ycrash

JVM Performance
Engineering and Troubleshooting

Compilation of Ram Lakshmanan's Blog and Articles

Author

Ram Lakshmanan
(yCrash)

Index

Garbage Collection

1. “I don’t have to worry about Garbage collection” – Is it true?

I have heard a few of my developer friends say: **“Garbage Collection is automatic. So, I do not have to worry about it.“** The first part is true, i.e., “Garbage Collection is automatic” on all modern platforms – Java, .NET, Golang, Python… But the second part i.e., “I don’t have to worry about it.” – may not be true. It is arguable, questionable. Here is my case to showcase the importance of Garbage Collection:

1. Unpleasant customer experience

When a garbage collector runs, it pauses the entire application to mark the objects that are in use and sweep away the objects that don’t have active references. During this pause period, all customer transactions which are in motion will be stalled (i.e., frozen). Depending on the type of GC algorithm and memory settings that you configure, pause times can run anywhere from a few milliseconds to a few minutes. Frequent pauses in the application can cause stuttering, juddering, or halting effects to your customers. It will leave an unpleasant experience for your customers.

2. Millions of dollars wasted

Here is a white paper we published, explaining factually how enterprises are wasting millions of dollars due to garbage collection. Basically, in a nutshell, modern applications are creating thousands/millions of objects. These objects must be continuously investigated to determine whether they have active references or are they ready for garbage collection. Once objects are garbage collected, the memory becomes fragmented. Fragmented memory must be compacted. All these activities consume ***enormous compute cycles***. These compute cycles translate to millions of dollars in spending. If Garbage collection performance can be optimized, it can result in several millions of dollars in cost savings.

3. Low risk, high impact performance improvements

By virtue of optimizing Garbage collection performance, you are not only improving the Garbage collection pause time, but you are improving the overall application's response time. We recently helped to tune the garbage collection performance of one of the world's largest automobile companies. Just by modifying the garbage collection settings **without refactoring a single line** of code, we improved their overall application's response time significantly. The below table summarizes the overall response time improvement we achieved with each Garbage Collection setting change we made:

	Avg Response Time (secs)	Transactions > 25 sec (%)
Baseline	1.88	16
GC settings iteration #2	4GB	8
GC settings iteration #3	4GB	8
GC settings iteration #4	4GB	8
GC settings iteration #5	4GB	8
GC settings iteration #6	4GB	8
GC settings iteration #7	4GB	8
GC settings iteration #8	4GB	8

When we started the GC tuning exercise, this automobile application's overall response time was 1.88 seconds. As we optimized Garbage Collection performance with different settings, on iteration #8, we were able to improve the overall response time to 0.95 seconds. i.e., **49.46%** improvement in the response time. Similarly, percentages of transactions taking more than 25 seconds dropped from 0.7% to 0.31%, i.e., 55% improvement. This is a significant improvement to achieve without modifying a single line of code.

All other forms of response time improvement will require infrastructure change or architectural change, or code-level changes. All of them are expensive changes. Even if you embark on making those costly changes, there is no guarantee of the application's

response time improvement.

4. Predictive Monitoring

Garbage Collection logs expose vital predictive micrometrics. These metrics can be used for forecasting application's availability and performance characteristics. One of the micrometrics exposed in Garbage Collection is 'GC Throughput' (to read more about other micrometrics, refer to this article). What is GC Throughput? If your application's GC throughput is 98%, it means your application is spending 98% of its time processing customer activity and the remaining 2% of the time in GC activity. When the application suffers from a memory problem, several minutes before GC throughput will start to degrade. Troubleshooting tools like yCrash monitors 'GC throughput' to predict and forecast the memory problems before they surface in the production environment.

5. Capacity Planning

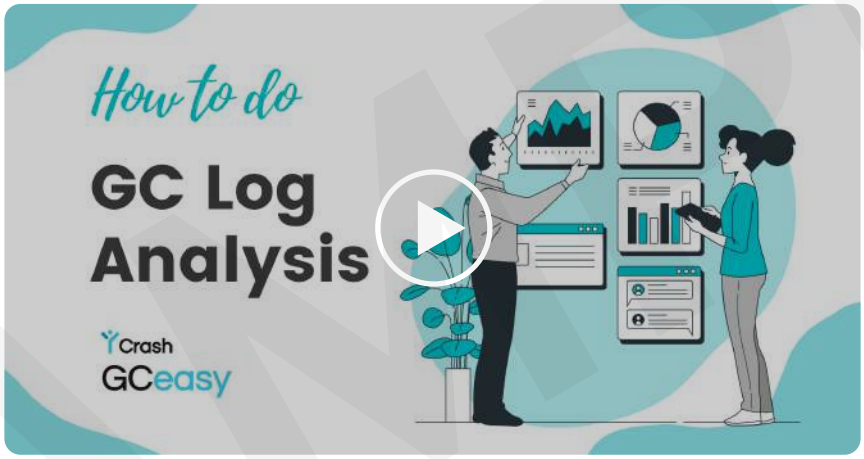
When you are doing capacity planning for your application, you need to understand your application's demand for memory, CPU, Network and storage. One of the best ways to study the demand for memory is by analyzing garbage collection behaviour. When you analyze garbage collection behaviour, you would be able to determine average object creation rate (example: 150 MB/sec), average object reclamation rate. Using these sort of micrometrics you can do effective capacity planning for your application.

Conclusion

Friends, in this post, I have made my best efforts to justify the importance of garbage collection analysis. I wish you and your team the best to benefit from the highly insightful garbage collection metrics.

2. How to do GC Log analysis?

Analyzing garbage collection log provides several advantages like: Reduces GC pause time, reduces cloud computing cost, predicts outages, provides effective metrics for capacity planning. To learn about the profound advantages of GC log analysis, please refer to this post. In this post let's learn how to analyze GC logs?



Watch video

https://www.youtube.com/watch?v=dZbmlMLCfZY

Here is an interesting video clip which walks through the best practices, KPIs, tips & tricks to effectively optimize Garbage collection performance.

Basically, there are 3 essential steps when it comes to GC log analysis:

Let's discuss these 3 steps now.

1. Enable GC Logs

Even though certain monitoring tools provide Garbage Collection graphs/metrics at real time , they don't provide a complete set of details to study the GC behavior. GC logs are the best source of information, to study the Garbage Collection behavior. You can enable GC logs, by specifying below JVM arguments in your application:

Java 8 & below versions:

If your application is running on Java 8 & below versions, then pass below arguments:

```
-XX:+PrintGCDetails -Xloggc:<gc-log-file-path>
Example:
-XX:+PrintGCDetails -Xloggc:/opt/tmp/myapp-gc.log
```

Java 8 & below versions:

If your application is running on Java 8 & below versions, then pass below arguments:

```
-XX:+PrintGCDetails -Xloggc:<gc-log-file-path>
Example:
-XX:+PrintGCDetails -Xloggc:/opt/tmp/myapp-gc.log
```

2. Measurement Duration & environment

It's always best practice to study the GC log for a 24-hour period during a weekday, so that application would have seen both high volume and low volume traffic tide.

It's best practice to collect the GC logs from the production environment, because garbage collection behavior is heavily influenced by the traffic patterns. It's hard to simulate production traffic in a test environment. Also overhead added by GC log in production servers is negligible, in fact it's not even measurable. To learn about overhead added by enabling GC logs, you can refer here.

3. Tools to analyze

Once you have captured GC logs, you can use one of the following free tools to analyze the GC logs:

1. GCeasy
2. IBM GC & Memory visualizer
3. HP Jmeter
4. Garbage Cat

3. Garbage collection patterns to predict outages

As the author of GCeasy – Garbage collection log analysis tool, I get to see few interesting Garbage Collection Patterns again & again. Based on the Garbage collection pattern, you can detect the health and performance characteristics of the application instantly. In this video and the post, let me share few interesting Garbage collection patterns that have intrigued me.

Watch video

https://www.youtube.com/watch?v=4jlfd3XCeTM

1. Healthy saw-tooth pattern

You will see a beautiful saw-tooth GC pattern when an application is healthy, as shown in the above graph. Heap usage will keep rising; once a 'Full GC' event is triggered, heap usage will drop all the way to the bottom.

In Fig 1, You can notice that when the heap usage reaches ~5.8GB, 'Full GC' event (red

triangle) gets triggered. When the 'Full GC' event runs, memory utilization drops all the way to the bottom i.e., ~200MB. Please see the dotted black arrow line in the graph. It indicates that the application is in a healthy state & not suffering from any sort of memory problems.

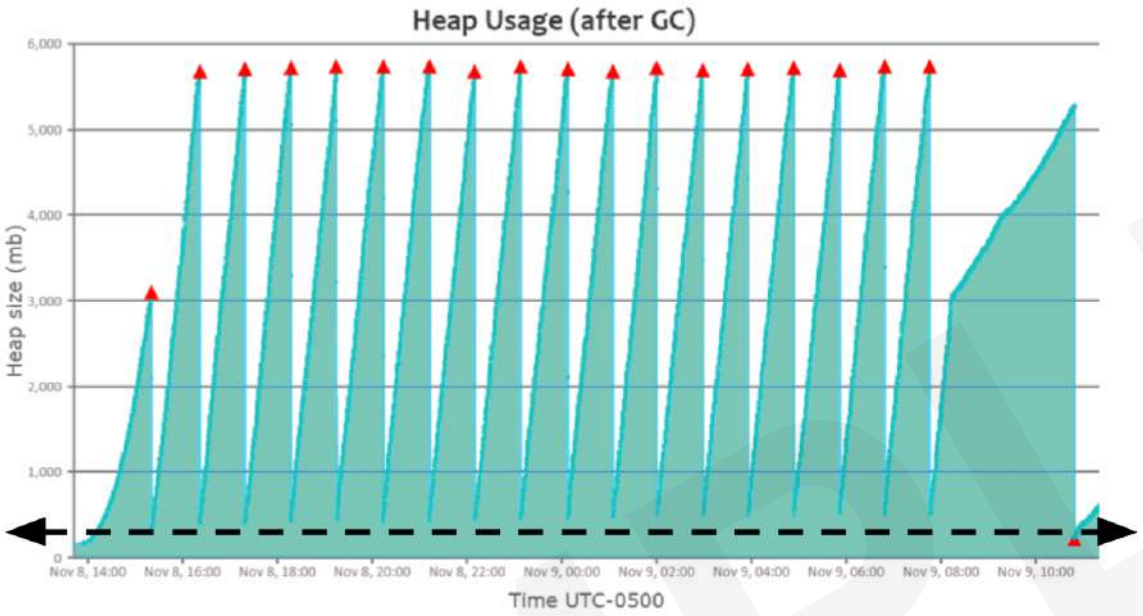


Fig 1: Healthy saw-tooth GC pattern

2. Heavy caching pattern

When an application is caching many objects in memory, 'GC' events wouldn't be able to drop the heap usage all the way to the bottom of the graph (like you saw in the earlier 'Healthy saw-tooth' pattern).

In Fig 2, you can notice that heap usage keeps growing. When it reaches around ~60GB, GC event (depicted as a small green square in the graph) gets triggered. However, these GC events aren't able to drop the heap usage below ~38GB. Please refer to the dotted black arrow line in the graph. In contrast, in the earlier 'Healthy saw-tooth pattern', you can see that heap usage dropping all the way to the bottom ~200MB. When you see this sort of pattern (i.e., heap usage not dropping till all the way to the bottom), it indicates that the application is caching a lot of objects in memory.

When you see this sort of pattern, you may want to investigate your application's heap using heap dump analysis tools like yCrash, HeapHero, Eclipse MAT and figure out whether you need to cache these many objects in memory. Several times, you might uncover unnecessary objects to be cached in the memory.

Here is the real-world GC log analysis report, which depicts this 'Heavy caching' pattern.

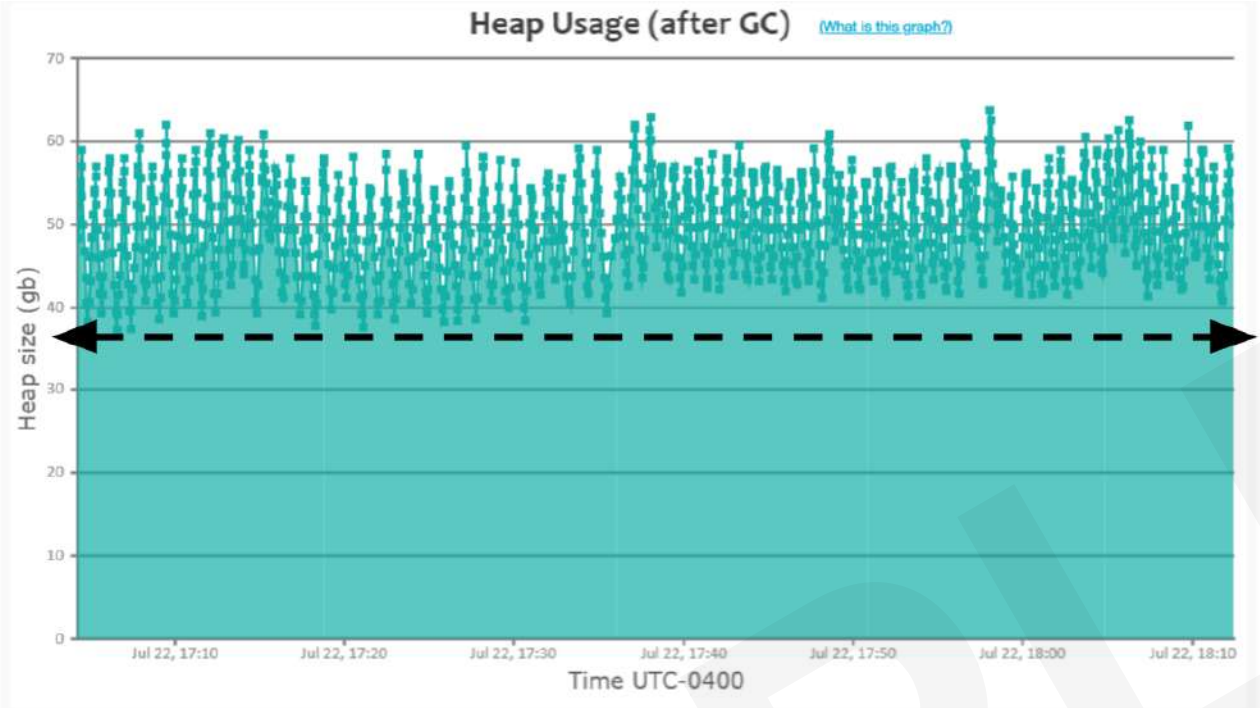


Fig 2: Healthy caching GC pattern

3. Acute memory leak pattern

Several applications suffer from this 'Acute memory leak pattern'. When an application suffers from this pattern, heap usage will climb up slowly, eventually resulting in OutOfMemoryError.

In Fig 3, you can notice that 'Full GC' (red triangle) event gets triggered when heap usage reaches around ~43GB. In the graph, you can also observe that amount of heap that full GC events could recover starts to decline over a period of time, i.e., you can notice that

a. When the first Full GC event ran, heap usage dropped to 22GB

b. When the second Full GC event ran, heap usage dropped only to 25GB

c. When the third Full GC event ran, heap usage dropped only to 26GB

d. When the final full GC event ran heap usage dropped only to 31GB

Please see the dotted black arrow line in the graph. You can notice the heap usage gradually climbing up. If this application runs for a prolonged period (days/weeks), it will experience OutOfMemoryError (please refer to Section #5 – 'Memory Leak Pattern').

Here is the real-world GC log analysis report, which depicts this 'Acute memory leak'

pattern.

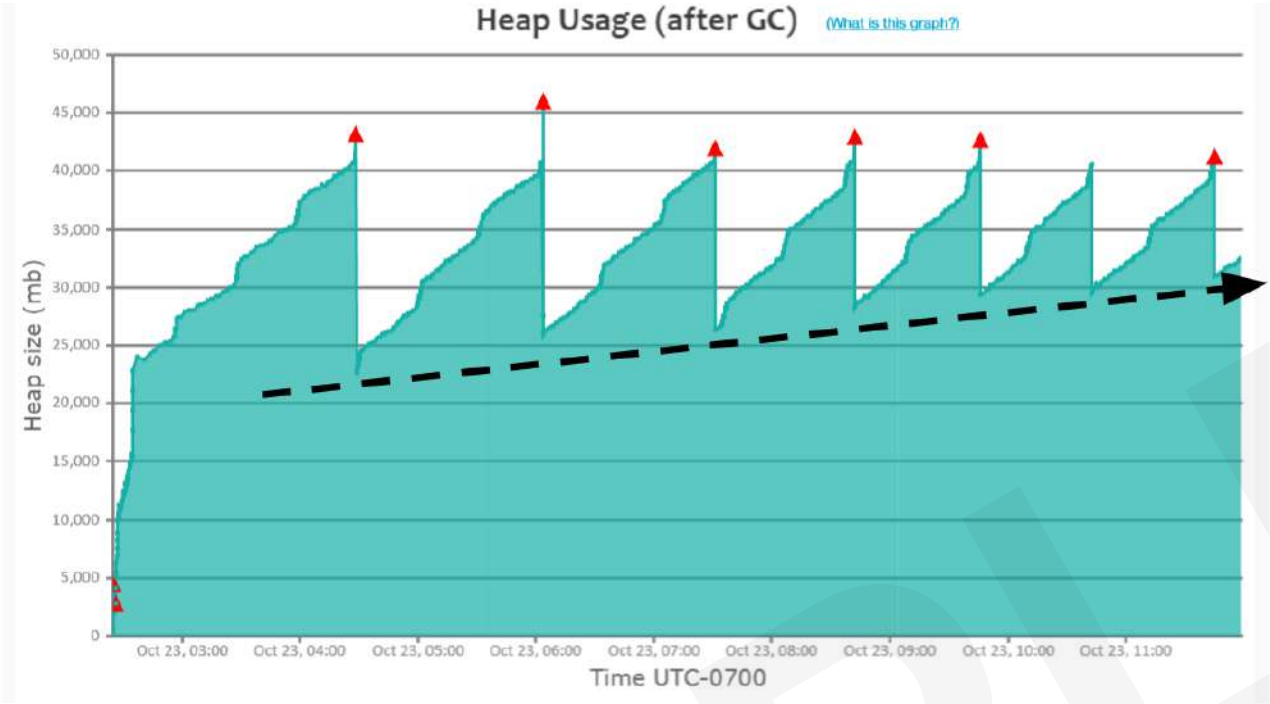


Fig 3: Acute memory leak pattern

4. Consecutive Full GC pattern

When the application's traffic volume increases more than JVM can handle, this Consecutive full GC pattern will become pervasive.

In Fig 4, please refer to the black arrow mark in the graph. From 12:02pm to 12:30 pm on Oct' 06, Full GCs (i.e., 'red triangle') are consecutively running; however, heap usage isn't dropping during that time frame. It indicates that traffic volume spiked up in the application during that time frame, thus the application started to generate more objects, and Garbage Collection couldn't keep up with the object creation rate. Thus, GC events started to run consecutively. Please note that when a GC event runs, it has two side effects:

a. CPU consumption will go high (as GC does an enormous amount of computation).

b. Entire application will be paused; no customers will get response.

Thus, during this time frame, 12:02pm to 12:30pm on Oct' 06, since GC events are consecutively running, application's CPU consumption would have been skyrocketing and customers wouldn't be getting back any response. When this kind of pattern surfaces, you can resolve it using one of the solutions outlined **in this post**.

Here is the real-world GC log analysis report, which depicts this 'Consecutive Full GC' pattern.

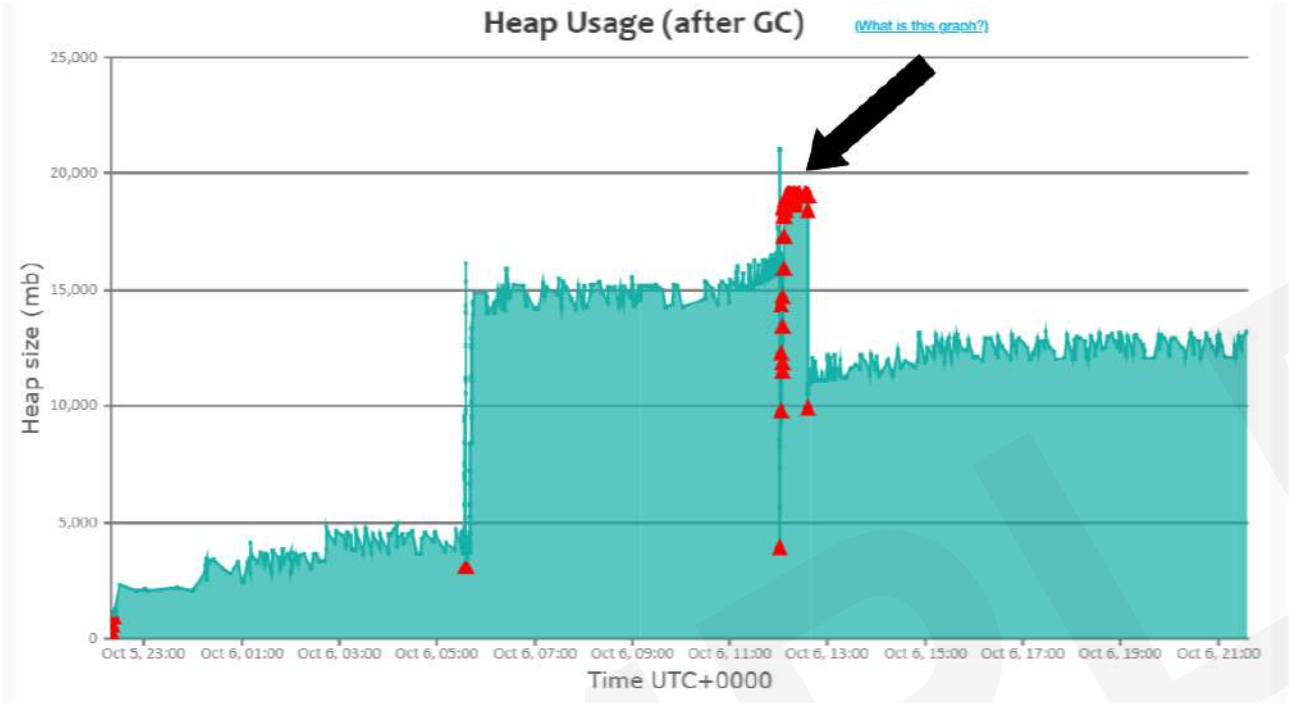


Fig 4: Consecutive full GC pattern

5. Memory Leak Pattern

This is a 'classic pattern' that you will see whenever the application suffers from memory problems. In Fig 5, please observe the black arrow mark in the graph. You can notice that Full GC (i.e., 'red triangle') events are continuously running. This pattern is similar to the previous 'Consecutive Full GC' pattern, with one sharp difference. In the 'Consecutive Full GC' pattern, application would recover from repeated Full GC runs and return back to normal functioning state, once traffic volume dies down. However, if the application runs into a memory leak, it wouldn't recover, even if traffic dies. The only way to recover the application is to restart the application. If the application is in this state, you can use tools like yCrash, HeapHero, Eclipse MAT to diagnose memory leak. Here is a more detailed post on how to diagnose Memory leak.

Here is the real-world GC log analysis report, which depicts this 'Memory Leak' pattern.

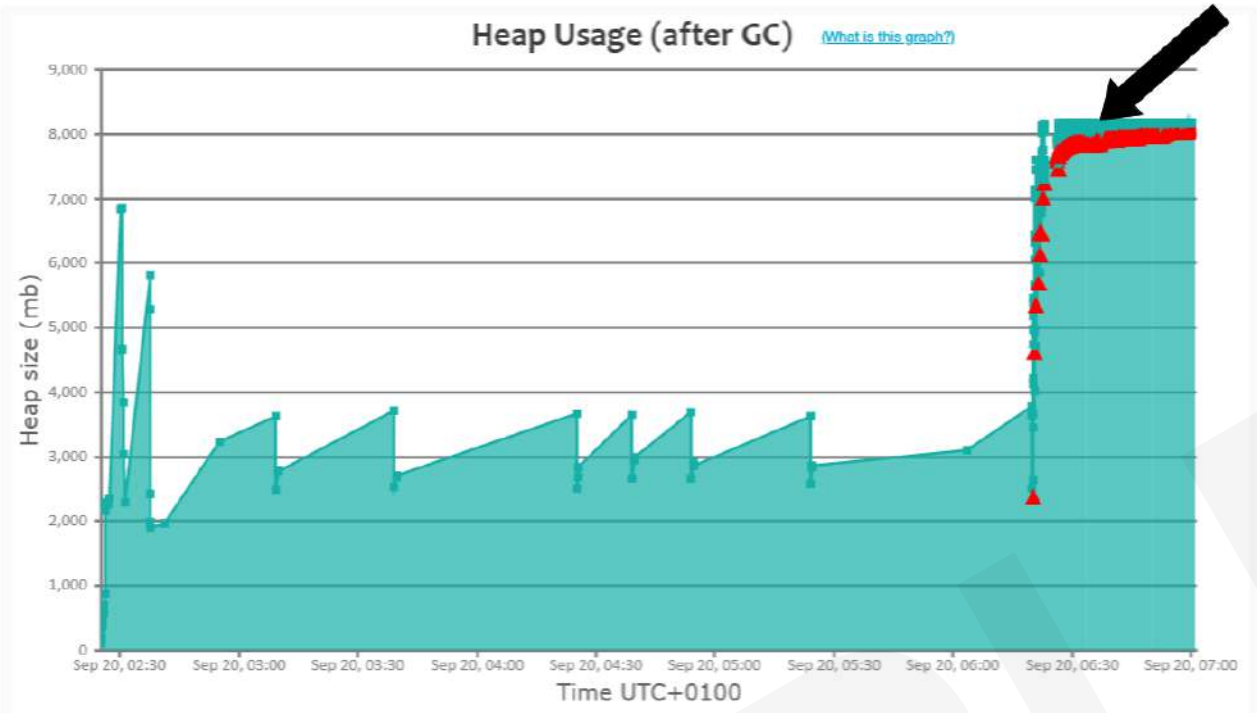


Fig 5: Memory leak GC pattern

6. Metaspace Memory problem Pattern

If you notice in this graph pattern, Full Garbage Collection events are consecutively triggered after 12:30am even though only 10% of maximum heap size is reached. Maximum available heap size for this application is 2.5GB, whereas Full GC events are triggered even memory is reaching 250MB (i.e., 10% of the maximum size). Typically, consecutive full GCs are triggered only when maximum heap size is reached. When you see this sort of pattern it's indicative that Metaspace region is reaching its maximum size. This can happen when

a. Metaspace region size is under allocated

b. Memory leak in the Metaspace region.

You can increase Metaspace region size by passing this JVM argument (-XX:MaxMetaspaceSize). You can refer to this post to see how to troubleshoot Metaspace memory problem.

Here is the real-world GC log analysis report, which depicts this 'Metaspace Memory problem' Pattern.

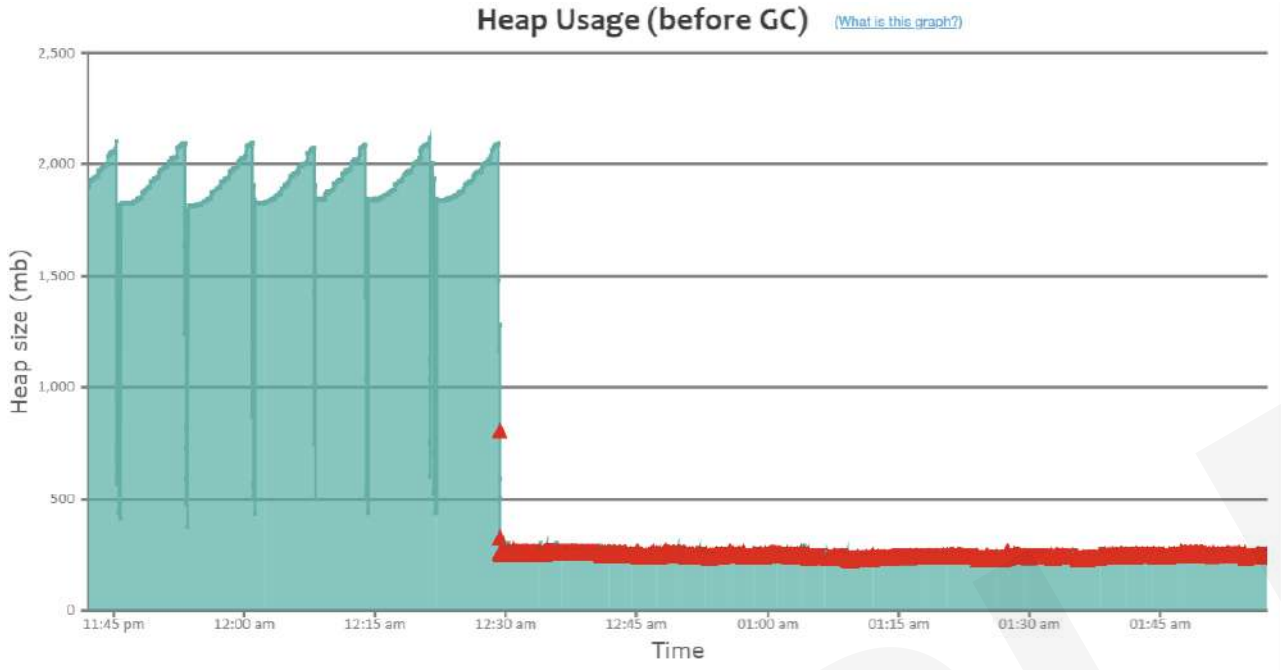


Fig 6: Metaspace memory problem pattern

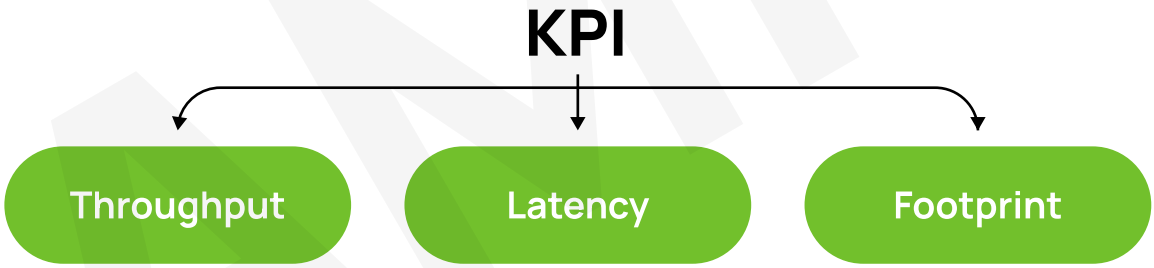
Conclusion

You can also consider enabling your application's Garbage collection log (as it doesn't add any measurable overhead to your application) and study the garbage collection behavior. It may reveal insightful views/perspectives about your application that you weren't aware of before.

4. Memory tuning: Key Performance Indicators

When you are tuning the application's memory & Garbage Collection settings, you should take well-informed decisions based on the key performance indicators. But there are overwhelming amount of metrics reported; which one to choose and which one to leave? This article intends to explain the right KPIs and right tools to source them.

What are the right KPIs?



Throughput

Throughput is the amount of productive work done by your application in a given time period. This brings the question what is productive work? what is non-productive work?

Productive Work: This is basically the amount of time your application spends in processing your customer's transactions.

Non-Productive Work: This is basically the amount of time your application spend in house-keeping work, primarily Garbage collection.

Let's say your application runs for 60 minutes. In this 60 minutes let's say 2 minutes is spent on GC activities.

It means application has spent 3.33% on GC activities (i.e. 2 / 60 * 100)

It means application throughput is 96.67% (i.e. 100 – 3.33).

Now the question is: What is the acceptable throughput %? It depends on the applicationand business demands. Typically one should target for more than 95% throughput.

Latency

This is the amount of time taken by one single Garbage collection event to run. This indicator should be studied from 3 fronts.

a. **Average GC Time:** What is the average amount of time spent on GC?

b. **Maximum GC time:** What is the maximum amount of time spent on a single GC event? Your application may have service level agreements such as "no transaction can run beyond 10 seconds". In such cases, your maximum GC pause time can't be running for 10 seconds. Because during GC pauses, entire JVM freezes – no customer transactions will be processed. So it's important to understand the maximum GC pause time.

c. **GC Time Distribution:**You should also understand how many GC events are completing with in what time range (i.e. within 0 – 1 second, 200 GC events are completed, between 1 – 2 second 10 GC events are completed …)

Footprint

Footprint is basically the amount CPU consumed. Based on your GC algorithm, based on your memory settings, CPU consumption will vary. Some GC algorithms will consume more CPU (like Parallel, CMS), whereas other algorithms such as Serial will consume less CPU.

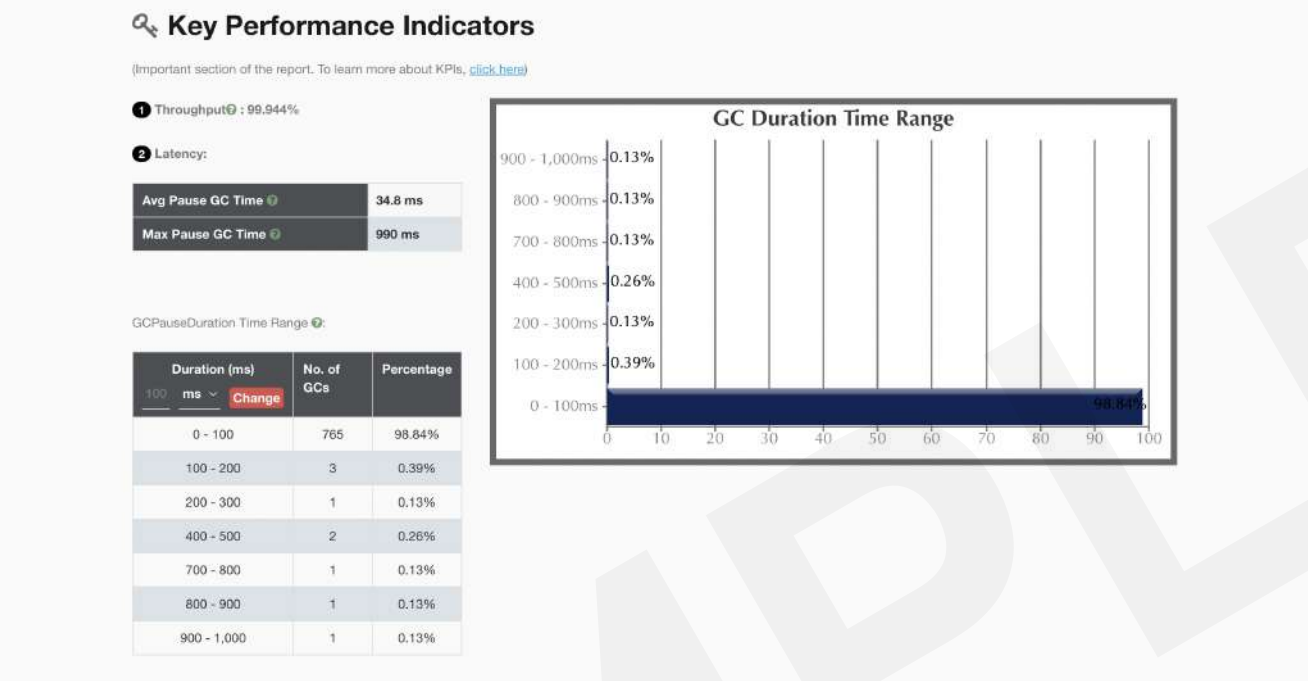
According to memory tuning Gurus, you can pick only 2 of them at a time.

- If you want good throughput and latency, then footprint will degrade.
- If you want good throughput and footprint, then latency will degrade.
- If you want good latency and footprint, then throughput will degrade.

Right Tools

Throughput and Latency can be obtained from analyzing Garbage collection Logs.

Upload your application's Garbage Collection log file in **http://gceasy.io/** tool. This tool can parse Garbage Collection logs and generates Throughput and Latency indicators for you. Below is the screen shot from the **http://gceasy.io/** tool showing the throughput and latency:



Key Performance Indicators

(Important section of the report. To learn more about KPIs, click here)

1 Throughput : 99.944%

2 Latency:

Avg Pause GC Time	34.8 ms
Max Pause GC Time	990 ms

GCPauseDuration Time Range:

Duration (ms) 100 ms Change	No. of GCs	Percentage
0 - 100	765	98.84%
100 - 200	3	0.39%
200 - 300	1	0.13%
400 - 500	2	0.26%
700 - 800	1	0.13%
800 - 900	1	0.13%
900 - 1,000	1	0.13%

Fig 1: KPI section from GCeasy.io report

Footprint (i.e. CPU consumption) can be obtained from the monitoring tools – Nagios, NewRelic, AppDynamics,…

5. Tips to reduce Long GC Pauses

Long GC Pauses are undesirable for applications. It affects your SLAs; it results in poor customer experiences, and it causes severe damages to mission critical applications. Thus in this article, I have laid out key reasons that can cause long GC pauses and potential solutions to solve them.

1. High Object Creation Rate

If your application's object creation rate is very high, then to keep with it, garbage collection rate will also be very high. High garbage collection rate will increase the GC pause time as well. Thus, optimizing the application to create less number of objects is THE EFFECTIVE strategy to reduce long GC pauses. This might be a time-consuming exercise, but it is 100% worth doing. In order to optimize object creation rate in the application, you can consider using java profilers like JProfiler, YourKit, JVisualVM….). These profilers will report

- What are the objects that created?
- What is the rate at which these objects are created?
- What is the amount of space they are occupying in memory?
- Who is creating them?

Always try to optimize the objects which occupy the most amount of memory. Go after big fish in the pond.

Tit-bit: How to figure out object creation rate?
Upload your GC log to the Universal Garbage Collection log analyzer tool **GCeasy.** This tool will report the object creation rate. There will be field by name 'Avg creation rate' in the section 'Object Stats.' This field will report the object creation rate. Strive to keep

this value lower always. See the image (which is an excerpt from the **GCeasy** generated report), showing the 'Avg creation rate' to be 8.83 mb.sec.

Object Stats

These are good metrics to compare with previous baseline

Total created bytes	3.82 tb
Total promoted bytes	16.48 tb
Avg creation rate	8.83 mb/sec
Avg promotion rate	38 kb/sec

2. Undersized Young Generation

When young Generation is undersized, objects will be prematurely promoted to Old Generation. Collecting garbage from old generation takes more time than collecting it from young Generation. Thus increasing young generation size has a potential to reduce the long GC pauses. Young Generation can be increased setting either one of the two JVM arguments

-Xmn: specifies the size of the young generation

-XX:NewRatio: Specifies ratio between the old and young generation. For example, setting -XX:NewRatio=3 means that the ratio between the old and young generation is 3:1. i.e. young generation will be fourth of the overall heap. i.e. if heap size is 2 GB, then young generation size would be 0.5 GB.

3. Choice of GC Algorithm

Choice of GC algorithm has a major influence on the GC pause time. Unless you are a GC expert or intend to become one or someone in your team is a GC expert – you can tune GC settings to obtain optimal GC pause time. Assume if you don't have GC expertise, then I would recommend using G1 GC algorithm, because of it's auto-tuning capability. In G1 GC, you can set the GC pause time goal using the system property '-XX:MaxGCPauseMillis.' Example:

```
-XX:MaxGCPauseMillis=20
```

As per the above example, Maximum GC Pause time is set to 200 milliseconds. This is a soft goal, which JVM will try it's best to meet it. If you are already using G1 GC algorithm

and still continuing to experience high pause time, then refer to this article.

4. Process Swapping

Sometimes due to lack of memory (RAM), Operating system could be swapping your application from memory. Swapping is very expensive as it requires disk accesses which is much slower as compared to the physical memory access. In my humble opinion – no serious application in a production environment should be swapping. When process swaps, GC will take a long time to complete.

Below is the script obtained from StackOverflow (thanks to the author) – which when executed will show all the process that are being swapped. Please make sure your process is not getting swapped.

```
#!/bin/bash
# Get current swap usage for all running processes
# Erik Ljungstrom 27/05/2011
# Modified by Mikko Rantalainen 2012-08-09
# Pipe the output to "sort -nk3" to get sorted output
# Modified by Marc Methot 2014-09-18
# removed the need for sudo

SUM=0
OVERALL=0
for DIR in `find /proc/ -maxdepth 1 -type d -regex "^/proc/[0-9]+"`
do
  PID=`echo $DIR | cut -d / -f 3`
  PROGNAME=`ps -p $PID -o comm –no-headers`
  for SWAP in `grep VmSwap $DIR/status 2>/dev/null | awk '{ print $2 }'`
  do
    let SUM=$SUM+$SWAP
  done
  if (( $SUM > 0 )); then
    echo "PID=$PID swapped $SUM KB ($PROGNAME)"
  fi
  let OVERALL=$OVERALL+$SUM
  SUM=0
done
echo "Overall swap used: $OVERALL KB"
```

If you find your process to be swapping then do one of the following:

a. Allocate more RAM to the server
b. Reduce the number of processes that running on the server, so that it can free up the memory (RAM).

c. Reduce the heap size of your application (which I wouldn't recommend, as it can cause other side effects).

5. Less GC Threads

For every GC event reported in the GC log, user, sys and real time are printed. Example:

```
[Times: user=25.56 sys=0.35, real=20.48 secs]
```

To know the difference between each of these times, please read the article. (I highly encourage you to read the article, before continuing this section). If in the GC events you consistently notice that 'real' time isn't significantly lesser than the 'user' time – then it might be indicating that there aren't enough GC threads. Consider increasing the GC thread count. Say suppose 'user' time 25 seconds, and you have configured GC thread count to be 5, then real time should be close to 5 seconds (because 25 seconds / 5 threads = 5 seconds).

WARNING: Adding too many GC threads will consume a lot of CPU and takes away a resource from your application. Thus you need to conduct thorough testing before increasing the GC thread count.

6. Background IO Traffic

If there is a heavy file system I/O activity (i.e. lot of reads and writes are happening) it can also cause long GC pauses. This heavy file system I/O activity may not be caused by your application. Maybe it is caused by another process that is running on the same server, still, can cause your application to suffer from long GC pauses. Here is a brilliant article from LinkedIn Engineers, which walks through this problem in detail.

When there is a heavy I/O activity, you will notice the 'real' time to be significantly more than 'user' time. Example:

```
[Times: user=0.20 sys=0.01, real=18.45 secs]
```

When this pattern happens, here are the potential solutions to solve it:
a. If high I/O activity is caused by your application, then optimize it.
b. Eliminate the processes which are causing high I/O activity on the server
c. Move your application to a different server where I/O activity is less

Tit-bit: How to monitor I/O activity?

You can monitor I/O activity, using the sar (System Activity Report), in Unix. Example:

```
sar -d -p 1
```

Above commands reports the reads/sec and writes/sec made to the device every 1 second. For more details on 'sar' command refer to this tutorial.

7. System.gc() calls

When System.gc() or Runtime.getRuntime().gc() method calls are invoked it will cause stop-the-world Full GCs. During stop-the-world full GCs, entire JVM is freezed (i.e. No user activities will be performed during period). System.gc() calls are made from one of the following sources:

1. Your own application developers might be explicitly calling System.gc() method.
2. It could be 3rd party libraries, frameworks, sometimes even application servers that you use could be invoking System.gc() method.
3. It could be triggered from external tools (like VisualVM) through use of JMX
4. If your application is using RMI, then RMI invokes System.gc() on a periodic interval. This interval can be configured using the following system properties:

– Dsun.rmi.dgc.server.gcInterval=n

– Dsun.rmi.dgc.client.gcInterval=n

Evaluate whether it's absolutely necessary to explicitly invoke System.gc(). If there is no need to then, please remove it. On the other hand, you can forcefully disable the System.gc() calls by passing the JVM argument: '-XX:+DisableExplicitGC'. For complete details on System.gc() problems & solution refer to this article.

Tit-bit: How to know whether System.gc() calls are explicitly called?

Upload your GC log to the Universal Garbage Collection log analyzer tool GCeasy. This tool has a section called 'GC Causes.' If GC activity is triggered because of 'System.gc()' calls then it will be reported in this section. See the image (which is an excerpt from the GCeasy generated report), showing that System.gc() was made 4 times during the lifetime of this application.

Object Stats

These are good metrics to compare with previous baseline

Cause	Count
Allocation Failure	242
System.gc() calls	4
Promotion Failure	1
GCLocker Initiated GC	2

8. Large Heap size

Large heap size (-Xmx) can also cause long GC pauses. If heap size is quite high, then more garbage will be get accumulated in the heap. When Full GC is triggered to evict the all the accumulated garbage in the heap, it will take long time to complete. Logic is simple: If you have small can full of trash, it's going to be quick and easy to dispose them. On the other hand if you have truck load of trash, it's going to take more time to dispose them.

Suppose your JVMs heap size is 18GB, then consider having three 6 GB JVM instances, instead of one 18GB JVM. Small heap size has great potential to bring down the long GC pauses.

CAUTION: All of the above mentioned strategies should be rolled to production only after thorough testing & analysis. All strategies may not apply to your application. Improper usage of these strategies can result in negative results.

9. Workload distribution

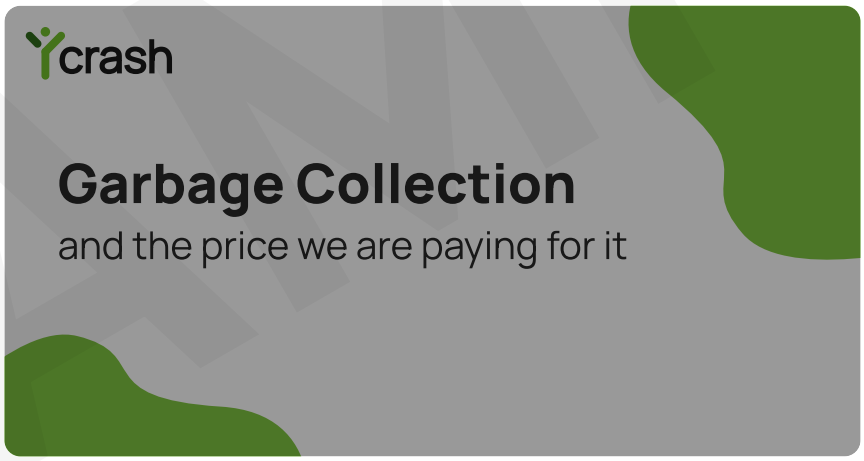
Eventhough there are multiple GC threads, sometimes work load is evenly distributed between GC worker Threads. There are multiple reasons why GC workloads may not be evenly broken up amoing GC threads. For example:

a. Scanning of large linear data structures currently can not be parallelized.b. Some times of events only triggers single thread collector (example when there is 'concurrent mode failure' in CMS collection)

If you happen to use CMS (Concurrent Mark & Sweep algorithm), you can consider passing -**XX:+CMSScavengeBeforeRemark** argument. This can create more balanced workloads among GC worker threads.

6. How many millions of dollars enterprises waste due to Garbage collection?

We truly believe enterprises are wasting millions of dollars in garbage collection. We equally believe enterprises are wasting these many millions of dollars even without knowing they are wasting. Intent of this post is to bring visibility on how several millions of dollars are wasted due to garbage collection.



Watch video

https://www.youtube.com/watch?v=N1_ScYlSlGs&t=7s

What is Garbage?

All applications have a finite amount of memory. When a new request comes, the application creates objects to service the request. Once a request is processed, all the objects created to service that request are no longer needed. In other terms those

objects become garbage. They have to be evicted/removed from the memory so that room is created to service new incoming requests.

Garbage collection evolution: Manual → Automatic

3 – 4 decades back, C, C++ programming languages were popularly used by the development community. In those-programming languages garbage collection needs to be done by the developers. i.e., application developers need to write code to dispose of unreferenced objects from the memory. If developers forget (or miss) to write that logic in their program, then the application will suffer from memory leak. Memory leaks will cause applications to crash. Thus, memory leaks were claimed to be quite pervasive back in those days.

In the mid-1990s when the Java programming language was introduced, it provided automatic garbage collection i.e., developers no longer have to write logic to dispose of unreferenced objects. Java Virtual machine will itself automatically remove unreferenced objects from memory. Definitely it was a great productivity improvement, developers enjoyed this feature. On top of it, a number of memory leak related crashes also came down. Sounds great so far, right? But there was one catch to this automatic garbage collection.

To do this automatic garbage collection, JVM has to pause the application to identify unreferenced objects and dispose them. This pausing can take anywhere from a few milliseconds to few minutes, depending on the application, workload & JVM settings. When an application is paused to do garbage collection, no customer transactions will be processed. Any customer transactions that are in the middle of processing will be halted. It will result in poor response time to the customers. So, this was the trade-off, i.e., for developer productivity and minimizing memory leak related crashes, application pause times got introduced in automatic garbage collection. By doing effective tuning we can bring down the pause time, but it cannot be eliminated.

This might sound like a minor performance hit to the customer's response time. But it does not stop there, today enterprises are losing millions of dollars because of this automatic garbage collection. Below are the interesting facts/details.

Garbage collection Throughput

'GC Throughput' is one of the key metrics that is studied when it comes to Garbage collection tuning. This metric is cleverly reported in percentage. What is 'GC Throughput %?'. It is basically the amount of time application spends in processing the customer transactions vs amount of time application spends in processing Garbage collection activities. Say suppose application has 98% as it's GC Throughput, it means application is spending 98% of its time in processing customer transactions and remaining 2% of time in processing Garbage collection activities.

Does 98% GC throughput sound good to you? Since human minds are trained to read 98% as A grade score, definitely 98% GC throughput should sound good. But in reality, it is not the case. Let us look at the below calculations.

In 1 day, there are **1440 minutes** (i.e. 24 hours x 60 minutes).

98% GC throughput means application is spending **28.8 minutes/day** in garbage collection. (i.e., the application is spending 2% of time in processing GC activities. 2% of 1440 minutes is 28.8 minutes).

What is this telling us? Even if your GC throughput is 98%, your application is spending 28.8 minutes/day (i.e., almost 30 minutes) in Garbage collection. For that 28.8 minutes period your application is pausing. **It's not doing anything for your customer.**

One way to visualize this problem is: Say you have bought a brand-new expensive car and you want to drive this car for a couple of hours. How will you feel if the car runs only for 1 hour and 50 minutes, but stops intermittently in the middle of the road for 10 minutes, and still ends up consuming gasoline? This is what is happening exactly in automatic garbage collection. JVM keeps pausing intermittently, while application is still processing customer transactions.

Dollars wasted

Even healthy application's GC throughput ranges from 99% to 95%. Sometimes it could go even below than that. In the below table I have summarized how many dollars mid-size(1K instances/year), large-size(10K instances/year) and very large(100K instances/year) enterprises would be wasting based on their application's GC throughput percentage.

GC Throughput %	99%	98%	97%	96%	95%
Minutes wasted by 1 instance per day	14.4 min	28.8 min	43.2 min	57.6 min	72 min
Hours wasted by 1 instance per year	87.6 hrs	175.2 hrs	262.8 hrs	350.4 hrs	438 hrs
Dollars wasted by mid-size company (1K Instances per year)	$50.07K	$100.14K	$150.21K	$200.28K	$250.36K
Dollars wasted by large size company (10K Instances per year)	$500.77K	$1.00M	$1.50M	$2.00M	$2.50M
Dollars wasted by X-Large size company (100K Instances per year)	$5.00M	$10.01M	$15.02M	$20.02M	$25.03M

Here are the assumptions I have used for our calculation:

1. Midsize enterprise would have their application running on 1000 EC2 instances. Large size enterprises would have their application running on 10,000 EC2 instances. Very large enterprises would have their application running on 100,000 EC2 instances.
2. For our calculation, I assume these enterprises are running on t2.2x.large 32G RHEL on-demand instances in US West (North California) EC2 instances. Cost of this type of EC2 instance is $ 0.5716/hour.

From all the below graphs you can notice the amount of money midsize, large size and very large size enterprise would be wasting due to garbage collection:

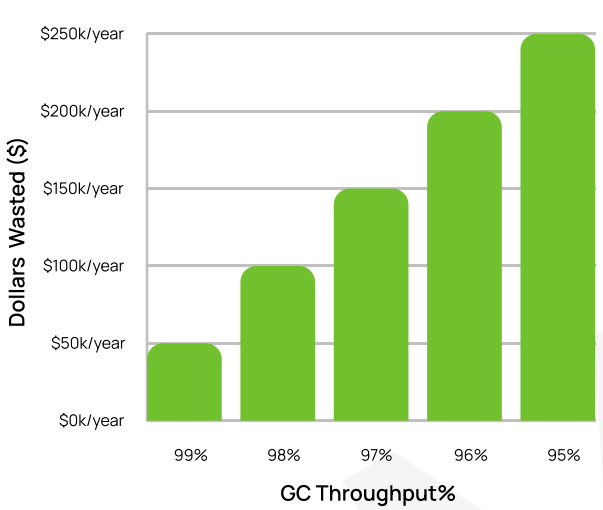


Fig 1: Money wasted by midsize enterprise due to Garbage Collection

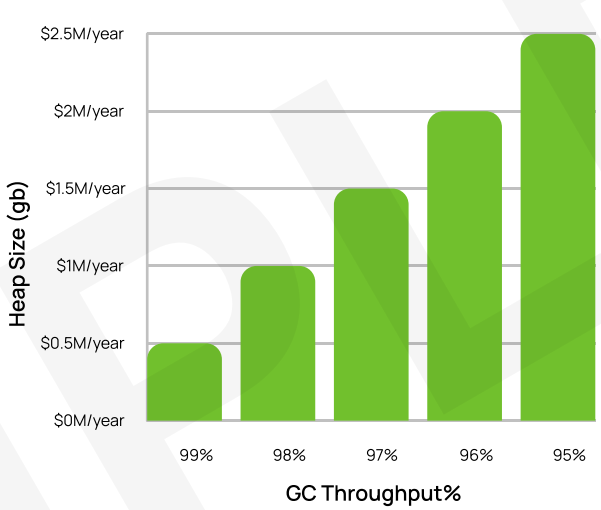


Fig 1.1: Money wasted by large size enterprise due to Garbage Collection

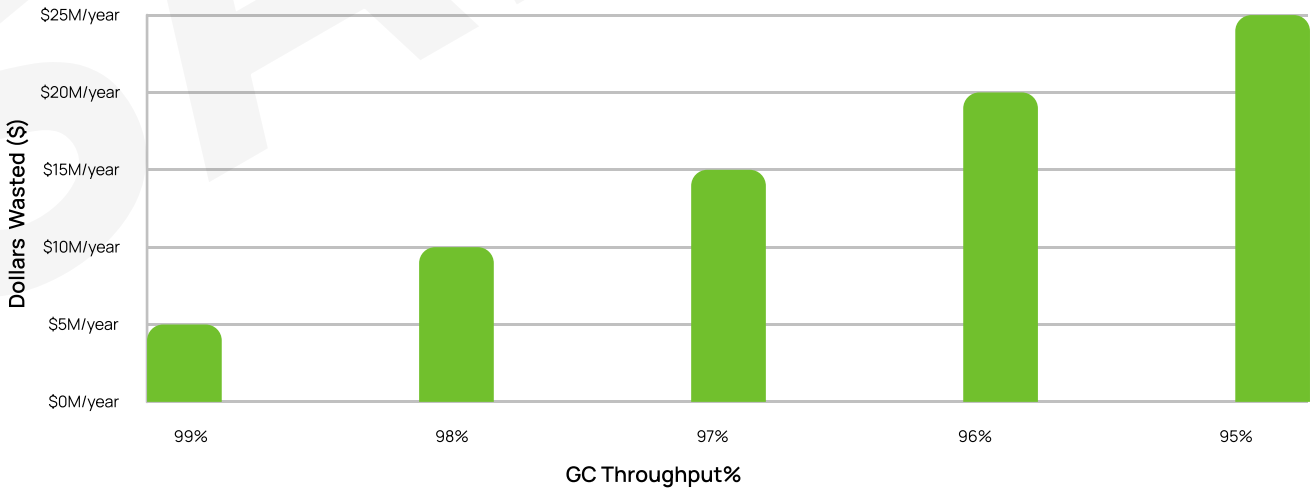


Fig 1.2: Money wasted by very large size enterprise due to Garbage Collection

Note 1: Here I have made calculations with assumptions GC throughput ranges only from 99% to 95%, several applications tend to have much poorer throughput. In such circumstances the amount of dollars wasted will be a lot more.

Note 2: I have used t2.2x.large 32G RHEL instance for calculation. Several enterprises tend to use machines with much larger capacity. In such circumstances, the amount of dollars wasted will be a lot more.

Counter arguments

Following are the counter arguments that can be placed against this study:

1. For my study I have used AWS EC2 on-demand instances, rather I could have taken dedicated instances for my calculations. Difference between on-demand and dedicated instances is only approximately 30%. So, the price point can fluctuate only by 30%. Still 70% of the above cost is outrageous.
2. Another argument can be AWS cloud is costly, I could have used some other cloud provider or bare metal machines or serverless architecture. Yes, these all are valid counter arguments, but they will shift the calculation only by a few percentages. But the case that garbage collection is wasting resources cannot be disputed.

You are open to articulate any other counter arguments in the comments section. I will try to respond to it.

Conclusion

In this post I have presented the case on how an exorbitant amount of money is wasted due to garbage collection. Unfortunate thing is: money is wasted even without our awareness. As applications developers/managers/executives we can do the following:

1. We should try to tune garbage collection performance , so that our applications starts to spend very less time on Garbage collection.
2. Modern applications tend to create tons of objects even to service simple requests. Here is our case study which shows the amount of memory wasted by the well celebrated spring boot framework. We can try to write efficient code, so that our applications tend to create very less number of objects to service the incoming requests. If our applications create a smaller number of objects, then very less garbage needs to be evicted from memory. If garbage is less, the pause time will also come down.

How To Get The Complete Ebook

To get the complete ebook and many other goodies. Buy JVM training course and become JVM expert today!

Checkout Course

Get a certificate on completing the course