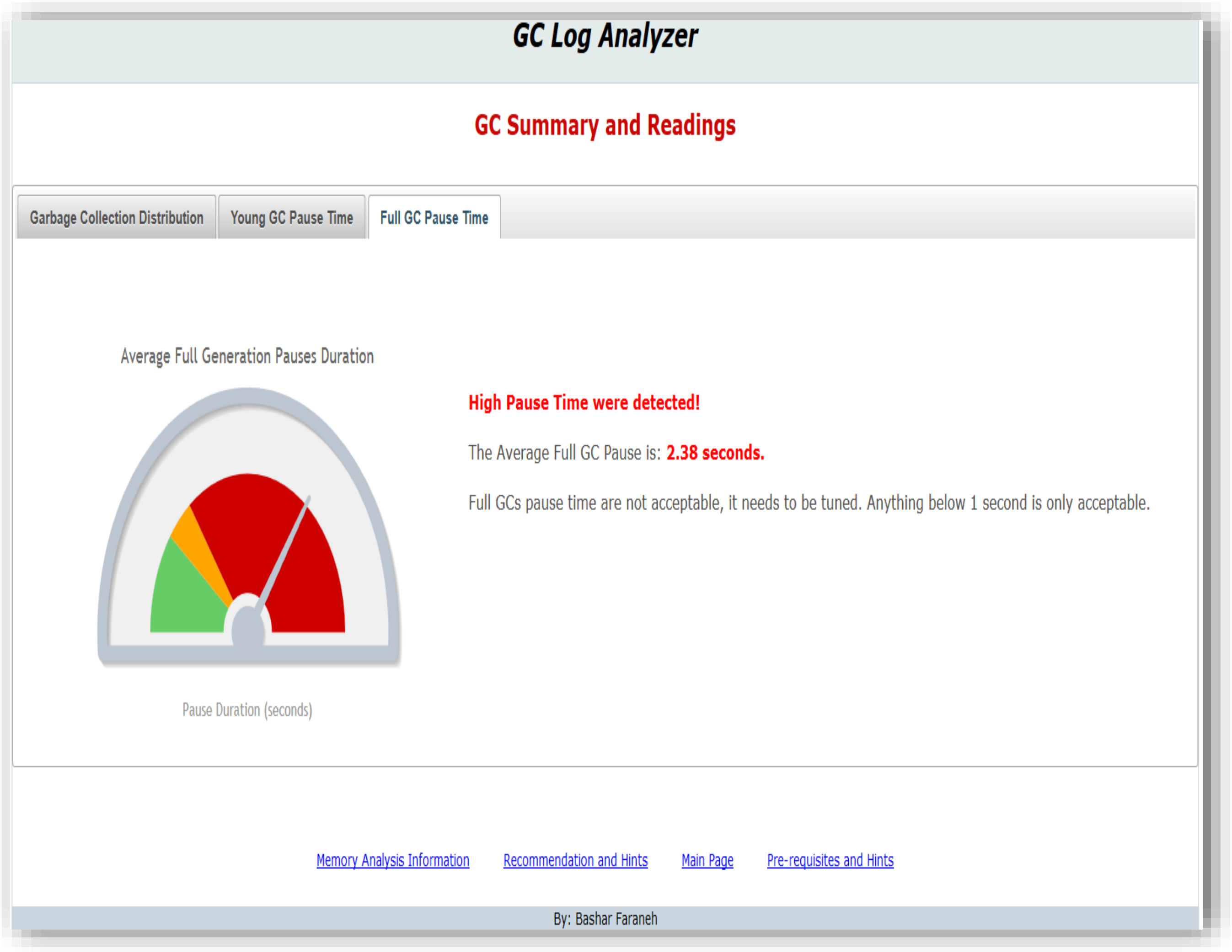
The Average Young GC Pause is: **0.18 seconds.**

The more the Young GCs are minimized, the better the performance is.

Figure 37: GC Log – Garbage Collection Analysis – Young GC Pause

5.4.3 Full GC Pause Time

The chart in figure 38 shows the pause duration caused by the Full garbage collector which are expected to take more time than minor garbage collectors because they work on an area larger than the Young space.



GC Log Analyzer

GC Summary and Readings

Garbage Collection Distribution
Young GC Pause Time
Full GC Pause Time
Average Full Generation Pauses Duration
Pause Duration (seconds)

High Pause Time were detected!

The Average Full GC Pause is: **2.38 seconds.**

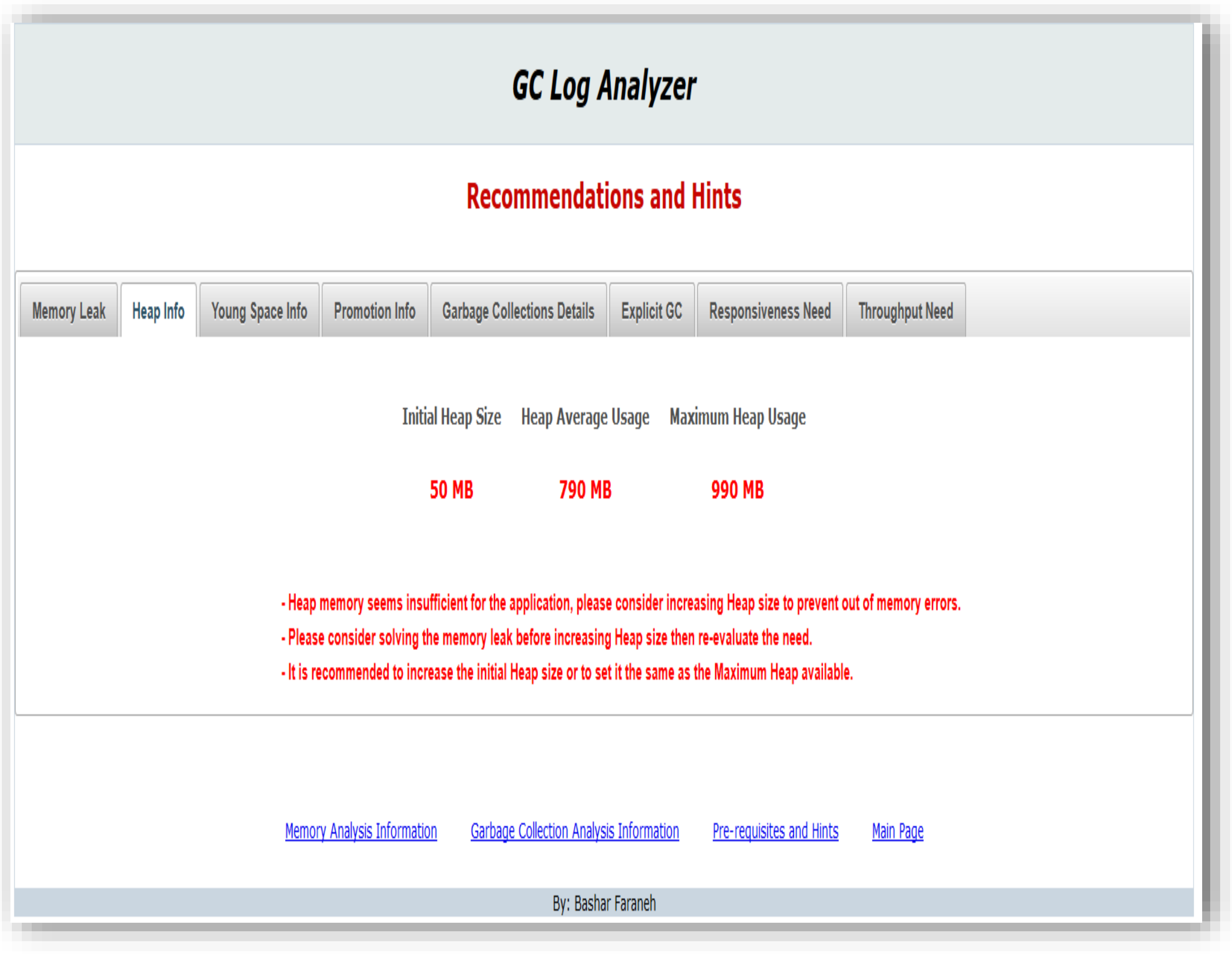
Full GCs pause time are not acceptable, it needs to be tuned. Anything below 1 second is only acceptable.

By: Bashar Faraneh

Figure 38: GC Log – Garbage Collection Analysis – Old GC Pause

1.45 5.5 Recommendations & Hints

5.5.1 Heap Usage Details



Initial Heap Size	Heap Average Usage	Maximum Heap Usage
50 MB	790 MB	990 MB

- Heap memory seems insufficient for the application, please consider increasing Heap size to prevent out of memory errors.
- Please consider solving the memory leak before increasing Heap size then re-evaluate the need.
- It is recommended to increase the initial Heap size or to set it the same as the Maximum Heap available.

Figure 39: GC Log – Recommendations & Hints – Heap Info

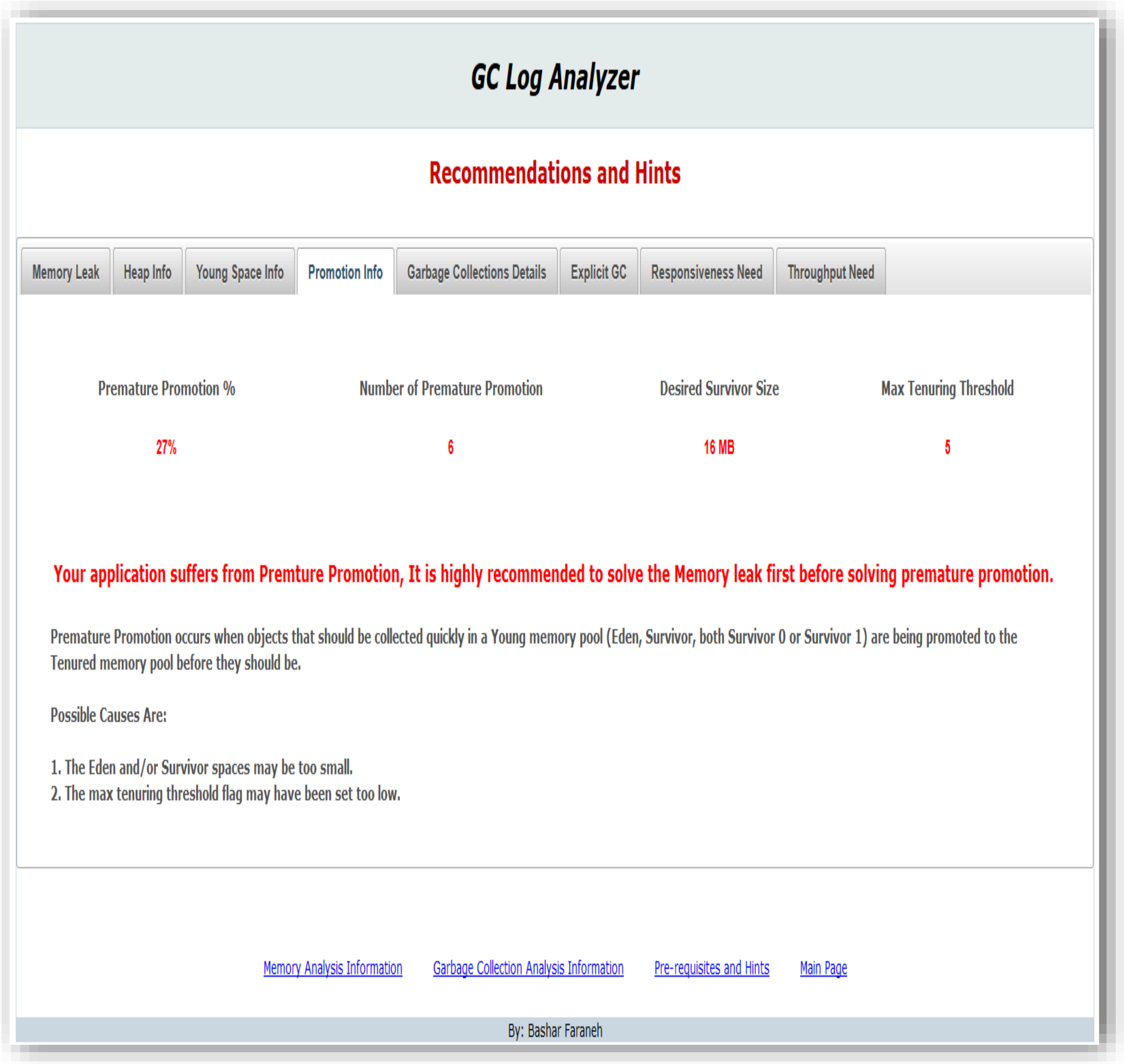
Figure 39 shows the Heap usage info:

- Initial Heap size: It represents the initial heap that the application started with.
- Average Heap Usage.
- Maximum Heap Usage.

What we can benefit here is that, if the initial Heap is a lot lower than the average heap usage, then the Initial Heap size should be increased a lot or should be set to the same as the maximum reserved heap size.

It also shows that the Heap memory seems insufficient and we might need to increase it, but as shown, it checks first if there is a memory leak, if so, it should be solved first before really increasing the heap.

5.5.2 Premature Promotion Information



Premature Promotion %	Number of Premature Promotion	Desired Survivor Size	Max Tenuring Threshold
27%	6	16 MB	5

Your application suffers from Premture Promotion, It is highly recommended to solve the Memory leak first before solving premature promotion.

Premature Promotion occurs when objects that should be collected quickly in a Young memory pool (Eden, Survivor, both Survivor 0 or Survivor 1) are being promoted to the Tenured memory pool before they should be.

Possible Causes Are:

1. The Eden and/or Survivor spaces may be too small.
2. The max tenuring threshold flag may have been set too low.

Memory Analysis Information
Garbage Collection Analysis Information
Pre-requisites and Hints
Main Page
By: Bashar Faraneh

Figure 40: GC Log – Recommendations & Hints – Promotion Information

Figure 38 illustrates the premature promotion information of objects. As we know, promotion is the process of moving objects from the Young space to the tenured space, which is a normal thing.

What is not normal is the premature promotion where we should minimize it as possible.

Premature promotion is when objects are moved from the Young space to the tenured space earlier than expected.

There is a specific threshold as discussed earlier by which objects are allowed to be tenured to the Tenured space, this is important because we want to minimize the rate of moving objects from the Young space to the Old space to minimize the Full garbage collection because it is a costly operation.

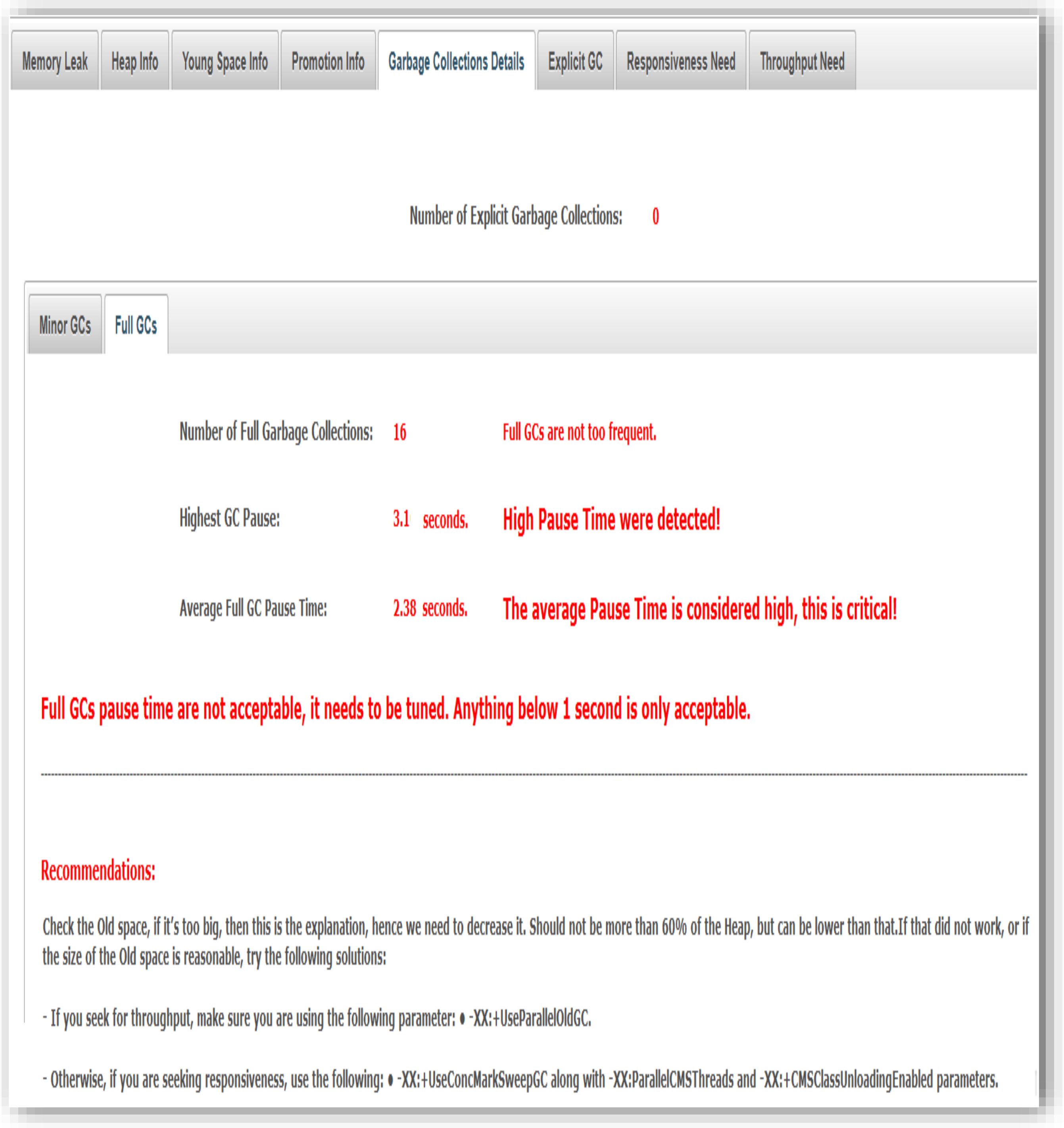
As shown it contains the premature promotion percentage which represents the rate of premature promotions against the total number of promotions.

It is also recommended to solve memory leaks before trying to solve premature promotion, because a memory leak could be one of the main causes for the leak.

If no memory leak is found, then the problem can be solved either:

- The Eden/Survivor spaces maybe too small.
- The max tenuring threshold may need to be increased.

5.5.3 Full GC Information



Number of Explicit Garbage Collections: 0

Number of Full Garbage Collections: 16 Full GCs are not too frequent.

Highest GC Pause: 3.1 seconds. High Pause Time were detected!

Average Full GC Pause Time: 2.38 seconds. The average Pause Time is considered high, this is critical!

Full GCs pause time are not acceptable, it needs to be tuned. Anything below 1 second is only acceptable.

Recommendations:

Check the Old space, if it's too big, then this is the explanation, hence we need to decrease it. Should not be more than 60% of the Heap, but can be lower than that.If that did not work, or if the size of the Old space is reasonable, try the following solutions:

- If you seek for throughput, make sure you are using the following parameter: • -XX:+UseParallelOldGC.

- Otherwise, if you are seeking responsiveness, use the following: • -XX:+UseConcMarkSweepGC along with -XX:ParallelCMSThreads and -XX:+CMSClassUnloadingEnabled parameters.

Figure 41: GC Log – Recommendations & Hints – Full GCs Information

Figure 39 is very important where it illustrates detailed information about the status of the different types of garbage collectors which are FULL and Minor.

As shown, it informs us with the following:

- Total number of Full garbage collections.

This implies if there are many garbage collections or not, still this does not mean a lot because it depends on the duration within the collection is happening.

- Implies if there is any frequent garbage collections or not.

This is important since it informs us if we have to worry about the number of collections, i.e.: whether the collections are too frequent or not. It could imply a lot, like if the application's time is being spent in collection or not.

- The Average pause time of the Full garbage collections.

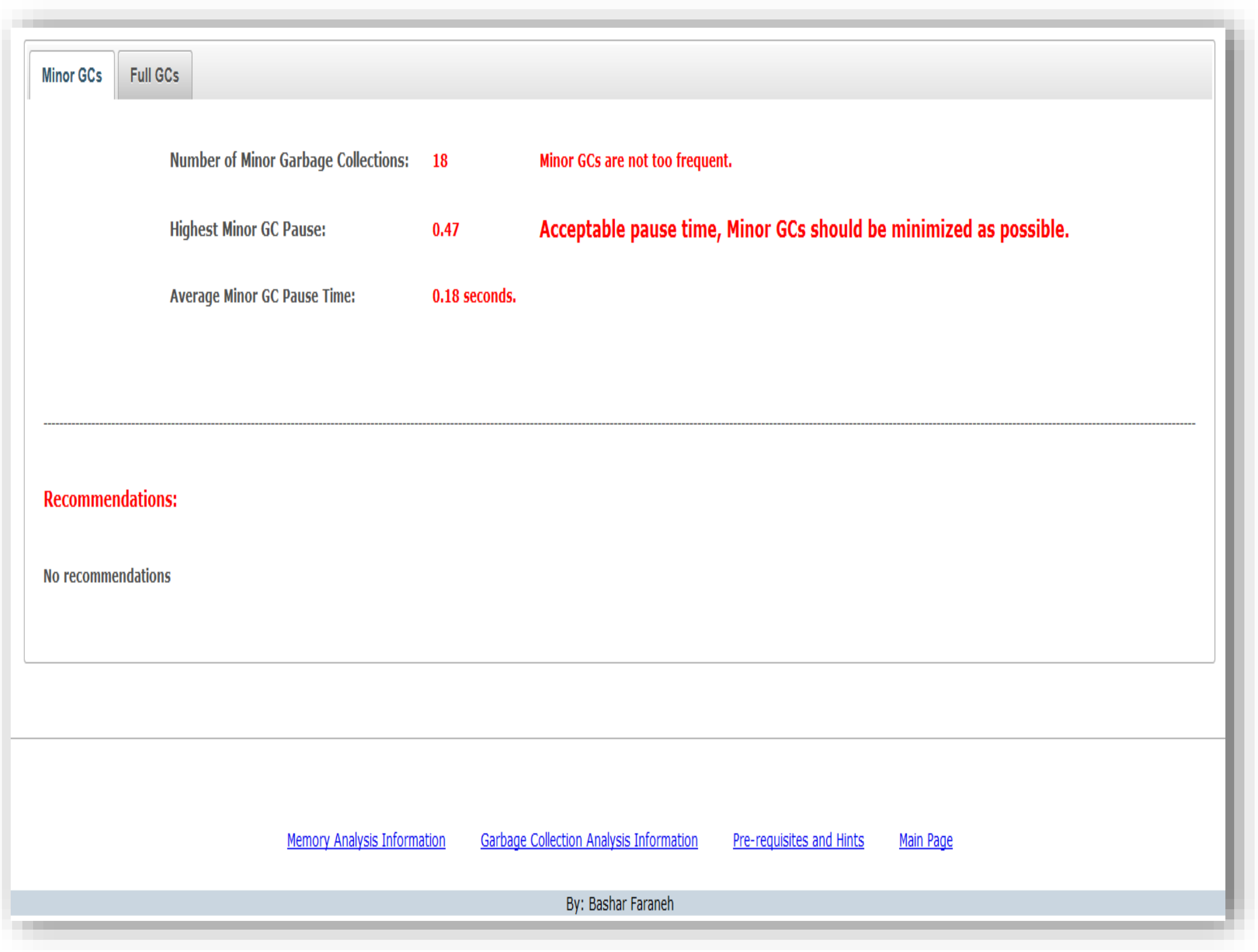
Implies the average pause time of the application, i.e.: the average pause time of the application not being spent for the application threads itself.

- The Highest pause time of Full garbage collections.

Implies the highest pause time of Full garbage collection, anything more than one second is not acceptable.

5.5.4 Minor GCs

Figure 42 illustrates the Minor garbage collection information as the Full Garbage collection information in the previous page.



Number of Minor Garbage Collections: 18 Minor GCs are not too frequent.

Highest Minor GC Pause: 0.47 Acceptable pause time, Minor GCs should be minimized as possible.

Average Minor GC Pause Time: 0.18 seconds.

Recommendations:

No recommendations

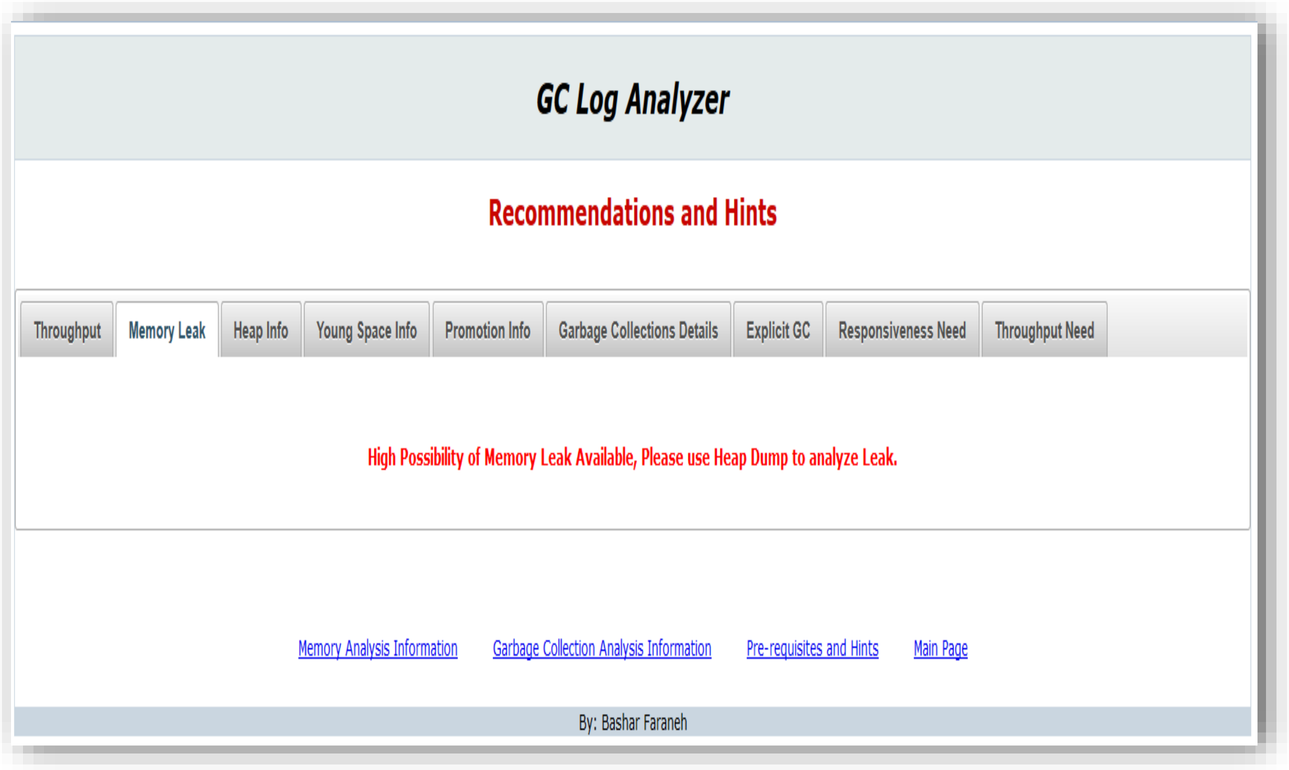
Memory Analysis Information
Garbage Collection Analysis Information
Pre-requisites and Hints
Main Page
By: Bashar Faraneh

Figure 42: GC Log – Minor GCs Details

The only difference here is what is acceptable for minor garbage collection is not acceptable for major (Full) garbage collection.

5.5.5 Memory Leak

Figure 43 shows if there is a memory leak or not, if there is, then nothing that this tool could do about, it has done a sufficient job figuring out if there is a leak or not.


GC Log Analyzer

Recommendations and Hints

High Possibility of Memory Leak Available, Please use Heap Dump to analyze Leak.

Memory Analysis Information
Garbage Collection Analysis Information
Pre-requisites and Hints
Main Page
By: Bashar Faraneh

Figure 43: GC Log – Memory Leak Information

Memory leaks are usually hard to discover.

The approach that was followed to discover a memory leak:

Scan all garbage collections of type "Full" and check the memory before and after the collection.

To figure out a memory leak, we should depend on:

- Memory Freed amount.
- Number of Full GCs.
- Amount of memory freed.
- The number of occurrences of possible memory leak.

Consider the following:

Memory used before leak is: **preUsedMemory**

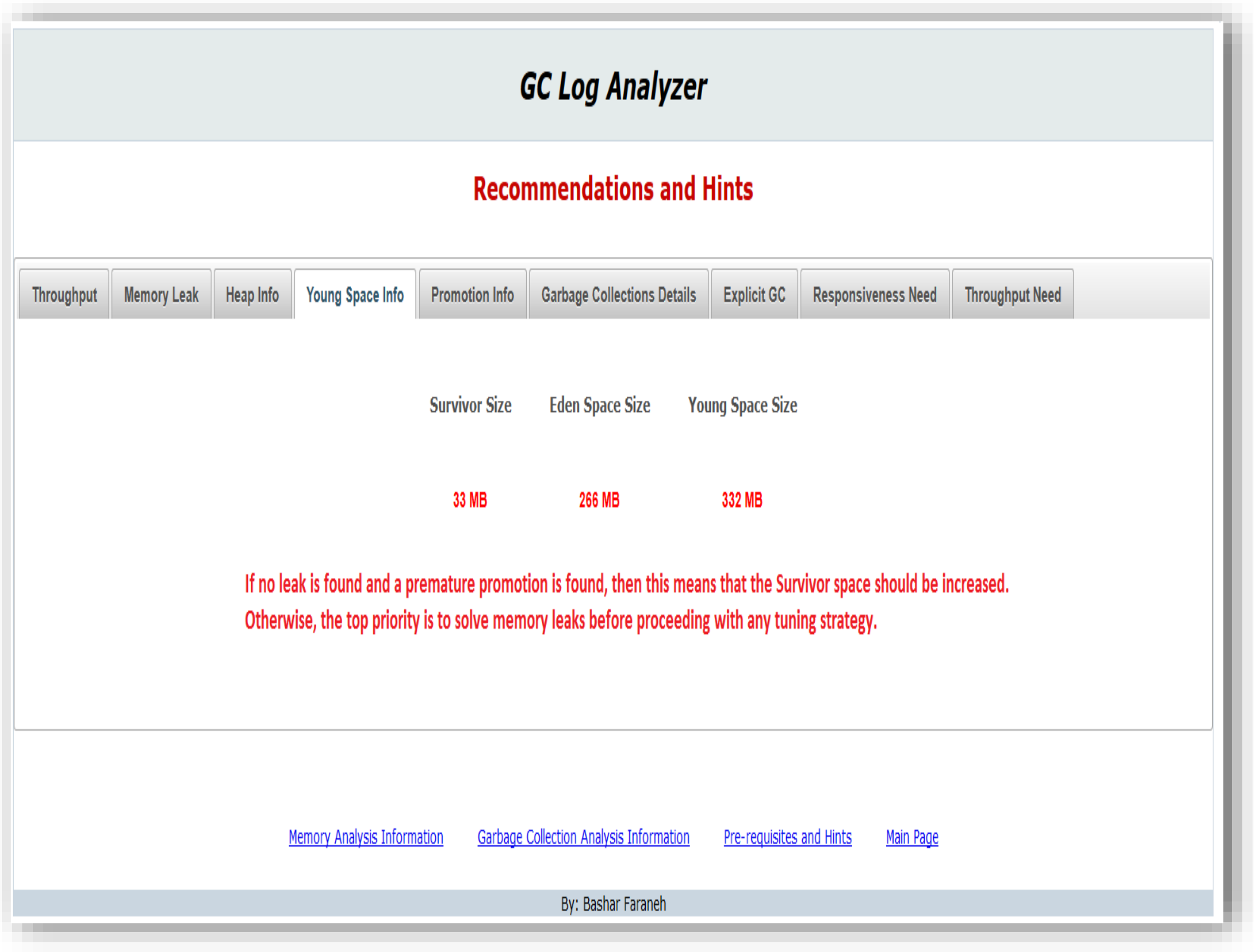
Memory used after leak is: **postUsedMemory**

- If the **postUsedMemory** is greater than or equal to the **preUsedMemory** >> Possible Leak, because what is expected from the Full garbage collection when it runs is to free a decent amount of memory. If memory was not freed after the Full GC or it did increase after it, then this could mean a possible memory leak, still may not be.
- If memory was freed but with a very small amount. Meaning that, objects that are not related to the leak were released only.
- The above hypotheses are only valid if a number of such readings occur. A single occurrence does not indicate a memory leak.

Discovering if a memory leak is found or not is half the way solving performance problems since they were hard to discover, it is important to solve because if a leak is found, if any process that needs a high memory will spike the memory in use and as a result would cause a system crash.

The next step after finding a leak is to analyze which objects are causing this, which is out of this tool’s league (Since this is an offline tool).

5.5.6 Young Space Info


GC Log Analyzer

Recommendations and Hints

Survivor Size	Eden Space Size	Young Space Size
33 MB	266 MB	332 MB

If no leak is found and a premature promotion is found, then this means that the Survivor space should be increased.
Otherwise, the top priority is to solve memory leaks before proceeding with any tuning strategy.

Memory Analysis Information
Garbage Collection Analysis Information
Pre-requisites and Hints
Main Page
By: Bashar Faraneh

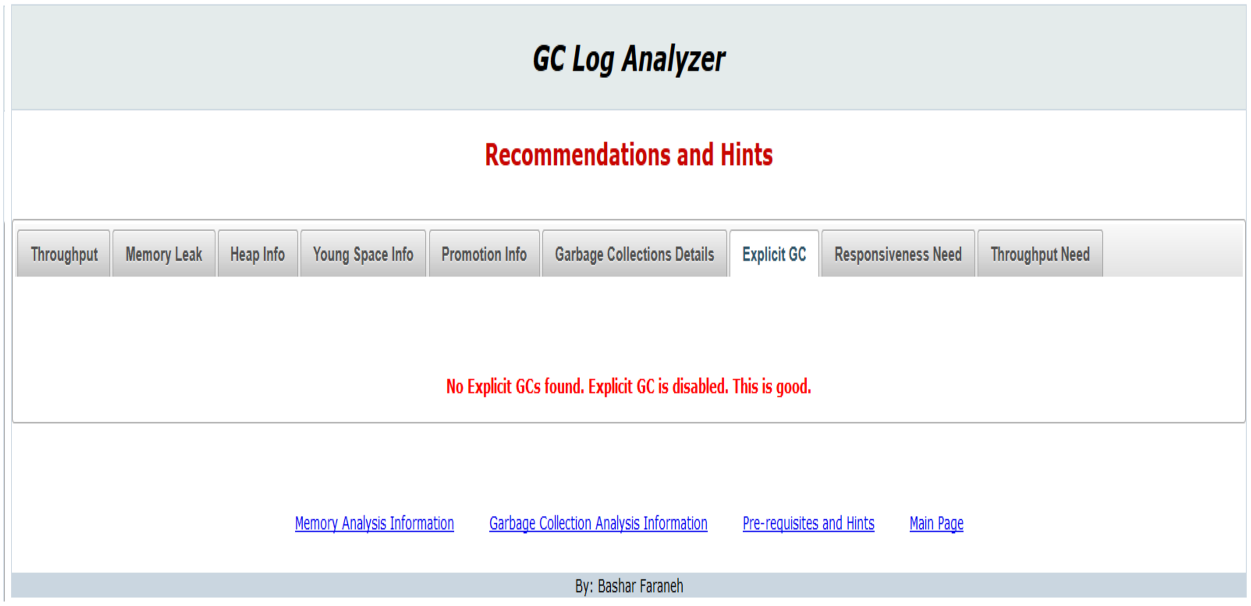
Figure 44: GC Log – Young Space Information

Figure 44 illustrates the Young space detailed information:

- Survivor Size.
- Eden Size.
- Total Young space size.

The small size of Survivor size could result in premature promotion which as a result increases the possibility of FULL garbage collections, which as a result, suspends the application for more time.

5.5.7 Explicit GC


GC Log Analyzer

Recommendations and Hints

No Explicit GCs found. Explicit GC is disabled. This is good.

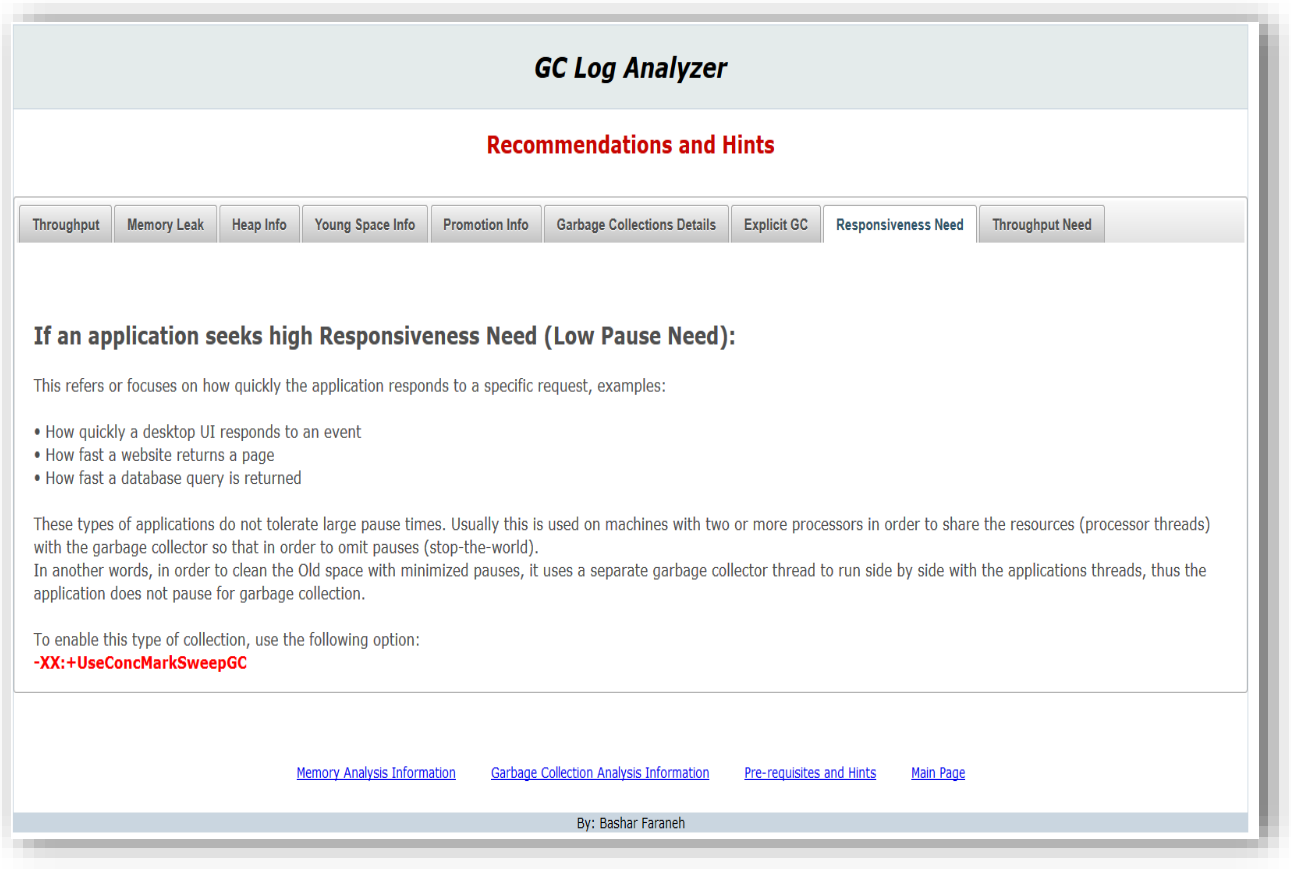
Memory Analysis Information
Garbage Collection Analysis Information
Pre-requisites and Hints
Main Page

By: Bashar Faraneh

Figure 45: GC Log – Explicit GC Information

Figure 45 shows if there is an explicit garbage collection which causes Full GCs. Explicit GCs should be avoided.

5.5.8 Responsiveness Need



If an application seeks high Responsiveness Need (Low Pause Need):

This refers or focuses on how quickly the application responds to a specific request, examples:

- How quickly a desktop UI responds to an event
- How fast a website returns a page
- How fast a database query is returned

These types of applications do not tolerate large pause times. Usually this is used on machines with two or more processors in order to share the resources (processor threads) with the garbage collector so that in order to omit pauses (stop-the-world).
In another words, in order to clean the Old space with minimized pauses, it uses a separate garbage collector thread to run side by side with the applications threads, thus the application does not pause for garbage collection.

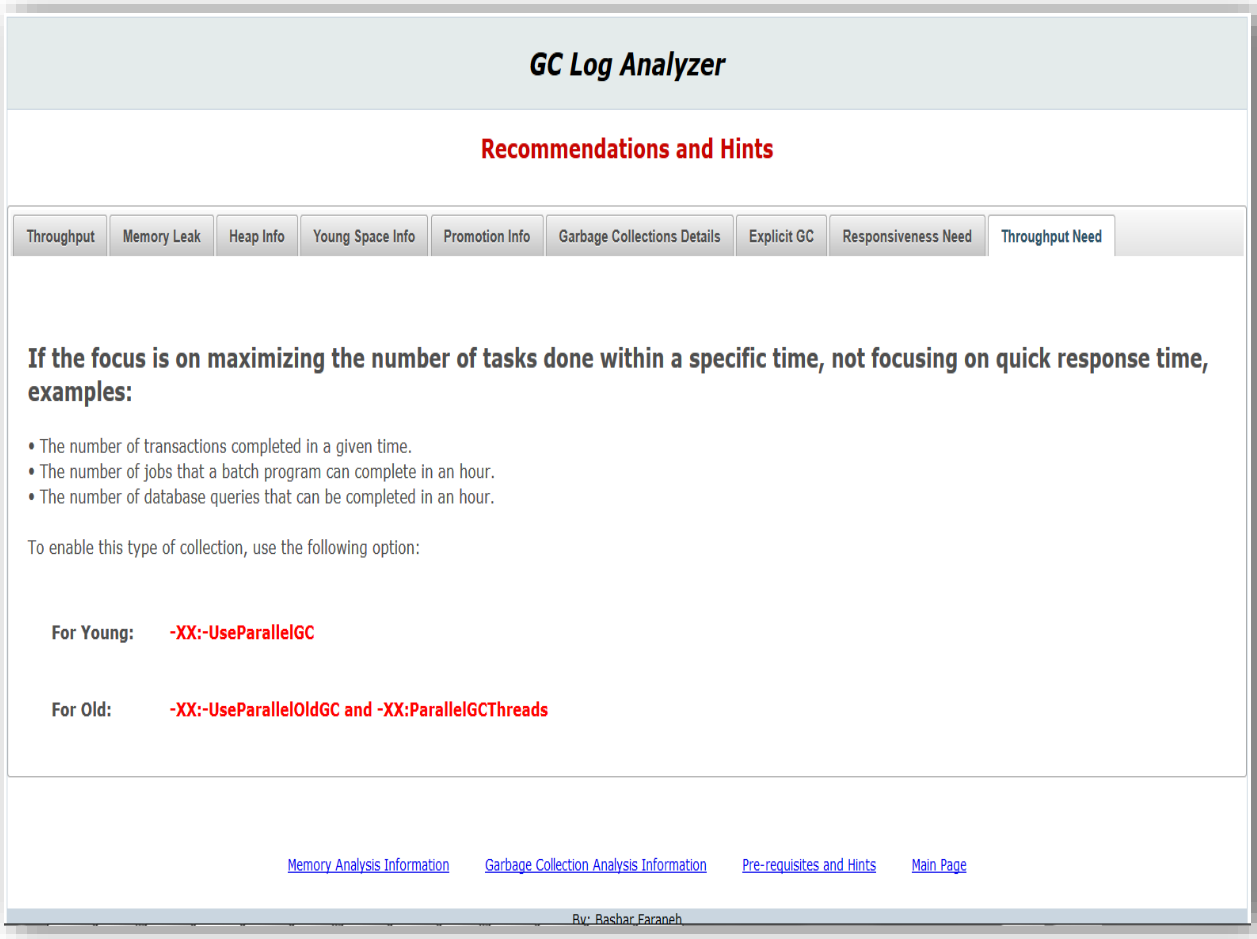
To enable this type of collection, use the following option:
-XX:+UseConcMarkSweepGC

Memory Analysis Information
Garbage Collection Analysis Information
Pre-requisites and Hints
Main Page
By: Bashar Faraneh

Figure 46: GC Log – Responsiveness Need

Figure 46 illustrates how to fulfill the responsiveness need of an application. As hints and guidelines with examples for real cases where the responsiveness is needed.

5.5.9 Throughput Need


GC Log Analyzer

Recommendations and Hints

If the focus is on maximizing the number of tasks done within a specific time, not focusing on quick response time, examples:

- The number of transactions completed in a given time.
- The number of jobs that a batch program can complete in an hour.
- The number of database queries that can be completed in an hour.

To enable this type of collection, use the following option:

For Young: **-XX:-UseParallelGC**

For Old: **-XX:-UseParallelOldGC and -XX:ParallelGCThreads**

Memory Analysis Information
Garbage Collection Analysis Information
Pre-requisites and Hints
Main Page
By: Bashar Faraneh

Figure 47: GC Log – Throughput Need

Figure 47 illustrates how to fulfill the throughput need of an application. As hints and guidelines with examples for real cases where the throughput is needed.

1.46 5.6 Tool Technical Details

Used tools/technologies:

- The used technology to implement such tool was the latest Java version which is 1.6.
- JSF based implementation; PrimeFaces was used to draw charts.
- Java I/O API's to read and parse information.
- Gcviewer API's that reads certain information from the gc log.
- Eclipse IDE for Java as the development.
- Java JMX for performance readings.

1.47 5.7 Samples from code

```
/**
 * This method is used to find possible indications of memory leak, then these readings
 * will be analyzed to check if the pattern indicates a memory leak.
 * @return
 */
public List getPossibleMemoryLeaks() {

    List possibleLeaks = new LinkedList();
```

```
                for    (Iterator    iterator    =    gcModel.getGCEvents();
iterator.hasNext();) {
                        GCEvent event = (GCEvent) iterator.next();
                        if (!event.isFull()) {
                                continue;
                        }


                        int preUsed = event.getPreUsed();
                        int postUsed = event.getPostUsed();


                        if((postUsed >= preUsed) || isPostLessByLittle(preUsed,
postUsed)) {
                                possibleLeaks.add(true);
                        } else {
                                possibleLeaks.add(false);
                        }


                }
                return possibleLeaks;
        /**
         *
```

```
 * @return
 */
public boolean isExplicitGCDisabled() {

        RuntimeMXBean                    runtimeMxBean                    =
ManagementFactory.getRuntimeMXBean();
        List<String> arguments = runtimeMxBean.getInputArguments();
        StringBuilder text = new StringBuilder();
        for (Iterator iterator = arguments.iterator(); iterator.hasNext();) {
            String string = (String) iterator.next();
            text.append(string);

        }
        return SystemInfoUtility
        .isParameterExists(text.toString(), "-XX:+DisableExplicitGC");
}
}
```

```
	/**
	 *
	 * @return
	 */
	public boolean isFullGCsTooFrequent() {

		Date firstCollectionTime = null;
		int numberOfGcs = 0;
		List possibleTooFrequentMinorGCsList = new ArrayList();
		for (Iterator iterator = gcModel.getFullGCEvents();
iterator.hasNext();) {
			GCEvent event = (GCEvent) iterator.next();
			numberOfGcs++;
			if (firstCollectionTime == null) {
				firstCollectionTime = event.getDatestamp();
				continue;
			}
			Date currentGCCollectionTime = event.getDatestamp();
			if (getDateDifference(firstCollectionTime,
currentGCCollectionTime) < 10) {
				possibleTooFrequentMinorGCsList.add(1);
			}
```

```
                        continue;


                }
                if (((possibleTooFrequentMinorGCsList.size() / numberOfGcs) *
100) >= 60) {
                        // most of the garbage collections are very frequent.. so this
is
                        // not normal
                        return true;
                }
                return false;
        }
```

```
public int getMaxTenuringThreshold() {

        try {

            BufferedReader br = new BufferedReader(new
FileReader(file));
            String line = br.readLine();

            while (line != null) {

                if (line.contains("Desired survivor size")) {

                    int indexOfSize = line.indexOf("max");
                    int indexOfBytes = line.indexOf(")");
                    String s1 = line.substring(indexOfSize + 3,
indexOfBytes);
                    return (Integer.valueOf(s1.trim()));
                }
                line = br.readLine();

            }
```

```
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

                return -1;
        }
        /**
         *
         * @param gcModel
         * @return
         */
        public boolean shouldSetMaxAndMinHeapSame(GCModel
gcModel) {

                double averageHeapAllocated =
gcModel.getHeapAllocatedSizes().average();

                double maxHeapAllocatedSize =
gcModel.getHeapAllocatedSizes().getMax();
                double minHeapAllocatedSize =
gcModel.getHeapAllocatedSizes().getMin();
```

```
                double maxAvgDifference = maxHeapAllocatedSize -
averageHeapAllocated;
                double minAvgDifference = averageHeapAllocated -
minHeapAllocatedSize;

                if ((minAvgDifference < 0) || (maxAvgDifference >
minAvgDifference)) {
                        return false;
                }
                return true;

        }
```

Chapter Six - Conclusions and Future Work

To summarize up what this work has presented:

- ✓ Detecting memory related issues during the life-time of the application in a very simple way.
- ✓ Provided a tool which helps targeting and solving memory issues by which:
  1. Provide an efficient/easy way to know performance issues related to memory before it turns out to be a show-stopper issue.
  2. Provide a decent way to know in details, how the garbage collector is performing.
  3. Provide a way to tune garbage collection performance.
  4. Provide a way to know if there is a memory leak, even if the application's performance seems fine.

  Up to this point, the researcher believes that this tool is good and sufficient for now, still could bare a number of modifications and additions to add up more features.

As an enhancement, adding the ability to analyze memory based on a full working business day. Doing so enables analyzing the performance of an application for a specific day in order to get more accurate readings.

As another addition would be to add up the system's admin email to enable the tool to send the log on a daily basis to analyze the performance of the customer's environment, even if the users are not complaining from anything.

What the tool is used to tackle is memory related issues, now in order to target most of the performance issues, Thread analyzer could be developed to provide features not available in the current thread dump analysis tools, by this, most of the performance issues are covered that are of type: Memory or CPU related.

Finally, the methodologies presented in this work shifts our way of approaching performance issues to a new level to be more proactive rather than reactive.
Rather than waiting for a serious/severe performance issues to come up, this new approach help discovering such issues before, thus preventing them from exploding.

References

[1] http://docs.oracle.com/cd/B14117_01/server.101/b10726/overview.htm. Accessed on Nov 2013

[2] Jose Manuel Velasco, David Atienza Katzalin Olcoz “Memory power optimization of Java-based embedded systems exploiting garbage collection information”, University of Madrid, August 2009

[3] https://plumbr.eu/blog/what-is-a-memory-leak Accessed on Oct 3, 2013

[4] http://pic.dhe.ibm.com/infocenter/isa/v4r1m0/index.jsp?topic=%2Fcom.ibm.java.diagnostics.memory.analyzer.doc%2Fheapdump.html Accessed on Sep 14, 2013

[5] http://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html accessed on Dec 4, 2013

[6] http://architects.dzone.com/articles/how-tune-java-garbage accessed on Sep 17, 2013

[7] Carol McDonald, Sun Microsystems, http://www.slideshare.net/caroljmcdonald/java-garbage-collection-monitoring-and-tuning, accessed on 4 Nov 2013.

[8] http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html, accessed on Sep 28, 2013

[9] http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html Accessed on Oct 5, 2013.

[10] http://www.c0t0d0s0.org/archives/6617-About-Java,-parallel-garbage-collection-and-processor-sets.html accessed on Nov 10, 2013

[11]
http://docs.oracle.com/cd/E13209_01/wlcp/wlss30/configwlss/jvmgc.html
accessed on Sep 14, 2013.

[12] Guoging Xu, Atanas Rountev “Precise Memory Leak Detection for Java Software Using Container Profiling”, University of Ohio, May 2008

[13] Leon Chen, Java consultant at Oracle, Introduction of Java GC Tuning and Java mission Control, http://www.slideshare.net/leonjchen/java-optimization-twjug

[14] http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.htm accessed on 17 Oct 2013

[15]
http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html accessed on Dec 4, 2013.

[16]
http://pic.dhe.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=%2Fcom.ibm.websphere.express.doc%2Finfo%2Fexp%2Fae%2Frprf_hotspot_parms.html, accessed on Sep 5, 2013.