**How to Reduce Long GC Pauses**

Garbage collection is necessary, but it can be a performance killer if not done well. Take these steps to make sure your GC pauses are minimal and short.

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 lead to long GC pauses as well as potential solutions to solve them.

## 1. High Object Creation Rate

If your application’s object creation rate is very high, then to keep up with it, the garbage collection rate will also be very high. A high garbage collection rate will increase the GC pause time as well. Thus, optimizing the application to create fewer 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 the object creation rate in the application, you can consider using Java profilers like [JProfiler](https://www.ej-technologies.com/products/jprofiler/overview.html" \t "_blank), [YourKit](https://yourkit.com/" \t "_blank), or JVisualVM. These profilers will report:

* What are the objects that have been 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 that occupy the most amount of memory. Go after the big fish in the pond.

### ****Tidbit: How to Figure Out the Object Creation Rate****

Upload your GC log to the Universal Garbage Collection log analyzer tool [GCeasy](http://gceasy.io/" \t "_blank). 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. See the image (which is an excerpt from the [GCeasy](http://gceasy.io/" \t "_blank)-generated report) showing the ‘Avg creation rate’ to be **8.83 mb.sec**.

## 2. Undersized Young Generation

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

**-Xmn**: specifies the size of the young generation.

**-XX:NewRatio**: Specifies the size of the young generation relative to the old generation. For example, setting -XX:NewRatio=2 means that the ratio between the old and young generation is 1:2. The young generation will be half the size of the overall heap. So, if the heap size is 2 GB, then the young generation size would be 1 GB.

## 3. Choice of GC Algorithm

Your GC algorithm has a major influence on the GC pause time. If you are a GC expert or intend to become one (or someone on your team is a GC expert), you can tune GC settings to obtain optimal GC pause time. If you don’t have a lot of GC expertise, then I would recommend using the G1 GC algorithm because of its **auto-tuning** capability. In G1 GC, you can set the GC pause time goal using the system property ‘-XX:MaxGCPauseMillis.’ Example:

-XX:MaxGCPauseMillis=200

As per the above example, the maximum GC pause time is set to 200 milliseconds. This is a soft goal, which JVM will try it’s best to meet it.

## 4. Process Swapping

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

Below is the script obtained from [StackOverflow](http://stackoverflow.com/questions/479953/how-to-find-out-which-processes-are-swapping-in-linux" \t "_blank) (thanks to the author) that, when executed, will show all the processes 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 processes are swapping, then do one of the following:

* Allocate more RAM to the serve.
* Reduce the number of processes that running on the server so that it can free up the memory (RAM).
* Reduce the heap size of your application (which I wouldn’t recommend, as it can cause other side effects. Still, it could potentially solve your problem).

## ****5. Tuning the Number of GC Threads****

For every GC event reported in the GC log, user, sys, and real times 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 this article](https://blog.gceasy.io/2016/04/06/gc-logging-user-sys-real-which-time-to-use/). (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 less than the ‘user’ time, then it might be indicating that there aren’t enough GC threads. Consider increasing the GC thread count. Suppose ‘user’ time is 25 seconds and you have configured the GC thread count to be 5, then the 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 take away resources from your application. Thus you need to conduct thorough testing before increasing the GC thread count.

## 6. Background IO Traffic

If there is 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 it can cause your application to suffer from long GC pauses. Here is a brilliant [article from LinkedIn Engineers](https://engineering.linkedin.com/blog/2016/02/eliminating-large-jvm-gc-pauses-caused-by-background-io-traffic) that walks through this problem in detail.

When there is heavy I/O activity, you will notice the ‘real’ time to be significantly higher than ‘user’ time. Example:

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

When this happens, here are some potential solutions to solve it:

* If high I/O activity is caused by your application, then optimize it.
* Eliminate the processes that are causing high I/O activity on the server.
* Move your application to a different server where there is less I/O activity.

### ****Tidbit: 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

The above command reports the reads/sec and writes/sec made to the device every 1 second. For more details on the ‘sar’ command, refer to [this tutorial](http://www.linuxtechi.com/generate-cpu-memory-io-report-sar-command/).

## 7. System.gc() Calls

When [System.gc()](https://docs.oracle.com/javase/7/docs/api/java/lang/System.html" \l "gc()" \t "_blank) or [Runtime.getRuntime().gc()](http://docs.oracle.com/javase/6/docs/api/java/lang/Runtime.html" \l "gc()" \t "_blank) method calls are invoked, it will cause stop-the-world full GCs. During stop-the-world full GCs, the entire JVM is frozen (i.e. No user activity will be performed during this period). System.gc() calls are made from one of the following sources:

1. Your own developers might be explicitly calling the System.gc() method.
2. It could be third-party libraries, frameworks, or sometimes even application servers that you use. Any of those could be invoking the System.gc() method.
3. It could be triggered from external tools (like VisualVM) through the 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 for it, 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 and solution [refer to this article](https://blog.gceasy.io/2016/11/22/system-gc/).

### ****Tidbit: How to Know Whether System.gc() Calls Are Explicitly Called?****

Upload your GC log to the Universal Garbage Collection log analyzer tool [GCeasy](http://gceasy.io/" \t "_blank). 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](http://gceasy.io/" \t "_blank)-generated report), showing that System.gc() was made four times during the lifetime of this application.

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