CHAPTER – ONE

BACKGROUND AND INTRODUCTION

1.1 Background

1.1.1 Java Virtual Machine JVM

When java compiler compiles the source code it produces an intermediate code known as bytecode for a machine that does not exist. This machine is called the java virtual machine (JVM) and it exists only inside the computer memory. It is a simulated computer within the computer and does all major functions of a real computer. Fig below illustrates the process of compiling a java program into bytecode which is also referred to as Virtual machine code [10]. The virtual machine code is not machine specific. The machine specific code (known as machine code) is generated by the java interpreter by acting as an intermediary between the virtual machine and the real machine as shown in figure below.

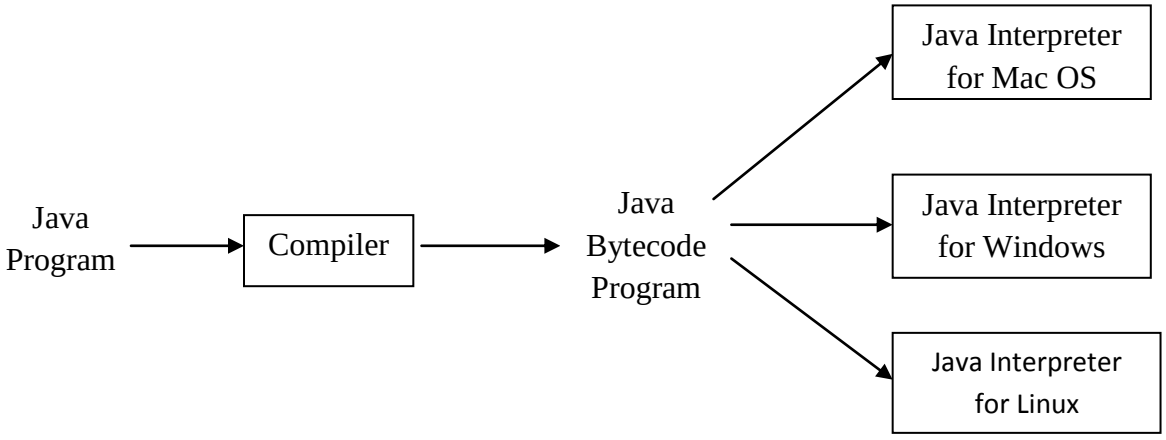


Fig 1.1: Compiling java program

Java Virtual Machine can be defined as a software module that executes java application bytecode and translates the byte code into hardware and operating system-specific instructions. By doing so, the JVM enables java program to be executed in different environments from where they were first written, without requiring any

changes to the original application code [7]. JVM performs the function of allocating memory as objects are created and freeing when they are no longer needed because different operating systems and hardware platform vary in the way they manage the memory. The process of freeing unused memory is called 'garbage collection' and is performed by JVM on memory heap during program execution [7].

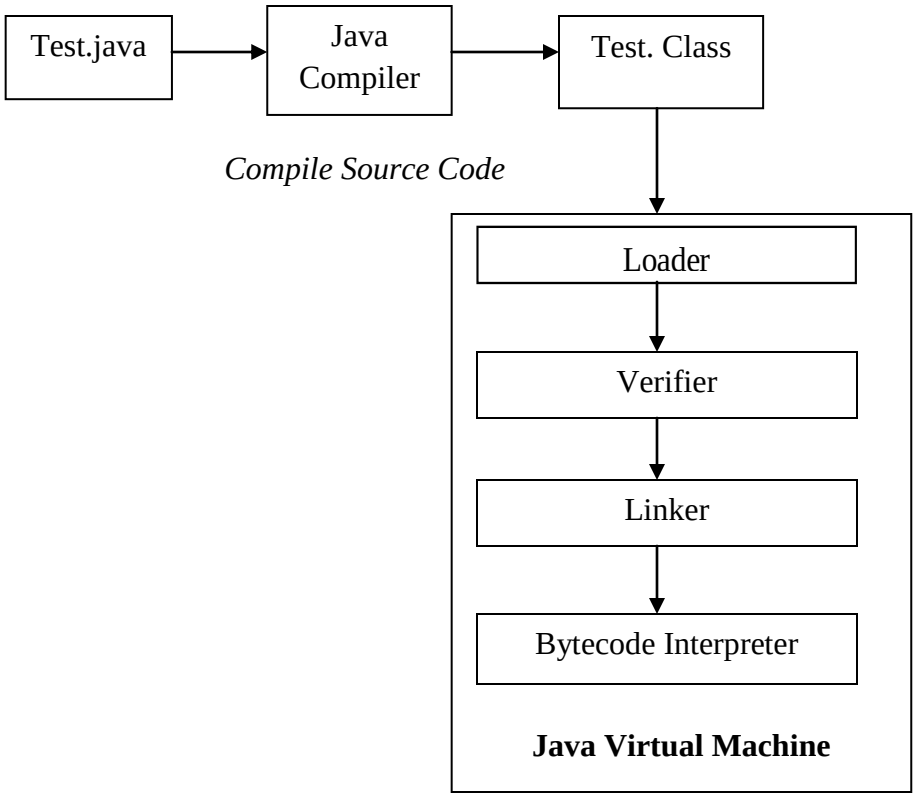


Fig 1.2: JVM Architecture

The java virtual machine (JVM) is a stack-based machine targeted towards the execution of compiled java programs. It receives as input a single class file, containing the definition of the main class of the program, and executes a particular method of that class. As the execution proceeds, the class files corresponding to the other classes needed by the program are loaded on demand.

One of the novel aspects of the JVM is that it verifies the class files after loading them, to make sure that their execution will not violate some defined safety properties. The checks performed by the verifier include a complete type checking of the program.

To make this type checking possible, all the instructions of the JVM are typed: the type of the operands they expect on the stack as well as the type of the result they push back can be inferred from the instructions themselves or their parameters. The type language of the JVM is very close to the one of the java language itself prior to version 5, i.e. without generics: a type is a class or interface name, or one of the nine primitive types (boolean, int, etc).

Although very java centric, the JVM has been used extensively over the recent years as a target platform for many compilers. A web page listing such compilers and interpreters currently contains close to 190 entries. This popularity seems to be due mostly to two factors [19]:

1. A large body of java libraries is available , and targeting the JVM instantly gives access to them,
2. The JVM is available on all major platforms, and compiled java programs run on all of them without needing any recompilation.

1.1.2 Explicit vs. Automatic Memory Management

Automatic memory management or garbage collection provides significant software engineering benefits over explicit memory management. Garbage collection frees the programmer from the burden of memory management and improves modularity, while preventing accidental memory overwrites ("dangling pointers") and security violations [20]. However garbage collection must perform more work than explicit memory managers, which rely on the programmer to indicate when to deallocate individual objects. Garbage collectors, on the other hand, must periodically identify objects reachable by pointer traversal, and reclaim those that are unreachable.

Memory management is the process of recognizing when allocated objects are no longer needed, deallocating (freeing) the memory used by such objects, and making it available for subsequent allocations. In some programming languages, memory management is the

programmer's responsibility. The complexity of that task leads to many common errors that can cause unexpected or erroneous program behavior and crashes. As a result, a large proportion of developer time is often spent debugging and trying to correct such errors.

One problem that often occurs in programs with explicit memory management is dangling references. It is possible to deallocate the space used by an object to which some other object still has a reference. If the object with that dangling reference tries to access the original object, but the space has been reallocated to a new object, the result is unpredictable and not what was intended. Another common problem with explicit memory management is space leaks. These leaks occur when memory is allocated and no longer referenced but is not released.

An alternative approach to memory management that is now commonly utilized, especially by most modern object oriented languages, is automatic management by a program called a garbage collector. Automatic memory management enables increased abstraction of interfaces and more reliable code.

1.1.3 Memory Fragmentation and Compaction

Fragmentation is the tendency of the memory to get broken up into smaller pieces. Contiguous dead space between objects may not be large enough to fit new object [26]. If subjected to mark and sweep repeatedly, overtime the heap gets fragmented. For example as shown in Figure 1.3, consider a scenario where a memory has exactly 8 blocks. After Sweep phase some of the objects may have been reclaimed. Now suppose object5 needs to be inserted into the memory array, there is no contiguous space to add the new object, in spite of having enough space.

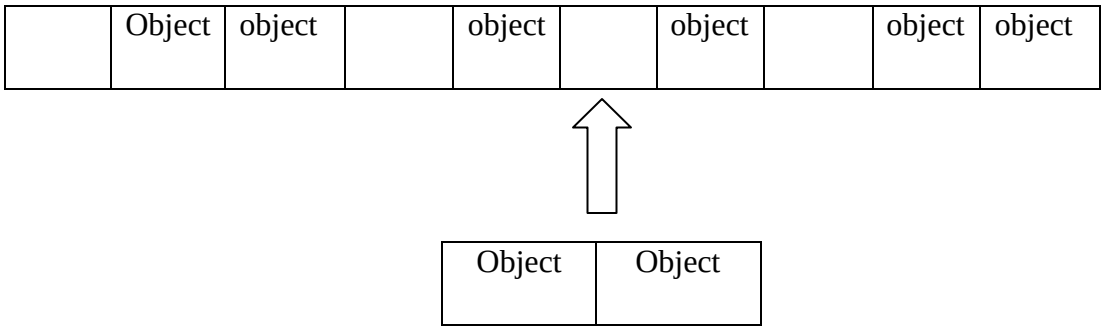
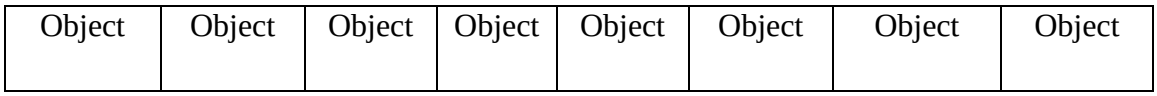


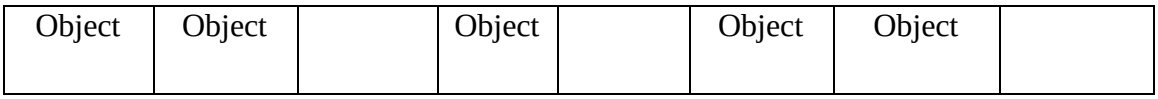
Figure 1.3: Memory Fragmentation

Fragmentation is taken care by another phase of garbage collection called Compaction. In this phase, objects are rearranged so that they occupy contiguous space. A compacting GC moves object during sweep phase. The three phases are summarized in Fig 1.4.

Initial State



After Mark Sweep



After Compaction

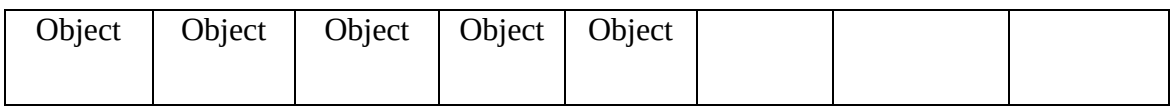


Figure 1.4: Memory Compaction

1.1.4 Garbage Collection

Historically Heap was managed explicitly by the programming by using allocate and release function calls in a library (such as malloc/free in C, and new/delete in C++) which may lead to many errors. As a result, large proportion of debugger time is often spent in debugging and trying to correct such errors [3]. Two problems are associated with explicit memory management; dangling references and memory leaks. Memory leaks occur, if we forget to free the memory which has been allocated and no longer referenced. If enough leaks occur, heap gets totally filled and program eventually runs out of memory. In order to avoid the problem of explicit memory management, an alternative approach, automatic memory management called garbage collector is applied by many modern object- oriented programming languages [3].

Garbage collection was invented by John McCarthy around 1959 to solve the problems of manual memory management in Lisp. It is portrayed as the opposite of manual memory management, which requires the programmer to specify which objects to deallocate and return to the memory system. However, many systems use a combination of the two approaches.

The basic principles of garbage collection are:

- Find data objects in a program that cannot be accessed in the future
- Reclaim the resources used by those objects

Garbage collection is the process of automatic storage reclamation in which those objects which are no longer referenced from any live objects or from program are collected. These objects are known as dead objects or garbage. The memory occupied by these objects is freed and added to the pool of free memory. In addition to freeing unreferenced object, a garbage collector may also combat heap fragmentation. Heap fragmentation occurs through the course of normal program execution. New objects allocated, and unreferenced objects are freed such that free portions of heap memory are left in between portions occupied by live objects. One of the advantages of garbage collection is that the garbage collection ensures program integrity. It is an important part of java security

strategy. Garbage collectors are becoming the essential part of compilers. Most of the high level languages like java and C# have incorporated garbage collectors for automatic memory management [6].

Garbage collectors are responsible for various memory management activities such as:-

- Memory allocation
- Preserving and ensuring object references are maintained in memory.
- Reclamation memory occupied by objects that are no longer in use or are unreachable from references.

Java puts all the newly created objects in a "heap". An object that is being use or is going to be used by the application is called a "Live objects". Opposite of a live object is garbage as in, the application cannot reference and cannot use the object. When the application no longer needs an object, the memory occupied by the object is cleared or reclaimed by the garbage collector so that the application can use it. Garbage collectors start collecting from the root object [23]. Root object is an object which can be directly accessed. i.e. without going other references. An object is considered live if it is referenced by a root object or other live object.

1.1.4.1 Desirable Garbage Collector Characteristics

Garbage collectors must have the following characteristics:

- A garbage collector must be both safe and comprehensive. That is, live data must never erroneously freed.
- It is also desirable that a garbage collector operate efficiently without introducing long pauses during application execution.
- Garbage collector rearranges the freed memory spaces into a single contiguous area such that there is always memory for allocation of a large object [17].

1.1.5. Java Heap Memory

The JVM allocates java heap memory from the OS and then manages the heap for the java application. When an application creates a new object, the JVM sub-allocates a contiguous area of heap memory to store it. An object in the heap that is referenced by any other object is "live," and remains in the heap as long as it continuous to be referenced. Object that is no longer referenced are garbage and can be cleared out of the heap to reclaim the space they occupy. The JVM performs a garbage collection (GC) to remove these objects, reorganizing the objects remaining in the heap [8].

The traditional HotSpot JVM algorithms divide the heap into "young" and "old" generations, creating most new objects in the young generation and only promoting them to the old generation after they survive minor garbage collections. The young generation occupies a small but highly active portion of the heap. It is organized to facilitate fast allocation of memory for new objects and frequent, fast garbage collections that do not have a heavy performance impact. The older generation holds longer-lived object in an area that occupies a much larger, less active portion of the heap. Garbage collections on the Old generation occur less frequently, but may have much greater impact on application performance if JVM threads are paused during the collection.

In java, the memory is divided into Heap memory and non Heap Memory. The non heap memory is actually two types. The method area or Permanent Generation is logically part of the Heap memory [25].

The heap holds the object data, Method Area/PermGen holds the class data and the per-class runtime constant pool. All class data are loaded into this (Method Area/PermGen) memory space. This includes the field and method data and the code for the methods and constructs.

The heap is divided into different regions to help the garbage collection process efficient and faster. The heap is broadly divided into a Young Generation Space and the Old or Tenured Generation area. The Young space is further divided into Eden and Survivor spaces. When objects are first allocated (new) they are stored in the Eden Space. When the object is in use for specific time or duration of the Garbage Collection Cycle, the object gets moved From the Eden to the Survivor Space. When it ages further it is promoted to the Old Generation or Tenured

Space. This mechanism ensures that there are enough space freed up for new or short lived objects and garbage collector can do an efficient job. The Young Generation is introduced based on the fact that most of the objects in the life time of a JVM is short lived and only a small percentage actually lives longer. Whenever the Garbage collector frees up objects in the Young generation, it is known as Minor Collection or Minor GC. The Garbage collection happening at the Old Generation or Tenured Generation is known as Major Collection or Major GC [8]. Fig: 1.2.1 shows java heap memory.

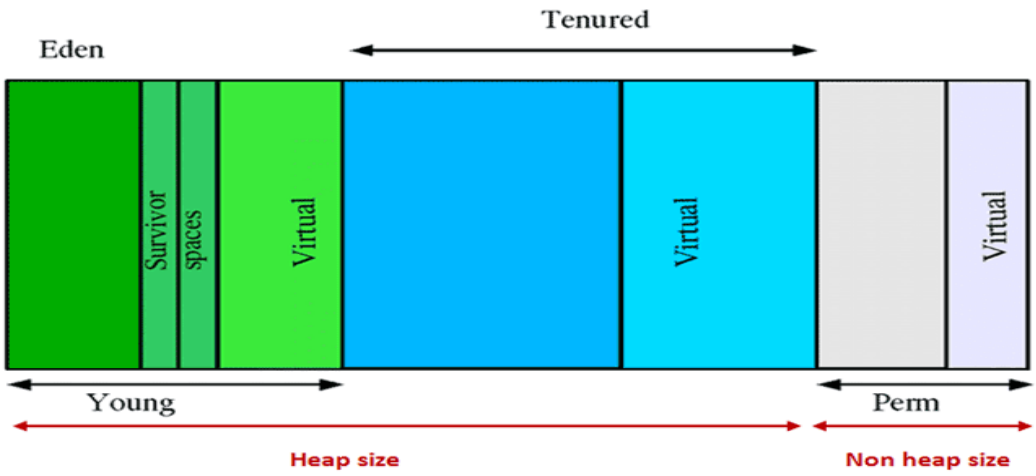


Fig: 1.5 Java Heap Memory

1.2. Introduction

A java virtual machine is a process virtual machine that can execute java bytecode. It is the code execution component of the java platform.

It is a software module that executes java application bytecode and translates the bytecode into hardware and operating system-specific instructions. By doing so, the JVM enables java programs to be executed in different environments from where they were first written, without requiring any changes to the original application code. The JVM frees the programmer from keeping track of references between objects and knowing how long they should be kept in the system. It also frees us from having us to decide exactly when to issue explicit instructions to free up memory [19].

The java platform is used for a wide array of applications ranging from small applet to web services on large servers. As part of memory management, java provides many garbage collectors, namely:

- Parallel Scavenge
- Serial Garbage Collector
- Parallel New +Serial GC
- CMS(Concurrent Mark Sweep)

1.2.1. Problem Definition

Garbage collection has greatly improved the productivity of software developers, making writing correct software easier in common sense. With garbage collection it is easier to reason about the correctness of a program, but usually harder to reason about its performance. Many different variations on the basic GC algorithms, all of which attempt to maximize some metrics for typical allocation patterns. As the java virtual machine has matured, there have been improvements introduced to minimize the impact of GC on applications.

This dissertation work will compare the performance of Serial, Parallel and Concurrent Mark Sweep Garbage collection algorithm in JVM in today's powerful PC environment.

1.2.2. Objective

The objective of this dissertation work is:

- To gain knowledge in depth of different garbage collection algorithm (Serial, Parallel and CMS) and also to measure the performance among these algorithm to find the better one.
- To determine the space-time trade-offs of these algorithms.

1.3. Performance Metrics

Several metrics are utilized to evaluate garbage collector performance, including:

- **Throughput:** The percentage of total time spent in garbage collection and allowing the application to perform, disregarding the pause times and memory required.
- **Pause time:** The length of time during which application execution is stopped while garbage collection is occurring.
- **Footprint:** Amount of memory required by the garbage collector to execute efficiently.
- **Promptness:** The time between when an object becomes garbage and when the memory becomes available.

1.4. Motivation

When memory is allocated dynamically, memory space for object of data is allocated in heap and remains allocated until it is freed manually by the programmer. Providing programmers with power of memory management may lead to many potential problems such as: freeing the memory earlier may create the problems of dangling references and forgetting to free the dynamically allocated memory may cause system crash if heap memory became full and there is no space for storing new objects or data in heap. Further, explicit memory management may results decreased productivity of programmers. To avoid these problems automatic memory management called garbage collection is evolved as one of the exciting area of research field.

1.5. Structure of Thesis

This dissertation work is organized into six chapters. Chapter one is the introduction part, which gives background and introduction of Java Virtual Machine (JVM), Memory Management and Garbage Collection. It deals the problem that arises with explicit memory management. It also deals with how the collector automatically manages the memory.

Chapter Two discuss the past related work in garbage collection. It also describes the methodology that is used in data collection.

Chapter Three provides the description and overview of garbage collector algorithms, namely: Serial garbage collector, Parallel garbage collector and Concurrent mark sweep garbage collector.

Chapter Four elaborate different types of tools that is used in this dissertation work for design and implementation.

Chapter Five named as Data Collection and Analysis. Since it is clear from the name that different primary data which are generated by executing garbage collector in JVM will be analysed in this chapter. Different tests are carried out to find percentage of CPU and memory used by three type's garbage collector.

Chapter Six is a conclusion section. The result obtained by comparing performance and analyzing the three different collectors will be illustrate in this section.

Finally, the references used for this dissertation work are shown at the last of the dissertation.

CHAPTER – TWO

RESEARCH METHODOLOGY AND LITERATURE REVIEW

2.1 Literature Review:

A number of different techniques for incremental garbage collection have been proposed in the past. Generational techniques attempt to visit newly–allocated objects more often than longer-lived ones, in the hope that the former are more likely to become garbage quickly. When this assumption holds, it is possible to collect most garbage objects by just looking at a small area of the heap (the young generation) where objects are allocated, and hence minimize the garbage collection pause time. Generational garbage collection was first proposed by Lieberman and Hewitt [3], but Ungar reported the first implementation [23].

One of the most famous incremental collectors is Baker's algorithm [23]. It is a two-space algorithm that works by copying live objects either eagerly (when they are accessed) or lazily (by a background process) from the from-space to the to-space, so that they are always manipulated in the later. When all live objects have been copied, the two spaces are flipped. An implementation of this algorithm exists for the research VM. Baker also proposed a variation on this algorithm, called the Treadmill, which removes the usually expensive two space requirement.

There have been several other variations of concurrent mark-sweep collection. The original mostly-parallel algorithm was invented by Boehm et al. [16] as a way to introduce incrementality in the Boehm-Demers-Weiser conservative garbage collector for C and C++ [5]. Doligez and Gonthier [24], present different aspects of an innovative concurrent mark-sweep collector. This collector scan mutator thread individually, never requiring all mutator threads to be stopped simultaneously. An algorithm that has been gaining popularity lately is the mature object space algorithm, usually called the train algorithm, originally proposed by Hudson and Moss [2] and first implemented and analysed by Seligmann and Grarup [4]. In this algorithm, heap is split into small regions (train), each of which can be collected independently.

Many researchers have explored how to avoid the work imbalance problem in early parallel copying collectors. It is impractical to avoid work imbalance: the collector does not know ahead

of time which roots leads to large data structures and which to small. The main technique therefore is to dynamically re-balance work from busy thread to idle threads.

Imai and tick [25] developed the first copying GC algorithm with dynamic work balancing. They divide to-space into blocks with each active GC thread having a "scan" block (of objects that it is tracing from) and a "copy" block (into which it copies objects it finds in from –space). If a thread fills its copy block then it adds to a shared work pool, allocates a fresh copy block, and continues scanning. If a thread finishes its scan block then it fetches a fresh block from the work pool. The size of the block provides a tradeoff between the time spent synchronizing on the work pool and the work imbalance.

Clement R. Attanasio and David F. Bacon [9] observed that when resources are sufficient, all the collectors behave in similar manner. But when memory is limited, the hybrid collector (using mark-sweep for the mature space and semi-space copying for the nursery) can delivers at least 50% better application throughput. Therefore parallel collector seems best for online transaction processing applications.

Katherine Barabash and Yoav Ossia [12] presented a modification of the concurrent collector, by improving the throughput of the application, stack, and the behavior of the cache of the collector without foiling the other good qualities (such as short pauses and scalability). They implemented their solution on the IBM production JVM and obtained a performance improvement of up to 26.7%, a reduction in the heap consumption by up to 13.4%, and no substantial change in the pause times. The proposed algorithm was incorporated into the IBM production of JVM.

Stephen M Blackburn and Perry Cheng analyzed that the overall performance of generational collectors as a function of heap size for each benchmark is mainly dictated by collector time. Mark sweep does better in small heaps and Semi Space is the best in large heaps. But the results are not satisfactory in small memory. Garbage collection algorithms still trade for space and time which needs to be better balanced for achieving the high performance computing. Stephen M Blackburn and Perry Cheng [11], experimental design shows key algorithmic features and how they match program characteristic to explain the direct and indirect costs of garbage collection as a function of heap size on the JVM benchmarks. They find that the contiguous allocation of copying collectors attains significant locality benefits over free list allocators.

Jurgen Heymann [18] presented an analytical model that compares all known garbage collection algorithms. The overhead functions are easy to measure and tune parameters and account for all relevant sources of time and space overhead of the different algorithms.

A recent literature on oldest –first techniques has pointed out that while the youngest-first technique of generational collection is effective at removing short-lived objects, there is little evidence that the same models apply to longer-lived objects. In particular, it is often a more effective heuristic to concentrate collection activity on the oldest of the longer lived object [6, 15]. The paper presented the implementation techniques that compared two older first collectors to traditional younger-first generational collectors. One of the older- first collectors performed well and were effective at reducing the first-order cost of collection relative to younger-first collectors. Older-first collectors performed the collection when all the objects have random lifetimes [15]. Mature Object Space Collector uses the idea of dividing collection of a large heap into incremental collection of smaller portions of the heap [22].

Incremental incrementally compacting garbage collection [14] algorithms picks, at each marking cycle, a section of the heap to be evacuated to another section kept free for that purpose; this is much like Garbage –First evacuation pause. However, the marking/compacting phase is neither concurrent nor parallel, and only one region is compacted per global marking. An algorithm for parallel incremental compaction [7] describes a similar system to that of [14]. They augment the collector described in [14] with the ability to choose a sub-region of the heap for compaction, constructing its remembered set during marking, then evacuating it in parallel in a stop world phase. This collector clearly has features similar to Garbage-First, there are also important differences. The region to be evacuated is chosen at the start of the collection cycle, and must be evacuated in a single atomic step. In contrast, Garbage –First allows compaction to be performed in a number of smaller steps.

Mark-copy garbage collector [18], a somewhat similar scheme to Garbage-First, in which a combination of marking and copying is used to collect and compact the old generation. The main algorithm described in this collector is neither parallel nor concurrent: its purpose is to decrease the space overhead of copying collection. Remembered sets are constructed during marking; like the Train algorithm, these are unidirectional, and thus require a fixed collection order. They sketch techniques that would make marking and remembered set maintenance more concurrent

(and thus more like Garbage-First), but they have implemented only the concurrent remembered set maintenance. The performance of Mark-copy garbage collector is compared with a non-generational mark-sweep garbage collector and a hybrid copying/mark-sweep generational garbage collector. It was found that the Mark-Copy collector runs in heap comparable in size to the minimum heap space required by Mark-sweep. It was also found that mark-copy collector is significantly faster than mark-sweep in small and moderate size heaps. When it is compared with hybrid collector, it was found that the mark –copy collector improves total execution time by about 5% for some benchmarks, partly by increasing the speed of execution of the application code [21].

2.2. Research Methodology

Research is a scientific and systematic search for pertinent information on a specific topic. In fact research is an art of scientific investigation. In a scientific method research at first problem is formulated then according to collected input data, output information is analyzed and finally the information is generalized. The main aim of research is to find out the truth which is hidden and which has not been discovered as yet.

This dissertation is based on quantitative research design. The main focus of this dissertation work is to experimentally evaluate the different garbage collection algorithms available in JVM and identify suitable garbage collector according to workloads. All data collected are primary data, which are traces generated by VisualVM application. VisualVM is used to monitor CPU and memory usage while executing java programs. Hence this dissertation work is based on trace driven simulation. Output information gathered is analyzed in a quantitative approach. Finally conclusion is drawn with the help of analyzed data. This work is only specialized for garbage collectors provided in JVM.

CHAPTER–THREE

GARBAGE COLLECTION ALGORITHMS

The java platform is used for a wide array of applications ranging from small applets to web services on large servers. As part of its memory management, java provides many garbage collectors; some of the garbage collectors that will be studied in this dissertation are described below.

3.1. Serial Garbage Collector

In Serial garbage collector both young and old generation are collected in a stop the world fashion. It is usually adequate for small applications (requiring heap up to 100 mb) [26]. In this collector application execution is halted while collection is taking place. Normally, a serial collector is a default copying collector which uses only one GC thread for the GC operation. In serial GC only one things happens at a time. Even when multiple CPU's are available only one is utilized to perform the collection.

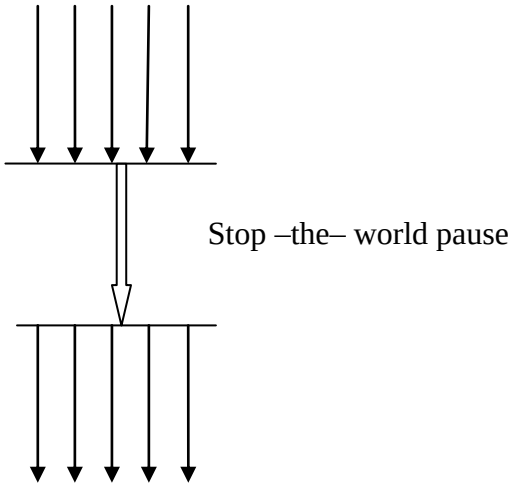


Figure 3.1.: Serial garbage collector

Figure 3.1 shows the serial garbage collector. The serial collector is choice of those applications which run on client –style machines and that do not have a requirement for low pause times

because throughput of serial collector is low. In the java 5 release, the serial collector was automatically chosen as the default garbage collector. The serial collector can be explicitly selected by using the XX: +UseSerialGC command line option. A garbage collection safe point is a point or range in a thread execution stack where the collector can identify all the references in that thread's execution stacks.

Figure 1.3 illustrates the operation of a young generation collection using the Serial collector. The live objects in Eden space are copied to the initially empty survivor space, labeled To, which is shown in figure below. The live objects in the occupied survivor space (labeled From) that are still relatively young are also copied to the other survivor space, while objects that are relatively old are copied to the old generation. Any objects remaining in Eden or the From space after live objects have been copied are not live, and they do not need to be examined. These garbage objects are marked with an X in the following figure, though in fact the collector does not examine or mark these objects [16].

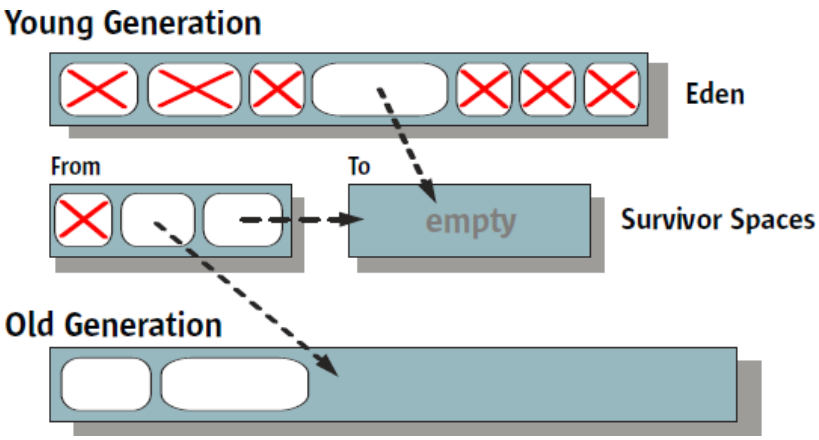


Figure 3.2: Serial young generation collection

3.2. Parallel Garbage Collector

In Parallel garbage collector minor collections are performed simultaneously while major collections are performed serially. It utilizes multiple threads simultaneously to do the GC work. Therefore parallel collector takes advantage of multiple CPU's with parallel execution of garbage collection. It is still a stop-the –world and copying collector, but performing the young generation collection in parallel, using many CPU's, decreases the garbage collection overhead and hence increases application throughput[16].

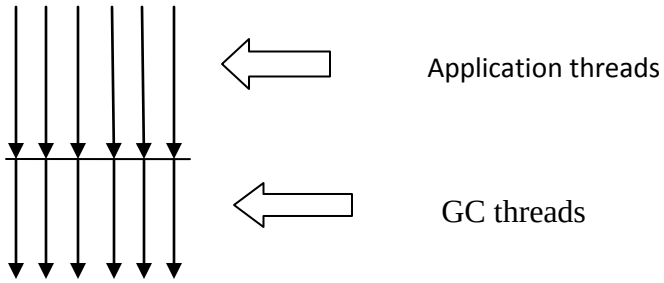


Figure 3.3: Parallel Garbage Collector

It is suitable for those applications that have large data sets. The parallel collector is appropriate on multiprocessor Systems. Those applications running on machines with more than one CPU's and do not have pause-time constraints can use parallel collector for garbage collection. Examples of applications for which the parallel collector is often appropriate, those that do batch processing, billing, payroll, scientific computing and so on. It can be enabled explicitly with –XX: +UseParallelGC command line option.

In parallel compacting collector, the old and permanent generations are collected in a stop the world, with mostly parallel fashion. The collector utilizes three phases. First, each generation is logically divided into fixed –sized regions. In the marking phase, the initial set of live objects which is directly reachable from application code is divided among garbage collection threads, and then all live objects are marked in parallel. As an object is identified as live, the data for the region is updated with information about the size and location of the object [6].

The summary phase operates on regions, but not on the objects. It is typical that some portion of the left side of each generation is dense, containing mostly live objects. The amount of space that could be recovered from such dense regions is not worth the cost of compacting them. So the first thing the summary does is to examine the density of the regions. The summary phase calculates and stores the new location of the first byte of live data for each compacted region. The summary phase is currently implemented as a serial phase; parallelization is possible but not as important to performance as parallelization of the marking and compaction phases.

In the compaction phase, the garbage collection threads use the summary data to identify regions that need to be filled, and the threads can independently copy data into the regions. This produces a heap that is densely packed on one end, with a single large empty block at the other end.

3.3. Concurrent Mark Sweep Garbage Collector

Concurrent Mark Sweep Collector is a generational stop the world collector which is based on the Mark and Sweep algorithms. It is used when applications demand quick response times. In CMS collector minor collections are performed in the same way as performed by parallel collector. While major collections are performed concurrently with the execution of application [15].

This collector consists of following four steps: Initial mark step, concurrent mark, remark and the concurrent sweep step. Firstly the initial mark step stops the application for a very short period of time while a single collection thread marks the first level of objects directly connected the root objects. After that, the application continues to work normally while the collector performs a concurrent mark, marking the rest of the objects while the application keeps running. Because the application might change references to the object graph by the time the concurrent marking is finished, the collector performs an additional step: the remarking, which stops the application for another brief period. When the application changes an object it is marked as changed. During the remarking step the application checks only those changed objects, and to make things faster it distributes the load between multiple threads. Finally the collector performs a concurrent sweep which does not compact the memory area in order to save time and operates concurrently with the application threads [8].

Figure 3.2 shows that concurrent Mark Sweep Algorithm performs garbage collection in four steps. In the initial mark, collector marks the first level of objects stopping the application for a short period of time. This is done in a stop the world pause manner where the entire program threads are halted for a short duration. Then the concurrent mark phase marks the rest level of objects and application keeps running. After concurrent marking is completed, application changes the references to the object and during remark phase, the changed objects are marked stopping the application for another period of time.

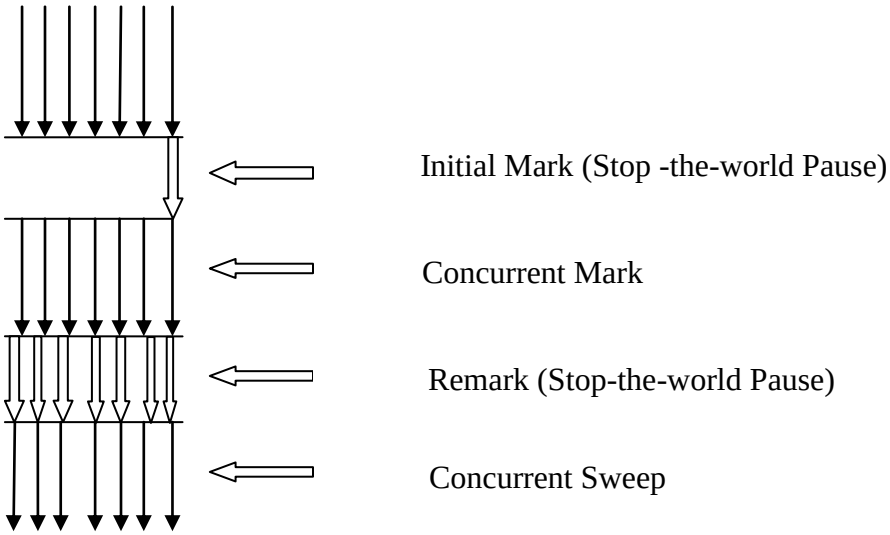


Figure 3.4: Concurrent Mark Sweep Algorithm

Finally, the collector performs a concurrent sweep which does not compact the memory area in order to save time and operates concurrently with the application threads.

If an application needs shorter garbage collection pauses and can afford to share processor resources with the garbage collector when the application is running we can use concurrent mark sweep garbage collector algorithm. An example would be web servers. If we want the CMS collector to be used, we must explicitly select it by specifying the command line option –XX: +UseConcMarkSweepGC [23].

CHAPTER–FOUR

DESIGN AND IMPLEMENTATION

4.1. Programming Language

Java is a computing programming language evolved from a language named Oak. Oak was developed in the early nineties at Sun Microsystems as a platform-independent language. It is intended to let application developers "write Once, run anywhere" (WORA) meaning that code that runs on one platform does not need to be recompiled to run on another. Java application is typically compiled to bytecode (class file) that can run on any Java virtual machine regardless of computer architecture. There are lots of applications and websites that will not work if java has not been installed, and more are created every day. Java is fast and secure. From laptops to datacenters, game consoles to scientific computers, cell phones to the internet, java is used everywhere. Since the main theme of this dissertation work is to evaluate the performance of garbage collector algorithm available in JVM, different application program that are used as test cases are written in java programming language [10].

4.2. Tools Used

Different tools are used to evaluate the performance of garbage collection. Some of the most commonly used tools are described below.

4.2.1. VisualVM

Java VisualVM is a tool that provides a visual interface for viewing detailed information about java applications while they are running on a java Virtual Machine (JVM), and for troubleshooting and profiling these applications. Various optional tools, including Java VisualVM, are provided with Sun's distribution of the Java Development Kit (JDK) for retrieving different types of data about running JVM software instances. For example, most of the previously standalone tools JConsole, jstat, jinfo, jstack, and jmap are part of Java VisualVM. Java VisualVM federates these tools to obtain data from the JVM software, then re-organizes and presents the information graphically [27].

4.2.2. Javac:- The java programming language compiler, javac, reads source files written in the java programming language, and compiles them into bytecode class files. Java source code must be contained in files whose filenames end with the .java extension. For every class defined in each source file compiled by javac, the compiler stores the resulting bytecodes in a class file with a name of the form classname.class. The javac tool reads class and interface definitions, written in the java programming language, and compiles them into bytecode class files. It can also process annotations in java source files and classes [9].

There are two ways to pass source code file names to javac:

- For a small number of source files, simply list the file names on the command line.
- For a large number of source files, list the file names in a file, separated by blanks or line breaks. Then use the list file name on the javac command line, preceded by an @ character.

4.2.3. Java:- The *java* command executes Java class files created by a Java compiler. *java,* the Java interpreter, is used to run Java applications from the command line. It takes as an argument the name of a class file to run, as in the following example:

Java GCTest

The class loaded by the Java interpreter must contain a class method called main () that takes the following form.

```
Public static void main (String [] arguments)
  {
     // Method here
  }
```

The Java interpreter runs bytecode – the compiled instructions that are executed by a Java virtual machine. After a Java program is saved in bytecode as a .class file, it can be run by different interpreters without modification [9]. If we have compiled a Java 2 program, it will be compatible with any interpreter that fully supports Java 2.

4.3. Data Structures

4.3.1. Integer object: - Each of java's primitive data types has a class dedicated to it. These are known as wrapper classes, because they 'wrap' the primitive data type into an object of that class. The wrapper classes are part of java.lang package, which is imported by default into all java programs. Integer is a wrapper class while int is a primitive data type. Vectors cannot handle primitive data types like int, float, long, char, and double. Primitive data types may be converted into object types by using the wrapper classes that are contained in the java .lang package. If int is data type then its corresponding wrapper class is Integer [9].

4.3.2. Vector: - A vector is an ordered set of elements in which each element is associated with, and is accessible by, a non-negative integer called its index. Vectors are commonly used instead of arrays, because they expand automatically when new data items are added to them. It implements a dynamic array. It can hold only objects and not primitive types (e.g. int) vector contain many useful methods. To add elements in vector add ( ) method can be used. Similarly to add element at fix position we have to use add (index, object) method. [9].

4.3.3. Strings: - String manipulation is the most common part of many java programs. In java Strings are object of class and implemented using two classes, namely, *String* and *StringBuffer*. A java string is an instantiated object of the string class. Java strings, as compared to C strings, are more reliable and predictable. This is basically due to C's lack of bound checking. A java string is not a character array and is not NULL terminated [9].

4.4. Programming Language Features

4.4.1. Multithreading: - Multithreading is one of the core features supported by Java. It allows creation of multiple objects that can simultaneously execute different operations. It is often used in writing programs that perform different tasks asynchronously such as printing and editing. Network applications that are responsible for serving remote clients requests have been implemented as multithreaded applications.

The main purpose of multithreading is to provide simultaneous execution of two or more parts of a program to maximum utilize the CPU time. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program called a thread. Each thread has a separate path of its execution. So this way a single program can perform two or more tasks simultaneously. Threads are lightweight processes; they share the same address space. In Multithreaded environment, programs make maximum use of CPU so that the idle time can be kept to minimum [9].

4.4.2. Generics: - Generics in java is one of the important feature added in java 5 along with Enum, autoboxing and varargs, to provide compile time type-safety. Using java generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with integer arrays, Double arrays, String arrays and so on, to sort the array elements.

We can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods [9]:

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).

- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.

- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments

- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

4.5. Experimental Setup

All the test cases are executed in hardware platform with Intel core i5 processor with a 2.30GHz clock, DDR3 memory of size 4GB, Level1 cache of 128KB, and a Level2 cache of 512KB. The processor in 32-bit mode and the system runs under windows 8.

The JVM command line options are specified in each case, specifying the garbage collector to be used. While the test was running, CPU and memory Usage of the running application is recorded by using the VisualVM.

4.6 Test Case Design

In this dissertation four test cases are used. For each test case CPU and Memory Usage of running application is recorded by VisualVM.

In test case1 arrays of integer objects are added to the vector which is immediately eligible for garbage collection. Code creates and adds 1024 integer array into vector. Arrays are removed during iterations. At every 10^{th} iterations, System.gc() is called, suggesting the java virtual machine to start garbage collection.

In test case2 string is concatenated with vector resulting into string and string object are eligible for garbage collection only after they are removed from garbage collection. The code creates and adds 1024 strings into vector and strings are removed during iteration.

The main thread class TestGCThread in test case3 enters a sleep period of variable duration (0-10,000 ms). After each sleep period, it creates a variable number of TestGCObject instances. The allocated object can be immediately garbage collected as they are not being used further.

Here test case 3 and 4 are almost similar but only the difference is case 3 is multithreaded while case 4 is single threaded.

4.6.1. Test Case 1

```
Import java.util.Vector;
class GCTest
{       Static Vector<Integer[]> v=new Vector<Integer[]>(1024);
          Public static void main(String[]args) throws exception
        {         Integer obj[];
                 Int index;
                Thread.sleep(5000);
                 for (int i=0;i< 200; i++)
              {          obj=new Integer[2048*2048];
                                  v.add(obj);
                    Thread.sleep(50);
               }          for (int i=0; i<200;i++)
                   {          index = v.size()-1;
                     v.remove (index);
                     if (i%10==0)
                       {    System.gc();
                          System.out.println("Garbage is Collected");
                       }  Thread.sleep(50);
                  }            System.gc();
                                  Thread.sleep(3000);
      }
 }
```

4.6.2. Test Case2

```
import java.util.*;
public class GCTest2
{       Static vector<String>sv= new Vector<String(1024);
        Public static void main(String[] args) throws Exception{
        String s;
        Int index;
        Thread.sleep(10000);
        for (int i=0; i<1024; i++)
        {       s= new String("Notepad is a basic text-editing program")
                Sv.add(s);
                Thread.sleep(50); 1
        }
        for (int i=0;i<1024; i++)
{
        Index =sv.size() – 1;
        Sv.remove(index) ;
        If(i%10==0)
        {       System.gc();
        }
        Thread.sleep(50);
}       System.gc();
        Thread.sleep(3000);}
}
```

4.6.3. Test Case 3

```
class TestGCThread extends Thread
{
        Public static void main (String[] args) throws Exception
        {       Thread.sleep(3000);
                new TestGCThread().start(); }
        public void run( )
        {       long start = System.currentTimeMillis();
                long then = start;
                while(true)
        {
                //sleep random delay
                Try
                {       int delay = (int) Math.round(100*Math.random( ));
                        Thread.sleep (delay); }
                Catch (Exception e){ }
                / / create random number of objects
                int count = (int) Math.round(Math.random)*1000);
                for (int i=0; i<count; i++)
                {new TestGCobject( ); }
                / / log stats to console
                long now = System.currentTimeMillis();
```

```
If (now- then > 1000)
{
        (System.out.println (TestGCobject.created +  "     objects created\t" +
        TestGCobject.freed +"                objects freed") ;
then =now; }
}
}
}
Class TestGCObject
{       static long created ;
        Static long freed;
        Public TestGCObject()
        {
        created++;
        }
        Public void finalize()
        {
        freed++ ;
        }
}
```

4.6.4. Test Case 4

```
Class TestThread1
{
        public static void main (String[] args) throws Exception
        Thread.sleep(3000);
        long start =  System.currentTimeMillis();
        long then = start;
        while (true)
{
        / / sleep random delay
        Try
        {
        Int delay = (int) Math.round(100*Math.random() ) ;
        Thread.sleep(delay) ;
        }
        catch(Exception e){ }
        / / create random number of objects
        Int count = (int) Math.round(Math.random()*1000);
        for  (int i=0; I < count;  i++)
        {
                New TestGCObject1() ;
        }
        / / log stats to console
```

```
        long now = System .currentTimeMillis();
        If (now – then > 1000)
        {
                System .out.println(TestGCObject1.created + "                objects created\t"
                +TestGCObject1.freed+ "      objects freed");
                then = now;
        }
}
}
}
Class TestGCObject1
{
        Static long created;
        Static long freed;
        Public TestGCobject1()
        {
                Created++;
        }
        Public void finalize()
        {
                freed++;
        }
}
```

CHAPTER–FIVE

DATA COLLECTION AND ANALYSIS

5.1. CPU and Memory Usage for Test Case 1

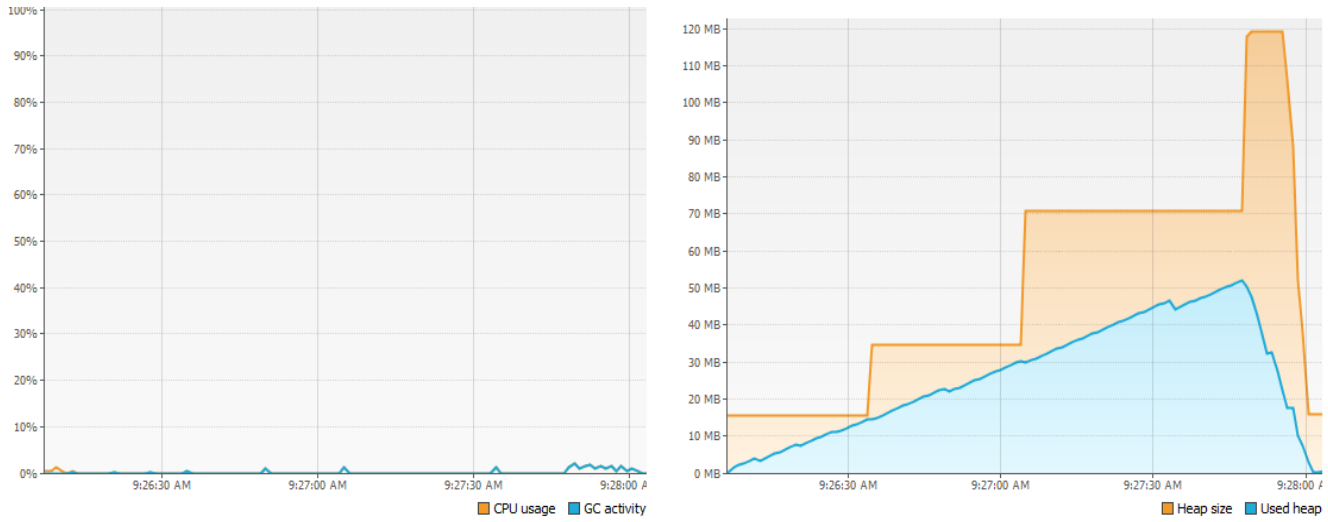


Figure 5.1. CPU and Memory Usage with SerialGC

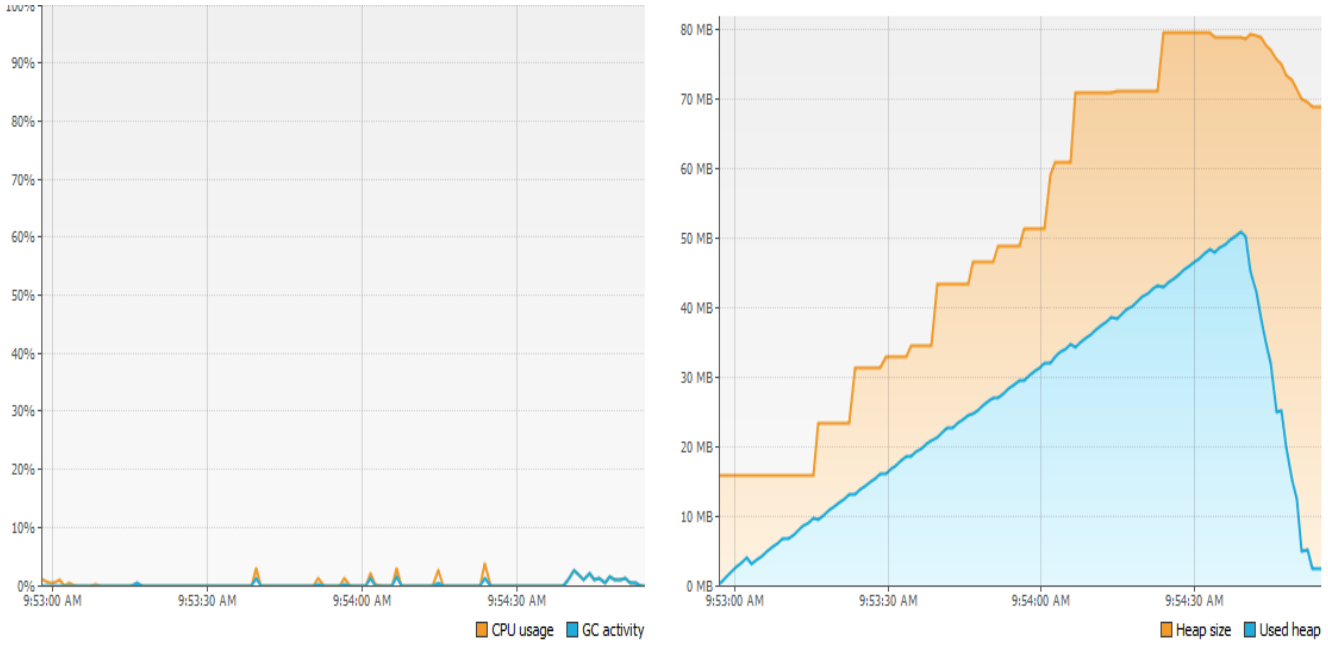


Figure 5.2. CPU and Memory Usage with Parallel GC

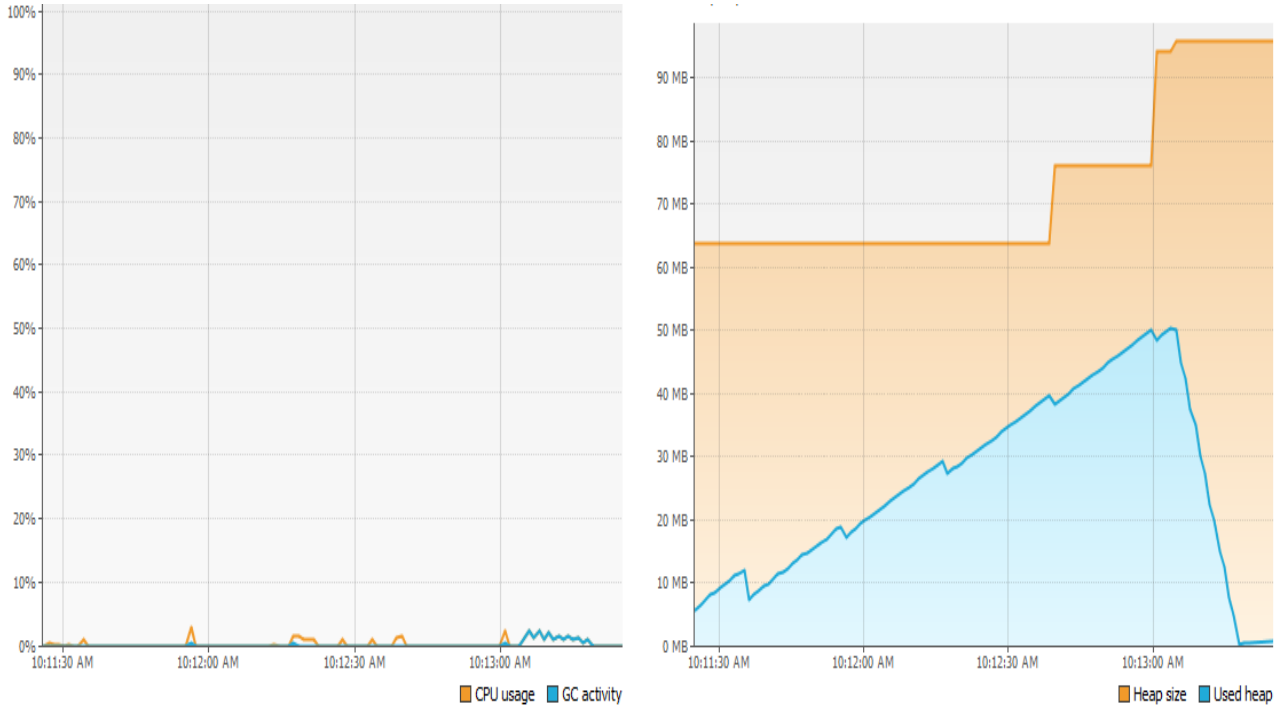


Figure 5.3. CPU and Memory Usage with CMS GC

From above figure it can be seen that for Serial garbage collector maximum CPU usage is 3.2% and maximum GC activity is 3.2%. The average CPU usage is 1.6% and average GC activity is also 1.6%. The Serial Garbage Collector uses Maximum heap of 52.06 MB. Similarly in case of Parallel Garbage Collector maximum CPU usage is 5.1% and maximum GC activity is 4.0%. Average CPU Usage is around 1.9% and average GC activity is 1.7%. The Parallel garbage collector uses maximum heap of 50 MB which is slightly lower than that of the Serial garbage collector.

If we see the graph for CMS collector there is little bit different than that of the serial and parallel GC. The maximum CPU usage for CMS is found to be 4.0% and maximum GC activity is c 2.7%. If we see average CPU usage and average GC activity it is found to be 1.9% and 1.5% respectively. The maximum heap memory used for CMS collector is 51.61MB which is slightly greatly than parallel GC and slightly lower than Serial GC.

5.2. CPU and Memory Usage for Test Case2

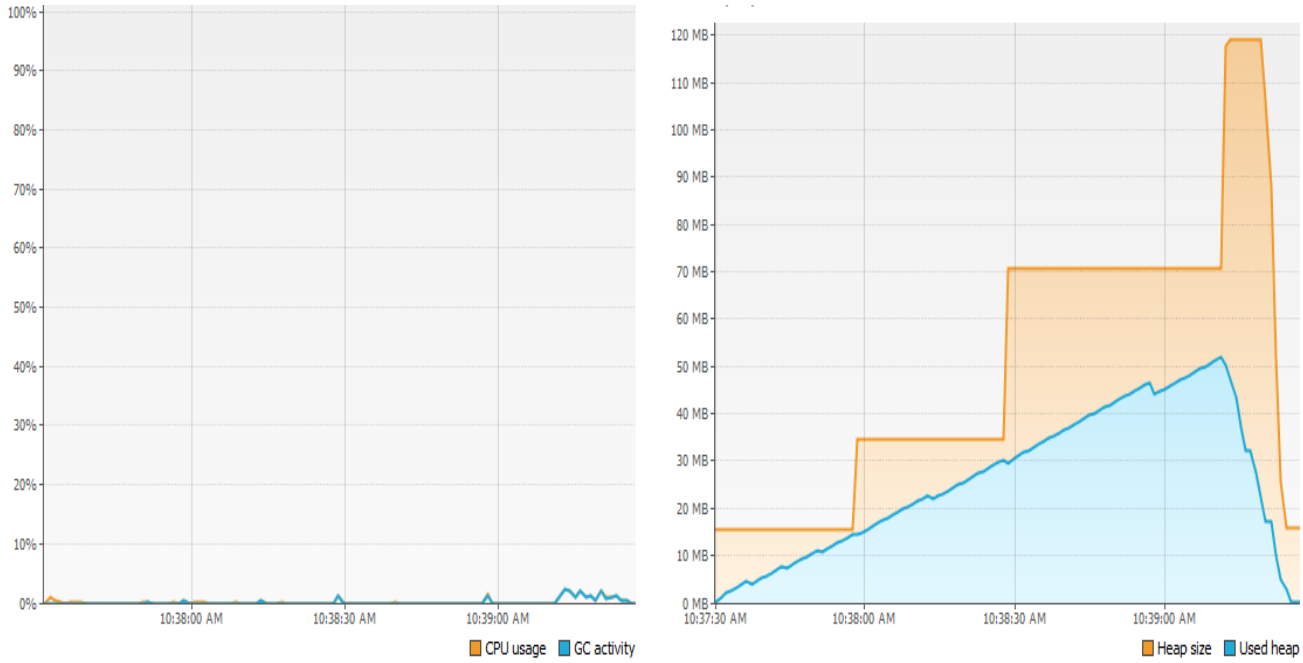


Figure 5.4. CPU and Memory Usage with Serial GC

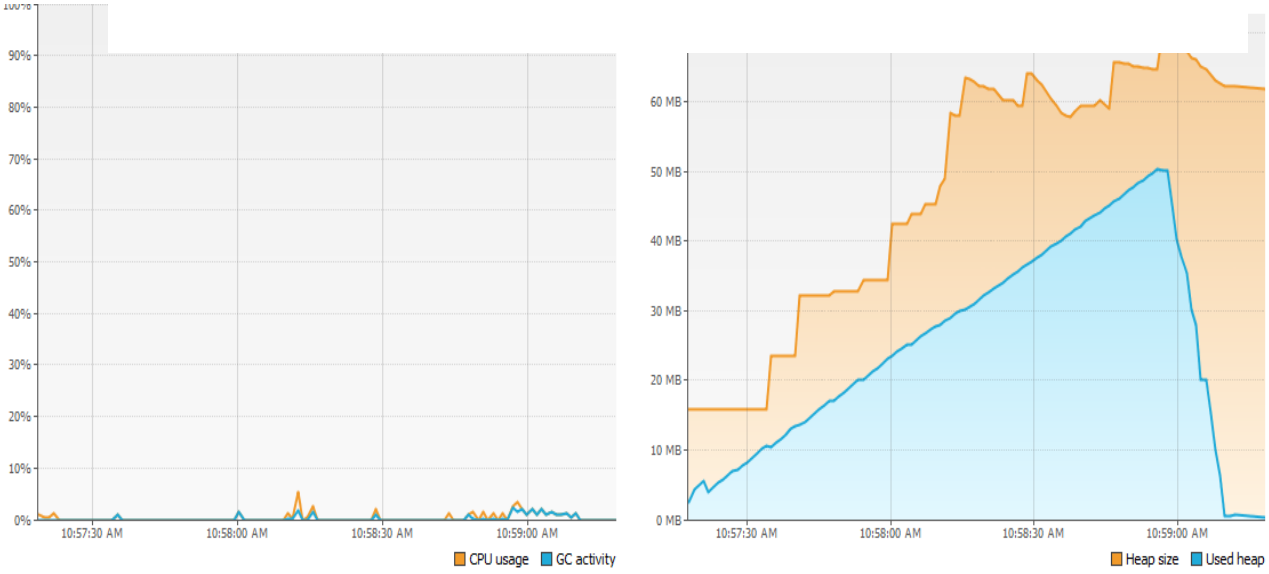


Figure 5.5 CPU and Memory Usage with Parallel GC

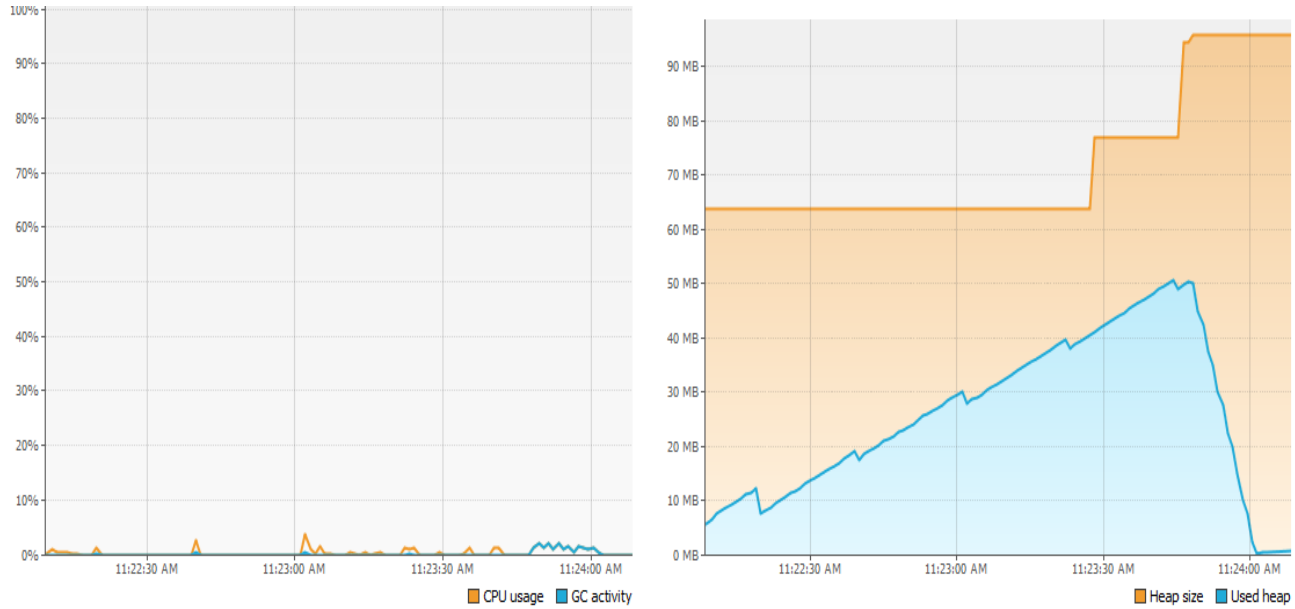


Figure 5.6. CPU and Memory Usage with CMS GC

In this case maximum CPU usage is 2.6% and maximum GC activity is 2.6% for Serial Garbage Collector. Also it has average CPU usage is around 1.38% and average GC activity is below 1.3%. The maximum heap used is about to be 52.13 MB. For parallel garbage collector maximum CPU usage is 5.5% and maximum GC activity is 2.3%. Average CPU usage and average GC activity for parallel collector are 2.04% and 1.26% respectively. Both Serial and Parallel garbage collector uses maximum heap of 52.13 and 50.46MB.

Similarly in case of Concurrent Mark sweep maximum CPU usage is 3.9%. The maximum GC activity used in this case is found to be 2.3%. CMS GC uses average CPU of 1.65% and average GC activity is more than 1.4%. Here, CMS GC uses maximum heap of 50.87 MB which is comparatively larger than parallel garbage collector and slightly smaller than serial garbage collector.

5.3. CPU and Memory Usage for Test Case3

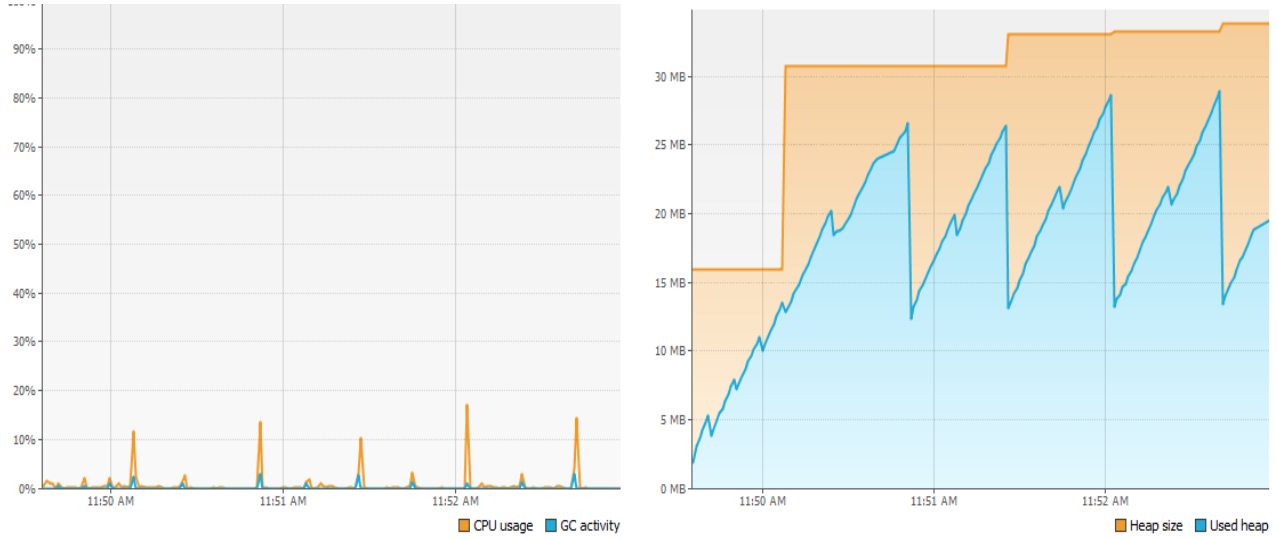


Figure 5.7: CPU and Memory Usage with Serial GC

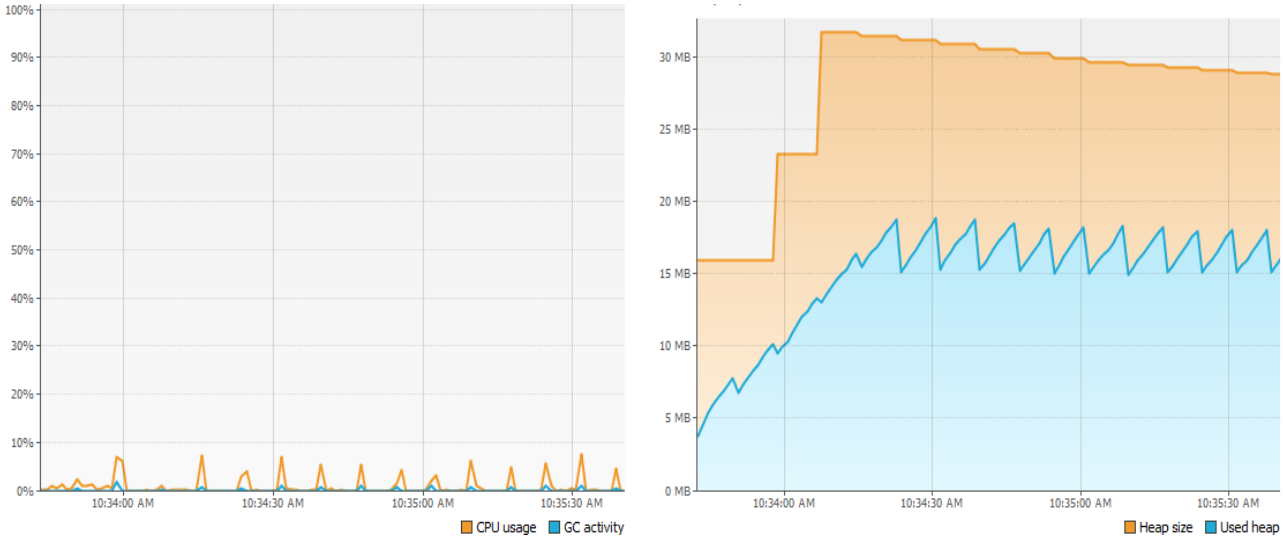


Fig 5.8 CPU and Memory Usage with Parallel GC

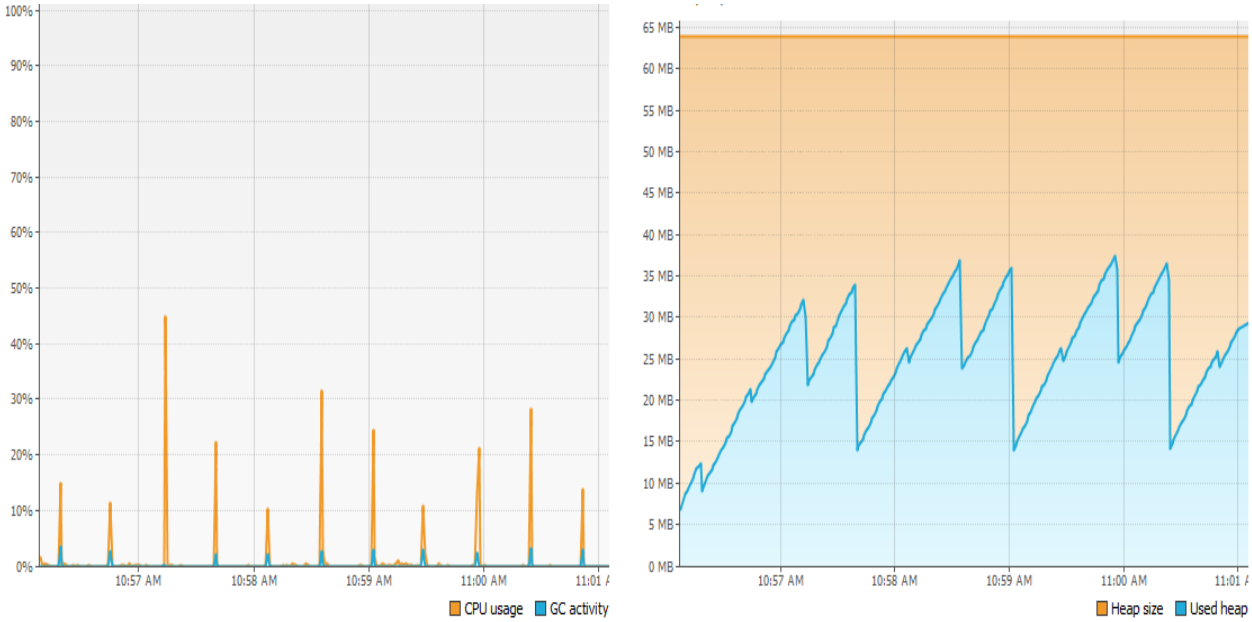


Figure: 5.9. CPU and Memory Usage with CMS GC

For the Test Case3 it is clear from the figure that the graph for each serial parallel and CMS Collector, graph is little more different than that for the graph of Test Case1 and Test Case2. The maximum CPU usage for Serial Collector is 17.4% and maximum GC activity is 3.2%. its average CPU usage is 5.39% and average GC activity is 1.9%. Similarly for Parallel GC the maximum CPU usage is 7.7% and maximum GC activity is 2.0%. The average CPU usage is found to be around 3.9% and average GC activity is above 1%.the heap memory used for Serial GC is found to be 29 MB and for Parallel GC is 18.93 MB.

The case for CMS collector is comparatively different than other collectors. Its maximum CPU Usage is 45.1%, which is highest than other collectors. The maximum GC activity is about 3.6%. Average CPU and GC Activity is found to be 17.07% and 2.65% respectively. The heap memory used for the CMS collector is found to be 37.60 MB.

5.4. CPU and Memory Usage for Test Case4

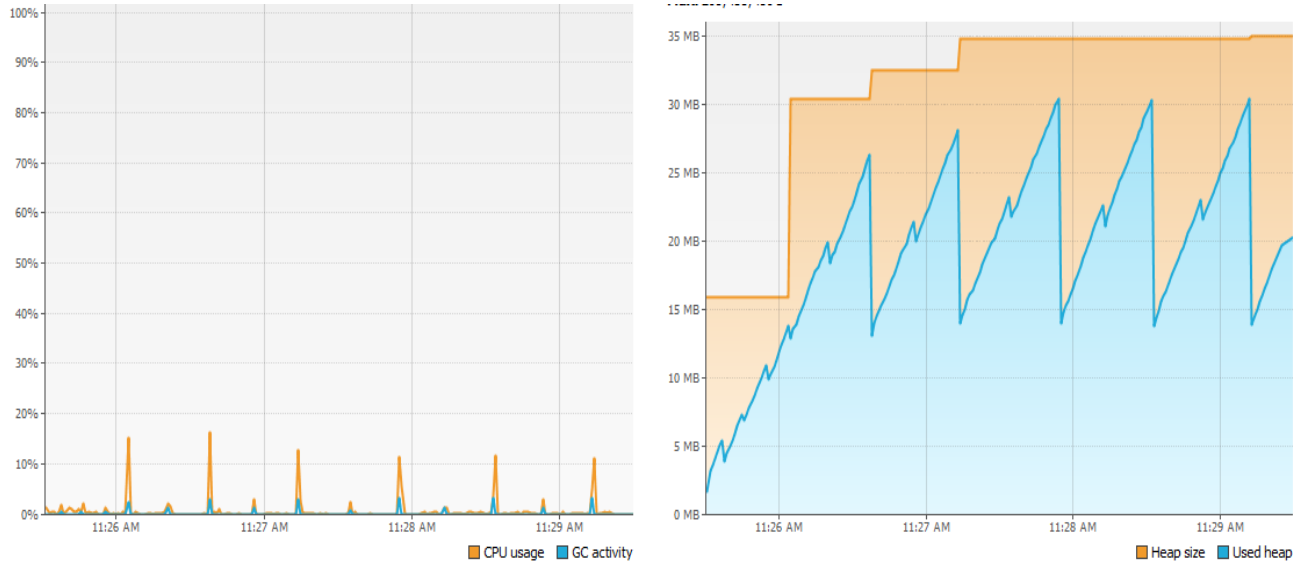


Figure 5.10: CPU and Memory Usage with Serial GC

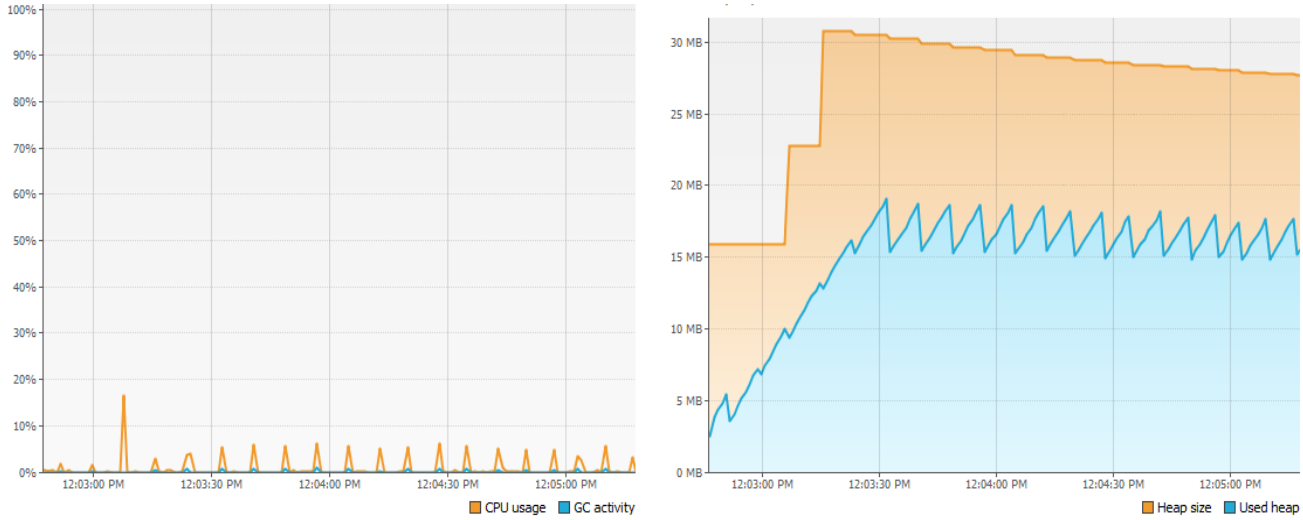


Figure 5.11: CPU and Memory Usage with Parallel GC

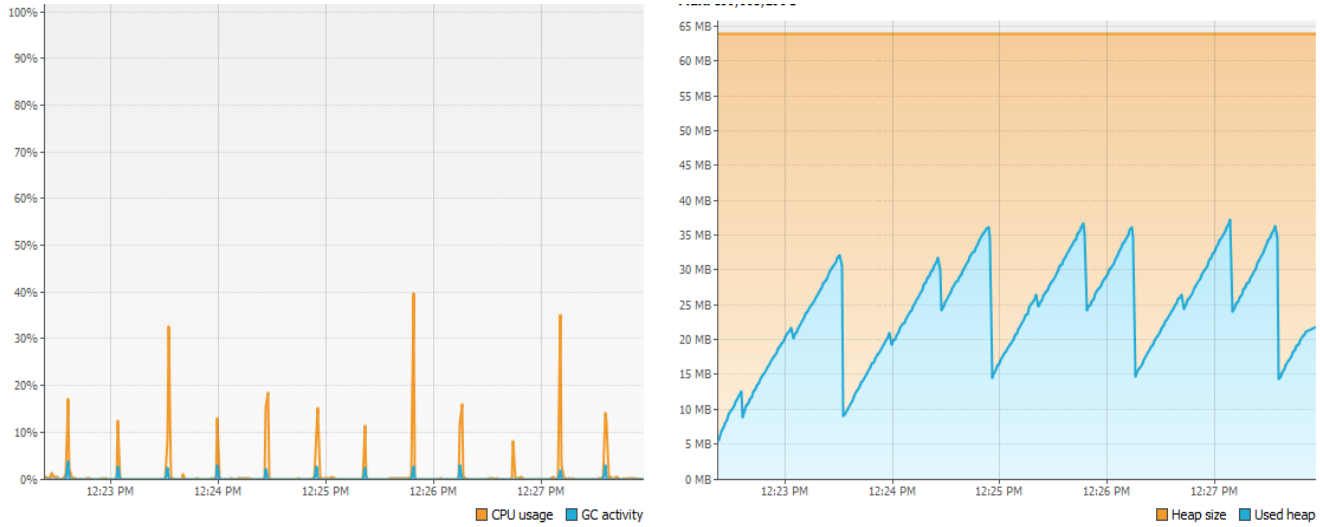


Figure 5.12: CPU and Memory Usage with CMS GC

Finally for the test case 4, it is seen that for serial garbage collector maximum CPU usage is 16.5% and maximum GC activity is 3.4%. Its average CPU is found to be 5.44% and average GC activity around 2.03%. Similarly in case of parallel garbage collector maximum CPU usage is 16.8% and maximum GC activity is 1.2%. Average CPU usage is around 4.95% and average GC activity is below 1%. Here both serial and parallel collector uses maximum 30.44 and 19.11 MB of available heap memory.

From above figure 5.12 it is seen that for CMS collector the maximum CPU usage is 40% and maximum GC activity is 3.9%. Its average CPU is found to be 14.08% and average GC activity is 2.8%. CMS collector uses 36.87 MB of available heap memory.

5.5. Summarizing GC Performance

Performance Metric	For Test Case1			For Test Case2			For Test Case3			For Test Case4		
	SGC	PGC	CMSGC	SGC	PGC	CMSGC	SGC	PGC	CMSGC	SGC	PGC	CMSGC
Max CPU Usage	3.2	5.1	4.0	2.6	5.5	3.9	17.4	7.7	45.1	16.5	16.8	40
Max GC Activity	3.2	4.0	2.7	2.6	2.3	2.3	3.2	2.0	3.6	3.4	1.2	3.9
Max Heap Used	52.06	50	50.67	52.13	50.46	50.87	29	18.93	37.60	30.44	19.11	36.87

Table 5.13: Summary of maximum CPU and Memory Usage

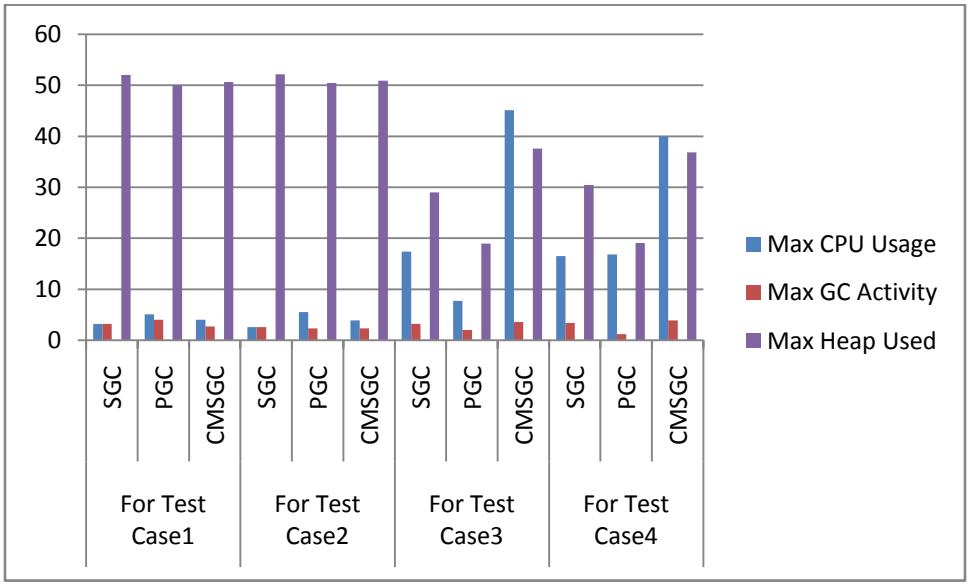


Figure 5.14: graph for Table 5.13

Performance Metric	For Test Case 1			For Test Case 2			For Test Case 3			For Test Case 4		
	SGC	PGC	CMSGC	SGC	PGC	CMSGC	SGC	PGC	CMSGC	SGC	PGC	CMSGC
Avg. CPU Usage	1.6	1.9	1.9	1.38	2.04	1.65	5.39	3.9	17.07	5.44	4.95	14.08
Avg. GC Activity	1.6	1.7	1.5	1.3	1.26	1.4	1.9	1.0	2.65	2.03	1.0	2.8

Table 5.15: Average CPU Usage and Memory Usage

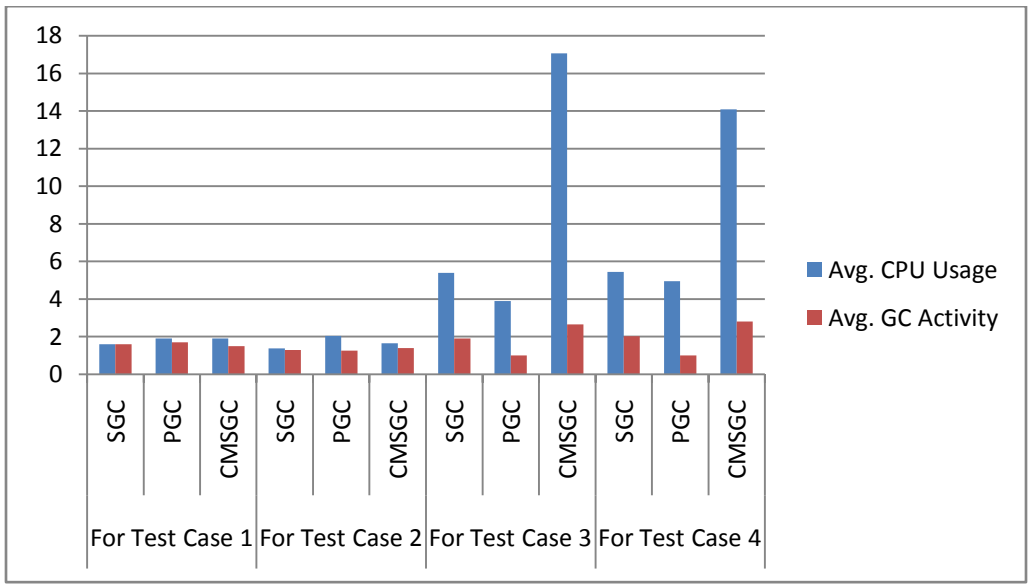


Figure 5.16: graph for Table 5.15

CHAPTER–SIX

CONCLUSION AND RECOMMENDATION

6.1. Conclusion

Garbage collection offers several clear advantages to software engineers over manual memory management. However automatic memory management does not come without cost. It is also of benefit when constructing large, modular systems. In manually memory-managed programs, it is necessary for programmers to determine who has the responsibility for deallocating a piece of memory. This may occur in a different module from that in which memory was allocated. Confusion between programmers can lead to objects being released multiple times, or not at all. The use of garbage collection alleviates this problem; objects are freed according to a global view of the system. As such, no programmer is responsible for deallocation.

From analysis of previous chapter it is shown that CPU usage of Serial GC is least among all other garbage collectors while heap usage of all garbage collectors is comparable (slightly higher for SGC) in case of test case 1 (figure 5.1 and figure 5.2). Similarly in case of test case 2 again CPU Usage for serial GC is least, 2.6% (figure 5.4 and figure 5.5) among others and heap usage is seemed to be almost similar. (Slightly higher for Serial GC)

Again if we look at analysis of test case 3 which is multithreaded program, it can be seen that CPU Usage and heap Usage of parallel GC is least among others (figure 5.7 and figure 5.8). And in case of test case 4, Serial GC wins the race again. It has least CPU Usage among others but in terms of heap usage parallel GC seems to be better (figure 5.10 and figure 5.11).

From this observation, it can be concluded that Serial GC is better choice if we have to use single threaded programs since it has least CPU usage among others and also have comparable heap usage. But parallel GC is better choice in case of multithreaded programs because it has lowest CPU usage and heap usage.

6.2. Further Recommendation

Observing others factors such as pause time, heap usage for permanent generation, throughput etc is the area of future research. There are many ideas for improving Concurrent Mark Sweep Garbage collector in the future. One of the ideas is to modify remembered set representations to increase the efficiency of remembered set processing. This is also one of the interesting future researches.

References

[1] Baker H.G, List Processing in Real –Time on a Serial Computer. 280-294.1990.

[2] Banny Van Houdt, A Mean Field Model for a Class of Garbage Collection Algorithm, ACM, 2013.

[3] Boehm H, Demers A.J. Demers A, J and Shenker S, Mostly Parallel Garbage Collection, June 1991.

[4] C.E. Hewitt and H.Lieberman, Real-Time Garbage Collector Based on the Lifetime of objects. 419-429, 1985.

[5] C.R Attanasio, D.F. Bacon, A.Cocchi, A Comparative Evaluation of Parallel Garbage Collector and Implementations, 177-192, 2003.

[6] Darko Stefanovic, Matthew Hertz, Stephen M. Blackburn, Kathryn S.Mc Kinley, and J.Eloit B.Moss, Older-First garbage Collection in Practice evaluation in a java virtual machine, 2002.

[7] Description of Hotspot GC's: Memory Management in the Java Hotspot Virtual Machine White Paper, 2006.

[8] Doligez Damien and Gonthier Georges, Unobtrusive Garbage Collection for Multiprocessor systems. 1994.

[9] E Balagurusamy, Programming with Java Third Edition, 2008

[10] George Andy, Eeckhout Lieven, Buytaert Dries, Java Performance Evaluation through Rigorous Replay Compilation. 2008.

[11] Grarup S. and Seligmann J, Incremental Mature Garbage Collection using Train Algorithm Aug, 1995.

[12] Hudson R.L. and Moss J.E.B Incremental Garbage Collection of Mature objects. September, 1992.

[13] Imai A. and Tick E, Evaluation of Parallel Copying Garbage Collection on a shared Memory Multiprocessor. 1993.

[14] J. Heymann, A Comparative Analytical Model for Garbage Collection Algorithms, August, 1991.

[15] K. Barabash, Y. Ossia, E.Petrank, Mostly Concurrent Garbage Collection, 255-268,ACM, 2003.

[16] Kotwal Nitin S, Jamwal Shubhnandan S, Empirical Analysis of Serial and Parallel Garbage Collectors, June 2013.

[17] Kumari, Ambika, Quantitative Evaluation of Garbage Collector Algorithm in JVM, December, 2012.

[18] Lars T.Hansen and William D. Clinger, An experimental Study of renewal- older-First Garbage Collection, 2002.

[19] Lindholm, T, Yellin F, The Java Virtual Machine Specification. April 2001.

[20] P.R Wilson, Uniprocessor Garbage Collection techniques, Y.Bekkers and J.Cohen, Editors International Workshop on Memory Management. September, 2000.

[21] Richard L. Hudson and J. Eliot B.Moss, Incremental Collection of mature objects, International Workshop on Memory Management, 1992.

[22] S.M. Blackburn, P.Cheng, K.S. McKinley, The Performance Impact of Garbage Collection, June 12-16, ACM Press, New York, 2004.

[23] Tauro Clarance J M, Prabhu Manjunath V, Saldanha Vernon J, CMS and G1 Collector in java 7 Hotspot: Overview, Comparison and Performance Metrics, April 2012.

[24] Ungar D.M, A non-disruptive high Performance Storage reclamation algorithm, 157-167, April 1985.

[25] Weiser M. and Boehm H, Garbage Collection in Uncooperative Environment. September, 1988.

[26] Witawas Srisa-an, Chia-Tien Dan Lo and J. Moms Chang, Scalable Hardware algorithm for Mark-Sweep Garbage Collection, 2002.

[27] http://docs.oracle.com/javase/7/docs/technotes/guides/visualvm/