CMS and G1 Collector in Java 7 Hotspot: Overview, Comparisons and Performance Metrics

Clarence J M Tauro
Center for Research
Christ University
Bangalore, India

Manjunath V Prabhu
Department of Computer Science
Christ University
Bangalore, India

Vernon J Saldanha
Department of Computer Science
Christ University
Bangalore, India

ABSTRACT
Java is used in large enterprise server applications. Enterprise applications are characterized by large amount of live heap data and considerable thread level parallelism. Garbage collectors are programs that attempts to reclaim garbage, or memory occupied by objects that are no longer in use by the main program [1]. The strength of Java platform is that it performs automatic memory management, thereby shielding developers from the complexity of explicit memory management.

This paper provides an overview of features shared by most Garbage collectors in the latest version of java (as of Jan-2012) Java7. This document also attempts to compare the CMS (Concurrent Mark and Sweep) collector against its replacement and a new implementation in Java7, G1 aka "Garbage first" [2].

General Terms
Garbage Collection, G1 collector, Concurrent Mark and Sweep (CMS).

Keywords
Java 7 Hotspot, G1, Garbage First, CMS, Concurrent Mark and Sweep, Performance, Comparisons.

1. INTRODUCTION
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).

However it is important to note that the logic behind choosing a particular garbage collector is out of the scope of this paper. It is also important to note that the comparison test results may vary based on the underlying hardware, but there will be an attempt made at logically reasoning and generalizing the results.

2. INTRODUCTION TO BASIC CONCEPTS OF GARBAGE COLLECTION
Moving forward, let us look at few concepts that form the basis for this paper.

2.1 Memory Addresses
Memory is array of bytes, with addresses. Fig 1 shows a diagrammatic representation of Memory and Memory addresses.

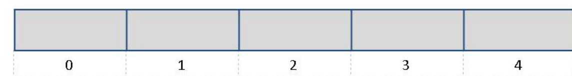

Fig 1: Empty Memory and Memory Address

A 32 bit Processor basically means that the processor can read 32 bits or 4 Bytes at a time. A local variable creation is a process of allocating a memory location that means, naming a memory location.

For instance, Let us assume an integer type with value "10". Fig 2 shows the integer populated in memory address 2.

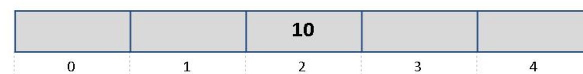

Fig 2: Populated Memory and Memory address

Java picks an appropriate memory address, assigns it and keeps track of the address as well. JVM has an invisible Data Structure, which keeps track of the free and allocated memory.

2.2 Basic Garbage Collection Concepts
Garbage collectors are programs which are responsible for various memory management activities such as –

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

Java puts all the newly created objects in a "heap". An object that is being used or is going to be used by the application is called a "Live Object" [3]. 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 [4]. 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.

2.3 Fragmentation

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.3, Consider a scenario where a memory has exactly 10 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.

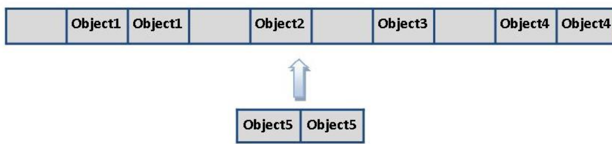

Fig 3: Example of fragmented memory space

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 4

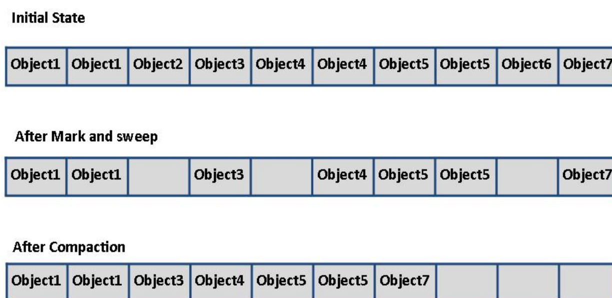

Fig 4: Mark, Sweep and Compact phases

2.4 References

A reference variable in Java contains an address, or a reference to an address (similar to pointer variables in C++). Java does not, however, allow this address to be arbitrarily (randomly) edited or changed in any way. An object can be referenced by many other objects. Figure.5 shows multiple references to an object in the memory.

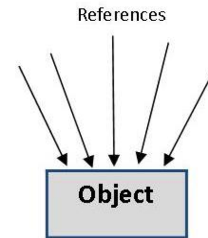

Fig 5: Object References

Since the memory is shifted during compaction, what will happen to the references?

In Sun JVM, a reference is known as a Handle and not a pointer [6]. A handle is a pointer to a pointer and is used to reliably move these references.

Fig 6 shows the references to an object, using handles.

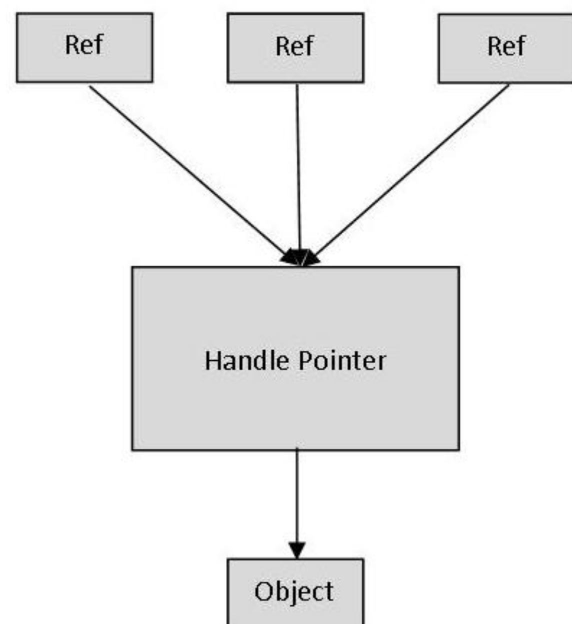

Fig 6: Java handle pointer

2.5 Characteristics of Garbage Collector

- A garbage collector must be both accurate and comprehensive, by not wrongly freeing up memory used by live objects and also reclaim garbage within a small number of collection cycles.
- A garbage collector should be efficient and be virtually untraceable, by not introducing long pauses during application execution.
- It is desired that the garbage collector rearranges the freed memory spaces into a single contiguous area such that there is always memory for allocation of a large object.
- Garbage collectors should be able to cater to scaling memory allocation and garbage collection needs in multithreaded or multi processor environments.

3. OVERVIEW OF VARIOUS GARBAGE COLLECTING ALGORITHMS

Some of the most widely used garbage collection algorithms are –

3.1 Mark and Sweep Algorithm.

This is the most commonly used algorithm. As the name suggests, this algorithm has two main phases

3.1.1 Mark Phase

Mark phase does a DFS from every root object. It basically "paints" or "Marks" all the live objects [7]. 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 [8]. All reachable objects will be marked live, and all non-reachable objects will be marked dead.

3.1.2 Sweep Phase:

In this phase, dead objects are "swept" which means, the memory occupied by the dead objects are reclaimed.

3.2 Copy Garbage Collection

Copy garbage collection is much faster than Mark and sweep algorithm because it has only one phase. In copy garbage collector Memory is divided into two separate spaces called the old space and the new space. It finds all the live objects using Depth first Search algorithm. When it finds a live object, it moves it to the new space immediately. Compaction is automatically taken care as the object will be moved to the first available memory location in the new space. Once all the objects are moved, it forgets the old space. Next time, new and old space trade places.

The disadvantage of the copy collector is the memory usage, which means, memory is cut into two halves.

3.3 Generational garbage collector

This collector is based on a theory that majority of the objects "Die young", that is, in a Heap the following are true -

- Most objects have short life time.
- Only a few live very long.
- Longer they live, more likely they live longer.

Generational garbage collector divides objects into generations and treats the old and new objects differently.

In Generational Garbage collection, the heap is divided into two or more generations namely nursery, young, and old. Nursery is nothing but newly created object. They may be of different sizes and may also change during execution.

Every cycle of garbage collection cycle survived by an object promotes it to the older generations from the younger or nursery generations. It works on a principle of "Longer the objects live, more likely they live longer". This principle allows the garbage collector not to worry about the older generations as much, thereby restricting much of the garbage collection to the younger generations. This leads to significantly less CPU usage and increase in performance.

4. CMS versus G1

CMS and G1 are one of the many garbage collectors provided by Java. This section delves into the characteristics of each of these collectors.

4.1 Concurrent Mark Sweep GC (aka CMS)

CMS is a Generational, stop-the-world collector which is based on the Mark and Sweep algorithm [1]. It is Mostly Concurrent, and is used when applications demand quick response times

Garbage collection using CMS follows the process of -

- Mark concurrently while mutator is running
- Track mutations in card marks
- Revisit mutated cards (repeat as needed)
- Stop-the-world to catch up on mutations, reference processing, etc.

The pauses in a CMS collector are relatively small due to the concurrent marking of live objects. It thereby ensures maximum response times.

Since CMS delivers on many fronts, response times are increased. However as a trade-off it usually suffers an overhead caused by revisiting the mutated cards. The revisiting is necessary to correct all the references that may have occurred while the collector was in the Concurrent Mark phase [1].

Another drawback of CMS is that it does not compact the fragmented memory [1]. Though it maintains a free list, objects are not moved around. Due to this fragmentation may occur. 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).

4.2 G1GC (aka "Garbage First Garbage collector")

Garbage-First is a server-style garbage collector, targeted for multi-processors with large memories, that meets a real-time goal [2]. In G1, there is no physical separation between the young and old generations. There is a single contiguous heap which is split into same-sized regions. The young generation is a set of potentially non-contiguous regions, and the same is true for the old generation. This allows G1 to flexibly move resources as needed from the old to the young generation, and vice versa [9]. Collection in G1 takes place through evacuation pauses, during which the survivors from a set of regions referred to as the collection set are evacuated to another set of regions (the to-space) so that the collection set regions can then be reclaimed. Evacuation pauses are done in parallel, with all available CPUs participating. Most evacuation pauses collect the available young regions, thus are the equivalent of young collections in other HotSpot™ GCs. Occasionally, select old regions may also be collected during these pauses because G1 piggybacks old generation collection activity on young collections [9].

G1 is both concurrent and parallel. G1 takes advantage of the parallelism that exists in hardware today [9]. It uses all available CPUs (cores, hardware threads, etc.) to speed up its "stop-the-world" pauses when an application's Java threads are stopped to enable GC. It also works concurrently with running Java threads to minimize whole-heap operations during stop-the-world pauses.

Concurrency involves refinement, marking, cleanup and parallelism involves multiple thread for various Stop the world phases. Currently Full GC is serial mark Sweep compact.

G1 GC has regionalized heap called heap regions. Heap is split into fixed equal sized heap region. They are fixed for the entire JVM process. However, if can be specified by the user as well. If the user does not specify the heap regions, then JVM chooses it heuristically [9]. Heuristics aims at creating 2000 Regions.

"`G1HeapRegionSize`" is the JVM parameter used to specify the heap region. . The size can vary from a minimum of 1 MB to a maximum of 32 MB

Heap regions are managed by Region lists which means, Master Free list, Secondary free list, survivor list and humongous list.

Like CMS collector G1 is generational Collector. The Young generation is not fixed, and is determined logically. Generational regions are just a set of regions and it may not be contiguous [2].

The young generation further comprises of "Eden" where objects are constantly created and destroyed. This the region used for application allocation. It is basically a place where most objects die. Regions used for application allocations. The allocations are done "on demand" from the free list. As the allocation demand come, when mutator allocation region gets filled. On demand heap space is pulled from free list and adds it to the Eden.

G1 also includes the survivor space. Survivor space contains live object that have survived previous Garbage collection cycles.

Fig 7 shows the Heap space distribution of free, Old, Survivor and Young spaces

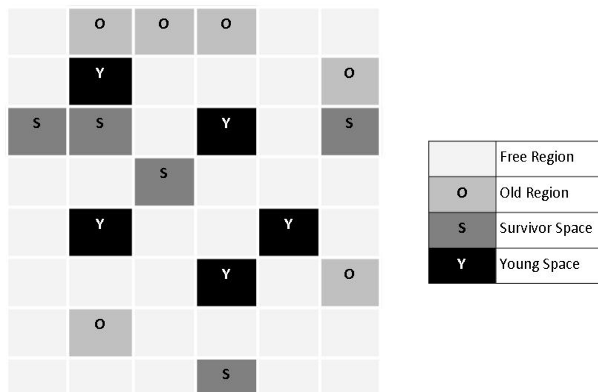

Fig 7: Heap space distribution

4.3 Comparison of Features of G1 and CMS collector

G1 and CMS collectors have many similar and dissimilar features. These features are compared in Table 1

Table 1: Feature comparison of G1 and CMS collector

Features	Garbage Collectors	
	Garbage First(G1)	**Concurrent Mark Sweep(CMS)**
Concurrent and Generational	Yes	Yes
Releases Max Heap memory after usage	Yes	No
Low-latency	Yes	Yes
Throughput	Higher	Lower
Compaction	Yes	No
Predictability	More	Less
Physical Separation between Young and old	No	Yes

4.4 Performance Metrics

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.

4.5 Tests

Tests conducted to measure performance and behavior of CMS and G1 collectors are as follows -

4.5.1 Test Description

- The code creates and adds 150 integer Arrays into an Array list.
- Each integer array reserves 4MB of memory that is

```
1 × 1024 × 1024 × 4Bytes  ≡ 4MB
4MB × 150iterations  ≡ 600MB
```

- Arrays are removed during iteration.
- At every 10th iteration, `System.gc()` is called, suggesting the Java Virtual Machine to start garbage collection
- Visual VM provided in JDK is used to capture the results.

4.5.2 Test Code

```
import java.util. ArrayList;
public class GCTest {
private static ArrayList < Integer[] > array
= new ArrayList < Integer[] > (150);
public static void main(Strin g[] args) throws
Interrupte dException {
   Thread.sle ep(15000);
   for (int i = 0; i < 150; i + +)
  {
        array.add( new Integer[1 * 1024 * 1024]);
        Thread.sle ep(50);
   }
   for (int i = 0; i < 150; i + +)
      {
       array.remo ve(array.s ize() - 1);
       if (i%10 == 0) {
          System.gc( );
       }
       Thread.sle ep(50); }
       System.gc( );
       Thread.sle ep(3000);
    }
}
```

4.5.3 CMS collector results

Command line used to test CMS collector

```
java - XX : +UseConcMarkSweepGC - XX : +PrintGCDetails
- Xloggc : G1GC.log - XX : GCTimeRatio = 49
- XX : MaxGCPauseMillis = 50 GCTest
```

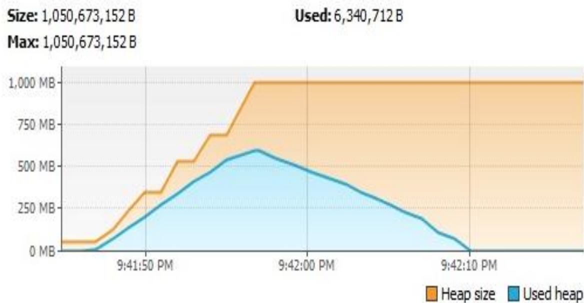

Fig 8: Max allocated heap size and Max used heap size for CMS collector

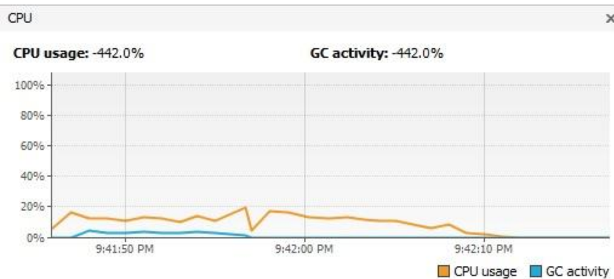

Fig 9: Application CPU usage and GC CPU usage for CMS collector

4.5.4 G1 Collector Results.

Command line used to test G1 collector

```
java - XX : +UseG1GC - XX : +PrintGCDe tails - Xloggc
: G1GC.log - XX : GCTimeRati o = 49
- XX : MaxGCPause Millis = 50 GCTest
```

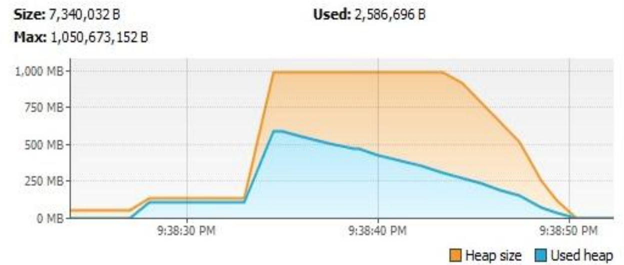

Fig 10: Max allocated heap size and Max used heap size for CMS collector

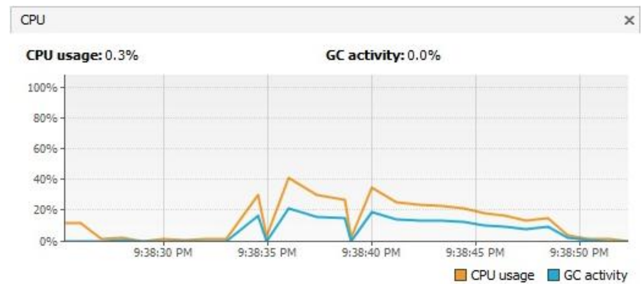

Fig 11: Application CPU usage and GC CPU usage for CMS collector

5. OBSERVATIONS

Observations based on multiple cycles of the tests on different machines.

1. When G1 collector is used, the Max heap size is reclaimed, but in case of CMS it is not reclaimed
2. In CMS Max used heap size is around 20 MB, but in G1 it is 600 MB.
3. Max heap size(available) in case of G1 it is 750 MB and CMS it is 65 MB
4. Max Throughput of G1 was 2.8%, but in case of CMS was well within 2%.

5.1 Results comparison

During execution of the test class, the following parameters have been considered and noted down. Results of the test are compared in Table 2

Table 2: Test results comparison

Parameters	G1 GC	CMS GC
Time taken for execution	31 Secs	31 Secs
Max CPU Usage	41.4%	20%

Max GC Activity	22%	5.1%
Max Heap Size	1000 MB	1000 MB
Max Used Heap	610 MB	610 MB

6. CONCLUSION

The paper has successfully presented an overview of the most commonly used garbage collection algorithms namely – Mark and Sweep, Copy GC, Generational GC. Its attempt to compare the mature CMS with the newly conceptualized G1 resulted in the following conclusions.

1. When G1 collector is used, the Max heap size is reclaimed, but in case of CMS it is not reclaimed
2. If a server has good CPU and RAM then G1 is a good option.
3. If a server has average CPU and good RAM, then CMS holds the edge over G1.
4. Application Performance is better in CMS than G1 owing to high CPU utilization.

7. ACKNOWLEDGMENTS

We are heartily thankful to Poonam Bajaj (Principal Member of Technical Staff at Oracle), Charlie Hunt (Principal Member of Technical Staff at Oracle), and Bengt Rutisson (Engineer Software Sr at Oracle), whose encouragement, guidance and support enabled us to develop an understanding of the subject.

8. REFERENCES

[1] Description of HotSpot GCs: Memory Management in the Java HotSpot Virtual Machine White Paper: http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf.

[2] The Garbage-First Garbage Collector, Oracle Technology network http://www.oracle.com/technetwork/java/javase/tech/g1-intro-jsp-135488.html.

[3] Tuning Garbage Collection with the 1.4.2 Java[tm] Virtual Machine, http://java.sun.com/docs/hotspot/gc1.4.2/

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

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

[6] The Java Language Environment, Memory Management and Garbage Collection, http://java.sun.com/docs/white/langenv/Simple.doc1.html#2333

[7] Formal Programming Language Theory, Dataflow Analysis http://www.cs.is.noda.tus.ac.jp/~mune/keio/m/chap2.pdf.

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

[9] The original G1 paper: Detlefs, D., Flood, C., Heller, S., and Printezis, T. 2004. Garbage-first garbage collection. In Proceedings of the 4th international Symposium on Memory Management (Vancouver, BC, Canada, October 24 - 25, 2004) http://labs.oracle.com/jtech/pubs/04-g1-paper-ismm.pdf