ISSN: 2277–3061 (online)

International Journal of Computers & Technology
Volume 2 No. 3, June, 2012

A Study Paper on Performance Degradation due to Excessive Garbage Collection in Java Based Applications using Profiler

Emi Retna. J

Abstract

Applications are becoming more complex, more larger and demand high quality. Application Server is the component on which most of the applications are hosted. It acts in the middle tier providing lot of functionalities like transaction management, caching ,persistence clustering etc. There are a variety of application servers to choose from like JBoss, Websphere, Tomcat etc, some are open source while others are proprietary. Quality parameters like performance, availability, scalability, maintainability, re-usability vary between different application servers. The application servers can be analyzed and monitored for performance using various tools and metrics. The quality parameters of an application server are affected by various factors such as memory leak, poor performing code, etc. It is necessary to evaluate the performance, to verify if the quality requirements are met. Software Engineering has several methodologies, metrics and calculations to evaluate the quality requirements. [1] discusses the various measurements and metrics that can be used to calculate the quality parameters.

Key Words - garbage collection, profiling, generation, calibration

Lot of factors affects the performance of a Java application. Factors that directly affect the performance of a Java application are heap size, threads and connection pooling. If more heap size is allocated then more java objects can be stored in the heap space. So the garbage collection can take place sparingly or the application can utilize more CPU cycles on performing the functionality rather than garbage collecting. But when garbage collection takes place, more objects needs to be searched for collecting the garbage object. On the other hand if less heap space is allocated the application does garbage collection very often. In this paper we discuss on the memory usage monitoring and garbage collection in Java based applications. Excessive garbage collection affects the performance of a Java based application though there are tunable parameters in JVM to perform garbage collection. The tool used for memory usage analysis is the Netbeans IDE profiler. The result shows that for high scaling applications improper garbage collection and memory leak can be a serious bottleneck that needs to be addressed.

This project is a motivation of a research work on performance study in Java based applications. This paper proposes an analysis on memory that may cause poor performance in an application. Problems like memory leak slow down the processing of jobs [2]. The memory leak needs to be identified at the early stage of deployment and remedial actions taken to fine tune the code so as to avoid memory leaks.

Experiments - Identifying Memory Leaks

There are several tools to analyze memory and monitor memory usage. The Netbeans IDE profiler is one among them that records the behavioral aspects of the application. The profiler is a tool that has been inbuilt with the latest version of Netbeans IDE. It helps in analyzing the memory, the heap size etc. The process involved in performing the profiling is out of the scope of this paper.

These experiments were conducted in uniprocessor system. The Java platform used was jdk1.6 in Windows operating environment.

Several case studies were performed and access the memory utilization and how they affect the overall performance of an application server. The results are discussed below.

Case Study 1 and Evaluation Results

The NetBaeans profiler was used to evaluate some of the Java applications developed. The Java application developed was deployed in NetBeans IDE and the JBOSS Application Server was started. Inorder to run the NetBeands profiler the following steps were followed;

1. Run the profiler calibration. Calibration is done on the JDK running on the machine.This is a one time action.
2. Integrate the profiler with the project
3. Analyze the results of memory profiling
   The entire application can be profiled or parts of application alone can be profiled. Further there is option of profiling all Java classes or only project classes. There is option of filtering out Java core classes alone. User defined profiling points alone can also be taken up for analysis.

The experiment was conducted to profile the entire application as well as only core java classes of the application.

Results

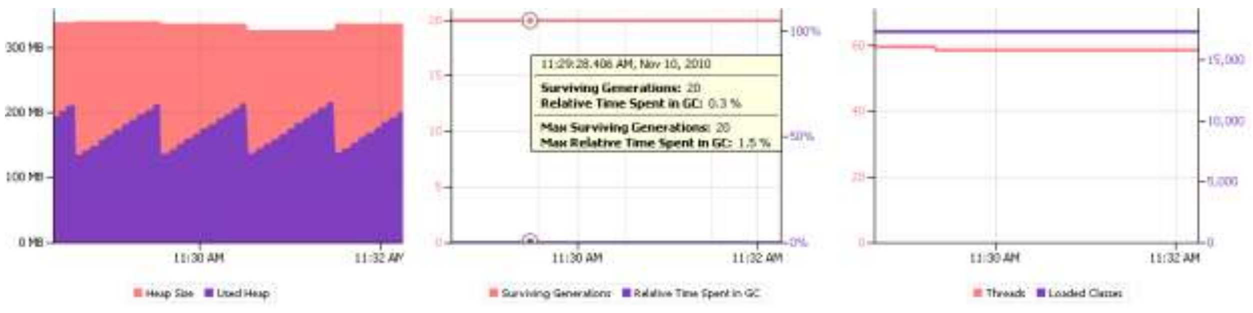


Fig 1: Memory profiling result of entire application

Inference:

- The red shade indicates the allocated size of the JVM heap size (left graph).
- The purple shade indicates the amount of JVM heap size actually in use (left graph).
- The red shade indicates the count of active threads in the JVM (right graph).
- The purple shade indicates the classes loaded in the JVM (right graph).
- The red shade indicates the surviving JVM objects that has not been garbage collected over time. As the object survives different garbage collections that occur, the age of the object increases. The age of the object is the number of garbage collection that the object has survived(centre graph)..
- The purple shade indicates the percentage of execution time spent by JVM doing garbage collection (centre graph). This percentage of execution time the JVM does not execute the application. If this percentage is more then the garbage collection parameters needs to be tuned in JVM.
- Count of generation = age of object1 + age of object 2 +….+age of object n

In the experiment:

- The allocated JVM heap size is above 300 megabytes (left graph).
- The JVM heap size actually in use varies from above 100 megabytes to 200 megabytes (left graph).

There is a provisioning to run the garbage collection forcibly. When the garbage collection was forcibly done, the results obtained are as shown in the below figure:

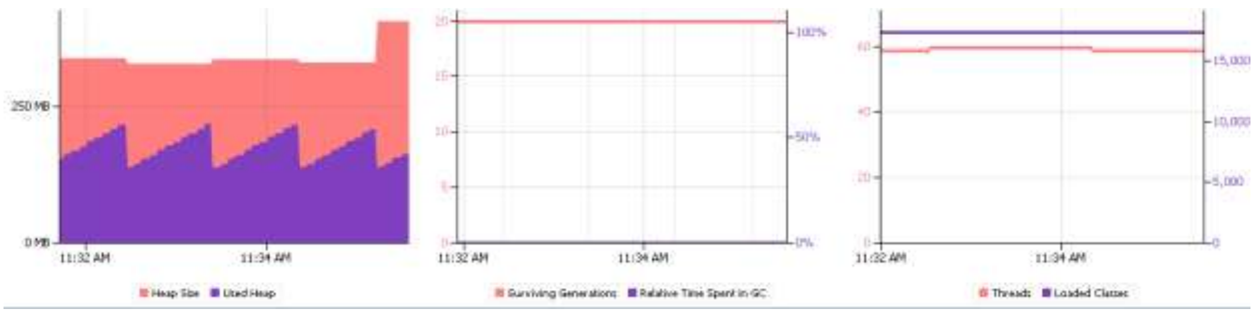
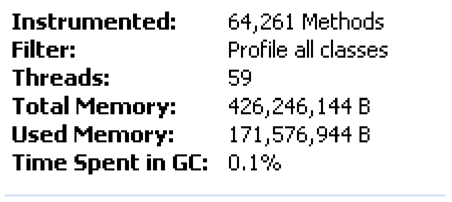


Fig 2: Memory profiling – forced garbage collection

The basic telemetry information of the project as given by the tool is as given below:



Instrumented:	64,261 Methods
Filter:	Profile all classes
Threads:	59
Total Memory:	426,246,144 B
Used Memory:	171,576,944 B
Time Spent in GC:	0.1%

Fig 3: Basic telemetry information – entire application

While the experiment is being conducted if there is a need to convert profiling to only project classes that can also be done. This need may arise due to the factors that too much data collected and the profiler ran out of memory. OutOfMemoryError may occur due to several reasons including HttpUnit doesn't support Java Script processing.

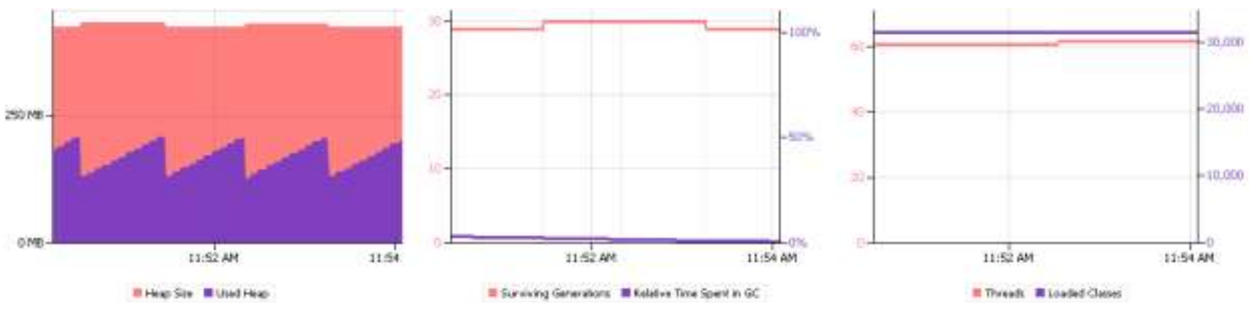
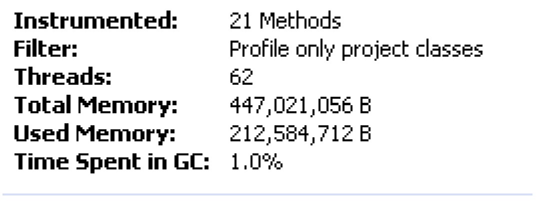


Fig 4: Memory profiling result of only project classes

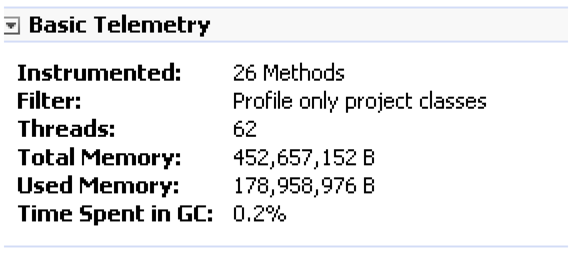
Now the basic profiler information is as given below:



Instrumented:	21 Methods
Filter:	Profile only project classes
Threads:	62
Total Memory:	447,021,056 B
Used Memory:	212,584,712 B
Time Spent in GC:	1.0%

Fig 5: Basic telemetry information – only project classes

When some major operation was performed the below data was obtained:

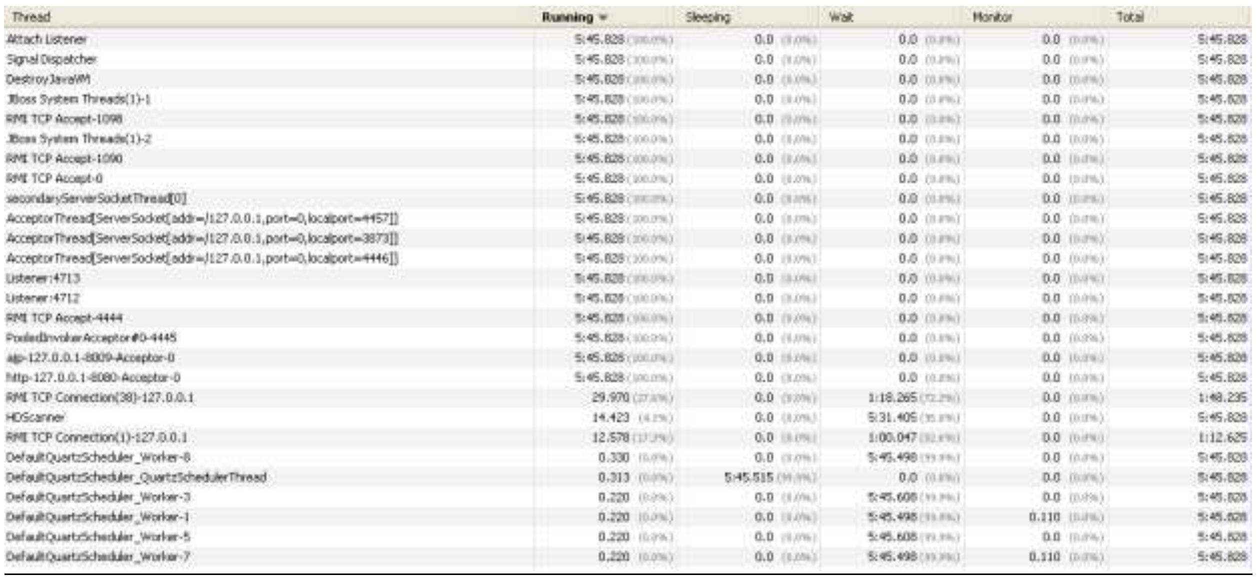


Basic Telemetry

Instrumented:	26 Methods
Filter:	Profile only project classes
Threads:	62
Total Memory:	452,657,152 B
Used Memory:	178,958,976 B
Time Spent in GC:	0.2%

Fig 6: Basic telemetry information – only project classes

Both object creation and garbage collection can be recorded and analyzed. The thread data can be viewed available is :



Thread	Running	Sleeping	Wait	Monitor	Total
Attach Listener	5:45.828 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	5:45.828
Signal Dispatcher	5:45.828 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	5:45.828
DestroyJavaVM	5:45.828 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	5:45.828
JBoss System Threads(1)-1	5:45.828 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	5:45.828
RMI TCP Accept-1098	5:45.828 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	5:45.828
JBoss System Threads(1)-2	5:45.828 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	5:45.828
RMI TCP Accept-1090	5:45.828 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	5:45.828
RMI TCP Accept-0	5:45.828 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	5:45.828
secondaryServerSocketThread[0]	5:45.828 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	5:45.828
AcceptorThread[ServerSocket[addr=/127.0.0.1,port=0,localport=4457]]	5:45.828 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	5:45.828
AcceptorThread[ServerSocket[addr=/127.0.0.1,port=0,localport=3873]]	5:45.828 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	5:45.828
AcceptorThread[ServerSocket[addr=/127.0.0.1,port=0,localport=4446]]	5:45.828 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	5:45.828
Listener:4713	5:45.828 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	5:45.828
Listener:4712	5:45.828 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	5:45.828
RMI TCP Accept-4444	5:45.828 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	5:45.828
PooledInvokerAcceptor#0-4445	5:45.828 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	5:45.828
ajp-127.0.0.1-8009-Acceptor-0	5:45.828 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	5:45.828
http-127.0.0.1-8080-Acceptor-0	5:45.828 (100.0%)	0.0 (0.0%)	0.0 (0.0%)	0.0 (0.0%)	5:45.828
RMI TCP Connection(38)-127.0.0.1	29.970 (27.8%)	0.0 (0.0%)	1:18.265 (72.2%)	0.0 (0.0%)	1:48.235
HDScanner	14.423 (4.2%)	0.0 (0.0%)	5:31.405 (95.8%)	0.0 (0.0%)	5:45.828
RMI TCP Connection(1)-127.0.0.1	12.578 (17.3%)	0.0 (0.0%)	1:00.047 (82.7%)	0.0 (0.0%)	1:12.625
DefaultQuartzScheduler_Worker-8	0.330 (0.0%)	0.0 (0.0%)	5:45.498 (99.9%)	0.0 (0.0%)	5:45.828
DefaultQuartzScheduler_QuartzSchedulerThread	0.313 (0.0%)	5:45.515 (99.9%)	0.0 (0.0%)	0.0 (0.0%)	5:45.828
DefaultQuartzScheduler_Worker-3	0.220 (0.0%)	0.0 (0.0%)	5:45.608 (99.9%)	0.0 (0.0%)	5:45.828
DefaultQuartzScheduler_Worker-1	0.220 (0.0%)	0.0 (0.0%)	5:45.498 (99.9%)	0.110 (0.0%)	5:45.828
DefaultQuartzScheduler_Worker-5	0.220 (0.0%)	0.0 (0.0%)	5:45.608 (99.9%)	0.0 (0.0%)	5:45.828
DefaultQuartzScheduler_Worker-7	0.220 (0.0%)	0.0 (0.0%)	5:45.498 (99.9%)	0.110 (0.0%)	5:45.828

Timestamp	Heap Size (Bytes)	Used Heap (Bytes)
Nov 10, 2010 11:20:24 AM	128647168	5897920
Nov 10, 2010 11:20:25 AM	128647168	8608272
Nov 10, 2010 11:20:27 AM	128647168	12663608
Nov 10, 2010 11:20:28 AM	128647168	25523112
Nov 10, 2010 11:20:29 AM	128647168	30235744
Nov 10, 2010 11:20:30 AM	128647168	9832240
….	…	…
Nov 10, 2010 11:28:29 AM	356909056	214367632
Nov 10, 2010 11:28:30 AM	356909056	214367632

<u>CaseStudy 2 and Evaluation Results</u>

The same experiment was repeated for several projects. Another result of a sample application is as given below:

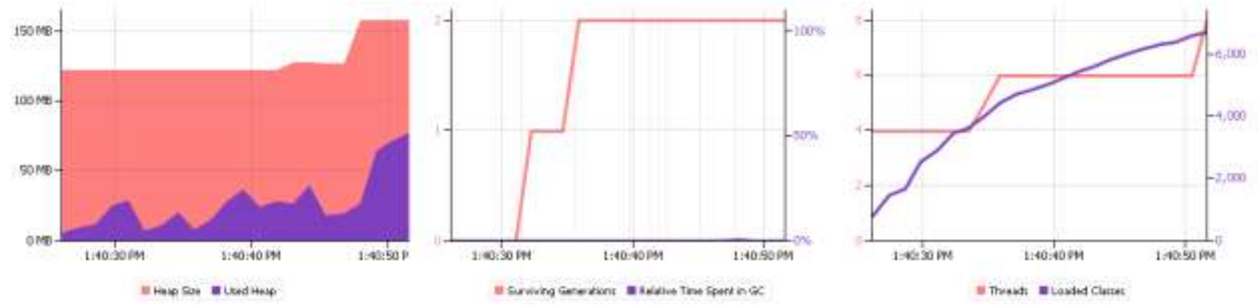
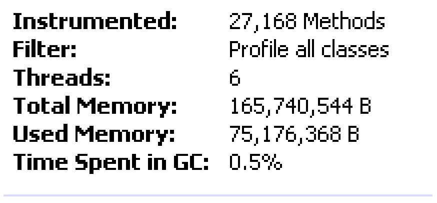


Fig 7: Memory profiling result of entire application



Instrumented:	27,168 Methods
Filter:	Profile all classes
Threads:	6
Total Memory:	165,740,544 B
Used Memory:	75,176,368 B
Time Spent in GC:	0.5%

Fig 8: Basic telemetry information – entire application

<u>CaseStudy 3 and Evaluation Results</u>

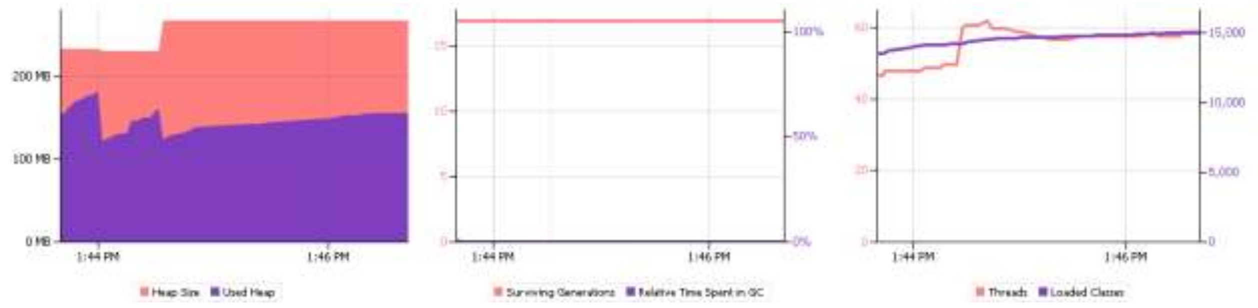
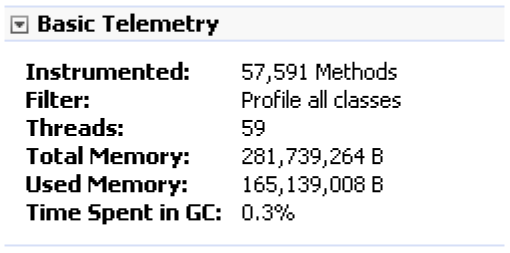


Fig 9: Memory profiling result of entire application



Basic Telemetry

Instrumented:	57,591 Methods
Filter:	Profile all classes
Threads:	59
Total Memory:	281,739,264 B
Used Memory:	165,139,008 B
Time Spent in GC:	0.3%

Fig 10: Basic telemetry information – entire application

Modifying the profiler to profile only project classes the following results were obtained

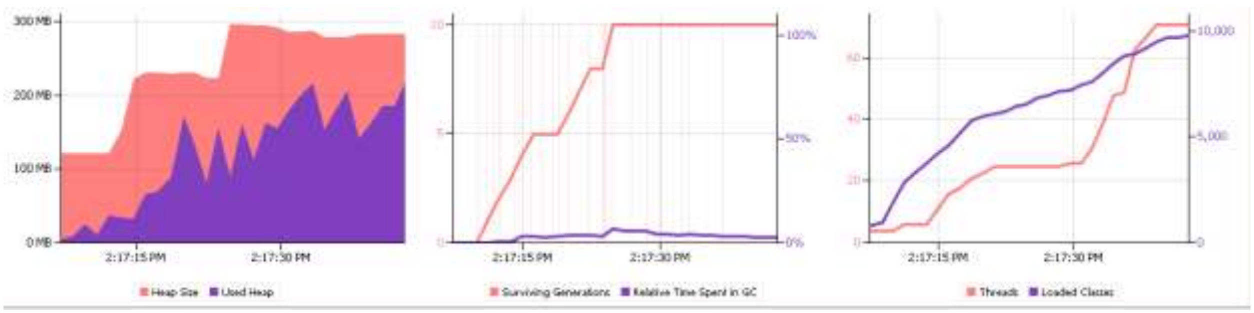
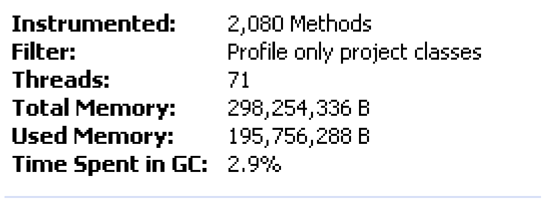


Fig 11: Memory profiling result of only project classes



Instrumented:	2,080 Methods
Filter:	Profile only project classes
Threads:	71
Total Memory:	298,254,336 B
Used Memory:	195,756,288 B
Time Spent in GC:	2.9%

Fig 12: Basic telemetry information – project classes only

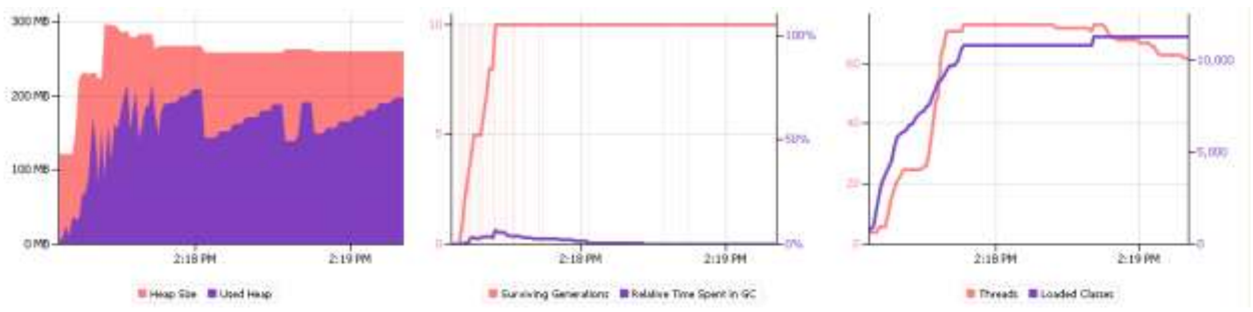
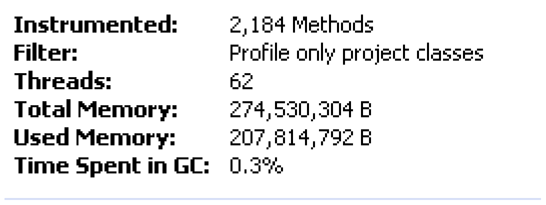


Fig 13: Memory profiling result of only project classes



Instrumented:	2,184 Methods
Filter:	Profile only project classes
Threads:	62
Total Memory:	274,530,304 B
Used Memory:	207,814,792 B
Time Spent in GC:	0.3%

Fig 14: Basic telemetry information – project classes only

On forcibly running GC the values obtained are as follows:

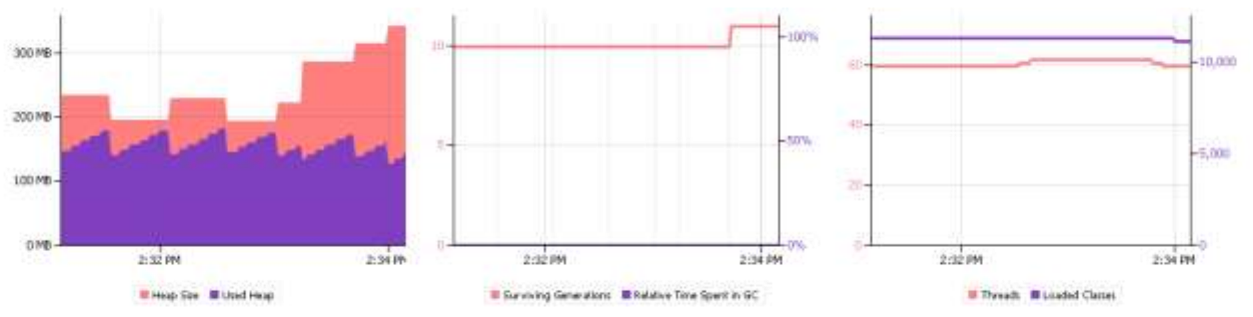
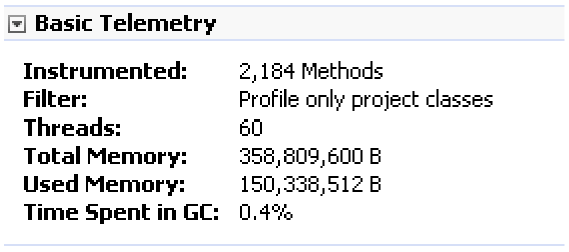


Fig 15: Memory profiling – forced garbage collection



Basic Telemetry

Instrumented:	2,184 Methods
Filter:	Profile only project classes
Threads:	60
Total Memory:	358,809,600 B
Used Memory:	150,338,512 B
Time Spent in GC:	0.4%

Fig 14: Basic telemetry information – project classes only on forced GC

Inference: When the GC is forcibly run the used memory is getting reduced. Reducing garbage collection times increase the performance. Garbage collection (GC) delays can be successfully lowered to by using parallel garbage collection algorithms. But most of the application do have modules which executes sequentially and does not do parallel processing.

Even if the JVM spends 1% of time in garbage collection in a uniprocessor system, it translates to more than a 20% loss in throughput on 32 processor systems [3]. Hence this is a major problem that needs to be addressed. Improvements in garbage collection process or avoiding memory leaks to a smaller extend can show better performance results. Garbage collection may become a principle bottleneck for large scale applications.

Acknowledgement

The author expresses deep gratitude to Karunya University for providing infrastructure for carrying out the research work. The author thanks the senior professors and the anonymous reviewers for roviding valuable suggestions to improve the quality of the research paper.

References

[1] J.Emi Retna, Greeshma Varghese, Merlin Soosaiya, Sumy Joseph, “A Study on Quality Parameters of Software and the Metrics for Evaluation” International journal of Computer Engineering & Technology (IJCET),ISSN Print : ISSN 0976 – 6367 ISSN Online: ISSN 0976 – 6375 Volume 1, Issue 1(2010)

[2] Aad P. A. van Moorsel, Katinka Wolter, “ Analysis of Restart Mechanisms in Software Systems” IEEE Transactions on Software Engineering Vol 32, No.8, August 2006

[3]http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html