CHAPTER – ONE

BACKGROUND AND INTRODUCTION

1.1. Background

1.1.1. Java Virtual Machine JVM

Java Virtual Machine is the software module that executes the java bytecode and translates the bytecode into hardware and operating system-specific instructions. Java programming language utilizes a managed runtime (java virtual machine or JVM) to improve developer's productivity and provide cross platforms portability. 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 [1].

At the heart of Java technology lies the Java virtual machine which is the abstract computer on which all Java programs run. Although the name "Java" is generally used to refer to the Java programming language, there is more to Java than the language. The Java virtual machine, Java API, and Java class file work together with the language to make Java programs run. The Java virtual machine is an abstract computer. Its specification defines certain features every Java virtual machine must have, but leaves many choices to the designers of each implementation. For example, although all Java virtual machines must be able to execute Java bytecode, they may use any technique to execute them.

The execution engine is one part of the virtual machine that can vary in different implementations. On a Java virtual machine implemented in software, the simplest kind of execution engine just interprets the bytecode one line at a time. Another kind of execution engine is one that is faster but requires more memory, is a just-in-time compiler. In this scheme, the bytecode of a method are compiled to native machine code the first time the method is invoked. The native machine code for the method is then cached, so it can be re-used the next time that same method is invoked. A third type of execution engine is an adaptive optimizer. In this

approach, the virtual machine starts by interpreting bytecode, but monitors the activity of the running program and identifies the most heavily used areas of code. As the program runs, the virtual machine compiles to native and optimizes just these heavily used areas. The rest of the code, which is not heavily used, remains as bytecode which the virtual machine continues to interpret. This adaptive optimization approach enables a Java virtual machine to spend typically 80 to 90% of its time executing highly optimized native code, while requiring it to compile and optimize only the 10 to 20% of the code that really matters to performance [2].

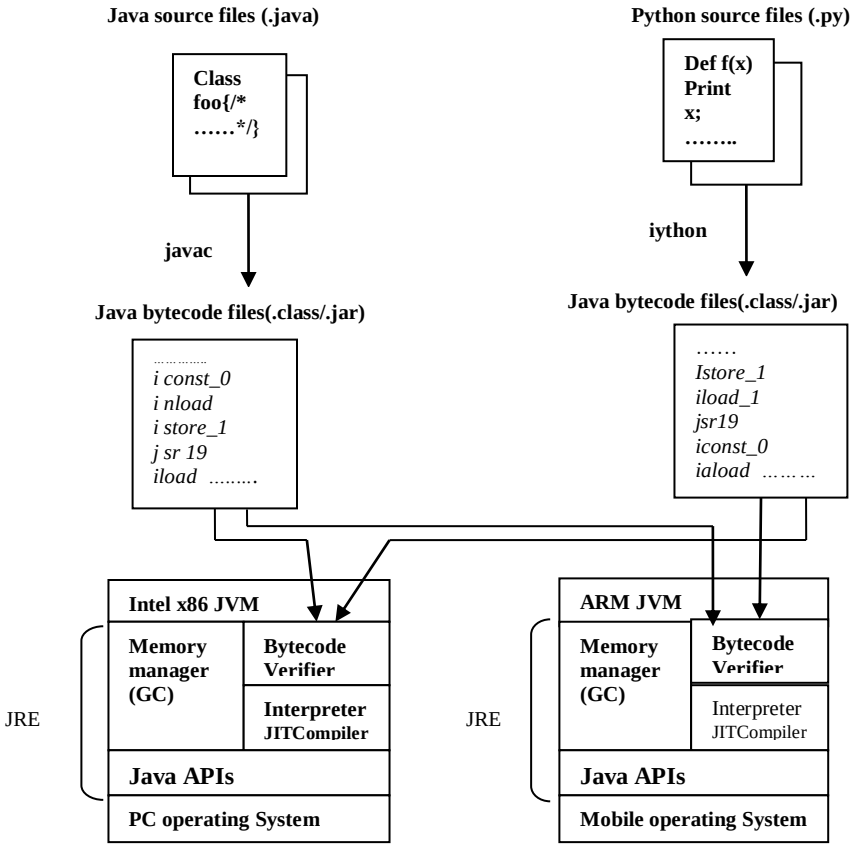

Figure 1.1: JVM

When running on a Java virtual machine that is implemented in software on top of a host operating system, a Java program interacts with the host by invoking native methods. In Java, there are two kinds of methods: Java and native. A Java method is written in the Java language, compiled to bytecode, and stored in class files. A native method is written in some other language, such as C, C++, or assembly, and compiled to the native machine code of a particular

processor. Native methods are stored in a dynamically linked library whose exact form is platform specific. While Java methods are platform independent, native methods are not. When a running Java program calls a native method, the virtual machine loads the dynamic library that contains the native method and invokes it [2].

1.1.2. Memory Management

Memory Management is the act of recognizing the objects are no longer needed, freeing the memory used by such objects, and making it available for subsequent allocations [3]. Memories are allocated either at compile time or run time. The memory allocated at compile time is static and the memory allocated at run-time is dynamic. Memory allocations are of three types; static allocation, automatic allocation and dynamic allocation. Static allocation is one of the simplest forms of allocation carried out during compile-time which remains fixed throughout the execution. Automatic allocation is carried out only when the function is being called. It is of fixed size and available during the execution of function call. Dynamic allocation is also known as heap allocated memory which is dynamically allocated at runtime. Heap is a block of memory within which pieces are allocated and stored in some relatively unstructured manner. It is of variable in size and is available until it is explicitly deallocated.

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 portion of debugger time is often spent in debugging and trying to correct such errors [4]. Two problems are associated with explicit memory management; dangling references and memory leaks. 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 is reallocated to a new object, the result is unpredictable and the system may crash. 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 problems of explicit memory management, an alternative approach, automatic memory management called garbage collector is applied by many modern object-programming languages [3].

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 objects [5]. If subjected to Mark and Sweep repeatedly, overtime the heap gets fragmented. For Example as shown in Figure 1.2, 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.

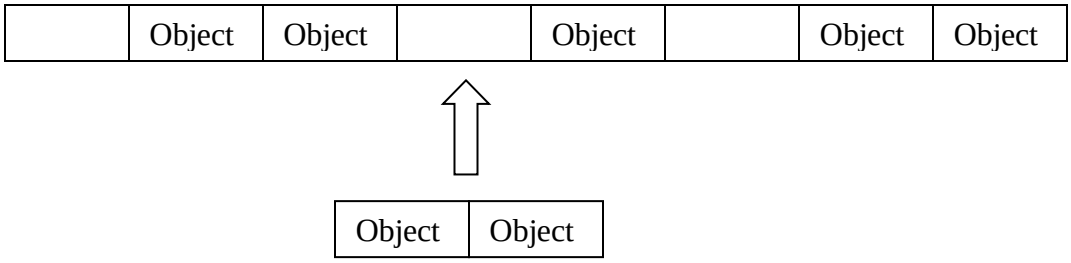

Figure 1.2: Memory Fragmentation

Fragmentation is taken care by another phase 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 1.3.

Initial State

Object	Object	Object	Object	Object	Object	Object	Object

After Mark Sweep

Object	Object		Object		Object	Object	

After Compaction

Object	Object	Object	Object	Object			

Figure 1.3: Memory Compaction

1.1.4. Garbage Collection concept

The name “garbage collection” implies that objects no longer needed by the program are “garbage” and can be thrown away. Java puts all the newly created objects in a “heap”. An object that is being used or is going to be used by the application is called a “Live Object” [6]. Opposite of a live object is garbage as in; the application cannot reference and cannot reuse 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 Objects [7]. Root object is an object which can be directly accessed that is, without going through other references. Root objects are Local variables in the stack or class variables that is, static objects. An object is considered live if it is referenced by a root object or other live object. A Depth First Search (DFS) algorithm is used from the root object and each object visited is tagged. This is not visible to main applications that are running on the JVM. In the process of freeing unreferenced objects, the garbage collector must run any finalizers of objects being freed. Methods for garbage collection comprise two separate phases; identifying storage that may be reclaimed and incorporating reclaimable storage into the memory area that can be made available to the user [8].

Garbage collection provides many software engineering benefits, most notably by eliminating a large class of insidious programming errors associated with manual memory management, such as dangling pointers and double frees. One downside of automatic memory management, however, is that programmers are left with less control and less information about the memory behavior of their programs [9]. Garbage Collectors are the programs responsible for various memory management activities such as; memory allocation, preserving and ensuring object references are maintained in memory and reclaiming memory occupied by objects that are no longer in use by program [6].

1.1.4.1 Garbage collection and the Java platform memory model

When the startup option –Xmx is specified on the command line of Java application (for instance: java –Xmx:2g MyApp) memory is assigned to a Java process. This memory is referred

to as the Java heap (or just heap). This is the dedicated memory address space where all objects created by Java program (or sometimes the JVM) will be allocated. As the Java program keeps running and allocating new objects, the Java heap (meaning that address space) will fill up. Eventually, the Java heap will be full, which means that an allocating threads is unable to find a large-enough consecutive section of free memory for the object it wants to allocate. At that point, the JVM determines that a garbage collection needs to happen and it notifies the garbage collector. A garbage collection can also be triggered when a Java program calls System.gc(). Using System.gc() does not guarantee a garbage collection. Before any garbage collection can start, a GC mechanism will first determine whether it is safe to start it. It is safe to start a garbage collection when all of the application's active threads are at a safe point to allow for it. Simply explained, it would be bad to start garbage collecting in the middle of an ongoing object allocation, or in the middle of executing a sequence of optimized CPU instructions, as context might be lost and thereby mess up end results [3].

A garbage collector should never reclaim an actively referenced object. A garbage collector is also not required to immediately collect dead objects. Dead objects are eventually collected during subsequent garbage collection cycles. While there are many ways to implement garbage collection, these two assumptions are true for all varieties. The real challenge of garbage collection is to identify everything that is live (still referenced) and reclaim any unreferenced memory, but do so without impacting running applications any more than necessary [3]. A garbage collector thus has two mandates:

1. To quickly free unreferenced memory in order to satisfy an application's allocation rate so that it doesn't run out of memory.
2. To reclaim memory while minimally impacting the performance (e.g., latency and throughput) of a running application.

1.1.4.2 Characteristics of Garbage Collector

Garbage collectors must have the following characteristics:

- It should be both accurate and comprehensive. It should not free up the memory used by live objects. Also it should reclaim garbage within small number of cycles.

- It should operate efficiently by not introducing long pauses during execution of the application.
- Garbage collector rearranges the free memory spaces into a single contiguous area so that is always space for the allocation of large objects.
- It should be scalable to operate in multithreaded and multiprocessor environments [6].

1.1.4.3. Design Choices

A number of choices must be made while selecting the garbage collection algorithms.

- **Serial versus Parallel:**

  Single task is performed at one time with serial collection. Only one CPU is used for collection even multiple CPU's are available. Parallel collection performs multiple tasks simultaneously on different CPU's. This enables collection to be performed more quickly, at the expense of some multiple potential and fragmentation [4].

- **Concurrent versus Stop-the-world:**

  Concurrent performs garbage collection concurrently with the application execution. Stop-the-world performs the garbage collection while the application is completely stopped. It is simpler than concurrent collection, since the heap size is frozen and objects are not changing during the collection. Sometimes it may be undesirable for some application to be paused. The pause time of concurrent collection is shorter. But care should be taken since it is operating over the objects that might be operated at the same time by the application. This adds some overhead to the concurrent collector that affects the performance and requires the larger heap size [4].

- **Compacting versus Non-Compacting Versus Copying:**

  Compacting garbage collector compacts the memory moving all the live objects together and completely reclaiming memory. Non-compacting garbage collector does not move all the live objects to create large reclaimed area. It only releases the space utilized by garbage objects. Third garbage collector is copying collector which copies objects to the different memory area considering the source area to be empty and available for faster and subsequent allocations. It consumes extra time and space for copying [4].

1.1.5. Java Heap Description

When a Java program started Java Virtual Machine gets some memory from Operating System. Java Virtual Machine or JVM uses this memory for all its need and part of this memory is call java heap memory. Heap in Java generally located at bottom of address space and move upwards. Whenever we create object using new operator or by any another means object is allocated memory from Heap and when object dies or garbage collected ,memory goes back to heap space in Java. The Java heap is divided into three main sections: Young Generation, Old Generation and the Permanent Generation as shown in the figure below.

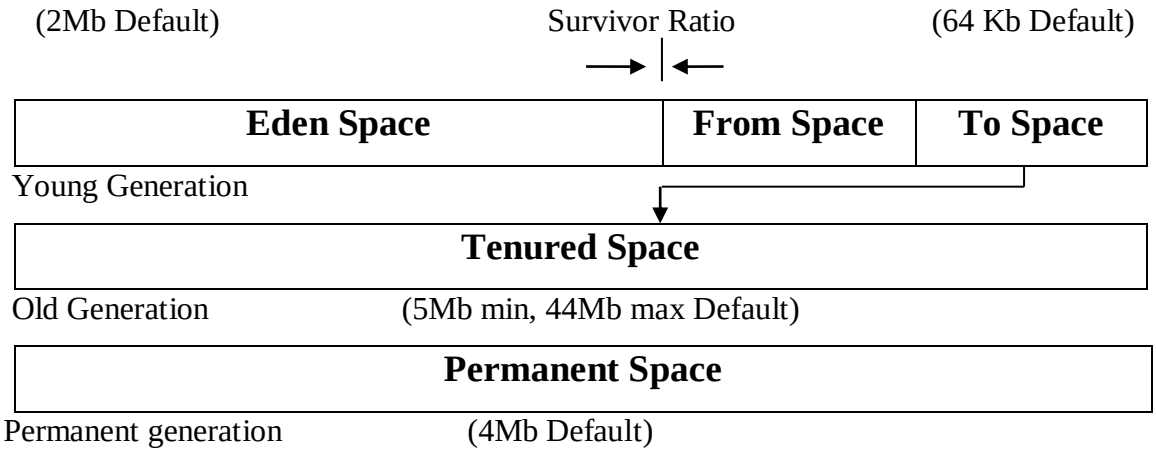

Figure 1.4: Java Heap

Young generation is further divided into two parts known as Eden space, and Survivor space. Survivor space is further divided into parts called 'to" and "from". When an object first created in heap its gets created in young generation inside Eden space and after subsequent Minor Garbage collection if object survives its gets moved to survivor 1 and then survivor 2 before Major Garbage collection moved that object to Old or tenured generation. Objects in this space are never garbage collected except in the two cases: Full Garbage Collection or Concurrent Mark-and-Sweep Garbage Collection. Permanent generation of Heap or Perm Area of Heap is somewhat special and it is used to store Meta data related to classes and method in JVM [4, 6].

When JVM starts JVM heap space is equal to the initial size of Heap specified by -Xms parameter, as application progress more objects get created and heap space is expanded to

accommodate new objects. JVM also run garbage collector periodically to reclaim memory back from dead objects. JVM expands Heap in Java somewhere near to Maximum Heap Size specified by -Xmx and if there is no more memory left for creating new object in java heap ,JVM throws “java.lang.OutOfMemoryError” and the application dies. JVM tries to run garbage collector to free any available space but even after that not much space available on Heap in Java it results into OutOfMemoryError [10].

1.2. Introduction

JVM is the 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 to decide exactly when to issue explicit instructions to free up memory. This feature of JVM is called garbage collection [4].

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, namely-

- ★ Parallel Scavenge.
- ★ Serial Garbage Collector
- ★ Parallel New + Serial GC
- ★ CMS
- ★ G1 (available in Java7)

1.2.1. Problem Definition

Garbage collection allows garbage to be created in order to avoid dangling references. Java's garbage collection (GC) has made life easier, perhaps, immeasurably, for developers. However, automatic memory management does not come without costs. Garbage collection cycles are unpredictable and applications may be susceptible to "pauses" or other performance issues while garbage collection is taking place. As the Java Virtual Machine (JVM) has matured,

there have been improvements introduced to minimize the impact of GC on applications. This study will compare and evaluate Mark Sweep, Concurrent Mark Sweep, and Garbage first garbage collection algorithms in JVM 1.7.

1.2.2. Objective of the study

The objective of this dissertation work is

- To gain insight knowledge about the different types of garbage collection algorithms in JVM and evaluate the performance of these algorithms to find the better one.
- To identify space-time trade-offs of theses algorithms.

1.3. Performance Matrix

Several metrics are utilized to evaluate garbage collector performance, but the three major attributes are:

- **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.

1.4. Motivation

In the field of computer programming dynamic memory allocation is a very important issue which is allocated in special area of memory called heap. Nowadays it sometimes seems that memory is not treated with the same accuracy as about twenty years before, when the main memory was a lot smaller than today. But especially in the field of embedded systems, main memory is still scarce. As the demand for larger and more complex programs also is ever increasing within embedded systems, possibilities have to be found which help to prevent that memory gets wasted.

When memory is allocated dynamically, memory space for the object of data is allocated in heap and remains allocated until it is freed manually by 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 in decreased productivity of programmers. To avoid these problems automatic memory management (garbage collection) is evolved as one of the exciting area of research area.

1.5. Organization of the study

The dissertation is organized into six chapters, chapter one is the introduction part. It deals with the background and introduction of JVM, memory management and how the memory is automatically managed by using garbage collector algorithms in java platform.

The past related works concerning the garbage collection have been discussed in chapter two. It also describes about the methodology used for collection of data in this dissertation.

Chapter three gives insight knowledge about the concepts about the three types of garbage collectors used for dissertation work namely Mark Sweep Algorithm, Concurrent Mark Sweep Garbage collector and Garbage First Garbage Collector.

The different types of tools used in dissertation for the design and implementation are discussed in chapter four.

Chapter five is named as Data collection and Analysis. In this chapter, the tests are carried out to find out the percentage of CPU and memory used by three types of garbage collectors. This is shown clearly with the help of graph individually for each algorithm in this chapter.

Chapter six is the conclusion part. It has been concluded that if a computer (generally server) has good CPU and RAM then G1GC is a efficient but if a computer has average CPU and good RAM, then CMS holds the edge over G1GC.

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

CHAPTER – TWO

RESEARCH METHODOLOGY AND LITERATURE REVIEW

2.1. Literature Review

A number of GC algorithms have been proposed in the past. A classical algorithm is "Reference Counting" [11], which associates with each allocated block of memory a counter the number of pointers pointing to that block. Each additional pointer referencing the block increments the counter, as pointers withdraw from pointing to it the count decrements. The block is reclaimed as free as its count drops to zero. The other class of GC algorithms is the tracing schemes. Instead of maintaining the state of each piece-wise memory being free or referenced at all time, it scans though a large portion of the memory space periodically to determine such states, and then processes the "garbage", or reclaims the free memory. One example of such schemes is the "Mark and Sweep" algorithm [12]. The algorithm traverses through the whole heap, marks all the live objects, and then "sweeps" all the unmarked "garbage" back to the main memory.

A out-grow of the tracing scheme are the moving collectors. This class of algorithms involves moving data to different area of memory depending of their state of liveness. A classical example of it is the Copy algorithm. It divides the heap into two subspaces, one labeled "FromSpace" and the other "ToSpace". After tracing through all the objects in one half of the memory space, it moves all the live objects in the "FromSpace" to the "ToSpace", and declares this half the recycled garbage, or the free memory pool. After a period of time, it does memory reclamation on the other half and reverse the role of the two half spaces. Each of these algorithms has its strength and shortcomings. As a result, new and improved algorithms kept coming out. One optimization to the tracing and moving collectors is the incremental version. The incremental schemes solve the problem of the active process having to be halted for a long delay while the GC routine is running. The concurrent collector interleaves GC with application execution. This makes it attractive for the user interactive and real-time system. A wildly acknowledged variation of the moving collector is the Generational Collector [13]. This algorithm divides memory into spaces. When one space is garbage collected, the live objects are

moved to another space. It has been observed that some objects are long-lived and some are short-lived. By carefully arranging the objects, objects of similar ages can be kept in the same space, thus causing more frequent and small-scaled GC's on certain spaces. The generational garbage collector achieve efficiency because newer records point to the older records by store operation to a previously created records and such operation are rare in many languages. Such a garbage collection can be so efficient that the allocation of records costs more than their disposal. This paper presents simple, efficient, low-overhead version of generational garbage collection with fast allocation suitable for implementation in UNIX environment [13].

A large number of concurrent collectors have also been proposed in the past [14, 15, 16]. The task of detecting an collecting garbage collection was performed by an additional processor operating concurrently with the processor devoted to the computation proper [14]. The paper reported the design and implementation of "quasi real time garbage collector for concurrent calm light, an implementation of ML with threads. The two generation system combines the fast, asynchronous copying collector on a young generation with a non disruptive concurrent marking collector on an old generation. This design crucially relies on a ML compile time distinction between mutable and immutable objects [15]. Mostly-concurrent garbage collection [17] algorithm begins by marking the roots and tracing concurrently with the program run. Since the object connectivity graph is modified by the program while the collector traverses the heap, a write barrier is required to ensure correctness. A variant of mostly-concurrent garbage collector [18] gains some synergies by implementation in a generational collector with mutator cooperation. This paper presents that the garbage collector is based on "mostly parallel" collection algorithm and can be used as old generation in generational computer system. It overloads efficient write barrier codes already generated to support generational garbage collection to also identify objects that were modified during concurrent marking. These objects must be rescanned to ensure that the concurrent marking marks all the live objects. The algorithm minimizes the maximum garbage collection pause time, while having only a small impact on average garbage collection pause time and overall execution time. Collectors that use parallelism to decrease evacuation pauses have been reported by [19]. The paper suggested that the pause time and throughput can be improved by exploring the parallel collectors and implementing the two parallel collectors [20].

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 objects [21, 22]. 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 was 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 [22]. Mature Object Space Collector uses the idea of dividing collection of a large heap into incremental collection of smaller portions of the heap [23].

Incremental incrementally compacting garbage collection [24] algorithm 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 a 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 [25] describes a similar system to that of [24]. They augment the collector described in [24] 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 constrast, Garbage-First allows compaction to be performed in a number of smaller steps.

Mark-Copy garbage collector [26], 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 [24].

Another interesting research area is that of hard real-time garbage collection. Mostly non-moving, dynamically defragmenting collector is presented in order to meet the real time bounds. A Real-time Garbage Collector with Low Overhead and Consistent Utilization is able to meet real-time bounds by avoiding copying in most cases, keeping space requirements low; and by fully incrementalizing the collector. A later paper describes heuristics for when to compact, and which and how many regions to compact. The major difference is that the [25], to guarantee hard real-time behavior, performs compaction in small atomic steps. The paper presented the real-time garbage collector for java, an analysis of its real time properties, and implementation was done to show that the applications were able to run with high mutator utilization and low variance in pause times [25].

2.2 Research Methodology

Research is a careful study performed to find out new things in a systematic way. In a scientific method of research at first problem is formulated then according as collected input data, output information is analyzed and finally the information is generalized. This dissertation work is truly scientific and flows in the same way

The design of this thesis is quantitative research design. The research strategy set up for this thesis work is pictorially shown in figure 2.1below. Internet is the main source for collection of articles related to this thesis work. The relevance of the articles is decided according to the topic and also using intuition of the researcher. Main focus of this dissertation is to experimentally evaluate the different garbage collection algorithms available in JVM 7 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 which is not in the generalized form. This work is only specialized for garbage collectors provided in JVM 7. Above mentioned steps can be formalized in a diagram as follows:

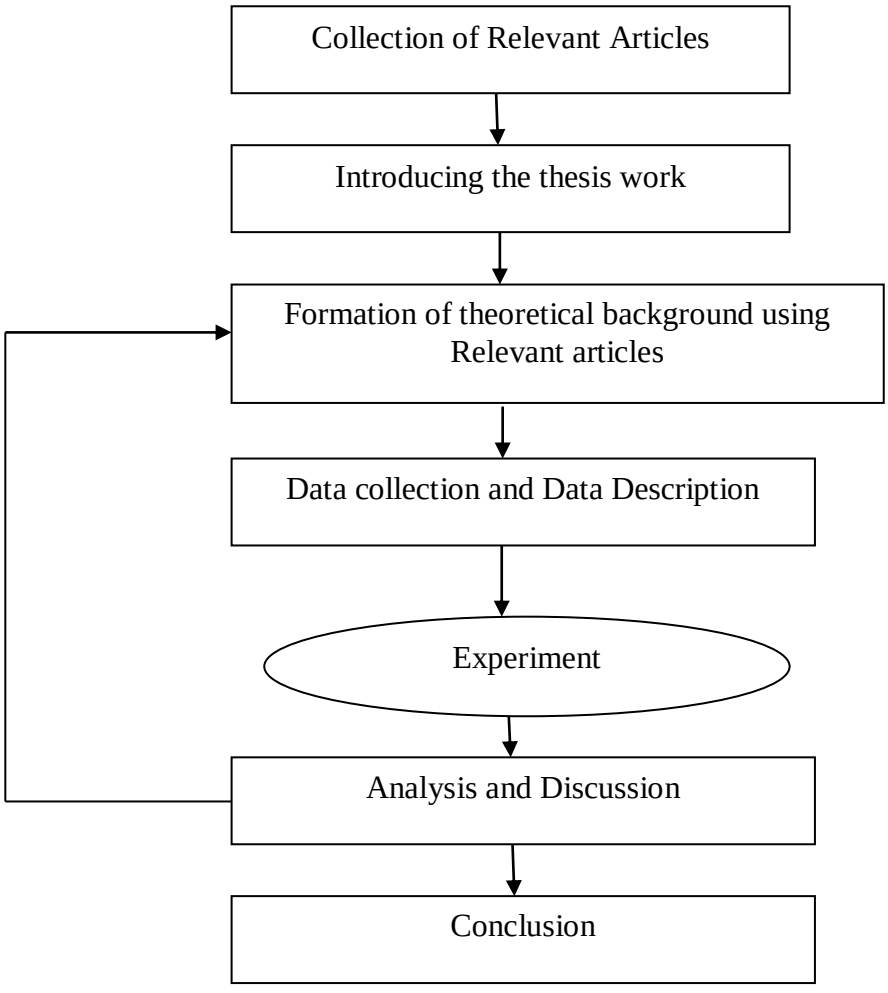

Figure 2.1: Research Methodology

CHAPTER – THREE

DESCRIPTION OF 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 them are described below:

3.1 Mark and Sweep Algorithm

Mark and sweep is one of the earliest and best-known garbage collection algorithms. As the name suggests, this algorithm has two main phases: Mark Phase and Sweep Phase.

Mark phase does a DFS from every root object. Root objects are the local variables or any static variables. It basically "paints" or "Marks" all the live objects [27]. During this phase, application execution is momentarily frozen. A Garbage collection safe point is a point or range in a thread's execution where the collector can identify all the references in that thread's execution stack [28]. All reachable objects will be marked live, and all non-reachable objects will be marked dead. In sweep phase, dead objects are "swept" which means, the memory occupied by the dead objects are reclaimed.

The mark-and-sweep algorithm consists of two phases: In the first phase, it finds and marks all accessible objects. The first phase is called the mark phase. In the second phase, the garbage collection algorithm scans through the heap and reclaims all the unmarked objects. The second phase is called the sweep phase. The algorithm can be expressed as follows:

```
for each root variable r
   mark (r);
sweep ();
```

The advantages of mark sweep algorithm are that all the memory can be used for allocation and there is no need of costly object copying and pointer updates. The disadvantage is that the heap can become fragmented after a few cycles making the allocation more complex and costly [29].

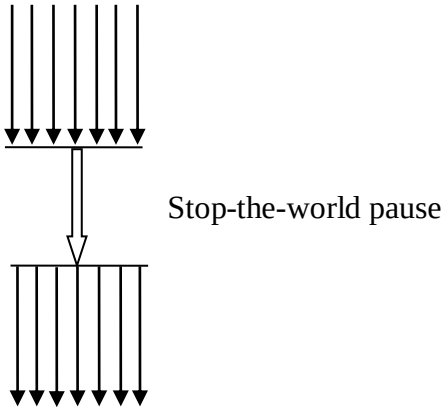

Figure 3.1: Mark Sweep Algorithm

Mark and Sweep algorithm is also known as simple stop-the-world garbage collector. It completely halts execution of the program to run a collection cycle during which new objects are not allocated and objects do not suddenly become unreachable while the collector is running.

3.2. Concurrent Mark Sweep Garbage Collector

Concurrent Mark Sweep (CMS) is one of HotSpot JVM low pause garbage collectors. Though CMS can do most of its work for reclaiming memory concurrently with application's execution, but still it requires few stop-the-world pauses to make its work.

CMS is a generational collector, it means that heap is separated into young and old space and these spaces are collected independently. For young space collection usual copy collector is used. Concurrent Mark Sweep is used only to collect old space.

Concurrent mark sweep garbage collector performs garbage collection concurrently with the application's execution. This collector consists of 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 to 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 [3]. This also means that there is a need for larger heap size for the concurrent marking and execution of the mutator (which continues to allocate memory for new objects).

The pause time of concurrent collection is shorter. But care should be taken since it is operating over the objects that might be operated at the same time by the application. This adds some overhead to the concurrent collector that affects the performance and requires the larger heap size [3].

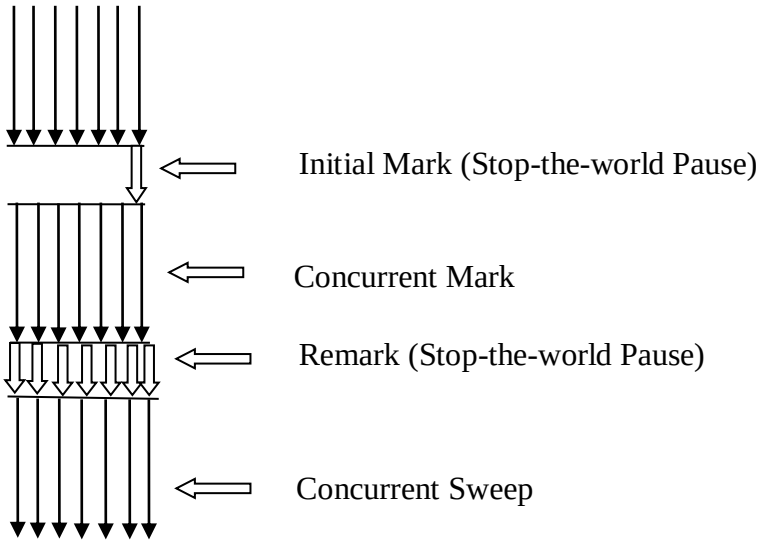

Figure 3.2: Concurrent Mark Sweep Algorithm

Fig 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 stop-the-world pause manner where the entire program threads, or 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. 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.

CMS collector is mostly used when the application needs shorter pauses and can afford to share processor resources with garbage collector when application is running. CMS is required for the application that have a relatively large set of long-lived data, and that run on machines with two or more processors. The CMS collector should be considered for any application with low pause time [3].

3.3. Garbage First Garbage collector

Garbage-First garbage collector (G1GC) is a server-style garbage collector, targeted for multi-processors with large memories, that meets a soft real-time goal with high probability. The G1 collector operates on a heap broken down into equally sized regions (Figure 3.3.1). In the strictest sense, the heap doesn't contain generational areas, although a subset of the regions can be treated as such. This provides flexibility in how garbage collection is performed, which is adjusted on-the-fly according to the amount of processor time available to the collector [9].

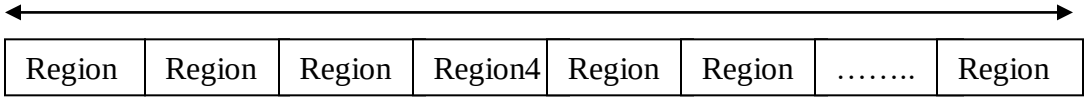

Figure 3.3: Heap Regions

Regions are further broken down into 512 byte sections called cards (figure 3.3.2). Each card has a corresponding one-byte entry in a global card table, which is used to track which cards are modified by mutator threads. Subsets of these cards are tracked, and referred to as Remembered Sets (RS). The G1 collector works in stages. The main stages consist of remembered set (RS) maintenance, concurrent marking, and evacuation pauses.

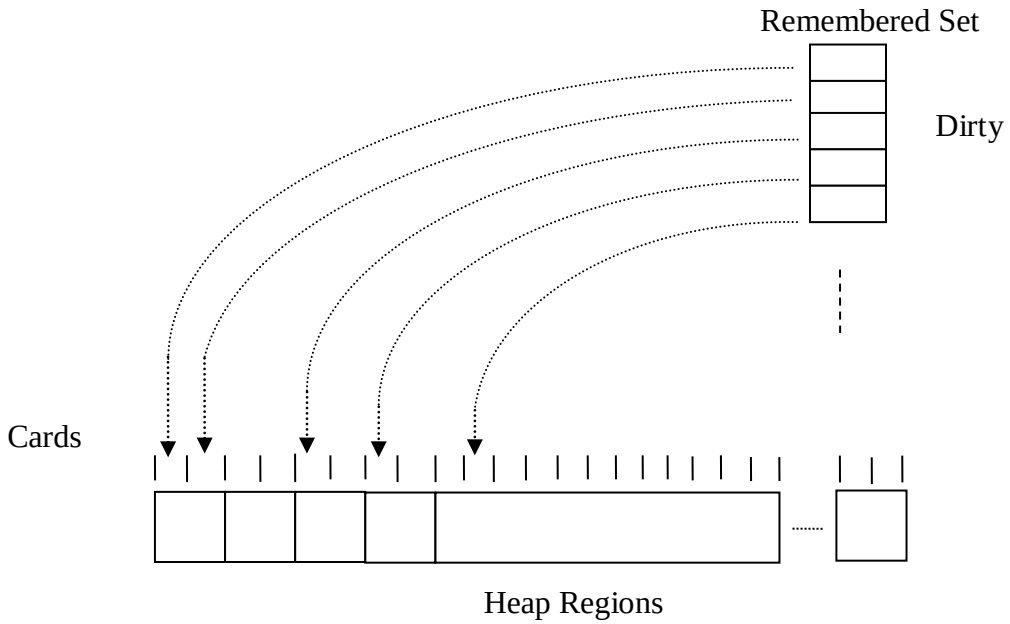

Figure 3.4: Division of Heap region

Each region maintains an associated subset of cards that have recently been written to, called the Remembered Set (RS). To be precise, for a particular region (i.e., region a), only cards that contain pointers from other regions to an object in region a are recorded in region a's RS. A region's internal references, as well as null references, are ignored. It's best to think of the RS as one logical set of dirty cards.

Concurrent marking identifies live data objects per region, and maintains the pointer to the next free byte, called top. A marking bitmap is maintained to create a summary view of the live objects within the heap. Each bit in the bitmap corresponds to one word within the heap. A bit in the bitmap is set when the object it represents is determined to be a live object. In reality there are two bitmaps: one for the current collection, and a second for the previously completed collection. This is one way that changes to the heap are tracked over time. When evacuation begins, all mutator threads are paused, and live objects are moved from their respective regions

and compacted into other regions. Although other garbage collectors might perform compaction concurrently with mutator threads, it's far more efficient to pause them. Since this operation is only performed on a portion of the heap -- it compacts only the collection set of regions -- it's a relatively quick, low-pause, operation. Once this phase completes, the GC cycle is complete [29].

CHAPTER –FOUR
DESIGN AND IMPLEMENTATION

4.1. Programming Language

Java is a programming language and computing platform first released by Sun Microsystems in 1995. It is the underlying technology that powers state-of-the-art programs including utilities, games, and business applications. Java runs on more than 850 million personal computers worldwide, and on billions of devices worldwide, including mobile and TV devices. There are lots of applications and websites that won't work unless Java has been installed, and more are created every day. Java is fast, secure, and reliable. From laptops to datacenters, game consoles to scientific supercomputers, cell phones to the Internet, Java is everywhere. Since main purpose of this dissertation is to evaluate garbage collectors available in JVM 7, application programs that are used as test cases are written in java programming language [31].

4.2. Tools Used

4.2.1. PrintGCDetails

One of the easiest ways to get initial information about garbage collections is to specify the command line option –XX:+PrintGCDetails. For every collection, these results in the output of information such as the size of live objects before and after garbage collection for the various generations, the total available space for each generation, and the length of time the collection took [32].

4.2.2. PrintGCTimeStamps

This outputs a timestamp at the start of each collection, in addition to the information that is output if the command line option –XX:+PrintGCDetails is used. The timestamps can help you correlate garbage collection logs with other logged events [32].

4.2.3. Xloggc

The java command accepts the directive - Xloggc that specifies a file name where the garbage collection diagnostic data is written. Details about the garbage collection activity, such

as size of the young and old generation before and aft

er garbage collection, size of total heap, elapsed time it takes for a garbage collection to happen in young and old generation, and size of objects promoted at every garbage collection, and other data, can be recorded by using the both the - XX:+PrintGCDetails and -XX:+PrintGCTimeStamps argument [3].

4.2.4. 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, to enable you to view different data about multiple Java applications uniformly, whether they are running locally or on remote machines. Furthermore, developers can extend Java VisualVM to add new functionality by creating and posting plug-ins to the tool's built-in update center.

Java VisualVM can be used by Java application developers to troubleshoot applications and to monitor and improve the applications' performance. Java VisualVM can allow developers to generate and analyze heap dumps, track down memory leaks, perform and monitor garbage collection, and perform lightweight memory and CPU profiling. Java VisualVM was first bundled with the Java platform, Standard Edition (Java SE) in JDK version 6, update 7 [32].

4.3. Experimental Setup

All the test cases are executed in hardware platform with intel core i3 processor with a 2.4GHz clock, a 1333MHz DDR3 memory of size 2 GB, Level1 cache of size 64KB, and a Level2 cache of 256KB. Processor operates in 32-bit mode, and use 32-bit kernels. The system runs windows 7 kernels.

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

4.4. Test Case Design

4.4.1. Test Case1

In test case, integer arrays are added to the array list which is immediately eligible for garbage collection.

```
import java.util. ArrayList;
public class GCTest
{
	private static ArrayList <Integer[]> array= new ArrayList
<Integer[]>(200);
	public static void main(String[] args) throws InterruptedException
	{
		Thread.sleep(5000);
		for (int i = 0;i <200; i++)
		{
			array.add(new Integer[1*256*256]);
			Thread.sleep(50);
		}
		for (int i = 0;i <200; i++)
		{
			array.remove(array.size() - 1);
			if (i%10 == 0)
			{
				System.gc();
				System.out.println("Garbage is Collected");
			}
			Thread.sleep(50);
		}
		System.gc();
		Thread.sleep(3000);
	}
```

The code creates and adds 200 integer Arrays into an Array list. Each integer array reserves 256 KB of memory that is {256 x 256 x 4 byte = 256 KB, 200 x 156 KB= 50 MB}. Arrays are removed during iterations. At every 10th iteration, System.gc() is called, suggesting the Java Virtual Machine to start garbage collection. Arrays removed from array list are eligible for garbage collection.

4.4.2. Test Case 2

In test case 2, string is concatenated with integer array resulting into string and string objects are eligible for garbage collection only after they are removed from garbage collection.

```
import java.util.ArrayList;
public class GCTest2
{
        private static ArrayList<String> array = new ArrayList<String>(750);
        public static void main(String[] args) throws InterruptedException {
        Thread.sleep(5000);
        for (int i=0;i<750;i++)
        {
                        array.add(""+new int[1 * 256 * 256]);
                        Thread.sleep(50);
         }
        for (int i=0;i<750;i++)
        {
                 array.remove(array.size()-1);
                        if (i%10 == 0)
                {
                        System.gc();
                        }
                        Thread.sleep(50);
        }
        System.gc();
        Thread.sleep(3000);
   }
 }
```

The code creates and adds 750 strings into an Array list. “new int[1 x 256 x 256]” return starting address of the reserved integer array and is concatenated with empty string which results into an

string and is added to Array list. Since, all integer arrays are orphan, they are eligible for garbage collection immediately. Strings are removed during iterations. At every 10th iteration, System.gc() is called, suggesting the Java Virtual Machine to start garbage collection. String objects removed from array list are also eligible for garbage collection.

4.4.3. Test Case 3

```
public class TestGCThread extends Thread
{
	public static void main(String[]  args) throws InterruptedException
	{
		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(InterruptedException e){ }
			// create random number of objects
			int count = (int) Math.round(Math.random() * 10 * 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(((now - start)/1000) + " s\t" +
                            TestGCObject.created + " created\t" +
                            TestGCObject.freed + " freed"       );
                        then = now;
                        }
                }
        }
}
```

```
public class TestGCObject
{
        static long created;
        static long freed;
        public TestGCObject()
        {
                created++;
        }
        public void finalize()
        {
                freed++;
        }
}
```

The main thread class TestGCThread is presented in first box. To avoid CPU starvation, while running, this thread enters a sleep period of variable duration (0-100 ms). After each sleep period, it creates a variable number of TestGCObject instances (0-10,000 at a time). The allocated objects can be immediately garbage collected as they are not being used further. The local variables are used to count each time an instance of TestGCObject is created in a constructor or finalized in a finalize().

CHAPTER-FIVE

DATA COLLECTION AND ANALYSIS

5.1. Memory and CPU Usage for Test Case1

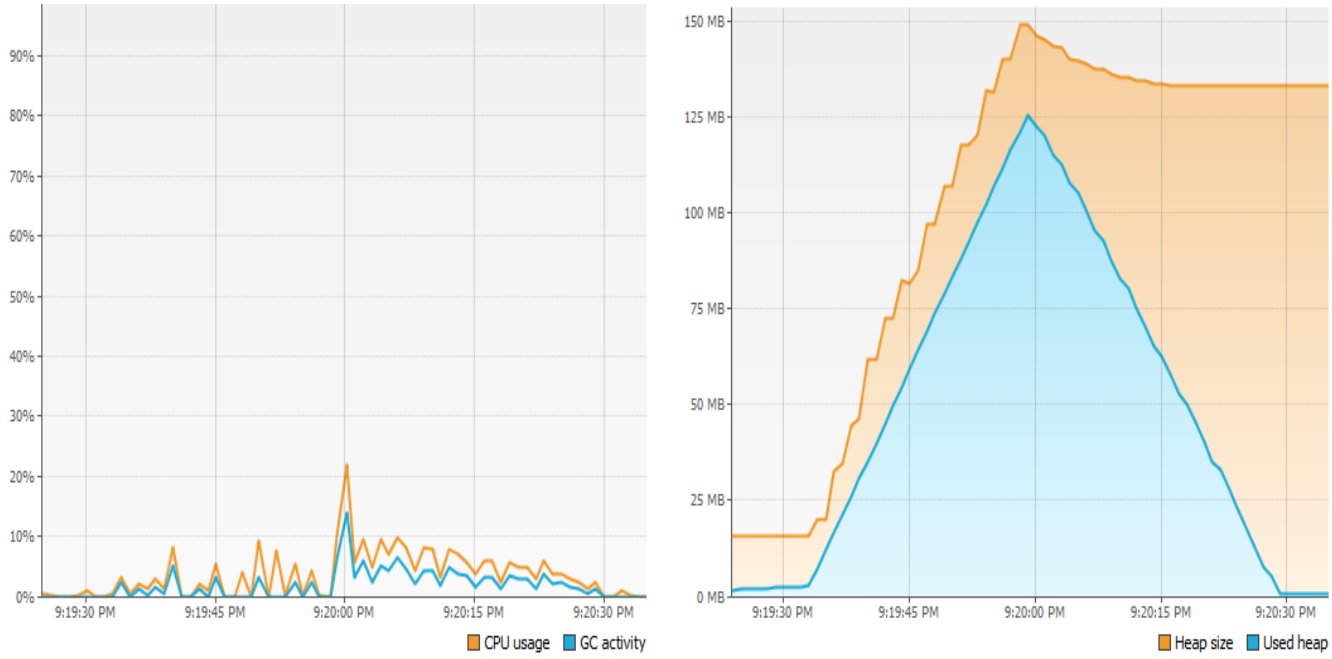

Figure 5.1: CPU and Memory Usage with Mark Sweep GC

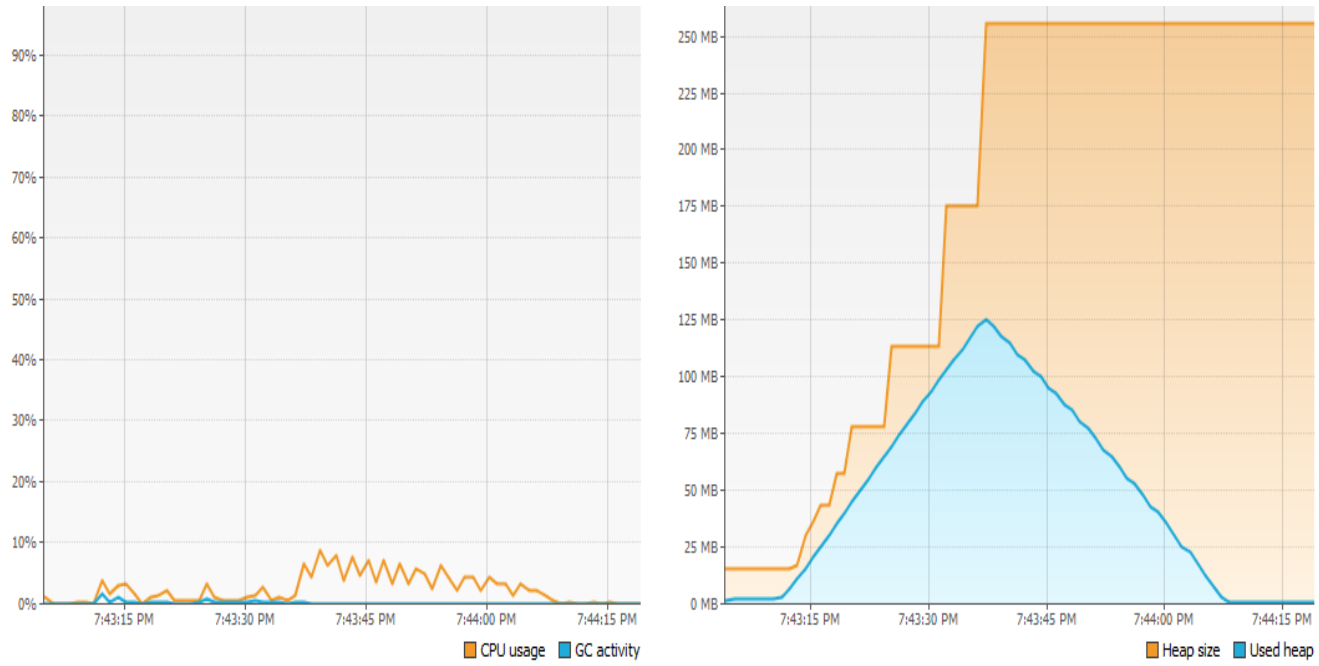

Figure 5.2: CPU and Memory Usage with Concurrent Mark Sweep

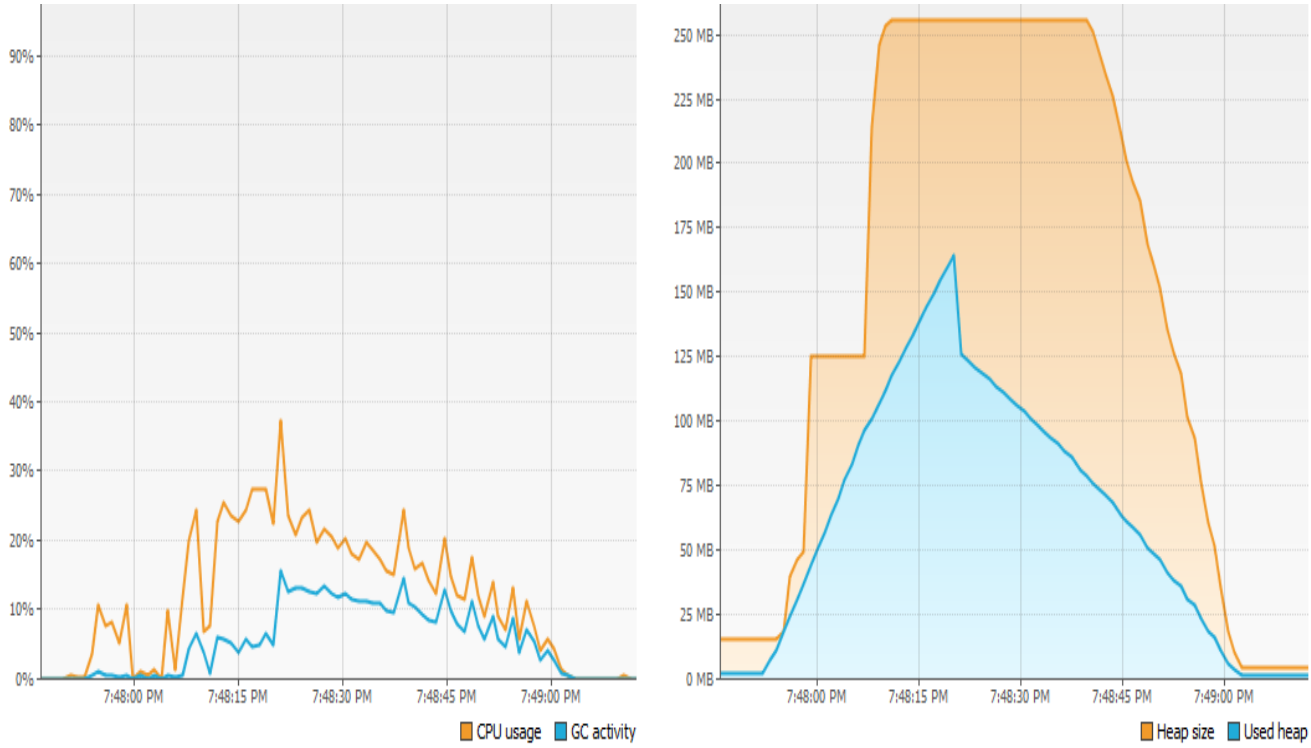

Figure 5.3: CPU and Memory Usage with G1GC

From above figures it can be seen that for mark sweep garbage collector maximum CPU usage is 21.1% and maximum GC activity is 14.2%. Also, it has average CPU usage above 7% and its average GC activity is around 4%. Similarly, in case of concurrent mark sweep garbage collector maximum CPU usage is 8.8% and maximum GC activity is 1.9%. Average CPU usage is around 4% and average GC activity is below 1% for CMS collector. Both mark sweep and concurrent mark sweep garbage collectors use maximum 125 MB of available heap memory and does not return deallocated memory to operating system.

The case with garbage first garbage collector is different. Its maximum CPU usage is 37.4% and maximum GC activity is 15.9%. G1GC have average CPU usage more than 15% and its average GC activity is above 10%. But G1GC use maximum 170 MB of heap memory available and returns deallocated memory to operation system.

5.2. CPU and Memory Usage for Test Case2

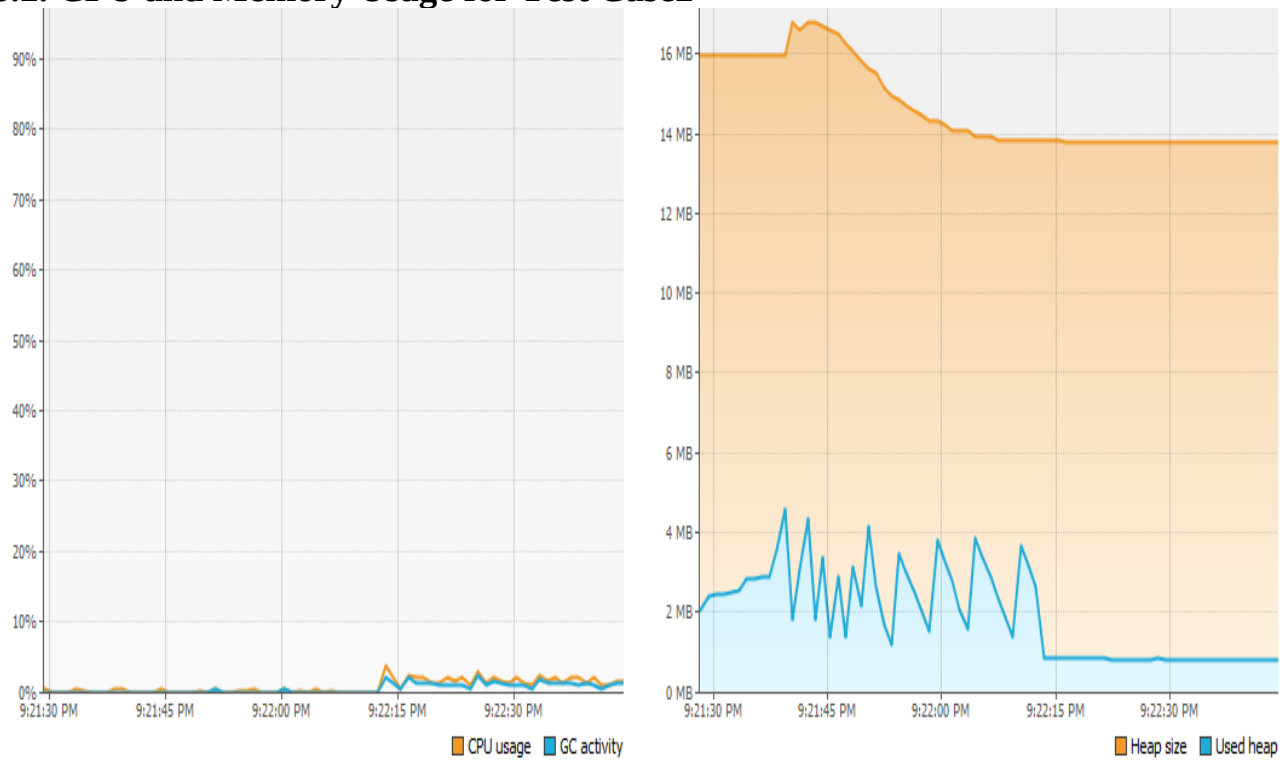

Figure 5.4: CPU and Memory Usage with Mark Sweep GC

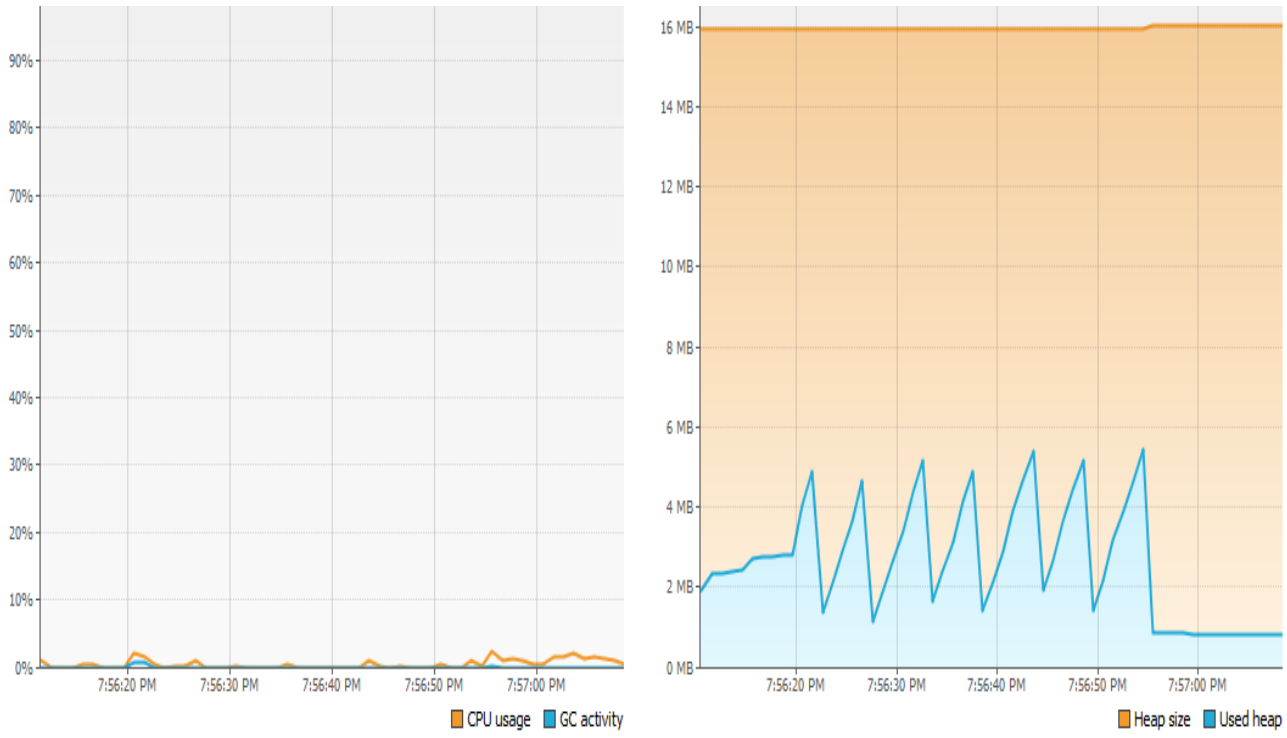

Figure 5.5: CPU and Memory Usage with Concurrent mark sweep

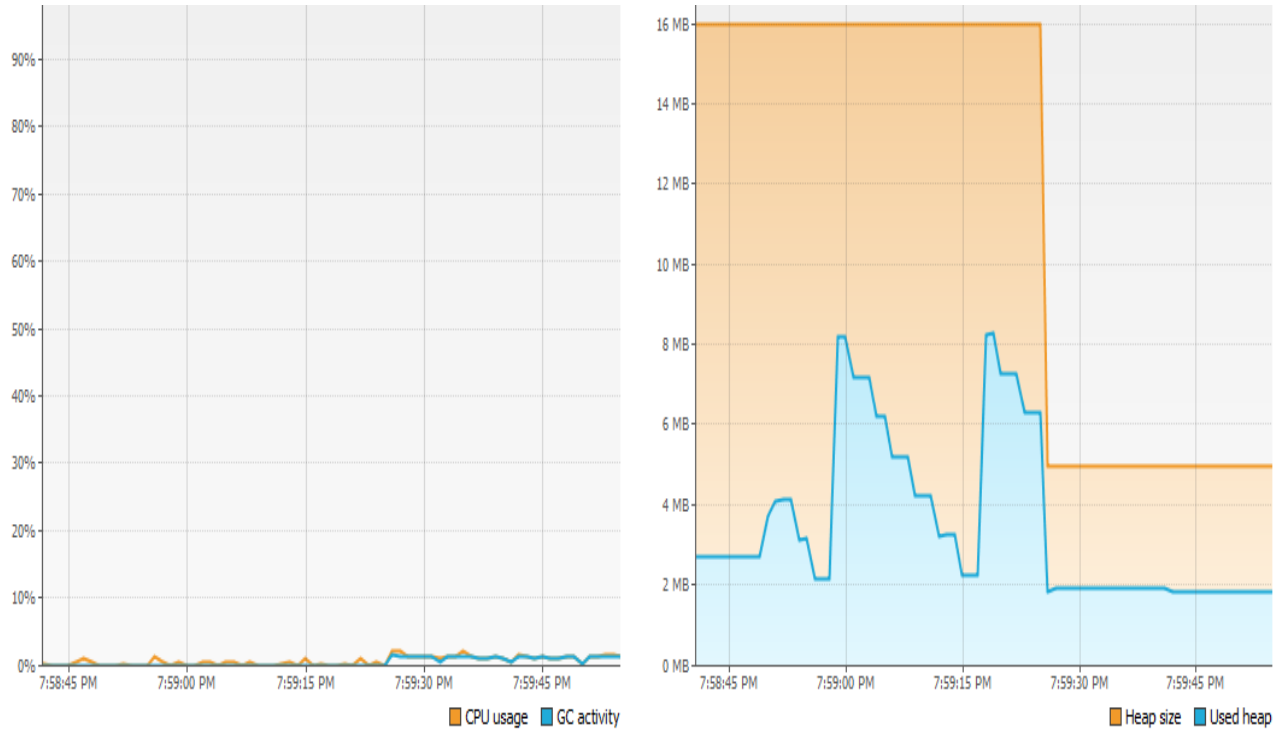

Figure 5.6: CPU and Memory Usage with G1GC

Here, behavior of garbage collectors is quite different from their behavior in test case1 even if two test cases are seems to be similar on surface. The reason behind this is that in case of test case2 integer arrays are not added to the array list rather string representations of addresses of integer arrays are added to array list and hence integer arrays are immediately eligible for garbage collection. And, string object are eligible for garbage collection only after they are removed from array lists.

In this case, maximum CPU usage is 3.9% and maximum GC activity is 2.5% for mark sweep garbage collector. Also, it has average CPU usage and GC activity below 2%. Similarly, in case of concurrent mark sweep garbage collector maximum CPU usage is 2.7% and maximum GC activity is 1.0%. And, it has average CPU usage and GC activity below 1.5% and 0.5% respectively. Both mark sweep and concurrent mark sweep garbage collectors use maximum around 5 MB of available heap memory and does not return deallocated memory to operating system.

Garbage first garbage collector has maximum CPU usage 2.3 and maximum GC activity 1.9%.

G1GC has average CPU usage and average GC activity is above 1%. But G1GC use

maximum 8.5 MB of heap memory available and returns deallocated memory to operating system

5.3. CPU and Memory Usage for Test Case3

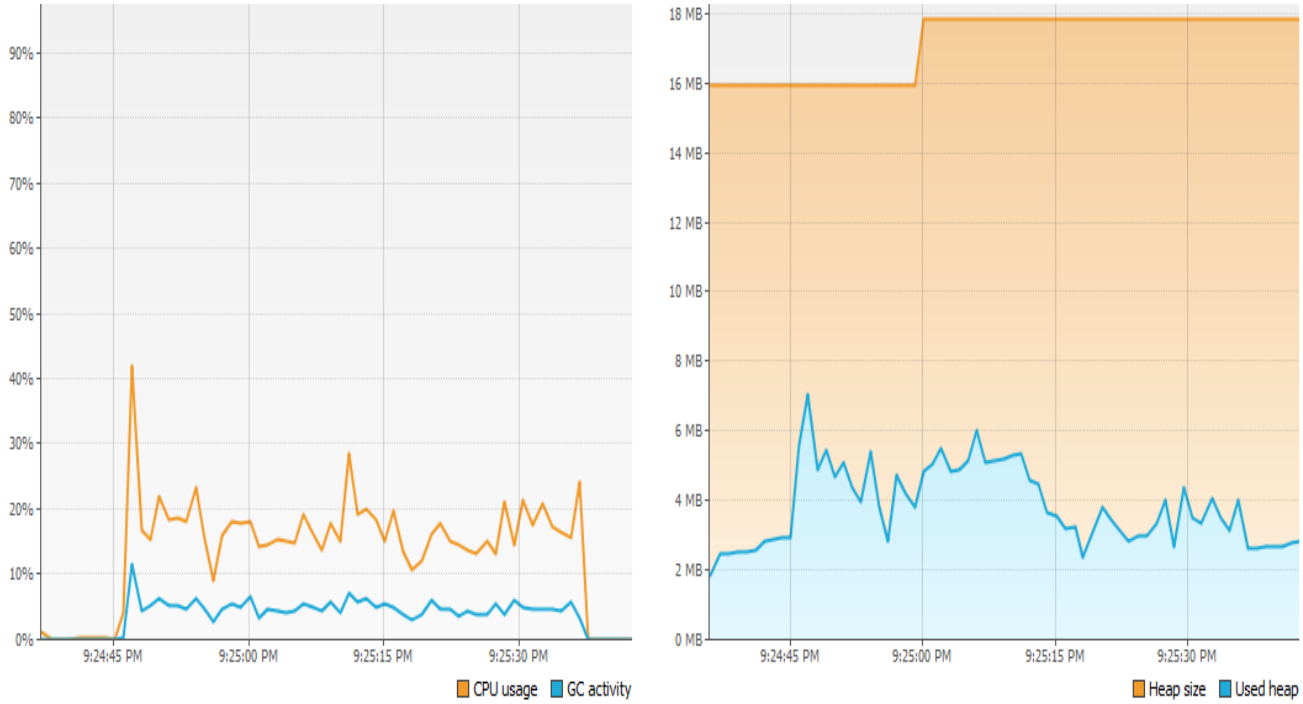

Figure 5.7: CPU and Memory Usage with Mark Sweep GC

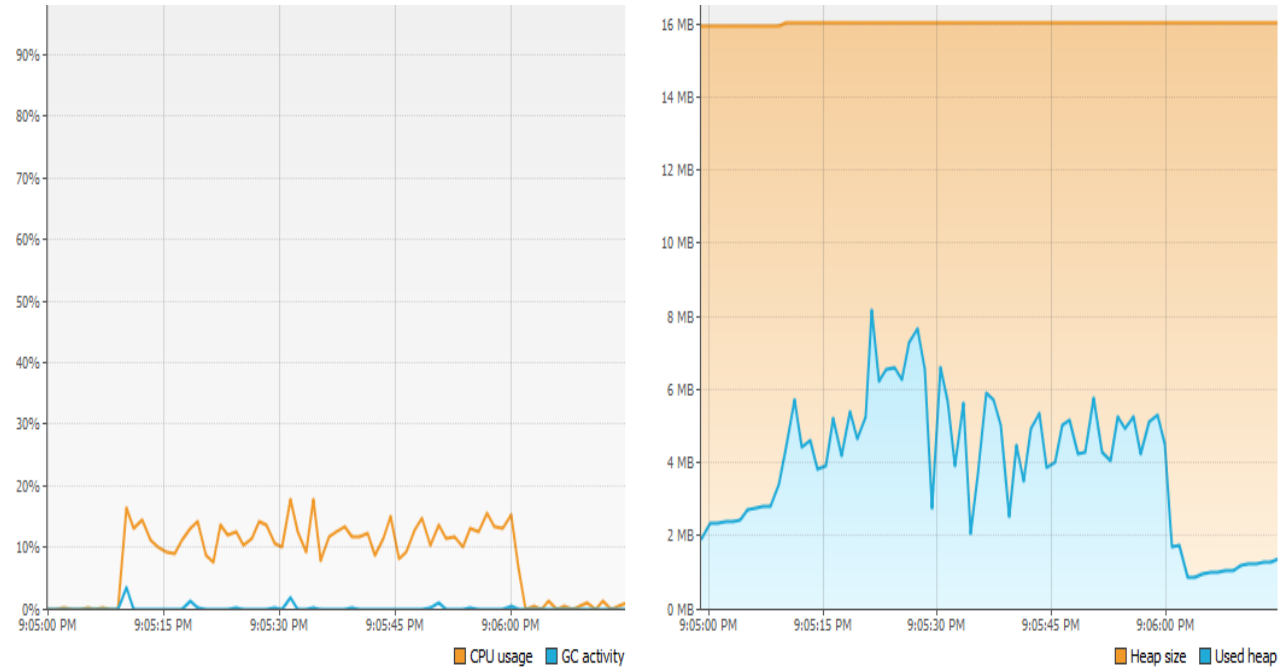

Figure 5.8: CPU and Memory Usage with Concurrent Mark Sweep

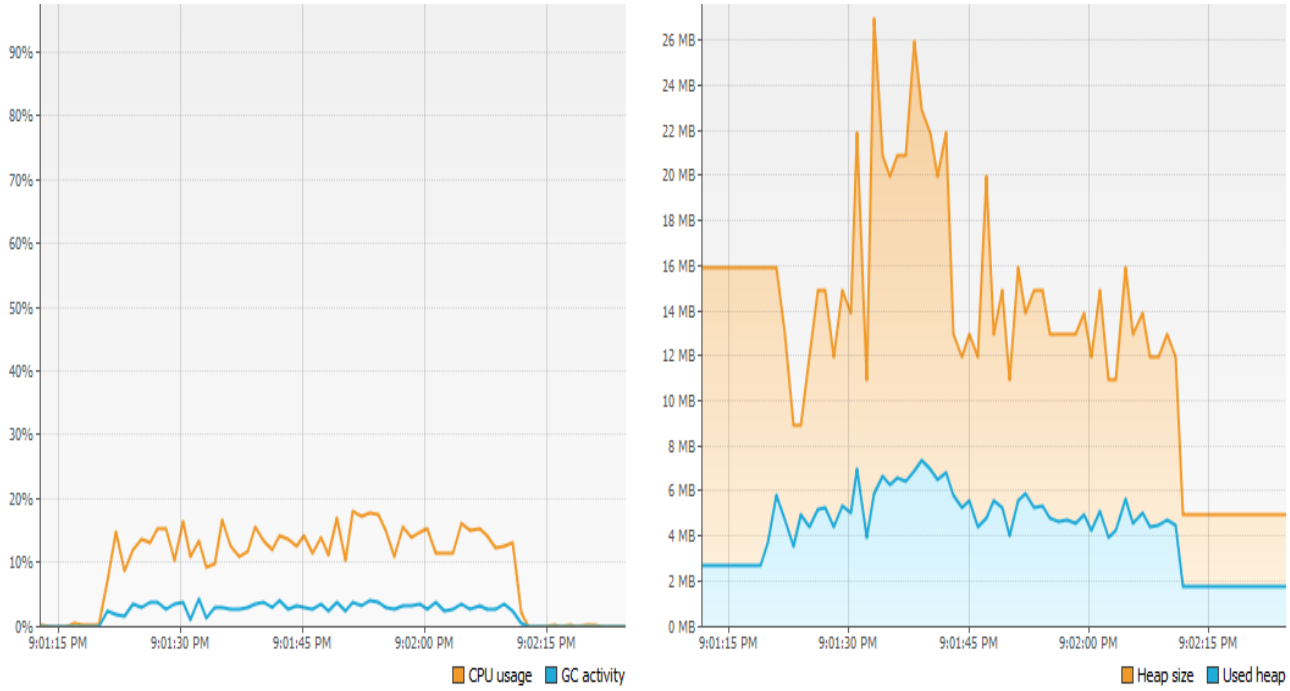

Figure 5.9: CPU and Memory Usage with G1GC

Finally, for test case 3, it can be seen that for mark sweep garbage collector maximum CPU usage is 42.2% and maximum GC activity is 11.6%. Its average CPU usage is above 20% and average GC activity is around 6%. Similarly, in case of concurrent mark sweep garbage collector maximum CPU usage is 17.9% and maximum GC activity is 3.8%. Average CPU usage is around 14% and average GC activity is below 1% for CMS collector. Here, both mark sweep and concurrent mark sweep garbage collectors use maximum 7 MB and 8.5 MB of available heap memory respectively and does not return deallocated memory to operating system.

Again, the case with garbage first garbage collector is different. Its maximum CPU usage is 18.4% and maximum GC activity is 4.5%. G1GC have average CPU usage more than 15% and average GC activity is around 3%. It uses maximum 7.75 MB of heap memory available and returns deallocated memory to operating system.

5.4. Summarizing GC Performance

Performance Metric	For Test Case1			For Test Case2			For Test Case3		
	MS	CMS	G1GC	MS	CMS	G1GC	MS	CMS	G1GC
Max CPU Usage	21.1	8.8	37.4	3.9	2.7	2.3	42.2	17.9	18.4
Max GC Activity	14.2	1.9	15.9	2.5	1	1.9	11.6	3.8	4.5
Max Used Heap	125	125	170	5	5	8.5	7	8.5	7.75

Table 5.10: Summary of maximum CPU and Memory Usage

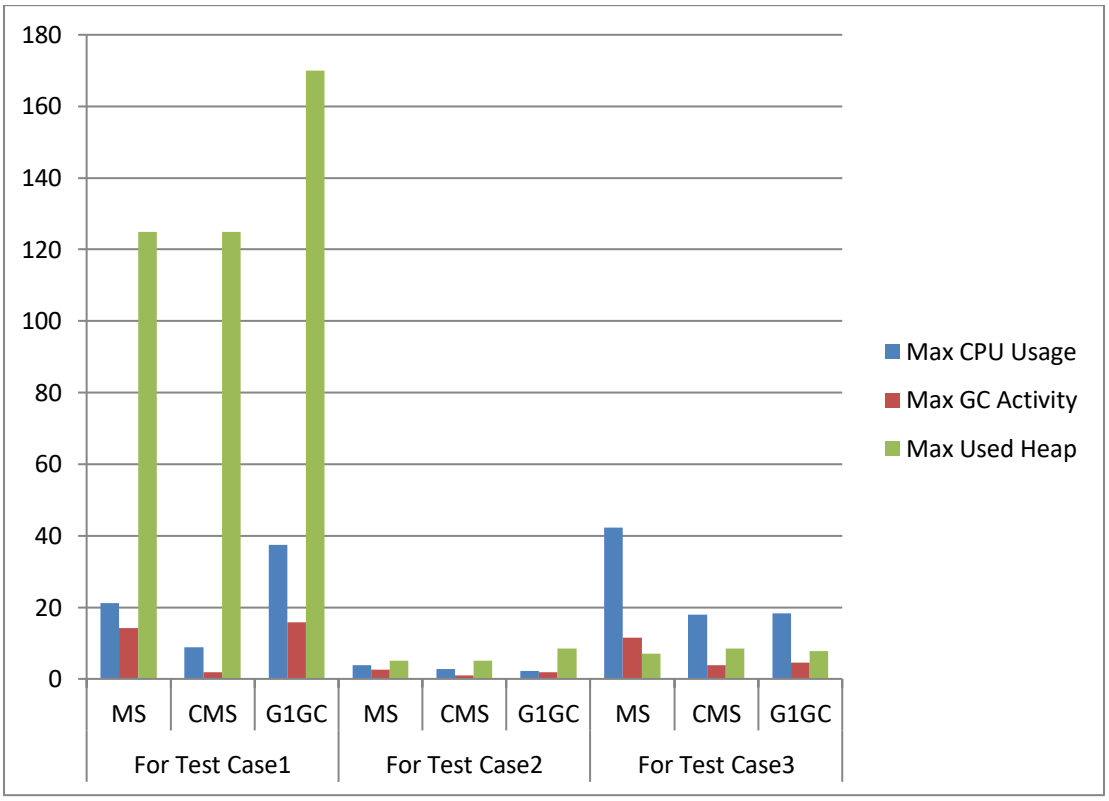

Figure 5.11: Graph for Table 5.10.

Performance Metric	For Test Case1			For Test Case2			For Test Case3		
	MS	CMS	G1GC	MS	CMS	G1GC	MS	CMS	G1GC
Avg. CPU Usage	7	4	15	2	1.5	1	20	14	15
Avg. GC Activity	4	1	10	2	0.5	1	6	1	3

Table 5.12: Average CPU and Memory Usage

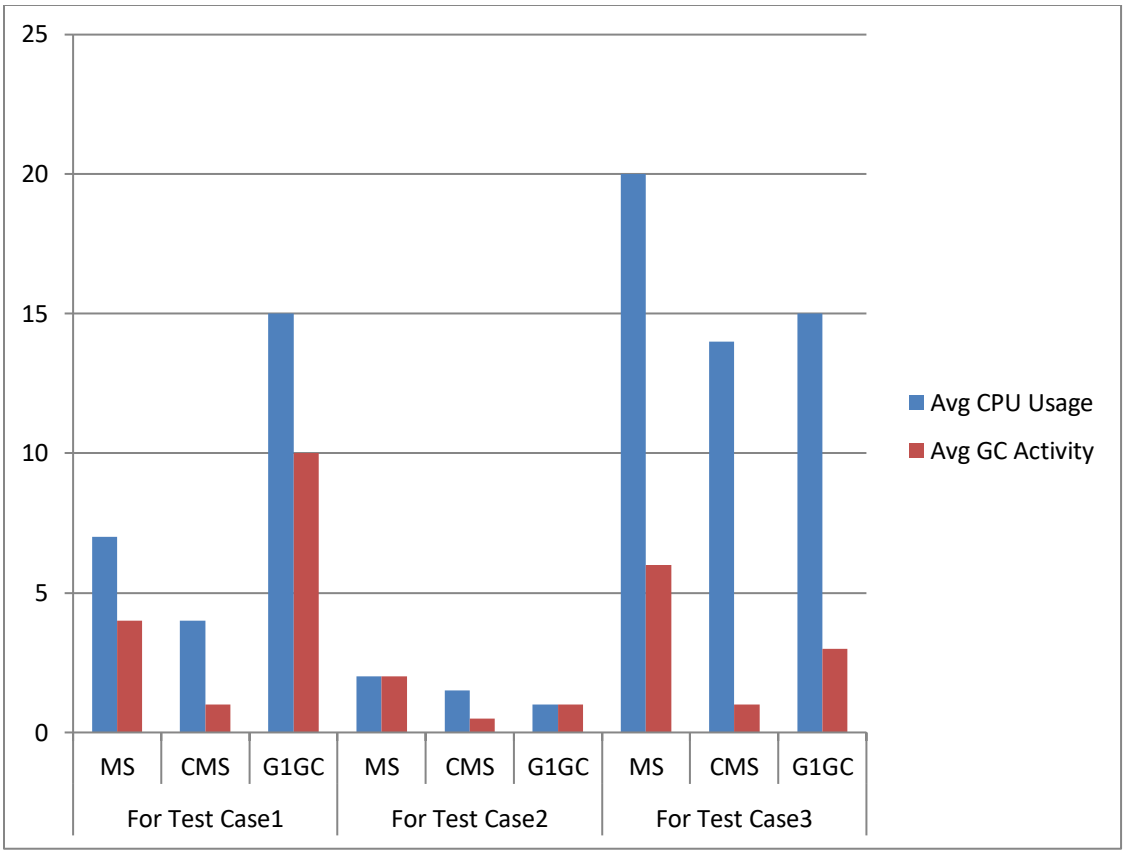

Figure 5.13: Graph for Table 5.12.

CHAPTER-SIX
CONCLUSION AND RECOMMENDATION

6.1. Conclusion

Java's garbage collection (GC) has made life easier, perhaps, immeasurably, for developers. However, automatic memory management does not come without costs. Garbage collection cycles are unpredictable and applications may be susceptible to "pauses" or other performance issues while garbage collection is taking place. As the Java Virtual Machine (JVM) has matured, there have been improvements introduced to minimize the impact of GC on applications. For example, now it is possible to choose what kind of GC to use for your application.

The dissertation has successfully evaluated three most widely used garbage collectors in JVM namely-Plain Mark Sweep, Concurrent Mark Sweep and Garbage First Garbage Collector. For test case 1 & 2 G1GC have more CPU usage and GC activity. But, in case of test case 3 Plain Mark Sweep garbage collectors have more CPU usage and GC activity whereas CPU usage and GC activity of G1 Collector and Concurrent Mark Sweep is comparable. On the other hand heap used by G1GC is also larger in test case 1 & 2 but in case of test case 3 heap used by Plain Mark Sweep is more than heap used by other two algorithms. Again, heap size used by G1GC and Concurrent Mark Sweep is comparable for test case 3. From this observation, it can be predicted than CPU and memory usage for G1GC and Plain Mark Sweep is normally higher than Concurrent Mark Sweep. Another observation that can be made from graphs (Figure 5.1, Figure 5.2, Figure 5.3, Figure 5.4, Figure 5.5, Figure 5.6, Figure 5.7, Figure 5.8, Figure 5.9, Figure 5.11, and Figure 5.12) in previous chapter is about heap size reclaimed by garbage collectors. Plain Mark Sweep and Concurrent Mark sweep never return deallocated memory to operating system whereas G1GC always returns garbage collected memory to operating system. Besides this, it is well known that pause time of Plain Mark Sweep garbage collector is larger than pause times of Concurrent Mark Sweep and Garbage First garbage collector.

Thus, from above observations it can be concluded that if a computer (generally server) has powerful CPU and RAM then G1GC is a good option. But for a computer with less powerful CPU and powerful RAM, CMS holds the edge over G1GC.

6.2. Further Recommendation

Observing other factors such as pause time, heap usage for permanent generation, throughput etc is the area of future research. There are many ideas for improving Garbage-First in the future. One of the ideas is to modify remembered set representations to increase the efficiency of remembered set processing. The research is also ongoing on investigating static analyses to remove write barriers in some cases. Further heuristic tuning may bring benefits. Finally, it believed that there are ways in which Garbage-First may make better use of compile-time escape analysis than standard generational systems can, and plan to investigate this.

REFERENCES

[1] Managed Runtime Initiative, Understanding Java Garbage Collection.

[2] Dietel Paul, Dietel Harvey. Java How to Program, 9th edition, 2011 march

[3] Description of HotSpot GCs: Memory Management in the Java HotSpot Virtual Machine White Paper, 2006

[4] Rick Byers, Garbage Collection Algorithms, Winter 2007.

[5] Witawas Srisa-an, Chia-Tien Dan Lo, and J. Moms Chang, "Scalable Hardware-algorithm for Mark-sweep Garbage Collection, 2000

[6] Clarence J M Tauro, Manjunath V Prabhu, International Journal of Computer Applications: "CMS and G1 Collector in Java 7 Hotspot: Overview, Comparisons and Performance Metrics", 2012

[7] Luke Dykstra, Witawas Srisa-an, and J. Morris Chan, IEEE:"An Analysis of the Garbage Collection Performance in Sun's HotSpotTM Java Virtual Machine", 2002

[8] Jacques Cuhen and Alexandru Nicolau, Brandeis University, Comparision of Compactiong algorithms for garbage cloolection, 1983

[9] Edward E. Aftandilian, Samuel Z. Guyer, GC Assertions: Using the Garbage Collector to Check Heap Properties, 2009

[10] Michael Finocchiaro. Java Performance Turing,2008

[11] George E. Collins. A method for overlapping and erasure of lists. Communications of the ACM, 3(12):655-657, December 1960.

[12] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. Communications of the ACM, 3:184-195, 1960.

[13] Andrew W. Appel. Simple generational garbage collection and fast allocation. Software Practice and Experience, 19(2):171-183, 1989.

[14] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M.Ste_ens. On-the-fly garbage collection: An exercise in cooperation. CACM, 1978.

[15] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1993.

[16] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: Portland, Oregon,1994,

[17] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In Brent Hailpern, editor, Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, 1991.

[18] Tony Printezis and David Detlefs, A generational mostly-concurrent garbage collector. In Proceedings of the International Symposium on Memory Management, Minneapolis, Minnesota, 2000.

[19] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In Proceedings of Supercomputing'97 (CD-ROM), San Jose, CA, November 1997

[20] Christine H. Flood, David Detlefs, Nir Shavit, and Xiaolan Zhang. Parallel garbage collection for shared memory multiprocessors. In Proceedings of the Java. Virtual Machine Research and Technology Symposium, Monterey, April 2001.

[21] Darko Stefanovic, Matthew Hertz, Stephen M. Blackburn, Kathryn S. McKinley, and J. Eliot B. Moss. Older-first garbage collection in practice: evaluation in a java virtual machine. In Proceedings of the workshop on memory system performance 2002.

[22] Lars T. Hansen and William D. Clinger. An experimental study of renewal-older-First garbage collection. In Proceedings of the seventh ACM SIGPLAN international conference on Functional programming, 2002.

[23] Richard L. Hudson and J. Eliot B. Moss. Incremental collection of mature objects. In Yves Bekkers and Jacques Cohen, editors, International Workshop on Memory Management, Lecture Notes in Computer Science, 1992.

[24] B. Lang and F. Dupont. Incremental incrementally compacting garbage collection. In Proceedings SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques, 1987.

[25] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In Conference Record of the Thirtieth Annual ACM

Symposium on Principles of Programming Languages, ACM SIGPLAN Notices, New Orleans, LA, January 2003.

[26] Narendran Sachindran, J. Eliot B. Moss. Mark-Copy: Fast copying GC with less space overhead ,2003

[27] Munehiro Takimoto, Formal Programming Language Theory, Dataflow Analysis, 2012

[28] Sergey V. Rogov, Viacheslav A. Kirillin, and Victor V. Sidelnikov,"Optimization of Java Virtual Machine with Safe-Point Garbage Collection", 2006

[29] David Detlefs, Christine Flood, Steve Heller, Tony Printezis, GarbageFirst Garbage Collection, 2004

[30] Katherine Barabash, Scalable Garbage Collection on Highly Platforms, 2011.

[31] http://www.java.com/en/dowmload/faq/whatis_java.xml

[32] http://docs.oracle.com/javase/6/docs/technotes/guides/visualvm/index.html

APPENDICES

Appendix 1:

8.122: [GC 8.122: [ParNew: 4416K->512K(4928K), 0.0141254 secs] 4416K->861K(15872K), 0.0142818 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]

15.622: [GC 15.622: [ParNew: 4901K->512K(4928K), 0.0798081 secs] 5250K->3695K(15872K), 0.0799276 secs] [Times: user=0.16 sys=0.00, real=0.08 secs]

16.502: [GC 16.502: [ParNew: 4683K->256K(4928K), 0.0145173 secs] 7867K->7585K(15872K), 0.0146404 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]

16.518: [GC [1 CMS-initial-mark: 7329K(10944K)] 7841K(15872K), 0.0020021 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

16.520: [CMS-concurrent-mark-start]

16.554: [CMS-concurrent-mark: 0.034/0.034 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]

16.554: [CMS-concurrent-preclean-start]

16.555: [CMS-concurrent-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

16.555: [GC[YG occupancy: 512 K (4928 K)]16.555: [Rescan (parallel) , 0.0010324 secs]16.556: [weak refs processing, 0.0000332 secs]16.556: [scrub string table, 0.0002229 secs] [1 CMS-remark: 7329K(10944K)] 7841K(15872K), 0.0014802 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

16.557: [CMS-concurrent-sweep-start]

16.559: [CMS-concurrent-sweep: 0.002/0.002 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]

16.559: [CMS-concurrent-reset-start]

16.562: [CMS-concurrent-reset: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

17.321: [GC 17.321: [ParNew: 4421K->256K(4928K), 0.0109374 secs] 11599K->11530K(16892K), 0.0110792 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]

17.332: [GC [1 CMS-initial-mark: 11274K(11964K)] 11786K(16892K), 0.0018760 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

17.334: [CMS-concurrent-mark-start]

17.369: [CMS-concurrent-mark: 0.034/0.034 secs] [Times: user=0.03 sys=0.00, real=0.04 secs]

17.369: [CMS-concurrent-preclean-start]

17.369: [CMS-concurrent-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
17.370: [GC[YG occupancy: 512 K (4928 K)]17.370: [Rescan (parallel) , 0.0212876 secs]17.391: [weak refs processing, 0.0000531 secs]17.391: [scrub string table, 0.0001746 secs] [1 CMS-remark: 11274K(11964K)] 11786K(16892K), 0.0217540 secs] [Times: user=0.00 sys=0.00, real=0.02 secs]
17.392: [CMS-concurrent-sweep-start]
17.394: [CMS-concurrent-sweep: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
17.394: [CMS-concurrent-reset-start]
17.397: [CMS-concurrent-reset: 0.003/0.003 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
18.219: [GC 18.219: [ParNew: 4621K->256K(4928K), 0.0112194 secs] 15737K->15723K(23456K), 0.0113527 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]
18.231: [GC [1 CMS-initial-mark: 15467K(18528K)] 15979K(23456K), 0.0020183 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
18.233: [CMS-concurrent-mark-start]
18.267: [CMS-concurrent-mark: 0.034/0.034 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]
18.267: [CMS-concurrent-preclean-start]
18.267: [CMS-concurrent-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
18.268: [GC[YG occupancy: 512 K (4928 K)]18.268: [Rescan (parallel) , 0.0010223 secs]18.269: [weak refs processing, 0.0000105 secs]18.269: [scrub string table, 0.0001337 secs] [1 CMS-remark: 15467K(18528K)] 15979K(23456K), 0.0013529 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
18.269: [CMS-concurrent-sweep-start]
18.271: [CMS-concurrent-sweep: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
18.272: [CMS-concurrent-reset-start]
18.274: [CMS-concurrent-reset: 0.003/0.003 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
18.951: [GC 18.951: [ParNew: 4459K->388K(4928K), 0.0100849 secs] 19927K->19440K(30708K), 0.0102210 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]
18.961: [GC [1 CMS-initial-mark: 19051K(25780K)] 19696K(30708K), 0.0024721 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

18.964: [CMS-concurrent-mark-start]
19.001: [CMS-concurrent-mark: 0.037/0.037 secs] [Times: user=0.03 sys=0.00, real=0.04 secs]
19.001: [CMS-concurrent-preclean-start]
19.001: [CMS-concurrent-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
19.002: [GC[YG occupancy: 644 K (4928 K)]19.002: [Rescan (parallel) , 0.0013274 secs]19.004: [weak refs processing, 0.0000105 secs]19.004: [scrub string table, 0.0001528 secs] [1 CMS-remark: 19051K(25780K)] 19696K(30708K), 0.0018003 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
19.004: [CMS-concurrent-sweep-start]
19.006: [CMS-concurrent-sweep: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
19.006: [CMS-concurrent-reset-start]
19.008: [CMS-concurrent-reset: 0.002/0.002 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]
19.817: [GC 19.817: [ParNew: 4778K->256K(4928K), 0.0110480 secs] 23825K->23782K(36676K), 0.0111769 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]
19.828: [GC [1 CMS-initial-mark: 23526K(31748K)] 24038K(36676K), 0.0018469 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
19.830: [CMS-concurrent-mark-start]
19.871: [CMS-concurrent-mark: 0.040/0.040 secs] [Times: user=0.05 sys=0.00, real=0.04 secs]
19.871: [CMS-concurrent-preclean-start]
19.871: [CMS-concurrent-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
19.871: [GC[YG occupancy: 512 K (4928 K)]19.872: [Rescan (parallel) , 0.0011747 secs]19.873: [weak refs processing, 0.0000105 secs]19.873: [scrub string table, 0.0001459 secs] [1 CMS-remark: 23526K(31748K)] 24038K(36676K), 0.0016613 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
19.873: [CMS-concurrent-sweep-start]
19.875: [CMS-concurrent-sweep: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
19.875: [CMS-concurrent-reset-start]
19.878: [CMS-concurrent-reset: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

20.680: [GC 20.680: [ParNew: 4642K->256K(4928K), 0.0103883 secs] 28168K->28133K(44140K), 0.0105208 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
21.540: [GC 21.540: [ParNew: 4640K->256K(4928K), 0.0103122 secs] 32518K->32486K(44140K), 0.0104394 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
21.551: [GC [1 CMS-initial-mark: 32230K(39212K)] 32742K(44140K), 0.0019692 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
21.553: [CMS-concurrent-mark-start]
21.599: [CMS-concurrent-mark: 0.046/0.046 secs] [Times: user=0.05 sys=0.00, real=0.05 secs]
21.599: [CMS-concurrent-preclean-start]
21.600: [CMS-concurrent-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
21.600: [GC[YG occupancy: 512 K (4928 K)]21.600: [Rescan (parallel) , 0.0011029 secs]21.601: [weak refs processing, 0.0000105 secs]21.601: [scrub string table, 0.0001455 secs] [1 CMS-remark: 32230K(39212K)] 32742K(44140K), 0.0014469 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
21.602: [CMS-concurrent-sweep-start]
21.603: [CMS-concurrent-sweep: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
21.604: [CMS-concurrent-reset-start]
21.606: [CMS-concurrent-reset: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
22.403: [GC 22.403: [ParNew: 4634K->256K(4928K), 0.0104690 secs] 36865K->36838K(58648K), 0.0106071 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]
23.264: [GC 23.264: [ParNew: 4610K->256K(4928K), 0.0101987 secs] 41192K->41190K(58648K), 0.0103308 secs] [Times: user=0.02 sys=0.01, real=0.01 secs]
24.081: [GC 24.081: [ParNew: 4541K->266K(4928K), 0.0099702 secs] 45476K->45297K(58648K), 0.0101019 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
24.092: [GC [1 CMS-initial-mark: 45031K(53720K)] 45553K(58648K), 0.0020106 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]
24.094: [CMS-concurrent-mark-start]
24.149: [CMS-concurrent-mark: 0.055/0.055 secs] [Times: user=0.05 sys=0.00, real=0.06 secs]
24.149: [CMS-concurrent-preclean-start]

24.149: [CMS-concurrent-preclean: 0.001/0.001 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
24.149: [GC[YG occupancy: 780 K (4928 K)]24.149: [Rescan (parallel) , 0.0012285 secs]24.151: [weak refs processing, 0.0000113 secs]24.151: [scrub string table, 0.0001499 secs] [1 CMS-remark: 45031K(53720K)] 45811K(58648K), 0.0016102 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
24.151: [CMS-concurrent-sweep-start]
24.153: [CMS-concurrent-sweep: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
24.153: [CMS-concurrent-reset-start]
24.155: [CMS-concurrent-reset: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
24.943: [GC 24.943: [ParNew: 4644K->256K(4928K), 0.0114743 secs] 49675K->49646K(79980K), 0.0116165 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]
25.805: [GC 25.805: [ParNew: 4636K->256K(4928K), 0.0112311 secs] 54026K->53998K(79980K), 0.0113547 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]
26.666: [GC 26.666: [ParNew: 4636K->256K(4928K), 0.0119491 secs] 58379K->58351K(79980K), 0.0120821 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]
27.528: [GC 27.528: [ParNew: 4636K->256K(4928K), 0.0135043 secs] 62731K->62703K(79980K), 0.0136258 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]
28.342: [GC 28.342: [ParNew: 4457K->258K(4928K), 0.0237626 secs] 66904K->66801K(79980K), 0.0239011 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
28.367: [GC [1 CMS-initial-mark: 66543K(75052K)] 67057K(79980K), 0.0018590 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]
28.369: [CMS-concurrent-mark-start]
28.438: [CMS-concurrent-mark: 0.069/0.069 secs] [Times: user=0.06 sys=0.00, real=0.07 secs]
28.438: [CMS-concurrent-preclean-start]
28.438: [CMS-concurrent-preclean: 0.001/0.001 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
28.439: [GC[YG occupancy: 514 K (4928 K)]28.439: [Rescan (parallel) , 0.0010081 secs]28.440: [weak refs processing, 0.0000101 secs]28.440: [scrub string table, 0.0002188 secs] [1 CMS-remark: 66543K(75052K)] 67057K(79980K), 0.0014818 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

28.440: [CMS-concurrent-sweep-start]
28.442: [CMS-concurrent-sweep: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
28.443: [CMS-concurrent-reset-start]
28.445: [CMS-concurrent-reset: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
29.240: [GC 29.240: [ParNew: 4612K->256K(4928K), 0.0136676 secs] 71155K->71151K(115836K), 0.0137964 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]
30.104: [GC 30.104: [ParNew: 4655K->258K(4928K), 0.0132867 secs] 75551K->75506K(115836K), 0.0134192 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]
30.967: [GC 30.967: [ParNew: 4638K->256K(4928K), 0.0142729 secs] 79886K->79856K(115836K), 0.0144184 secs] [Times: user=0.02 sys=0.02, real=0.01 secs]
31.831: [GC 31.832: [ParNew: 4636K->256K(4928K), 0.0157316 secs] 84236K->84208K(115836K), 0.0158767 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]
32.697: [GC 32.697: [ParNew: 4636K->256K(4928K), 0.0152571 secs] 88589K->88560K(115836K), 0.0153686 secs] [Times: user=0.02 sys=0.01, real=0.02 secs]
33.527: [GC 33.527: [ParNew: 4469K->262K(4928K), 0.0136153 secs] 92774K->92663K(115836K), 0.0137498 secs] [Times: user=0.02 sys=0.02, real=0.01 secs]
34.393: [GC 34.394: [ParNew: 4643K->256K(4928K), 0.0154079 secs] 97044K->97012K(115836K), 0.0155501 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]
35.259: [GC 35.260: [ParNew: 4610K->256K(4928K), 0.0129690 secs] 101367K->101365K(115836K), 0.0131064 secs] [Times: user=0.01 sys=0.01, real=0.01 secs]
36.123: [GC 36.123: [ParNew: 4636K->256K(4928K), 0.0121858 secs] 105745K->105717K(115836K), 0.0123187 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
36.135: [GC [1 CMS-initial-mark: 105461K(110908K)] 105973K(115836K), 0.0018477 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]
36.137: [CMS-concurrent-mark-start]
36.228: [CMS-concurrent-mark: 0.091/0.091 secs] [Times: user=0.08 sys=0.00, real=0.09 secs]
36.228: [CMS-concurrent-preclean-start]
36.229: [CMS-concurrent-preclean: 0.001/0.001 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
36.230: [GC[YG occupancy: 512 K (4928 K)]36.230: [Rescan (parallel) , 0.0013375 secs]36.232: [weak refs processing, 0.0000267 secs]36.232: [scrub string table, 0.0001609 secs]

[1 CMS-remark: 105461K(110908K)] 105973K(115836K), 0.0017111 secs] [Times: user=0.03 sys=0.00, real=0.00 secs]

36.232: [CMS-concurrent-sweep-start]

36.234: [CMS-concurrent-sweep: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

36.235: [CMS-concurrent-reset-start]

36.237: [CMS-concurrent-reset: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

37.087: [GC 37.087: [ParNew: 4636K->256K(4928K), 0.0145922 secs] 110018K->109990K(179712K), 0.0147312 secs] [Times: user=0.02 sys=0.02, real=0.02 secs]

38.048: [GC 38.048: [ParNew: 4636K->256K(4928K), 0.0123280 secs] 114370K->114342K(179712K), 0.0124577 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]

38.872: [GC 38.872: [ParNew: 4454K->260K(4928K), 0.0118191 secs] 118540K->118442K(179712K), 0.0119467 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]

39.734: [GC 39.734: [ParNew: 4640K->256K(4928K), 0.0121461 secs] 122823K->122792K(179712K), 0.0122697 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]

40.596: [GC 40.596: [ParNew: 4636K->256K(4928K), 0.0142567 secs] 127173K->127144K(179712K), 0.0144245 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]

40.960: [Full GC (System) 40.960: [CMS: 126888K->128614K(174784K), 0.2533512 secs] 128939K->128614K(179712K), [CMS Perm : 4505K->4455K(12288K)], 0.2542296 secs] [Times: user=0.23 sys=0.00, real=0.25 secs]

41.716: [Full GC (System) 41.716: [CMS: 128614K->126054K(174784K), 0.1887107 secs] 128651K->126054K(253440K), [CMS Perm : 4455K->4455K(12288K)], 0.1888707 secs] [Times: user=0.19 sys=0.00, real=0.19 secs]

42.406: [Full GC (System) 42.406: [CMS: 126054K->123494K(174784K), 0.1861061 secs] 126089K->123494K(253440K), [CMS Perm : 4455K->4455K(12288K)], 0.1862605 secs] [Times: user=0.19 sys=0.00, real=0.19 secs]

43.099: [Full GC (System) 43.099: [CMS: 123494K->120934K(174784K), 0.1833836 secs] 123494K->120934K(253440K), [CMS Perm : 4455K->4455K(12288K)], 0.1835270 secs] [Times: user=0.19 sys=0.00, real=0.18 secs]

43.784: [Full GC (System) 43.784: [CMS: 120934K->118376K(174784K), 0.2340392 secs] 121242K->118376K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.2341754 secs] [Times: user=0.23 sys=0.00, real=0.23 secs]

44.519: [Full GC (System) 44.519: [CMS: 118376K->115816K(174784K), 0.1744556 secs] 118411K->115816K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1746079 secs] [Times: user=0.17 sys=0.00, real=0.17 secs]
45.194: [Full GC (System) 45.194: [CMS: 115816K->113256K(174784K), 0.1720114 secs] 115816K->113256K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1721581 secs] [Times: user=0.17 sys=0.00, real=0.17 secs]
45.867: [Full GC (System) 45.867: [CMS: 113256K->110695K(174784K), 0.1676588 secs] 113291K->110695K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1678319 secs] [Times: user=0.16 sys=0.00, real=0.17 secs]
46.536: [Full GC (System) 46.536: [CMS: 110695K->108135K(174784K), 0.1656600 secs] 110730K->108135K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1658748 secs] [Times: user=0.17 sys=0.00, real=0.17 secs]
47.203: [Full GC (System) 47.203: [CMS: 108135K->105575K(174784K), 0.1622876 secs] 108135K->105575K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1624351 secs] [Times: user=0.16 sys=0.00, real=0.16 secs]
47.866: [Full GC (System) 47.866: [CMS: 105575K->103015K(174784K), 0.1571238 secs] 105610K->103015K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1572713 secs] [Times: user=0.16 sys=0.00, real=0.16 secs]
48.524: [Full GC (System) 48.524: [CMS: 103015K->100455K(174784K), 0.1555456 secs] 103122K->100455K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1557555 secs] [Times: user=0.16 sys=0.00, real=0.16 secs]
49.180: [Full GC (System) 49.180: [CMS: 100455K->97895K(174784K), 0.1521363 secs] 100455K->97895K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1522931 secs] [Times: user=0.14 sys=0.00, real=0.15 secs]
49.833: [Full GC (System) 49.833: [CMS: 97895K->95335K(174784K), 0.1471216 secs] 97925K->95335K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1472788 secs] [Times: user=0.14 sys=0.00, real=0.15 secs]
50.481: [Full GC (System) 50.481: [CMS: 95335K->92774K(174784K), 0.1480353 secs] 95365K->92774K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1481796 secs] [Times: user=0.16 sys=0.00, real=0.15 secs]

51.130: [Full GC (System) 51.130: [CMS: 92774K->90214K(174784K), 0.1447594 secs] 92774K->90214K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1451159 secs] [Times: user=0.14 sys=0.00, real=0.14 secs]
51.776: [Full GC (System) 51.776: [CMS: 90214K->87654K(174784K), 0.1381069 secs] 90245K->87654K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1382540 secs] [Times: user=0.14 sys=0.00, real=0.14 secs]
52.415: [Full GC (System) 52.415: [CMS: 87654K->85094K(174784K), 0.1376279 secs] 87685K->85094K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1379391 secs] [Times: user=0.14 sys=0.00, real=0.14 secs]
53.053: [Full GC (System) 53.053: [CMS: 85094K->82534K(174784K), 0.1324569 secs] 85094K->82534K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1326157 secs] [Times: user=0.13 sys=0.00, real=0.13 secs]
53.686: [Full GC (System) 53.686: [CMS: 82534K->79974K(174784K), 0.1318094 secs] 82641K->79974K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1319852 secs] [Times: user=0.12 sys=0.00, real=0.13 secs]
54.319: [Full GC (System) 54.319: [CMS: 79974K->77413K(174784K), 0.1284953 secs] 80004K->77413K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1286590 secs] [Times: user=0.13 sys=0.00, real=0.13 secs]
54.949: [Full GC (System) 54.949: [CMS: 77413K->74853K(174784K), 0.1242165 secs] 77414K->74853K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1243773 secs] [Times: user=0.13 sys=0.00, real=0.12 secs]
55.574: [Full GC (System) 55.574: [CMS: 74853K->72293K(174784K), 0.1229826 secs] 74884K->72293K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1231322 secs] [Times: user=0.09 sys=0.00, real=0.12 secs]
56.198: [Full GC (System) 56.198: [CMS: 72293K->69733K(174784K), 0.1145789 secs] 72293K->69733K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1147876 secs] [Times: user=0.11 sys=0.00, real=0.12 secs]
56.813: [Full GC (System) 56.813: [CMS: 69733K->67173K(174784K), 0.1144651 secs] 69764K->67173K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1146118 secs] [Times: user=0.11 sys=0.00, real=0.11 secs]

57.428: [Full GC (System) 57.429: [CMS: 67173K->64613K(174784K), 0.1109059 secs] 67204K->64613K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1110692 secs] [Times: user=0.12 sys=0.00, real=0.11 secs]

58.040: [Full GC (System) 58.041: [CMS: 64613K->62052K(174784K), 0.1062960 secs] 64613K->62052K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1064528 secs] [Times: user=0.09 sys=0.00, real=0.11 secs]

58.647: [Full GC (System) 58.648: [CMS: 62052K->59492K(174784K), 0.1049941 secs] 62364K->59492K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.1051424 secs] [Times: user=0.09 sys=0.00, real=0.11 secs]

59.253: [Full GC (System) 59.254: [CMS: 59492K->56932K(174784K), 0.0989616 secs] 59493K->56932K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0991091 secs] [Times: user=0.11 sys=0.00, real=0.10 secs]

59.853: [Full GC (System) 59.854: [CMS: 56932K->54372K(174784K), 0.0965389 secs] 56967K->54372K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0966994 secs] [Times: user=0.09 sys=0.00, real=0.10 secs]

60.451: [Full GC (System) 60.451: [CMS: 54372K->51812K(174784K), 0.0971508 secs] 54407K->51812K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0972991 secs] [Times: user=0.09 sys=0.00, real=0.10 secs]

61.049: [Full GC (System) 61.049: [CMS: 51812K->49252K(174784K), 0.0905587 secs] 51812K->49252K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0907240 secs] [Times: user=0.09 sys=0.00, real=0.09 secs]

61.640: [Full GC (System) 61.640: [CMS: 49252K->46692K(174784K), 0.0871765 secs] 49287K->46692K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0873224 secs] [Times: user=0.09 sys=0.00, real=0.09 secs]

62.228: [Full GC (System) 62.228: [CMS: 46692K->44132K(174784K), 0.0831120 secs] 46692K->44132K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0832721 secs] [Times: user=0.08 sys=0.00, real=0.08 secs]

62.812: [Full GC (System) 62.812: [CMS: 44132K->41571K(174784K), 0.0833073 secs] 44166K->41571K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0834573 secs] [Times: user=0.09 sys=0.00, real=0.08 secs]

63.396: [Full GC (System) 63.396: [CMS: 41571K->39011K(174784K), 0.0784158 secs] 41687K->39011K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0785702 secs] [Times: user=0.08 sys=0.00, real=0.08 secs]
63.976: [Full GC (System) 63.976: [CMS: 39011K->36451K(174784K), 0.0748720 secs] 39011K->36451K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0750305 secs] [Times: user=0.08 sys=0.00, real=0.08 secs]
64.552: [Full GC (System) 64.552: [CMS: 36451K->33891K(174784K), 0.0699384 secs] 36486K->33891K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0700887 secs] [Times: user=0.08 sys=0.00, real=0.07 secs]
65.123: [Full GC (System) 65.123: [CMS: 33891K->31331K(174784K), 0.0682718 secs] 33891K->31331K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0684343 secs] [Times: user=0.06 sys=0.00, real=0.07 secs]
65.692: [Full GC (System) 65.692: [CMS: 31331K->28771K(174784K), 0.0705871 secs] 31366K->28771K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0707998 secs] [Times: user=0.08 sys=0.00, real=0.07 secs]
66.264: [Full GC (System) 66.264: [CMS: 28771K->26210K(174784K), 0.0619378 secs] 28771K->26210K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0620967 secs] [Times: user=0.06 sys=0.00, real=0.06 secs]
66.828: [Full GC (System) 66.828: [CMS: 26210K->23650K(174784K), 0.0575865 secs] 26245K->23650K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0577972 secs] [Times: user=0.06 sys=0.00, real=0.06 secs]
67.386: [Full GC (System) 67.386: [CMS: 23650K->21090K(174784K), 0.0565370 secs] 23685K->21090K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0566934 secs] [Times: user=0.05 sys=0.00, real=0.06 secs]
67.943: [Full GC (System) 67.943: [CMS: 21090K->18530K(174784K), 0.0546541 secs] 21090K->18530K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0548615 secs] [Times: user=0.06 sys=0.00, real=0.06 secs]
68.498: [Full GC (System) 68.498: [CMS: 18530K->15970K(174784K), 0.0503404 secs] 18636K->15970K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0504903 secs] [Times: user=0.05 sys=0.00, real=0.05 secs]

69.049: [Full GC (System) 69.049: [CMS: 15970K->13410K(174784K), 0.0458351 secs] 15970K->13410K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0459947 secs] [Times: user=0.05 sys=0.00, real=0.05 secs]
69.595: [Full GC (System) 69.595: [CMS: 13410K->10850K(174784K), 0.0446722 secs] 13445K->10850K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0448217 secs] [Times: user=0.05 sys=0.00, real=0.05 secs]
70.141: [Full GC (System) 70.141: [CMS: 10850K->8289K(174784K), 0.0394820 secs] 10850K->8289K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0396392 secs] [Times: user=0.05 sys=0.00, real=0.04 secs]
70.681: [Full GC (System) 70.681: [CMS: 8289K->5729K(174784K), 0.0367640 secs] 8324K->5729K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0369078 secs] [Times: user=0.03 sys=0.00, real=0.04 secs]
71.218: [Full GC (System) 71.218: [CMS: 5729K->3169K(174784K), 0.0330670 secs] 5729K->3169K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0332210 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]
71.752: [Full GC (System) 71.752: [CMS: 3169K->865K(174784K), 0.0298563 secs] 3204K->865K(253440K), [CMS Perm : 4457K->4457K(12288K)], 0.0300021 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]
Heap
 par new generation   total 78656K, used 2038K [0x039b0000, 0x08f00000, 0x08f00000)
  eden space 69952K,   2% used [0x039b0000, 0x03bad878, 0x07e00000)
  from space 8704K,   0% used [0x07e00000, 0x07e00000, 0x08680000)
  to   space 8704K,   0% used [0x08680000, 0x08680000, 0x08f00000)
 concurrent mark-sweep generation total 174784K, used 865K [0x08f00000, 0x139b0000, 0x139b0000)
 concurrent-mark-sweep perm gen total 12288K, used 4470K [0x139b0000, 0x145b0000, 0x179b0000)

Appendix 2:

Heap

garbage-first heap total 16384K, used 0K [0x03d00000, 0x04d00000, 0x13d00000)

region size 1024K, 1 young (1024K), 0 survivors (0K)

compacting perm gen total 12288K, used 1471K [0x13d00000, 0x14900000, 0x17d00000)

the space 12288K, 11% used [0x13d00000, 0x13e6ff90, 0x13e70000, 0x14900000)

No shared spaces configured.

Appendix 3:

6.445: [GC [PSYoungGen: 4160K->624K(4800K)] 4160K->750K(15744K), 0.0081444 secs] [Times: user=0.00 sys=0.02, real=0.01 secs]

15.531: [GC [PSYoungGen: 4750K->624K(4800K)] 4876K->2918K(15744K), 0.0084507 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]

16.290: [GC [PSYoungGen: 4536K->616K(4800K)] 6831K->6750K(15744K), 0.0080026 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]

16.298: [Full GC [PSYoungGen: 616K->0K(4800K)] [ParOldGen: 6134K->6652K(15488K)] 6750K->6652K(20288K) [PSPermGen: 4444K->4440K(12288K)], 0.1000357 secs] [Times: user=0.09 sys=0.00, real=0.10 secs]

17.163: [GC [PSYoungGen: 4128K->544K(4800K)] 10780K->10524K(20288K), 0.0062914 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]

17.969: [GC [PSYoungGen: 4687K->544K(4800K)] 14668K->14620K(20288K), 0.0069507 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]

17.976: [Full GC [PSYoungGen: 544K->0K(4800K)] [ParOldGen: 14076K->14423K(24064K)] 14620K->14423K(28864K) [PSPermGen: 4464K->4443K(12288K)], 0.0581667 secs] [Times: user=0.09 sys=0.00, real=0.06 secs]

18.834: [GC [PSYoungGen: 4136K->2560K(6400K)] 18559K->18519K(30464K), 0.0068287 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]

19.491: [GC [PSYoungGen: 5893K->3328K(6976K)] 21852K->21847K(31040K), 0.0082931 secs] [Times: user=0.02 sys=0.01, real=0.01 secs]

20.149: [GC [PSYoungGen: 6684K->4096K(6976K)] 25204K->25175K(31040K), 0.0088903 secs] [Times: user=0.02 sys=0.02, real=0.01 secs]
20.158: [Full GC [PSYoungGen: 4096K->1280K(6976K)] [ParOldGen: 21079K->23894K(32448K)] 25175K->25174K(39424K) [PSPermGen: 4443K->4443K(12288K)], 0.0654938 secs] [Times: user=0.09 sys=0.00, real=0.07 secs]
20.724: [GC [PSYoungGen: 3868K->3840K(7744K)] 27763K->27734K(40192K), 0.0144593 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]
21.238: [GC [PSYoungGen: 6402K->5120K(7360K)] 30296K->30294K(39808K), 0.0144306 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]
21.608: [GC [PSYoungGen: 7080K->6160K(8192K)] 32254K->32102K(40640K), 0.0682820 secs] [Times: user=0.06 sys=0.00, real=0.07 secs]
22.027: [GC [PSYoungGen: 7956K->7200K(8064K)] 33898K->33910K(40512K), 0.0165854 secs] [Times: user=0.02 sys=0.02, real=0.02 secs]
22.194: [GC [PSYoungGen: 7970K->7968K(9216K)] 34680K->34678K(41664K), 0.0165971 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]
22.361: [GC [PSYoungGen: 8738K->8736K(9728K)] 35448K->35446K(42176K), 0.0173398 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]
22.378: [GC [PSYoungGen: 8736K->8736K(10496K)] 35446K->35446K(42944K), 0.0177102 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]
22.396: [Full GC [PSYoungGen: 8736K->3072K(10496K)] [ParOldGen: 26710K->32348K(39872K)] 35446K->35420K(50368K) [PSPermGen: 4452K->4452K(12288K)], 0.0767537 secs] [Times: user=0.11 sys=0.01, real=0.08 secs]
24.596: [Full GC [PSYoungGen: 3136K->3072K(10496K)] [ParOldGen: 43356K->43357K(50752K)] 46493K->46429K(61248K) [PSPermGen: 4452K->4452K(12288K)], 0.0594443 secs] [Times: user=0.08 sys=0.00, real=0.06 secs]
26.596: [Full GC [PSYoungGen: 3135K->2816K(10496K)] [ParOldGen: 53341K->53600K(60864K)] 56476K->56416K(71360K) [PSPermGen: 4452K->4452K(12288K)], 0.0717528 secs] [Times: user=0.11 sys=0.00, real=0.07 secs]
26.687: [GC [PSYoungGen: 2879K->2080K(11456K)] 56736K->56704K(72320K), 0.0595662 secs] [Times: user=0.05 sys=0.00, real=0.06 secs]

29.593: [Full GC [PSYoungGen: 2144K->1792K(11456K)] [ParOldGen: 69217K->69475K(76480K)] 71361K->71267K(87936K) [PSPermGen: 4452K->4452K(12288K)], 0.0865943 secs] [Times: user=0.14 sys=0.00, real=0.09 secs]

31.593: [Full GC [PSYoungGen: 1856K->1792K(11456K)] [ParOldGen: 79460K->79460K(86464K)] 81316K->81252K(97920K) [PSPermGen: 4452K->4452K(12288K)], 0.0898561 secs] [Times: user=0.16 sys=0.02, real=0.09 secs]

31.692: [GC [PSYoungGen: 1856K->32K(11712K)] 81572K->81540K(98176K), 0.1217541 secs] [Times: user=0.17 sys=0.00, real=0.12 secs]

33.594: [GC [PSYoungGen: 95K->16K(11584K)] 90820K->90741K(102528K), 0.0488667 secs] [Times: user=0.08 sys=0.00, real=0.05 secs]

33.643: [Full GC [PSYoungGen: 16K->0K(11584K)] [ParOldGen: 90725K->90723K(97664K)] 90741K->90723K(109248K) [PSPermGen: 4455K->4454K(12288K)], 0.0974494 secs] [Times: user=0.19 sys=0.00, real=0.10 secs]

36.592: [Full GC [PSYoungGen: 64K->0K(11584K)] [ParOldGen: 105315K->105315K(112256K)] 105379K->105315K(123840K) [PSPermGen: 4454K->4454K(12288K)], 0.1093677 secs] [Times: user=0.19 sys=0.00, real=0.11 secs]

36.708: [GC [PSYoungGen: 64K->16K(128K)] 105635K->105587K(112384K), 0.0043141 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]

38.592: [GC [PSYoungGen: 80K->16K(10816K)] 115124K->115060K(125952K), 0.0029782 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

38.596: [Full GC [PSYoungGen: 16K->0K(10816K)] [ParOldGen: 115044K->115044K(121856K)] 115060K->115044K(132672K) [PSPermGen: 4456K->4456K(12288K)], 0.1168768 secs] [Times: user=0.20 sys=0.00, real=0.12 secs]

40.595: [GC [PSYoungGen: 64K->16K(128K)] 124837K->124789K(125184K), 0.0033214 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

40.599: [Full GC [PSYoungGen: 16K->0K(128K)] [ParOldGen: 124773K->124772K(131456K)] 124789K->124772K(131584K) [PSPermGen: 4456K->4456K(12288K)], 0.2052421 secs] [Times: user=0.31 sys=0.00, real=0.21 secs]

41.598: [GC [PSYoungGen: 64K->16K(10048K)] 128933K->128885K(141504K), 0.0687990 secs] [Times: user=0.08 sys=0.00, real=0.07 secs]

41.667: [Full GC [PSYoungGen: 16K->0K(10048K)] [ParOldGen: 128869K->128846K(135424K)] 128885K->128846K(145472K) [PSPermGen: 4456K->4428K(12288K)], 0.2035699 secs] [Times: user=0.30 sys=0.00, real=0.20 secs]
41.871: [GC [PSYoungGen: 8K->32K(128K)] 128854K->128878K(135552K), 0.0025122 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
41.874: [Full GC (System) [PSYoungGen: 32K->0K(128K)] [ParOldGen: 128846K->128593K(135424K)] 128878K->128593K(135552K) [PSPermGen: 4428K->4428K(12288K)], 0.1800886 secs] [Times: user=0.28 sys=0.00, real=0.18 secs]
42.058: [GC [PSYoungGen: 64K->48K(9152K)] 128657K->128641K(144576K), 0.0028578 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
42.064: [GC [PSYoungGen: 112K->64K(128K)] 128705K->128657K(135552K), 0.0036650 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
42.070: [GC [PSYoungGen: 128K->64K(8256K)] 128721K->128689K(143680K), 0.0037322 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
42.077: [GC [PSYoungGen: 128K->32K(128K)] 128753K->128709K(135552K), 0.0032014 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
42.083: [GC [PSYoungGen: 96K->64K(7360K)] 128773K->128765K(142784K), 0.0026034 secs] [Times: user=0.03 sys=0.00, real=0.00 secs]
42.574: [GC [PSYoungGen: 98K->16K(128K)] 128800K->128737K(135552K), 0.0023311 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
42.576: [Full GC (System) [PSYoungGen: 16K->0K(128K)] [ParOldGen: 128721K->126115K(135424K)] 128737K->126115K(135552K) [PSPermGen: 4449K->4449K(12288K)], 0.2131151 secs] [Times: user=0.34 sys=0.00, real=0.21 secs]
43.290: [GC [PSYoungGen: 26K->0K(6592K)] 126141K->126115K(142016K), 0.0017249 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
43.292: [Full GC (System) [PSYoungGen: 0K->0K(6592K)] [ParOldGen: 126115K->123555K(135424K)] 126115K->123555K(142016K) [PSPermGen: 4449K->4449K(12288K)], 0.1159943 secs] [Times: user=0.20 sys=0.00, real=0.12 secs]
43.909: [GC [PSYoungGen: 26K->0K(128K)] 123581K->123555K(135552K), 0.0035827 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

43.913: [Full GC (System) [PSYoungGen: 0K->0K(128K)] [ParOldGen: 123555K->120995K(135424K)] 123555K->120995K(135552K) [PSPermGen: 4449K->4449K(12288K)], 0.1333353 secs] [Times: user=0.17 sys=0.00, real=0.13 secs]
44.547: [GC [PSYoungGen: 0K->0K(5760K)] 120995K->120995K(141184K), 0.0016682 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]
44.549: [Full GC (System) [PSYoungGen: 0K->0K(5760K)] [ParOldGen: 120995K->118434K(135424K)] 120995K->118434K(141184K) [PSPermGen: 4449K->4449K(12288K)], 0.1118147 secs] [Times: user=0.20 sys=0.00, real=0.11 secs]
45.161: [GC [PSYoungGen: 26K->0K(128K)] 118460K->118434K(135552K), 0.0017837 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
45.163: [Full GC (System) [PSYoungGen: 0K->0K(128K)] [ParOldGen: 118434K->115874K(135424K)] 118434K->115874K(135552K) [PSPermGen: 4449K->4449K(12288K)], 0.1053855 secs] [Times: user=0.20 sys=0.00, real=0.11 secs]
45.769: [GC [PSYoungGen: 25K->0K(5120K)] 115900K->115874K(140544K), 0.0016183 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
45.771: [Full GC (System) [PSYoungGen: 0K->0K(5120K)] [ParOldGen: 115874K->113314K(135424K)] 115874K->113314K(140544K) [PSPermGen: 4449K->4449K(12288K)], 0.1045747 secs] [Times: user=0.19 sys=0.00, real=0.10 secs]
46.376: [GC [PSYoungGen: 0K->0K(128K)] 113314K->113314K(135552K), 0.0014935 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
46.378: [Full GC (System) [PSYoungGen: 0K->0K(128K)] [ParOldGen: 113314K->110754K(135424K)] 113314K->110754K(135552K) [PSPermGen: 4449K->4449K(12288K)], 0.1025832 secs] [Times: user=0.17 sys=0.00, real=0.10 secs]
46.597: [GC [PSYoungGen: 64K->32K(4416K)] 110818K->110786K(139840K), 0.0027748 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
46.607: [GC [PSYoungGen: 96K->48K(128K)] 110850K->110822K(135552K), 0.0026743 secs] [Times: user=0.03 sys=0.00, real=0.00 secs]
46.981: [GC [PSYoungGen: 99K->0K(3776K)] 110874K->110782K(139200K), 0.0015150 secs] [Times: user=0.03 sys=0.00, real=0.00 secs]

46.983: [Full GC (System) [PSYoungGen: 0K->0K(3776K)] [ParOldGen: 110782K->108214K(135424K)] 110782K->108214K(139200K) [PSPermGen: 4455K->4455K(12288K)], 0.1777299 secs] [Times: user=0.26 sys=0.00, real=0.18 secs]
47.661: [GC [PSYoungGen: 26K->0K(128K)] 108240K->108214K(135552K), 0.0656211 secs] [Times: user=0.06 sys=0.00, real=0.07 secs]
47.727: [Full GC (System) [PSYoungGen: 0K->0K(128K)] [ParOldGen: 108214K->105654K(135424K)] 108214K->105654K(135552K) [PSPermGen: 4455K->4455K(12288K)], 0.1028701 secs] [Times: user=0.17 sys=0.00, real=0.10 secs]
48.330: [GC [PSYoungGen: 0K->0K(3328K)] 105654K->105654K(138752K), 0.0016597 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
48.332: [Full GC (System) [PSYoungGen: 0K->0K(3328K)] [ParOldGen: 105654K->103094K(135424K)] 105654K->103094K(138752K) [PSPermGen: 4455K->4455K(12288K)], 0.0999219 secs] [Times: user=0.17 sys=0.00, real=0.10 secs]
48.933: [GC [PSYoungGen: 28K->16K(128K)] 103122K->103110K(135552K), 0.0016374 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
48.935: [Full GC (System) [PSYoungGen: 16K->0K(128K)] [ParOldGen: 103094K->100534K(135424K)] 103110K->100534K(135552K) [PSPermGen: 4455K->4455K(12288K)], 0.0963825 secs] [Times: user=0.17 sys=0.00, real=0.10 secs]
49.532: [GC [PSYoungGen: 0K->0K(2816K)] 100534K->100534K(138240K), 0.0015025 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
49.534: [Full GC (System) [PSYoungGen: 0K->0K(2816K)] [ParOldGen: 100534K->97974K(135424K)] 100534K->97974K(138240K) [PSPermGen: 4455K->4455K(12288K)], 0.0943326 secs] [Times: user=0.17 sys=0.00, real=0.09 secs]
50.129: [GC [PSYoungGen: 26K->0K(128K)] 98000K->97974K(135552K), 0.0015134 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
50.131: [Full GC (System) [PSYoungGen: 0K->0K(128K)] [ParOldGen: 97974K->95414K(135424K)] 97974K->95414K(135552K) [PSPermGen: 4455K->4455K(12288K)], 0.0967362 secs] [Times: user=0.19 sys=0.00, real=0.10 secs]
50.728: [GC [PSYoungGen: 26K->0K(2496K)] 95440K->95414K(137920K), 0.0016123 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

50.730: [Full GC (System) [PSYoungGen: 0K->0K(2496K)] [ParOldGen: 95414K->92853K(135424K)] 95414K->92853K(137920K) [PSPermGen: 4455K->4455K(12288K)], 0.0917568 secs] [Times: user=0.17 sys=0.00, real=0.09 secs]

51.322: [GC [PSYoungGen: 0K->0K(128K)] 92854K->92853K(135552K), 0.0019522 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

51.324: [Full GC (System) [PSYoungGen: 0K->0K(128K)] [ParOldGen: 92853K->90293K(135424K)] 92853K->90293K(135552K) [PSPermGen: 4455K->4455K(12288K)], 0.0880072 secs] [Times: user=0.16 sys=0.00, real=0.09 secs]

51.600: [GC [PSYoungGen: 64K->32K(2112K)] 90357K->90325K(137536K), 0.0019607 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

51.912: [GC [PSYoungGen: 68K->0K(128K)] 90362K->90297K(135552K), 0.0016540 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

51.914: [Full GC (System) [PSYoungGen: 0K->0K(128K)] [ParOldGen: 90297K->87733K(135424K)] 90297K->87733K(135552K) [PSPermGen: 4455K->4455K(12288K)], 0.0996650 secs] [Times: user=0.16 sys=0.00, real=0.10 secs]

52.514: [GC [PSYoungGen: 0K->0K(1856K)] 87733K->87733K(137280K), 0.0013043 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

52.516: [Full GC (System) [PSYoungGen: 0K->0K(1856K)] [ParOldGen: 87733K->85173K(135424K)] 87733K->85173K(137280K) [PSPermGen: 4455K->4455K(12288K)], 0.0846672 secs] [Times: user=0.17 sys=0.00, real=0.09 secs]

53.101: [GC [PSYoungGen: 26K->0K(128K)] 85199K->85173K(135552K), 0.0022553 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

53.103: [Full GC (System) [PSYoungGen: 0K->0K(128K)] [ParOldGen: 85173K->82613K(135424K)] 85173K->82613K(135552K) [PSPermGen: 4455K->4455K(12288K)], 0.0842652 secs] [Times: user=0.14 sys=0.00, real=0.08 secs]

53.690: [GC [PSYoungGen: 26K->0K(1536K)] 82639K->82613K(136960K), 0.0430489 secs] [Times: user=0.03 sys=0.00, real=0.04 secs]

53.733: [Full GC (System) [PSYoungGen: 0K->0K(1536K)] [ParOldGen: 82613K->80053K(135424K)] 82613K->80053K(136960K) [PSPermGen: 4455K->4455K(12288K)], 0.0865874 secs] [Times: user=0.16 sys=0.00, real=0.09 secs]

54.321: [GC [PSYoungGen: 0K->0K(128K)] 80053K->80053K(135552K), 0.0014879 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
54.323: [Full GC (System) [PSYoungGen: 0K->0K(128K)] [ParOldGen: 80053K->77492K(135424K)] 80053K->77492K(135552K) [PSPermGen: 4455K->4455K(12288K)], 0.0804025 secs] [Times: user=0.14 sys=0.00, real=0.08 secs]
54.904: [GC [PSYoungGen: 25K->0K(1408K)] 77518K->77492K(136832K), 0.0011710 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
54.906: [Full GC (System) [PSYoungGen: 0K->0K(1408K)] [ParOldGen: 77492K->74932K(135424K)] 77492K->74932K(136832K) [PSPermGen: 4455K->4455K(12288K)], 0.0796108 secs] [Times: user=0.16 sys=0.00, real=0.08 secs]
55.486: [GC [PSYoungGen: 0K->0K(128K)] 74932K->74932K(135552K), 0.0011994 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
55.488: [Full GC (System) [PSYoungGen: 0K->0K(128K)] [ParOldGen: 74932K->72372K(135424K)] 74932K->72372K(135552K) [PSPermGen: 4455K->4455K(12288K)], 0.0786261 secs] [Times: user=0.14 sys=0.00, real=0.08 secs]
56.067: [GC [PSYoungGen: 26K->0K(1088K)] 72398K->72372K(136512K), 0.0015093 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
56.069: [Full GC (System) [PSYoungGen: 0K->0K(1088K)] [ParOldGen: 72372K->69812K(135424K)] 72372K->69812K(136512K) [PSPermGen: 4455K->4455K(12288K)], 0.0775123 secs] [Times: user=0.13 sys=0.00, real=0.08 secs]
56.601: [GC [PSYoungGen: 64K->32K(128K)] 69876K->69844K(135552K), 0.0019166 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
56.648: [GC [PSYoungGen: 43K->0K(1024K)] 69855K->69816K(136448K), 0.0019380 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
56.650: [Full GC (System) [PSYoungGen: 0K->0K(1024K)] [ParOldGen: 69816K->67252K(135424K)] 69816K->67252K(136448K) [PSPermGen: 4455K->4455K(12288K)], 0.0764612 secs] [Times: user=0.13 sys=0.00, real=0.08 secs]
57.228: [GC [PSYoungGen: 26K->0K(128K)] 67278K->67252K(135552K), 0.0014619 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

57.230: [Full GC (System) [PSYoungGen: 0K->0K(128K)] [ParOldGen: 67252K->64692K(135424K)] 67252K->64692K(135552K) [PSPermGen: 4455K->4455K(12288K)], 0.0730722 secs] [Times: user=0.16 sys=0.00, real=0.07 secs]
57.815: [GC [PSYoungGen: 26K->0K(832K)] 64718K->64692K(136256K), 0.0012966 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
57.817: [Full GC (System) [PSYoungGen: 0K->0K(832K)] [ParOldGen: 64692K->62132K(135424K)] 64692K->62132K(136256K) [PSPermGen: 4455K->4455K(12288K)], 0.0664784 secs] [Times: user=0.11 sys=0.00, real=0.07 secs]
58.384: [GC [PSYoungGen: 0K->0K(896K)] 62132K->62132K(136320K), 0.0011074 secs] [Times: user=0.03 sys=0.00, real=0.00 secs]
58.386: [Full GC (System) [PSYoungGen: 0K->0K(896K)] [ParOldGen: 62132K->59571K(135424K)] 62132K->59571K(136320K) [PSPermGen: 4455K->4455K(12288K)], 0.0692763 secs] [Times: user=0.11 sys=0.00, real=0.07 secs]
58.988: [GC [PSYoungGen: 26K->0K(960K)] 59598K->59571K(136384K), 0.0012131 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
58.990: [Full GC (System) [PSYoungGen: 0K->0K(960K)] [ParOldGen: 59571K->57011K(135424K)] 59571K->57011K(136384K) [PSPermGen: 4455K->4455K(12288K)], 0.0648281 secs] [Times: user=0.14 sys=0.00, real=0.07 secs]
59.555: [GC [PSYoungGen: 0K->0K(896K)] 57011K->57011K(136320K), 0.0009879 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
59.557: [Full GC (System) [PSYoungGen: 0K->0K(896K)] [ParOldGen: 57011K->54451K(135424K)] 57011K->54451K(136320K) [PSPermGen: 4455K->4455K(12288K)], 0.0653824 secs] [Times: user=0.11 sys=0.00, real=0.06 secs]
60.122: [GC [PSYoungGen: 27K->0K(1024K)] 54478K->54451K(136448K), 0.0014048 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
60.124: [Full GC (System) [PSYoungGen: 0K->0K(1024K)] [ParOldGen: 54451K->51891K(135424K)] 54451K->51891K(136448K) [PSPermGen: 4455K->4455K(12288K)], 0.0648119 secs] [Times: user=0.11 sys=0.00, real=0.07 secs]
60.689: [GC [PSYoungGen: 30K->0K(1024K)] 51922K->51891K(136448K), 0.0021204 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

60.692: [Full GC (System) [PSYoungGen: 0K->0K(1024K)] [ParOldGen: 51891K->49331K(135424K)] 51891K->49331K(136448K) [PSPermGen: 4455K->4455K(12288K)], 0.0863260 secs] [Times: user=0.11 sys=0.00, real=0.09 secs]
61.281: [GC [PSYoungGen: 0K->0K(1152K)] 49331K->49331K(136576K), 0.0009161 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
61.282: [Full GC (System) [PSYoungGen: 0K->0K(1152K)] [ParOldGen: 49331K->46771K(135424K)] 49331K->46771K(136576K) [PSPermGen: 4455K->4455K(12288K)], 0.0595217 secs] [Times: user=0.11 sys=0.00, real=0.06 secs]
61.843: [GC [PSYoungGen: 115K->16K(1088K)] 46886K->46787K(136512K), 0.0009210 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
61.844: [Full GC (System) [PSYoungGen: 16K->0K(1088K)] [ParOldGen: 46771K->44210K(135424K)] 46787K->44210K(136512K) [PSPermGen: 4455K->4455K(12288K)], 0.0603321 secs] [Times: user=0.09 sys=0.00, real=0.06 secs]
62.404: [GC [PSYoungGen: 0K->0K(1216K)] 44211K->44210K(136640K), 0.0010353 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
62.406: [Full GC (System) [PSYoungGen: 0K->0K(1216K)] [ParOldGen: 44210K->41650K(135424K)] 44210K->41650K(136640K) [PSPermGen: 4455K->4455K(12288K)], 0.0688010 secs] [Times: user=0.11 sys=0.00, real=0.07 secs]
62.976: [GC [PSYoungGen: 29K->0K(1216K)] 41680K->41650K(136640K), 0.0018971 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
62.978: [Full GC (System) [PSYoungGen: 0K->0K(1216K)] [ParOldGen: 41650K->39090K(135424K)] 41650K->39090K(136640K) [PSPermGen: 4455K->4455K(12288K)], 0.0674821 secs] [Times: user=0.09 sys=0.00, real=0.07 secs]
63.548: [GC [PSYoungGen: 0K->0K(1280K)] 39090K->39090K(136704K), 0.0010272 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
63.549: [Full GC (System) [PSYoungGen: 0K->0K(1280K)] [ParOldGen: 39090K->36530K(135424K)] 39090K->36530K(136704K) [PSPermGen: 4455K->4455K(12288K)], 0.0553255 secs] [Times: user=0.11 sys=0.00, real=0.06 secs]
64.105: [GC [PSYoungGen: 34K->0K(1280K)] 36564K->36530K(136704K), 0.0010061 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

64.106: [Full GC (System) [PSYoungGen: 0K->0K(1280K)] [ParOldGen: 36530K->33970K(135424K)] 36530K->33970K(136704K) [PSPermGen: 4455K->4455K(12288K)], 0.0587595 secs] [Times: user=0.11 sys=0.00, real=0.06 secs]
64.666: [GC [PSYoungGen: 34K->0K(1344K)] 34004K->33970K(136768K), 0.0626267 secs] [Times: user=0.06 sys=0.00, real=0.06 secs]
64.729: [Full GC (System) [PSYoungGen: 0K->0K(1344K)] [ParOldGen: 33970K->31410K(135424K)] 33970K->31410K(136768K) [PSPermGen: 4455K->4455K(12288K)], 0.0546355 secs] [Times: user=0.09 sys=0.00, real=0.05 secs]
65.285: [GC [PSYoungGen: 0K->0K(1344K)] 31410K->31410K(136768K), 0.0010685 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
65.286: [Full GC (System) [PSYoungGen: 0K->0K(1344K)] [ParOldGen: 31410K->28850K(135424K)] 31410K->28850K(136768K) [PSPermGen: 4455K->4455K(12288K)], 0.0475138 secs] [Times: user=0.08 sys=0.00, real=0.05 secs]
65.834: [GC [PSYoungGen: 37K->0K(1408K)] 28887K->28850K(136832K), 0.0009117 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
65.835: [Full GC (System) [PSYoungGen: 0K->0K(1408K)] [ParOldGen: 28850K->26289K(135424K)] 28850K->26289K(136832K) [PSPermGen: 4455K->4455K(12288K)], 0.0445595 secs] [Times: user=0.08 sys=0.00, real=0.04 secs]
66.380: [GC [PSYoungGen: 0K->0K(1344K)] 26289K->26289K(136768K), 0.0012229 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
66.381: [Full GC (System) [PSYoungGen: 0K->0K(1344K)] [ParOldGen: 26289K->23729K(135424K)] 26289K->23729K(136768K) [PSPermGen: 4455K->4455K(12288K)], 0.0441166 secs] [Times: user=0.08 sys=0.00, real=0.04 secs]
66.926: [GC [PSYoungGen: 134K->32K(1408K)] 23864K->23761K(136832K), 0.0009174 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
66.927: [Full GC (System) [PSYoungGen: 32K->0K(1408K)] [ParOldGen: 23729K->21154K(135424K)] 23761K->21154K(136832K) [PSPermGen: 4458K->4458K(12288K)], 0.0648913 secs] [Times: user=0.08 sys=0.00, real=0.07 secs]
67.493: [GC [PSYoungGen: 0K->0K(1408K)] 21154K->21154K(136832K), 0.0009745 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

67.494: [Full GC (System) [PSYoungGen: 0K->0K(1408K)] [ParOldGen: 21154K->18594K(135424K)] 21154K->18594K(136832K) [PSPermGen: 4458K->4458K(12288K)], 0.0414865 secs] [Times: user=0.08 sys=0.00, real=0.04 secs]
68.037: [GC [PSYoungGen: 43K->0K(1472K)] 18637K->18594K(136896K), 0.0008853 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
68.038: [Full GC (System) [PSYoungGen: 0K->0K(1472K)] [ParOldGen: 18594K->16034K(135424K)] 18594K->16034K(136896K) [PSPermGen: 4458K->4458K(12288K)], 0.0382227 secs] [Times: user=0.08 sys=0.00, real=0.04 secs]
68.577: [GC [PSYoungGen: 0K->0K(1472K)] 16034K->16034K(136896K), 0.0008104 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
68.578: [Full GC (System) [PSYoungGen: 0K->0K(1472K)] [ParOldGen: 16034K->13474K(135424K)] 16034K->13474K(136896K) [PSPermGen: 4458K->4458K(12288K)], 0.0391976 secs] [Times: user=0.05 sys=0.00, real=0.04 secs]
69.118: [GC [PSYoungGen: 24K->0K(1472K)] 13498K->13474K(136896K), 0.0008708 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]
69.119: [Full GC (System) [PSYoungGen: 0K->0K(1472K)] [ParOldGen: 13474K->10914K(135424K)] 13474K->10914K(136896K) [PSPermGen: 4458K->4458K(12288K)], 0.0350622 secs] [Times: user=0.05 sys=0.00, real=0.03 secs]
69.655: [GC [PSYoungGen: 0K->0K(1472K)] 10914K->10914K(136896K), 0.0013683 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
69.656: [Full GC (System) [PSYoungGen: 0K->0K(1472K)] [ParOldGen: 10914K->8353K(135424K)] 10914K->8353K(136896K) [PSPermGen: 4458K->4458K(12288K)], 0.0340046 secs] [Times: user=0.05 sys=0.00, real=0.03 secs]
70.193: [GC [PSYoungGen: 24K->0K(1536K)] 8378K->8353K(136960K), 0.0006394 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
70.194: [Full GC (System) [PSYoungGen: 0K->0K(1536K)] [ParOldGen: 8353K->5793K(135424K)] 8353K->5793K(136960K) [PSPermGen: 4458K->4458K(12288K)], 0.0327842 secs] [Times: user=0.06 sys=0.00, real=0.03 secs]
70.727: [GC [PSYoungGen: 25K->0K(1472K)] 5819K->5793K(136896K), 0.0011499 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

70.728: [Full GC (System) [PSYoungGen: 0K->0K(1472K)] [ParOldGen: 5793K->3233K(135424K)] 5793K->3233K(136896K) [PSPermGen: 4458K->4458K(12288K)], 0.0320528 secs] [Times: user=0.05 sys=0.00, real=0.03 secs]
71.261: [GC [PSYoungGen: 0K->0K(1536K)] 3233K->3233K(136960K), 0.0006216 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
71.262: [Full GC (System) [PSYoungGen: 0K->0K(1536K)] [ParOldGen: 3233K->929K(135424K)] 3233K->929K(136960K) [PSPermGen: 4458K->4458K(12288K)], 0.0281970 secs] [Times: user=0.06 sys=0.00, real=0.03 secs]
Heap
 PSYoungGen   total 1536K, used 691K [0x12230000, 0x123d0000, 0x17780000)
  eden space 1408K, 49% used [0x12230000,0x122dcfd0,0x12390000)
  from space 128K, 0% used [0x123b0000,0x123b0000,0x123d0000)
  to   space 128K, 0% used [0x12390000,0x12390000,0x123b0000)
 ParOldGen   total 135424K, used 929K [0x07780000, 0x0fbc0000, 0x12230000)
  object space 135424K, 0% used [0x07780000,0x078685b0,0x0fbc0000)
 PSPermGen   total 12288K, used 4471K [0x03780000, 0x04380000, 0x07780000)
  object space 12288K, 36% used [0x03780000,0x03bddef0,0x04380000)