UNIVERSITAT DE BARCELONA

Trabajo de Fin de Grado

Grado de Ingeniería Informática

Facultad de Matemáticas

Universitat de Barcelona

Estudio y análisis del Garbage Collector de la Máquina Virtual de Java

---

Autor: Fernando Padilla Sánchez

Director: Lluis Garrido Ostermann
Realizado en: Departament de Matemàtiques i Informàtica
Barcelona, 26 de enero de 2017

Agradecimientos,

A mis padres por por apoyarme a seguir siempre hacia delante.

A Alejandro, por estar ahí desde que somos pequeños.

A Gustavo, por ayudarme a idear este TFG. Uno de los amigos que me llevo de la Universidad.

A todos los compañeros con los que he compartido malos y buenos momentos durante la carrera.

Y a Lluis Garrido, mi tutor, por animarme y apoyarme durante estos meses.

Gracias a todos

Contenido

Abstract

In some programming languages is well known they have an automatic memory management, like Java and Python, in spite of C, where the programmer manages that. Precisely Java has an internal proccess, known as the Garbage Collector, available in the Java Virtual Machine.

Garbage Collector is responsible to freeing memory while the execution is on, when objects are being created and allocated to one memory, Heap.

Some applications have been developed with a unic goal, study, analize and try to optimize the results obtained by the GC. To achieve that, there has been a study about GC and how does it work.

The report has confirmed that applications with a medium-large data set running on a multicore machine, Parallel GC is the one GC with better results as long as the generations are kept balanced.

OpenJDK has some internal tools, and those tools have been used to test the GC, to monitor CPU usage and its threads, as well as the number of allocations made.

Resumen

En distintos lenguajes de programación es bien conocido que tienen una gestión de memoria automatizada, como lo es Java o Python, en comparación con C, que la realiza el mismo programador. En concreto Java tiene un proceso interno, propio de la Java Virtual Machine, conocido por Garbage Collector.

GC es el encargado de liberar memoria durante la ejecución de una aplicación, cuyos objetos se han creado en la memoria propia de JVM, Heap.

Se han desarrollados distintos programas con tal de analizar el funcionamiento del GC, estudiar, analizar e intentar de optimizar los resultados obtenido por el GC. Se ha realizado un estudio previo sobre los fundamentos del GC de la JVM, propiedad de Oracle, en concreto la Java Sun HotSpot.

El análisis de los datos ha demostrado que en aplicaciones con sets de datos medios corriendo bajo una máquina multicore, el GC en paralelo es el que consigue mejores tiempos con un balance de las distintas generaciones, encargadas de almacenar los objetos creados y desechados, en proporción al tamaño total del Heap.

Se han utilizado las propias herramientas disponibles en el OpenJDK de Java 8, VisualVM, un software libre para monitorizar el uso de CPU y threads, así como la cantidad de allocations que se realizan en tiempo de ejecución.

Resum

En diferents llenguatges de programació es coneix que existeix una gestió de memoria automàtica, Python o Java en són dos exemples, en comparació amb C, que la gestió es responsabilitat del programador. En concret Java té un procès intern propi de la Java Virtual Machine, conegut com el Garbage Collector.

GC es l'encarregat d'alliberar memòria durant l'execució d'una aplicació, que crea objectes i aquest reserven espais de memòria del Heap.

S'han desenvolupat diferents programes per tal de d'analitzar el funcionament del GC, estudiar i intentar optimitzar els resultats obtinguts pel GC. Per assolrir aquest objectiu, s'ha realitzat un estudi previ sobre els fonaments del GC disponible al JVM, en concret la Java Sun HotSpot.

L'anàlisis de les dades ha demostrat que en aplicacions amb set de dades d'un tamany mitj-gran , i utilitzant el ParallelGC en una màquina multicore, és la opció que aconsegueix millors resultats, sempre i quan el tamany de les generacions es mantinguin en proporció.

S'han utilitzat les propies eines disponibles al OpenJDK de Java 8 i VisualVM, un software lliure per observar l'ús de CPU i els threads, aíxi com la quantitat d'asignacions que es realitzen en temps d'execució.

1. Introducción

En distintos lenguajes de programación es bien conocido que tienen una gestión de memoria automatizada, como lo es Java o Python, en comparación con C, que la realiza el mismo programador. En concreto Java tiene un proceso interno, propio de la Java Virtual Machine, conocido como Garbage Collector (GC).

GC es el encargado de liberar memoria durante la ejecución de una aplicación, cuyos objetos se han creado en la memoria propia de *JVM*, *Heap*.

Se han desarrollados distintos programas con tal de analizar el funcionamiento del GC, estudiar e intentar optimizar los resultados obtenido por el GC. Se ha realizado un estudio previo sobre los fundamentos del GC de la JVM, propiedad de Oracle, en concreto la Java Sun HotSpot.

1.1 Ámbito el proyecto

Este TFG relaciona varias asignaturas del Grado de Ingeniería Informática con Java. La asignatura que introdujo y enseñó las bases de Java, Programación I, seguida de Programación II, centrada en la programación orientada a objetos (POO). Se adquirieron más conocimientos con otras asignaturas también con uso de Java, como es Diseño de Software y Software Distribuido.

1.2 Motivación

La gestión de memoria es un aspecto a tener muy cuenta debido a la cantidad de configuraciones que existen entre RAM y CPU. Un estudiante sin conocimientos del Garbage Collector desconoce que está pasando internamente en la JVM, ni si los recursos disponibles están siendo aprovechados.

Conocer como funciona internamente la gestión de memoria de Java cuando se desarrolla una aplicación ayudará a entender mejor Java y a mitigar problemas que surjan en futuros desarrollos.

1.3 Objetivos generales

El objetivo del TFG es utilizar los conocimientos adquiridos del GC para realizar un análisis de los datos obtenidos con tal de buscar fallos en su definición, aprender el funcionamiento del GC y optimizar, si es posible, los tiempos de ejecución.
El GC se ha observado en el sistema operativo Ubuntu (con kernel de Linux), arquitectura de 64 bits.

1.4 Objetivos específicos

Los objetivos que se pretenden realizar en este proyecto constaran de las siguientes partes:

- Entender el funcionamiento del *Parallel Garbage Collector*. En concreto ha sido este PGC ya que la máquina donde se han realizado los tests por defecto utilizaba dicho GC.
- Estudiar las distintas herramientas para extraer datos del GC.
- Ejecutar diversas aplicaciones desarrolladas con tal de analizar el funcionamiento del GC.
- Conclusiones y resultados
- Tests modificando parámetros del heap.

1.5 Organización de la memoria

La memoria del proyecto está formada por las siguientes partes:

1. Antecedentes: introducción de la JVM del GC y las herramientas con las que se puede analizar.
2. Análisis de datos: análisis de los datos obtenidos después de largas horas de ejecución acumulando datos del funcionamiento del GC. A continuación se detallan las herramientas usadas para el análisis:
    - *VisualVM*, herramienta para observar en tiempo de ejecución el uso de *Central Processing Unit* (*CPU*), *Threads*, o la cantidad de memoria reservada por objetos.
    - *Jstat*, herramienta interna de la JVM con la que se pueden obtener datos *raw* sobre el GC.
    - *Jmap,* herramienta parecida a Jstat pero se centra en el muestreo de la *CPU*.
    - Máquina Ubuntu que ha realizado la ejecución de los programas y la posterior extracción de datos.

3. Resultados: después de analizar por separado cada aplicación, se observarán los resultados.
4. Conclusiones: valoración personal de este proyecto y que aspectos pueden aplicarse en el futuro.

2 Tecnologias utilizadas

2.1 Java Virtual Machine

Java es un lenguaje de programación concurrente, basado en clases y orientado a objetos. *Write Once, Run Everywhere (WORA),* es la principal característica que ofrece este lenguaje, ya que permite, gracias a la JVM, ejecutar su código compilado independientemente del sistema operativo.

La JVM (*Java Virtual Machine*) es la piedra angular de la plataforma de *Java,* componente software responsable de independizar el lenguaje sea cual sea el *hardware* o SO bajo el que se ejecute. Lo más importante de la JVM es el disminuido tamaño de su código compilado y la estabilidad que ofrece al proteger a los usuarios frente a *malware*.
Tal y como cualquier sistema operativo, tiene su set de instrucciones y puede manipular varias áreas de memoria en tiempo de ejecución.

Sun Microsystems presentó *Java HotSpot Virtual Machine* como una motor de alto rendimiento enfocado para el lenguaje *Java*. El primer prototipo de JVM (implementado por Sun Mycrosystems) emulaba un set de instrucciones hospedado en un dispositivo que se asemejaba a un smartphone actual. Actualmente las implementaciones de *Oracle* emulan esta *JVM* en móviles, ordenadores de escritorios y dispositivos remotos, aunque la *JVM* no asuma ninguna implementación tecnológica, de tipo servidor o sistema operativo. No es del todo interpretado, pero puede ser implementado por un conjuntos de instrucciones compiladas en *CPU*.

Las mejoras que introdujo Java HotSpotVM incluían compilación dinámica y optimizada, mejor sincronización de *threads* y una mejor gestión de memoria para el GC.

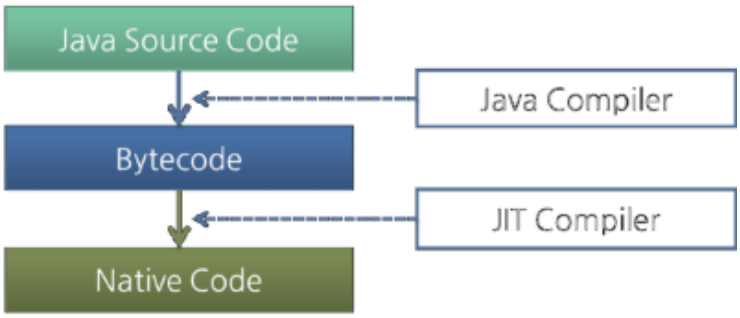


Ilustración 1: Recorrido Java, Bytecode y código nativo

Curiosamente la Máquina Virtual de Java desconoce el lenguaje, interpreta su formato binario, el archivo en formato **.**class. Dicho archivo contiene instrucciones de la JVM (o *bytecode*), una tabla de símbolos además de información auxiliar. Es gracias a estos archivos que la *JVM*, juntamente con la *API* (Application Programming Interface) de Java es capaz de ejecutar el programa. El proceso de traducción que realiza se explica en el apartado Bytecode.

Las principales características de la *Java HotSpot Virtual Machine* son las siguientes:

- Máquina virtual basada en pila.
- Referencias simbólicas.
- *Garbage Collection.*
- Garantiza independencia a plataforma definiendo su datos primarios primitivos.
- Network byte order.

2.2 Heap

Heap es un espacio de memoria que se comparte entre todos los *threads* de JVM. Es un espacio de uso en tiempo de ejecución donde las instancias de las clases y *arrays* son alojadas.

Heap no existe permanentemente, se crea cuando se inicia la JVM. El almacenamiento para los objetos se reclama automáticamente por el *GC*, según los parámetros que tenga establecidos por defecto, basados en las características de la máquina, *CPU* y la *RAM* (*Memory Access Memory*). El espacio *Heap* puede ser de un tamaño fijo, expandido o reducido según se necesite durante el proceso del GC.

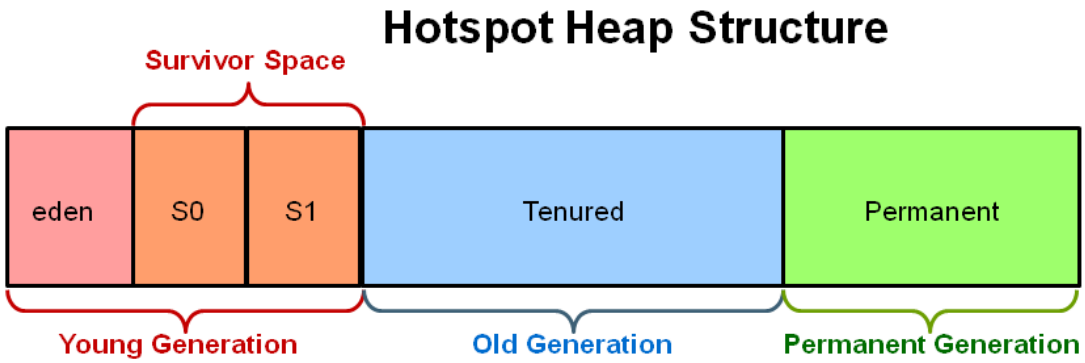


Ilustración 2: Estructura Heap de la JVM

En el momento de inicialización de la aplicación, se reserva la máxima capacidad de memoria disponible, virtual y física. El espacio físico puede llegar a usar la parte virtual si fuera necesario.

Los eventos GC ocurren cuando las generaciones, las tres que se pueden observar en la Ilustración 2, ocupan todo su espacio o ciertos objetos son candidatos a ser copiados a otra generaciones. El rendimiento es inversamente proporcional a la cantidad de memoria disponible.

Por defecto, JVM aumenta o disminuye el tamaño del heap en cada GC para intentar mantener una proporción entre el espacio disponible y los objetos *alive* (aquellos que siguen referenciados) en cada evento GC dentro de un rango establecido por los siguientes parámetros:

- MinHeapFreeRatio
- *MaxHeapFreeRatio*
- El tamaño total del heap está limitado por -*Xms* y -*Xmx*.

Cuando se altera el tamaño del Heap, la JVM debe recalcular la OG y la YG para mantener un equilibrio fijado por *NewRatio* (proporción de tamaño entre YG y OG).

Tal y como se ha comentado con anterioridad, Heap está formada por tres generaciones (*Ilustración 2*): *Young Generation*, *Old Generation* o *Tenured Space* y *Permanent Generation*.

- La primera generación, YG, consiste en la zona Eden y dos zonas supervivientes. Eden Space es responsable de almacenar los objetos creados en el instante inicial de cualquier aplicación, mientras que S0 y S1 van alternando estos objetos en su espacio.
- OG se encarga de almacenar los objetos más pesados y con más edad.
- La tercera y última generación, PG, almacena información para la JVM, aunque también almacena métodos y objetos descriptivos de clases.

La JVM, aumenta o disminuye en cada evento GC el tamaño del Heap para mantener una proporción entre el espacio libre y los objetos vivos en un rango especificado.

2.3 Java Bytecode

Java esta diseñado para correr en cualquier máquina virtual sin depender de la máquina física para implementar *WORA*(*Write Once Run Anywhere*), y para conseguir tal implementación, la JVM hace uso del *Java Bytecode,* un lenguaje intermediario entre el lenguaje Java y el código máquina. A diferencia del lenguaje C, cuyo lenguaje es lo más parecido al código máquina. En Java se traduce a código máquina mediante bytecode para que el sistema lo pueda comprender.

Ya que Java Bytecode no depende de ninguna plataforma, funciona en cualquier SO siempre y cuando la JVM haya sido instalada. Cualquier clase *.java* desarrollada y compilada puede ser usada en cualquier máquina Windows, MacOS o Ubuntu.

El tamaño del código compilado es casi idéntico al tamaño del código original, facilitando así la transferencia y ejecución del código compilado a través de red.

Una peculiaridad sobre Java HotSpot es que su nombre tiene un origen y está relacionado con el código nativo. Si en la implementación de nuestro código tenemos un loop o un método que se ejecuta de media 1500 veces, Java recoge la parte de código donde ha detectado esa acumulación de código y lo compila a código nativo, para ahorrar tiempo en compilarlo en Java Bytecode.

Existe la opción de convertirlo a Java Bytecode, pero el rendimiento en lenguaje de máquina es superior, a pesar de que la independencia respecto a plataformas es mejor en Bytecode.

2.4 Gargage Collector (GC)

Java Memory Management, juntamente con los múltiples algoritmos Garbage Collector que tiene disponible, es uno de los logros más importantes en un lenguaje de programación. Permite a los desarrolladores crear nuevos objetos sin la preocupación explicita de reservar memoria y eliminarla, y también de evitar *leaks.* Ocurren cuando un bloque de memoria reservada no ha sido liberada en un programa de computación de memoria.

A pesar de que sin duda aportan muchísimo al desarrollo de aplicaciones, los GC son conocidos por ser lentos e intrusivos, incluso provocando en ciertas ocasiones largas paradas durante la ejecución de un programa, hecho que se ha podido observar durante la ejecución de los programas realizado en este TFG. Se estima que la JVM puede gastar alrededor de hasta un 60% del tiempo de gestión de memoria, en el peor de los casos.

La última versión de JVM mencionada anteriormente, *Sun's HotSpot VM*, es el resultado de intentar mitigar esos tiempos con múltiples esquemas de GC, usados en distintas generaciones con el fin de reducir el tiempo de parada provocada por los GC.

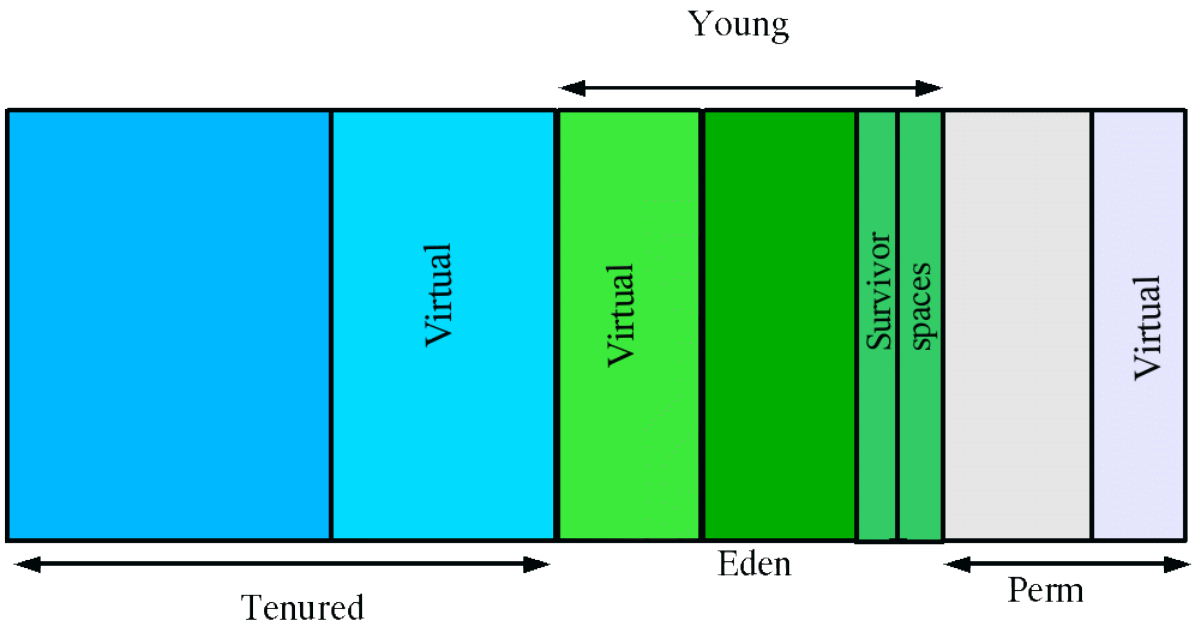


Ilustración 3: Estructura heap para Parallel Collector

Existen hasta cuatro tipos de Garbage Collector:

1. **Serial Collector**: utiliza un sólo *thread* para realizar todo el proceso GC, relativamente eficiente desde que no hay gastos generales de comunicación entre *threads*. Más eficiente en máquinas con un solo procesador, ya que no puede aprovechar las ventajas de un *hardware* multicore, aunque se pueda usar para aplicaciones con pequeños sets de datos (aproximadamente de 100MB). *SerialCollector* se usa por defecto en ciertas configuraciones de *hardware* en determinados sistemas operativos, aunque también cualquier usuario puede escoger utilizar este *GC*. El comando para activar este GC es el siguiente:

   -XX:+UseSerialGC

2. **Parallel Collector**: hoy en día es común trabajar sobre máquinas con CPU multicores y con mucha memoria disponible. También conocido por T*hrouhgput Collector,* fue desarrollado para aprovechar la todos los cores disponible antes que dejarlos sin usar. Es capaz de reducir el gasto computacional. Enfocado para aplicaciones con un set de datos medio-largo. Existe una diferencia en la estructura interpretada por el Parallel Garbage Collector en el heap, Ilustración 6.

   -XX:+UseParallelGC

3. **Parallel Compacting Collector**: disponible el actualización 6 en J2SE 5.0 La principal diferencia con PC es que usa un nuevo algoritmo para la OG. A la larga, PCC reemplazará a PC.

   -XX:+UseParallelOldGC

4. **Concurrent Mark-Sweep Collector**: disponible el actualización 6 en J2SE 5.0, fue diseñado para aplicaciones que priorizan menos tiempos de GC y que puedan compartir los recursos de los procesadores con el GC mientras que la aplicación se esta ejecutando. Normalmente las aplicaciones que tienen un set de datos con un historial de vida en OG relativamente amplio se suelen beneficiar de este GC. Sin embargo, este GC se debería considerar para cualquier aplicación con unos tiempos bajos de parada; por ejemplo, se han obtenido buenos resultados con aplicaciones interactivas con tamaños de OG moderados con un simple procesador, especialmente usando el modo incremental.
   El comando para activar este GC es:

   -XX:+UseConcMarkSweepGC

JVM escoge por defecto el Garbage Collector que mejor le convenga según las prestaciones de la máquina donde se va a ejecutar.

2.4.1. Funcionamiento Garbage Collector

A pesar de que se tenga la idea de el GC recolecte y se deshaga de objetos, la realidad es otra. Los objetos considerados *alive* son los que realmente se rastrean y todo el resto se clasifica como basura. Objetos que ya no son referenciados, se ven obligados a devolver la memoria que ocupan.

- Una *allocation* simplemente pide una parte del array de memoria y mueve el puntero offset hacia delante. La siguiente *allocation* comienza en el offset y vuelve a demandar una porción de la tabla.

- Cuando un objeto ya no se usa, el GC reclama ese trozo de memoria y lo usa para futuras *allocations*. Esto significa que no hay borrado de objetos y ningún trozo de memoria es devuelto al sistema operativo.

Todos los objetos alojados en el área denominada heap están controlados por la JVM. Cada objeto que el desarrollador usa es tratado de la misma manera, incluyendo los objetos clases, las variables estáticas, incluso el código. Una vez el objeto ya no está referenciado y no pueda ser usado por el mismo código, el GC lo remueve y reclama la memoria ocupada por éste.

2.4.2. Objetos candidatos para el GC

Cuando se produce un evento GC, no todos los objetos se consideran iguales, hay diferencias entre aquellos que pueden ser candidatos o no.
Cualquier objeto que cumpla este requisito será candidato:

- Aquellos objetos cuyas referencia sean *null*.

2.4.3. El método Write Barrier

Antes de entrar en el funcionamiento del GC, es necesario conocer el método *Write Barrier.* Es necesario para el GC consultar una porción del heap cada vez que hay un ciclo. Para lograr esto la JVM tiene que implementar los siguientes métodos.

Existen dos tipos de esta escritura implementadas en HotSpot: *dirty card* y S*napshot at the Beginning (SATB)*.

El principio de este sistema es sencillo, cada vez que un programa modifica alguna referencia en memoria, debería marcar esa página como modificada o *dirty.* Para que esto funcione hay disponible una tabla llamada *card table* en la JVM donde cada página de 512 bytes de memoria está asociada a un *card table*.

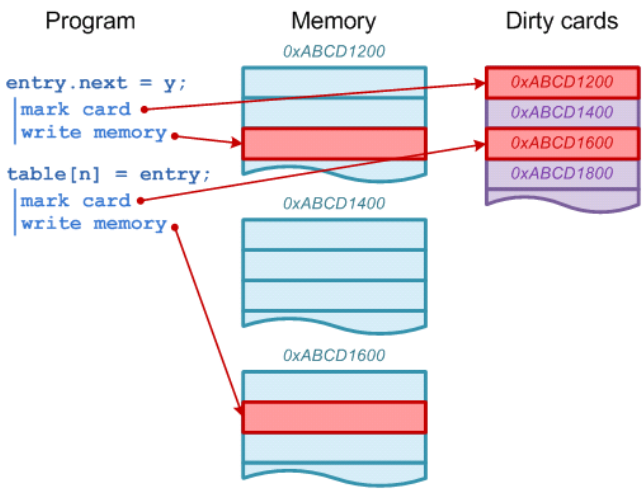


Ilustración 4: Procedimiento Write Barrier

2.4.4. Orden de procesamiento de cada zona

La mayoría de nuevos objetos creados se almacenan en el área de Eden, aunque los objetos de mayor peso pueden pasar directamente a la OG. Cuando este espacio se llena con objetos, se ejecuta un Minor GC y todos los objetos que han sobrevivido se mueven a una de las dos zonas de supervivientes, S0 o S1. Todos estos objetos tienen una edad que indicará el número de GC que han sobrevivido, ya que a medida que pasan los ciclos de GC, estos objetos envejecerán y serán candidatos a cambiar de espacio, hacia la OG. Esto se controla con un valor en concreto, TT y MTT (*Maximum Tenuring Threshold*). Cuando se predice que el objeto sobrepasará el valor de TT u ocupa demasiado, el objeto se copia a OG.

El funcionamiento es senzillo, el GC habría encontrado datos que hubieran superado el límite de edad establecido por TT, así que JVM reducirá el TT para forzar a ciertos objetos a cambiar a la OG, obligando al GC a realizar Minor GC. DSS disminuye su capacidad máxima a medida que el uso de los distintos espacios disminuyen también.

Un datos importante es que unas de las dos zonas, S0 o S1, permanecerán vacías. Si existen datos en ambas zonas, o el uso de las dos es 0, es un señal de que algo no está funcionando correctamente.

2.4.5. Algoritmo de recolección en Young Generation (Minor GC)

Anotar primero que el algoritmo usado en esta generación se enfoca más en la velocidad de su procesamiento.

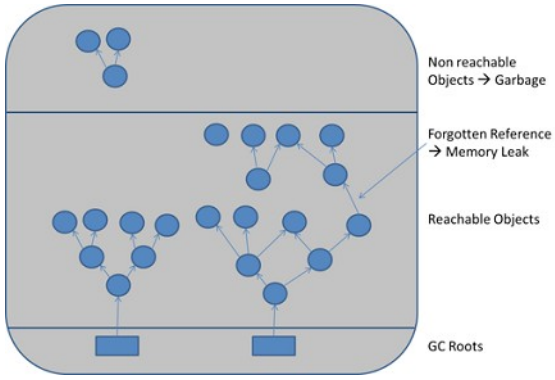


Ilustración 5: Recolección de objetos

Parallel Collector hace uso de un algoritmo paralelo de YG utilizado por *Serial Collector.* Es un algoritmo tipo *stop-the-world* y copiador, pero su rendimiento recolectando en paralelo decrece, aun usando múltiples CPUs aumenta su interferencia con la aplicación.

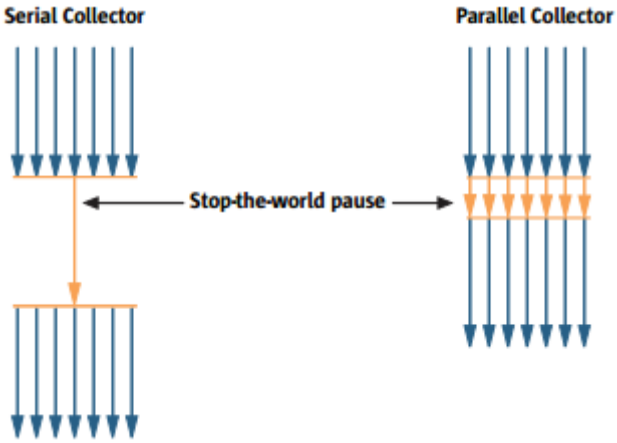


Ilustración 6: Ilustración Serial Collector frente Parallel Collector.

Para llevar acabo los Minor GC en esta generación, es necesario que el GC encuentre todas las referencias roots, disponibles en OG, Ilustración 9. Minor GC considera las referencias roots como referencias hacia la pila.

Normalmente la recolección de todas las referencias de la OG necesitarán un escaneo a través de los objetos en la OG. Por eso mismo se necesita el método *Write Barrier.* Todos los objetos en YG han sido creados o copiados desde la última reinicio de *Write-Barrier*, así que las paginas non-dirty no pueden tener referencias en la YG. De esta manera se puede escanear objetos en dirty-pages, enfocando así el algoritmo en velocidad.

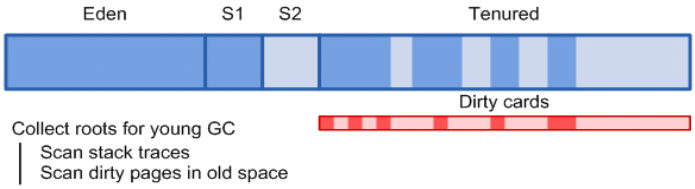


Ilustración 7: Primera iteración Write Barrier

Una vez tenemos el set de referencias listo, las *dirty cards* se resetean y la JVM comienza a copiar todos los objetos vivos desde Eden Space a una de las dos zonas. El único requisito de la JVM es gastar tiempo en los objetos vivos.

Mientras que la JVM esta actualizando las referencias a los objetos realojados, las páginas de memoria se marcan otra vez, para asegurarnos de que en el siguiente YGC solo las dirty pages tienen referencias hacia la Young Space.

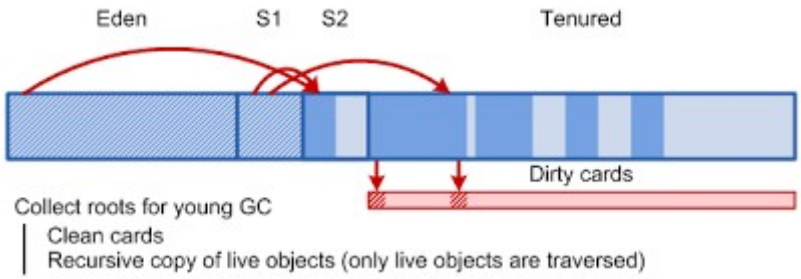


Ilustración 8: Tercera iteración Write Barrier

Finalmente tenemos el espacio Eden y una de las zonas Survivor vacías (y lista para ser referenciada) y una zona superviviente llena con objetos.

Ilustración 9: Fase final Write Barrier

2.4.6 Old Generation

Esta generación contiene los objetos cuyo tamaño son grandes, por lo tanto, pasan directamente a OG. Los objetos, después de sobrevivir a múltiples eventos GC, ven incrementad su *edad generacional* lo suficiente como para ser copiado a esta generación. Los eventos que ocurren en este espacio se denominan Major GC o Full GC.

2.4.7. Major GC

Una de las primeras diferencias respecto a Minor GC que diferencia a Major GC es que su ejecución suspende la JVM, lo que se traduce en un empeoro en el rendimiento, y se debería evitar a toda costa. Mientras que todos los eventos Major GC recolectan objetos de la Old Generation, no todas las Old Generation GC son Major Collection. Esta diferencia se debe a que Major GC y Full GC son eventos muy similares.

Full GC recolecta todas las áreas del heap, incluyendo la Young Generation, y en el caso de la Oracle HotSpot, la Permanent Generation. Para hacer más hincapié, realiza su tarea durante el evento conocido como *stop the world event.*

2.4.8. Stop the World Event

Todos los GC son eventos *Stop the World* (SWE) dado que todos los *threads* de las aplicaciones se pararían hasta que las operaciones correspondientes se parasen. Dado que la YG guarda objetos con baja edad, Minor GC es muy rápida en procesarlos y la aplicación no se ve afectada por eso. Sin embargo, Major GC/Full GC ocupa más tiempo y por consiguiente, suspende la aplicación, llegando a durar segundos o inclusos minutos.

2.4.9. Full GC

El concepto de FullGC y Major GC no está definido *per se,* ya que existen ciertos casos en el que FullGC no limpia el heap entero o Major GC si que se ejecuta en la totalidad del heap. Sin embargo si se pueden diferenciar en un aspecto importante:

1. Major GC limpia la Old Generation. Se activan por culpa de Minor GC.
2. Full GC limpia el heap entero, incluyendo YG, OG y PG.

Muchos GC realizan la limpieza de OG parcialmente, así que el término *collector* es parcialmente correcto.

2.4.10. Permanent Generation

La JVM tiene una representación propia de los objetos y clases Java, siendo entre ellos muy similares. Tanto que durante un proceso de GC ambos son vistos como objetos y se recolectan de la misma manera. Sin embargo, ambos objetos se distribuyen por espacios distintos.

Hay dos explicaciones que justifican esta distribución.

- La primera es que las clases son parte de la implementación de la JVM y no se debería rellenar el heap con ese tipo de estructuras.
- Originalmente no existía esta generación. Objetos y clases estaban juntos. Se observó que los *class loaders* no se usaban demasiado y se decidió establecer un espacio nuevo. Con este cambio el rendimiento se mejoró notablemente. Aunque con la cantidad de cargas de *class loader* que existen en ciertas aplicaciones, todavía no se tiene claro su ganancia real.

PG cambió el paradigma del algoritmo de detección de objetos durante un GC. La misma generación contiene las referencias necesarias para conocer que tamaño y que contiene el objeto X. Antes de explicar este proceso en la PG, definir este característica de la PG.

- Objeto X: objeto Java copiado a la PG.
- Classe K del objeto X, responsable de proporcionar información sobre X.

Durante un evento GC es necesario tener una descripción del objeto Java, de su tamaño y contenido. Si se apunta a cualquier objeto X, se necesita su clase K para saber como es el objeto X. Con una PG, el GC primero visita dicha generación para saber las nuevas localizaciones de K antes de buscar estos objetos.

2.4.11. Contenido Permanent Generation

Si se pudiera inspeccionar el contenido de la PG, nos encontraríamos datos como los que así se describen:

- Métodos de clases (incluyendo su *bytecode*).
- Nombres de las clases (el forma de objeto que apunta al String también en generación permanente).
- Tabla de información constante (información extraída del archivo .class, por ejemplo una array de objetos que contienen referencias a métodos).
- Array de objetos y tipo de arrays asociadas con una clase(java/lang/Object o java/lang/exception)
- Objetos internos creados por la JVM.
- Información usada para la optimización de los compiladores (JITs).

3 Metodología y herramientas

A continuación se detallan en profundidad las herramientas que se han usado para la extracción y monitorización de datos. En mayor o menor medida todos los datos que se explican a continuación se han empleado para realizar el análisis del GC.

3.1 HPROF Tool

La plataforma de Java 2 (J2SE) siempre ha proporcionado una línea de comando para obtener datos llamada HPROF para el análisis del Heap y CPU.

HPROF es una librería nativa de la JVM que se carga dinámicamente a través de lineas de comando, en el momento de carga de la JVM, y se convierte en parte del proceso de la JVM. Los usuarios pueden pedir varios tipos de análisis. Los datos generados pueden estar en forma textual o binaria, y pueden ser usados para rastrear problemas de rendimiento que afecten a memoria a causa de un código mal diseñado.

HPROF es capaz de presenta el uso de la CPU, las estadísticas de *allocation* en el heap, y monitorizar perfiles de contención. Además, puede reportar un informe completo, volcados de memoria y los estados de todos los procesos y threads en la JVM.

3.2 Comandos HPROF

A continuación se presenta una lista de comandos disponibles en la herramientas HPROF. Si se quisiera obtener distinta información haciendo uso de HPROF, el comando a continuación invocaría cualquier de las descripciones incluidas en la primera columna de la Tabla 2.

javac -J-agentlib:hprof=heap=sites MainApplication.java

-J es necesario para indicar que se usará Java. En caso contrario, si se usará Java directamente, no haría falta añadir dicha instrucción.

Option Name and Value	Description	Default
heap=dump|sites|all	**heap profiling**	**all**
cpu=samples|times|old	**CPU usage**	**off**
monitor=y|n	monitor contention	n
format=a|b	**text(txt) or binary output**	**a**
file=<file>	write data to file	java.hprof[.txt]
net=<host>:<port>	send data over a socket	off
depth=<size>	stack trace depth	4
interval=<ms>	sample interval in ms	10

cutoff=<value>	output cutoff point	0.0001
lineno=y|n	line number in traces?	y
thread=y|n	thread in traces?	n
doe=y|n	dump on exit?	y
msa=y|n	Solaris micro state accounting	n
force=y|n	force output to <file>	y
verbose=y|n	print messages about dump	y

Tabla 2 Lista de comandos HPROF

Por defecto, la información obtenida del heap se almacena en el archivo java.hprof.txt en formato ASCII. Dentro de estos archivos de texto en formato .txt (comando format = a), encontramos la siguiente información. Véase la Ilustración 10.

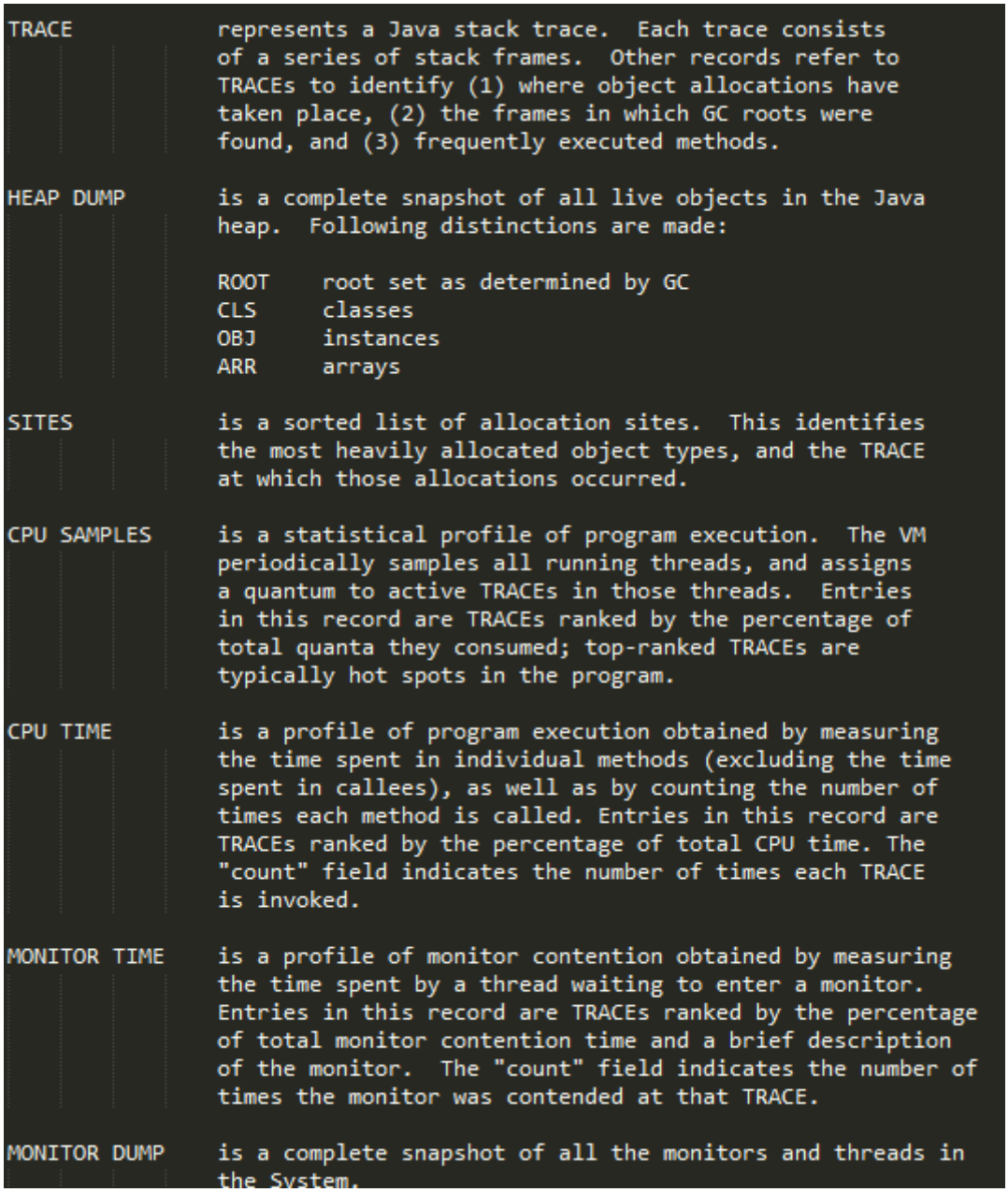


Ilustración 10: Salida HPROF tool

Los comandos que ofrecen más información en la herramienta HROF se presentan a continuación.

3.2.1. Heap Dump

Heap Dump genera un archivo java.hprof.txt que contiene la información acumulada (Ilustración 10) desde el inicio de la ejecución del programa hasta el instante del volcado.
Es posible analizar el archivo con VisualVM, software que se ha usado durante el trabajo y que posteriormente se explica.

El comando para obtener dicho archivo es el siguiente:

```
javac -J-agentlib:hprof=heap=dump YourProgram
```

3.2.2. CPU Usage Sampling Profiles

A continuación se presenta una porción del archivo generado por el compilador javac. Ofrece, por cada clase, el tiempo acumulado en CPU, *count* (cuántas veces se ha invocado) y trace, el *frame* correspondiente en la pila.
El comando para obtener este archivo es el siguiente:

javac -J-agentlib:hprof=cpu=samples MainApplication.java

Ejemplo de salida del programa realizado para este TFG

Rank	Self	Accum	Count	Trace	Class name
1	16,67%	16,67%	6	300107	java.lang.ClassLoader.defineClass1
2	5,56%	22,22%	2	300151	java.util.zip.ZipFile.read
3	2,78%	25%	1	3000055	java.util.jar.JarFile.initializeVerifier
4	2,78%	27,78%	1	300109	java.lang.ClassLoader.findBootstrapClass

Tabla 3 Uso de CPU

3.2.3. CPU Usage Times Profile (cpu=times)

HPROF es capaz de acumular el uso de CPU por cada hilo disponible. A continuación se presenta la Tabla 4 con la salida (reducida) del archivo en cuestión.

Rank	Self	Accum	Count	Trace	Method
1	32,21%	32,21%	11	303615	java.lang.Object.wait
2	32,21%	64,33%	690	303637	java.lang.ref.ReferenceQueue.remove
3	32,21%	66,69%	25259	312341	com.sun.tools.javac.file.ZipFileIndex$ZipDirectory.readEntry
4	32,21%	67,76%	665	313124	com.sun.tools.javac.util.List.reverse

Tabla 4 Muestra obtenida con el comando

javac -J-agentlib:hprof=cpu=samples MyProgram.java

3.3. Herramienta Jstat

Esta herramienta es una de las más completas y extensas que presenta la JVM de Sun Oracle, ya que es capaz de ofrecer los tiempos que duran los Minor GC o Major GC o de obtener el tamaño que se está ocupando en las distintas generaciones del heap. Algunas de sus opciones pueden imprimir estadísticas de comportamiento relacionadas con el uso del heap y sus capacidades.

A continuación se explica las instrucciones que se usan en el análisis de este TFG, con los datos que se pueden obtener y que significa cada uno.

El comando que se presenta a continuación contiene todas las opciones que se pueden cambiar. En total hay disponible seis y se presentan a continuación:

- *General Option*: un comando único general que indica que método usar.
- *Output Options*: una o más opciones de tipo output, además de las opciones -t, -h y -j.
  - -*t* se usa para añadir una columna de tiempo. Dicho tiempo es el tiempo total desde que se inició la JVM.
  - -*J* indica que se usará Java.
  - -*h n* añade una columna más cada *n* muestras, siendo n un entero positivo.

- Por defecto es 0, y muestra el encabezado de la columna sobre la primera fila de datos.

- *Vmid*: identificador de máquina virtual. Un string indicando el id del programa en ejecución.
- *Interval* [s/ms]: la unidad por defecto son ms. Podemos definir un lapso de tiempo en el que jstat tomará muestras del método que le indiquemos.
- *Count*: número de muestras que mostrar. El valor por defecto es infinito.

jstat [generalOption | outputOptions vmid [interval[s|ms] [count]]]

3.3.1 StatOption

La tabla a continuación muestra la lista las opciones disponibles. La columna Option muestra todas las opciones disponibles que tiene jstat disponible. Algunas muestran la misma información aunque pierden otras para mostrar datos como el total de eventos FGC y sus tiempos. Las opciones en negrita indican cuáles se han usado.

Las capacidades y usos de las generaciones se miden en KiloBytes.

Option	Displays
class	Estadísticas del comportamiento del class loader
compiler	Estadísticas del comportamiento del compilador Just in Time de HotSpot
gc	Estadísticas del comportamiento del garbage collector
gccapacity	Estadísticas de las capacidades de las generaciones y su espacio correspondiente.
gccause	Sumario de las estadísticas del Gargabe Collector
gcnew	Estadísticas del comportamiento de la nueva generación.
gcnewcapacity	Estadísticas de del tamaño de las nuevas generaciones y su correspondiente espacio.
gcold	Estadísticas del comportamiento del tamaño de la generaciones permanente.
gcoldcapacity	Estadísticas del tamaño de la generación antigua.
gcpermcapacity	Estadísticas del tamaño de la generación permanente.
gcutil	Sumario de las estadísticas del Garbage Collector.
printcompilation	Recopilación de las estadístacias de los métodos.

Tabla 5 Según la opción, la salida del archivo mostrará información distinta

3.3.1.1 Option GC

La tabla siguiente detalla por cada apartado que información nos muestra. GC es la opción de jstat más completa de toda las disponibles.

Command: **jstat -gc MainApplication 1520 1000**

Zona del heap	Descripción
S0C	Espacio actual del espacio superviviente 0.
S1C	Espacio actual del espacio superviviente 1.
SOU	Espacio utilizado en el Survivor 0.
S1U	Espacio utilizado en el Survivor 1.
EC	Capacidad del espacio Eden.
EU	Espacio usado en el espacio Eden.
OC	Capacidad actual de la Old Space.
OU	Espacio usado en la Old Space.
PC	Capacidad de la Perm Generation.
PU	Espacio usado de la Perm Generation.
YGC	Numero de GC eventos en la Young Generation.
YGCT	Tiempo acumulado del Gargabe Collection en la Young Generation.
FGC	Numero de eventos Full GC.
FGCT	Tiempo acumulado de Full Gargabe Collection.
GCT	Total del tiempo del garbage collector.

Tabla 6 GC

3.3.1.2. Opción gcnew

Todas las opciones disponibles en jstat tienen el mismo formato y comando:

<jstat>, <StartOption>, <Program> <pid> <ms>

Command: **jstat -gcnew MainApplication 1520 1000**

Zona del heap	Descripción
S0C	Capacidad del espacio superviviente 0
S1C	Capacidad del espacio superviviente 1
S0U	Espacio usado del espacio superviviente 0
S1U	Espacio usado del espacio superviviente 1
TT	Tenuring Threshold
MTT	Maximum Tenuring Threshold
DSS	Espacio máximo deseado
EC	Capacidad actual del espacio eden
EU	Espacio usado de Eden Space
YGC	Numero de eventos GC en la generación joven
YGCT	Tiempo de recolección de la generación joven

Tabla 7 gcnew

3.3.1.3. Opción gcold

Como el nombre indica, gcold se encarga de mostrar la información de la OG, y además, de la PG. Dicha opción nos da la posibilidad de comparar ambas generaciones y observar relaciones entre ambas.

Command: **jstat -gcold MainApplication 1520 1000**

Zona del heap	Descripción
PC	Capacidad de espacio permanente actual
PU	Utilización del espacio permanente
OC	Capacidad de espacio antiguo actual
OU	Utilización del espacio antiguo
YGC	Número de eventos GC de generación joven
FGC	Número de eventos GC completos
FGCT	Tiempo completo de recolección de basura
GCT	Tiempo total de recolección de basura

Tabla 8 gccold

3.3.1.4. Opción gcutil

Esta última tabla es la única que no muestra sus resultados en Kb, sino que lo hace en porcentaje. Los últimos 5 datos son lo mismos tiempos y número de eventos que las tablas anteriores.

Command: **jstat -gcutil MainApplication 1520 1000**

Zona del heap	Descripción
S0	Espacio sobreviviente 0 utilización como porcentaje de la capacidad actual del espacio
S1	Utilización del espacio de supervivencia 1 como porcentaje de la capacidad actual del espacio
E	Utilización del espacio de Eden como un porcentaje de la capacidad actual del espacio
O	Utilización del espacio antiguo como porcentaje de la capacidad actual del espacio
P	Utilización del espacio permanente como porcentaje de la capacidad actual del espacio
YGC	Número de eventos GC de generación joven
YGCT	Tiempo de recolección de basura de la generación joven
FGC	Número de eventos GC completos
FGCT	Tiempo total del Full GC.
GCT	Tiempo del GC.

Tabla 9 gcutil

3.4 Heap Histogram, herramienta de Jmap

Utilidad que fue introducida en una de las actualizaciones del JDK en Solaris Operating Enviroment (no disponible en Windows). Imprime todas las estadísticas relacionadas con la JVM.

Si *jmap* se usa con un proceso sin una línea de comandos que indique alguna opción en concreto, imprime la lista de objetos compartidos y cargados, similar a lo que pmap de Solaris imprime. Sus principales opciones son:

- heap
- histogram
- permstat

Tal y como se ha visto en las anteriores herramientas, existe una estructura de comando para activar *jmap* y obtener la información requerida. La estructura es la misma, se indica que herramienta, en este caso jmap, que opción (heap, histo o permstat) y el id del programa en ejecución.

jmap -heap 18623

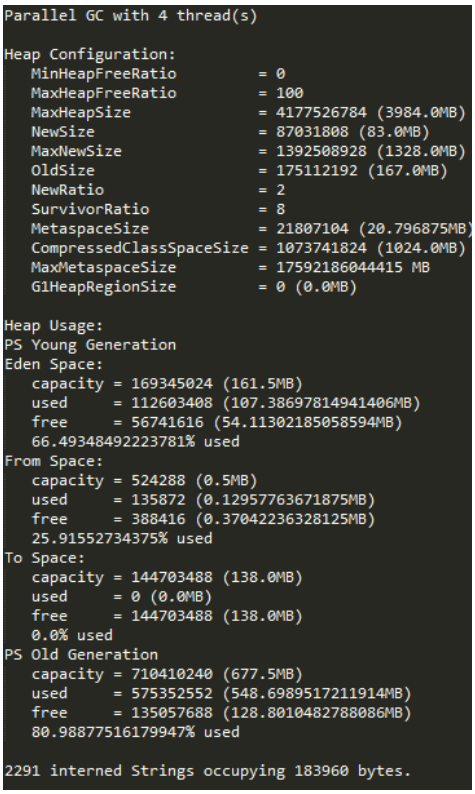


Ilustración 11: Salida jmap -clstats

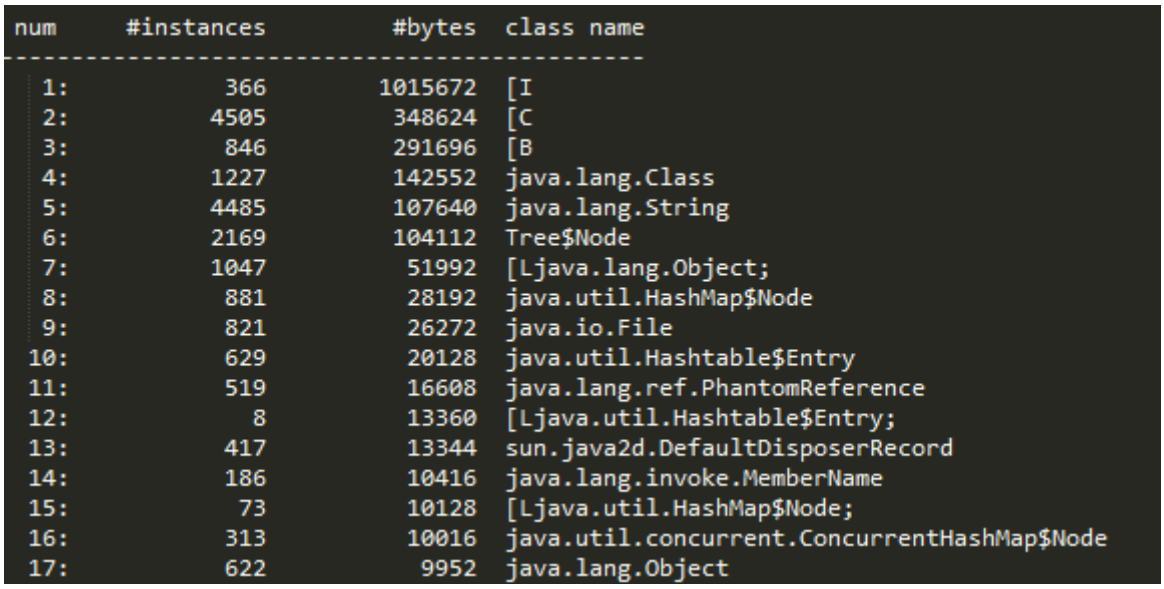
La información que nos muestra la Ilustración 11 es la siguiente:

- Información específica del algoritmo GC, incluyendo el nombre del algoritmo GC (como se aprecia en la Ilustración 11).
- Configuración del heap que pueda haber sido configurada como línea de comandos o seleccionada por la JVM basada en la configuración de la máquina.
- El uso de la memoria del heap. Para cada generación o área del heap, la herramienta imprime la capacidad total del heap, el uso en memoria, y la memoria disponible. Si una generación se organiza como una colección de espacios, como podría ser la nueva generación, se da por hecho pues que se incluye una nuevo zona de memoria. Los datos de Ilustración 11 son del momento del volcado.

3.4.1. Heap Histogram, herramienta de Jmap

El comando -histo se usa para obtener el histograma de las clases del heap. Dependiendo del parámetro, el comando *jmap -histo* puede imprimir el histograma del heap de un proceso ejecutándose. Cuando dicho caso se cumple, la herramienta imprime el número total de objetos, tal y como se puede observar en la Ilustración 12, el tamaño de la memoria en bytes, y todos los nombres de cada clase. Las clases internas de la JVM están entre corchetes angulares ([]).

Un histograma es útil para conocer que clases han realizado más instancias y cuantos bytes han ocupado.



num	#instances	#bytes	class name
1:	366	1015672	[I
2:	4505	348624	[C
3:	846	291696	[B
4:	1227	142552	java.lang.Class
5:	4485	107640	java.lang.String
6:	2169	104112	Tree$Node
7:	1047	51992	[Ljava.lang.Object;
8:	881	28192	java.util.HashMap$Node
9:	821	26272	java.io.File
10:	629	20128	java.util.Hashtable$Entry
11:	519	16608	java.lang.ref.PhantomReference
12:	8	13360	[Ljava.util.Hashtable$Entry;
13:	417	13344	sun.java2d.DefaultDisposerRecord
14:	186	10416	java.lang.invoke.MemberName
15:	73	10128	[Ljava.util.HashMap$Node;
16:	313	10016	java.util.concurrent.ConcurrentHashMap$Node
17:	622	9952	java.lang.Object

Ilustración 12: Salida histograma

3.4.2. Permanent Generation Statistics, herramienta de Jmap

La PG es el área del heap que contiene todos los datos sensibles de la Máquina Virtual, tales como clases y métodos de objetos. También se le conoce por *method area*.

El comando para mostrar esta información es el siguiente:

jmap -permstat <pid> 29620

3.5. VisualVM

Herramienta que proporciona una interfaz visual para observar al detalle información sobre las aplicaciones de java mientras están ejecutándose en la JVM. Muchas de las herramientas usadas y explicadas en este trabajo son parte de VisualVM, que permiten que tales herramientas obtengan datos del software de JVM.

Java VisualVM se introdujo con la versión 6 del JDK, actualización 7.

3.5.1. Procedimiento de instalación y utilización de VisualVM

Primeramente hay que descargar el programa alojado en la página web que se muestra a continuación:

https://visualvm.github.io/.

Según el IDE que se quiera utilizar, se deberá proceder de manera distinta. Se explicará el procedimiento en IntellIJ.

3.5.2 IntellIJ

Para realizar el desarrollo de la distintas aplicaciones, se ha usado el IDE (*Integrated Development Enviroment*) *IntellIJ,* compatible con VisualVM.

IntellIJ es un ambiente de desarrollo integrado (IDE) para el desarrollo de programas informáticos.

Desarrollado por JetBrains, no está basado en Eclipse como MyEclipse o Oracle Enterprise Pack para Eclipse. Para poder usar VisualVM, ya que no detecta los procesos ejecutados por IntellIJ por defecto, será necesario descargar un archivo .jar que contiene el plugin de VisualVM para IntellIJ.

En las figuras posteriores se muestra la interfaz de IntellIJ y los pasos a seguir para realizar la instalación del plugin necesario para usar VisualVM en el IDE IntellIJ.

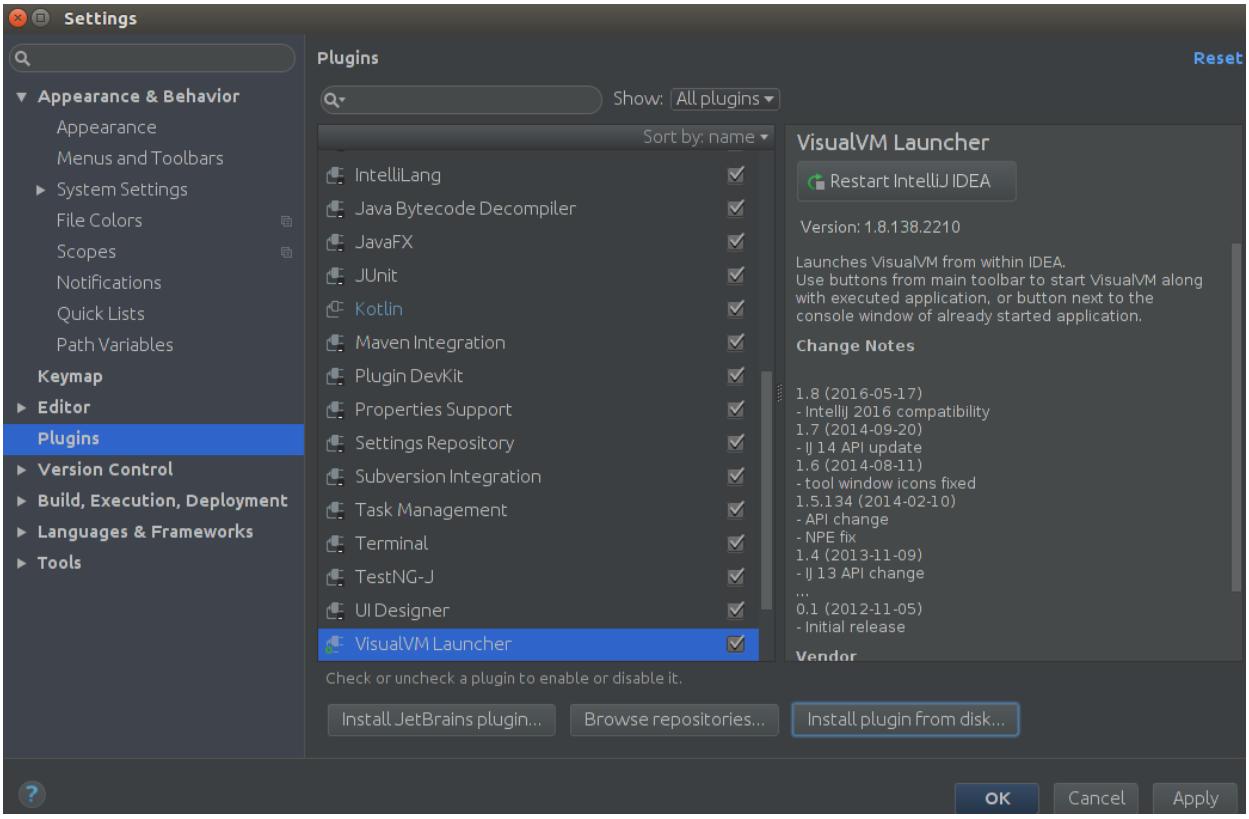


Ilustración 13: Interfaz VisualVM

Tal y como muestra la Ilustración 13, en el apartado Settings→ Plugins, si se realiza una búsqueda por VisualVM, habiendo seleccionado el plugin, se puede realizar una instalación de dicho archivo desde disco.

Una vez se tenga el plugin instalado, estarán disponibles las opciones de VisualVM en la barra de herramientas del IDE, tal y como se observa en la figura. En vez de ejecutar cualquier aplicación con el botón *run* en verde, se ejecutará el programa con el botón 1 que se indica en la Ilustración 14, y con el botón 2, que realizar el debug de la aplicación. Ambos ejecutarán VisualVM.

Ilustración 14: Barra herramientas VisualVM

Como se observa a en la Ilustración 15, es necesario tres parámetros.

1. Campo VisualVm executable: path que dirija al archivo ejecutable de IntellIJ.
2. El segundo campo indica el tiempo durante el cual VisualVM acumulará datos.
3. Delay for starting hace referencia al lapso de tiempo entre un muestro del heap y el siguiente.

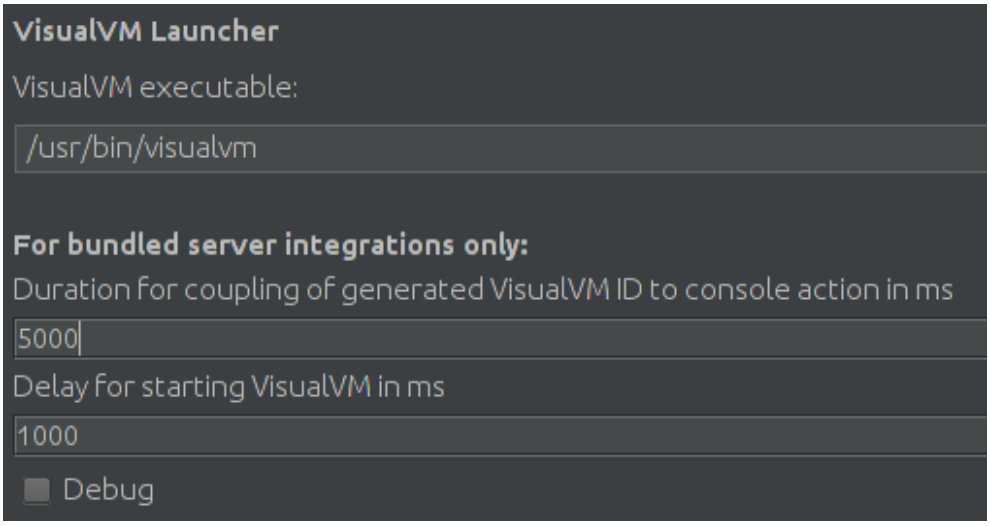


Ilustración 15: Configuración VisualVM

3.5.3. Interfaz Monitor de VisualVM

En el apartado Monitor hay disponibles 4 ventanas.

- CPU Usage: el uso en tiempo de ejecución de la aplicación. Ilustración 16.
- El tamaño y uso de Heap: gráfica que muestra el uso y capacidad total del heap. Ilustración 18.
- Classes: muestra el número de clases cargadas. Ilustración 23.
- Threads: muestra el número de threads activo y los que han sido finalizados. Ilustración 17.

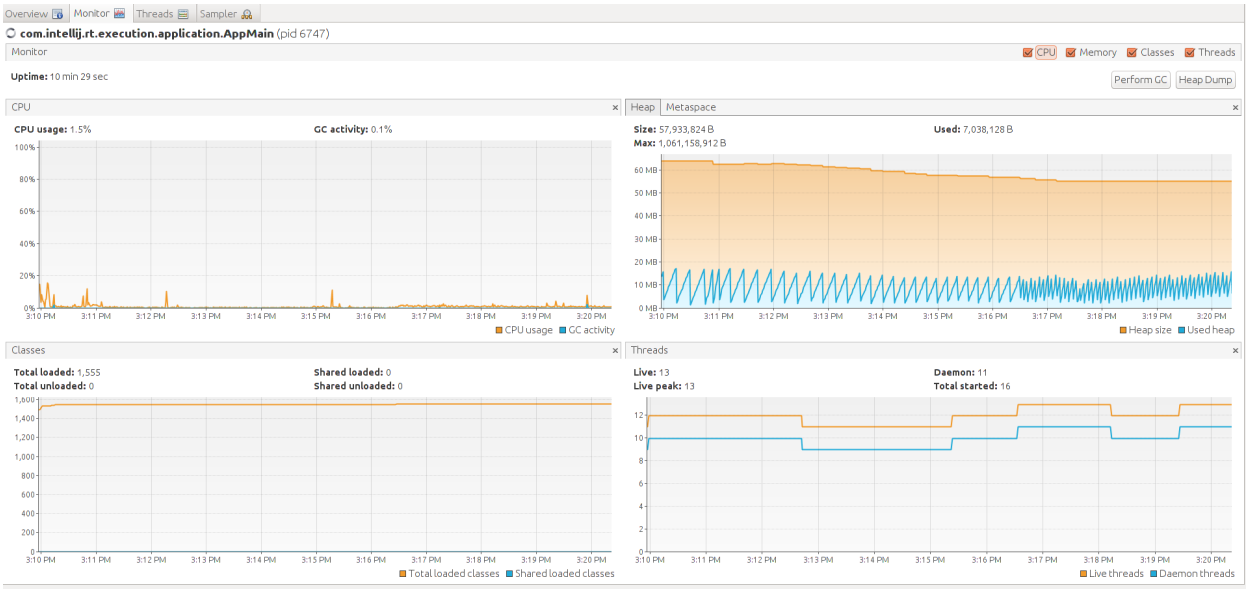


Ilustración 16: Interfaz monitor, 4 paneles

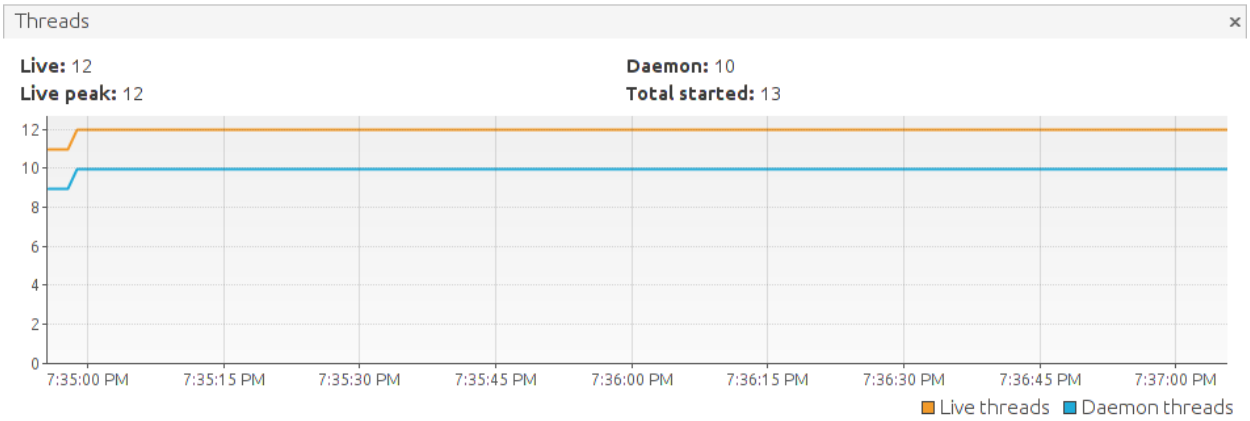


Ilustración 17: Threads

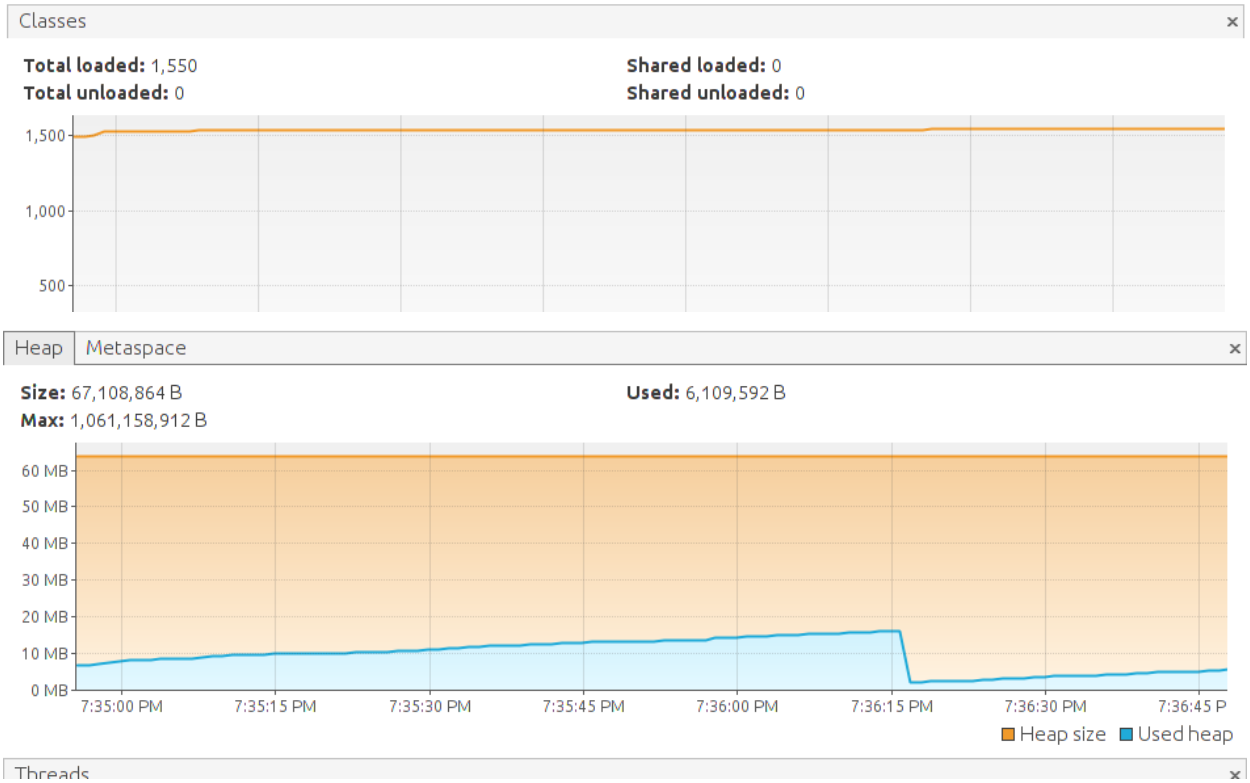


Ilustración 18: Heap

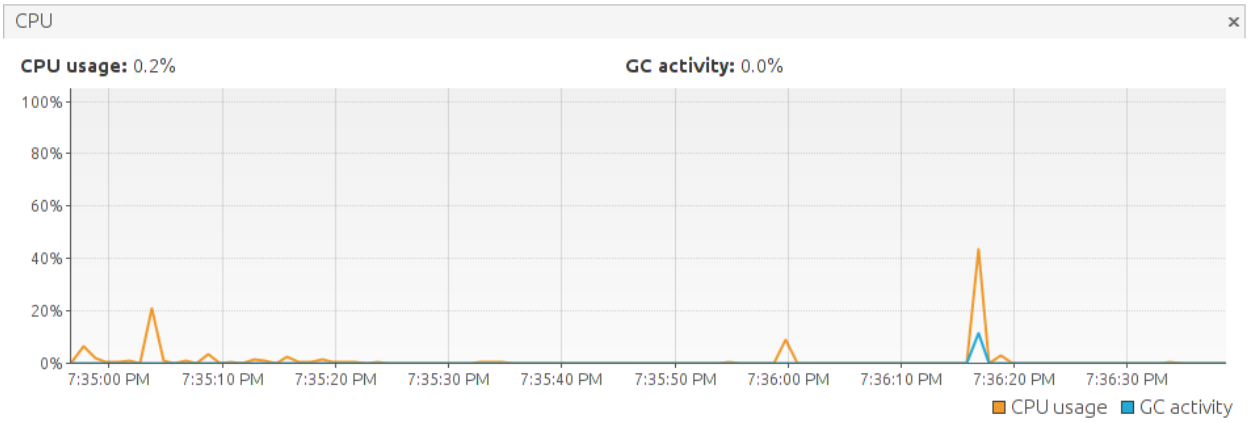


Ilustración 19: CPU Usage

3.5.4. Sampler Thread CPU Time

La Ilustración a continuación muestra el uso de la CPU por cada Thread.

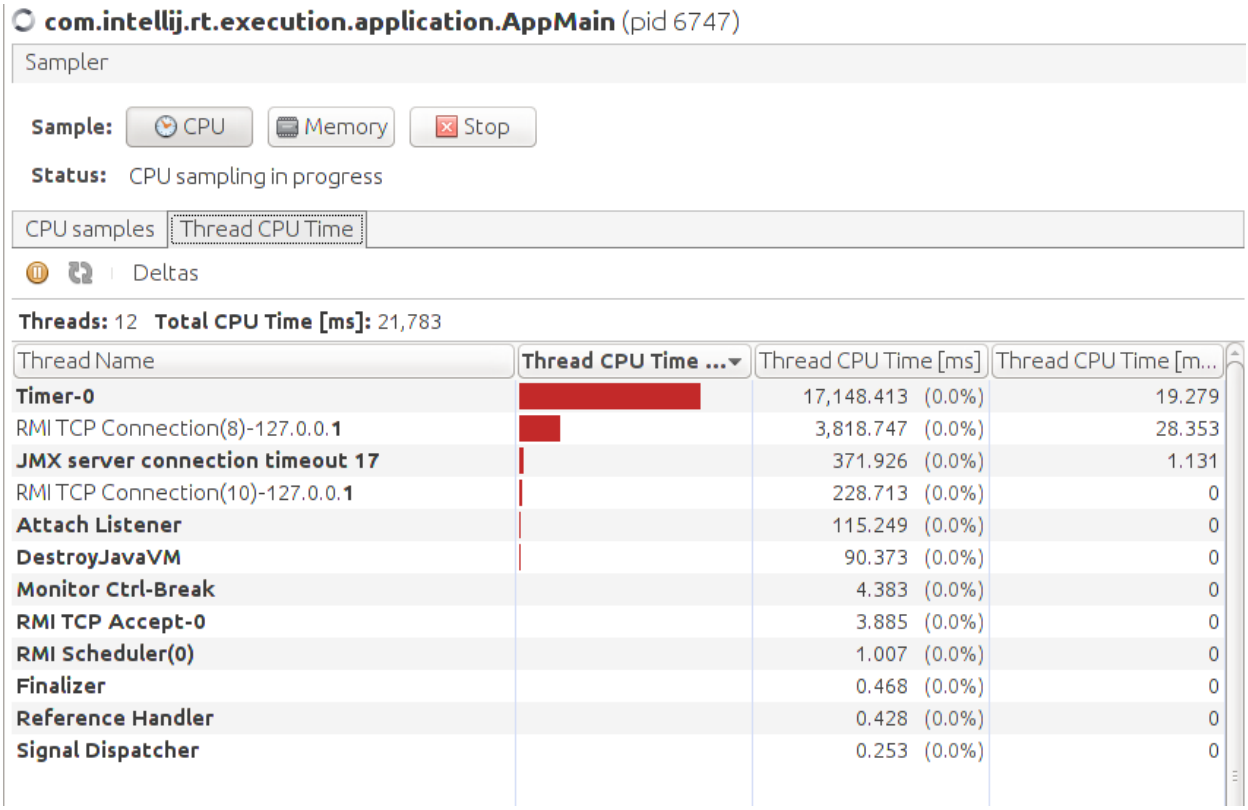


Ilustración 20: Tiempos de CPU

3.5.5. CPU Samples

El tiempo que ocupa en CPU métodos de la aplicación desarrollada.

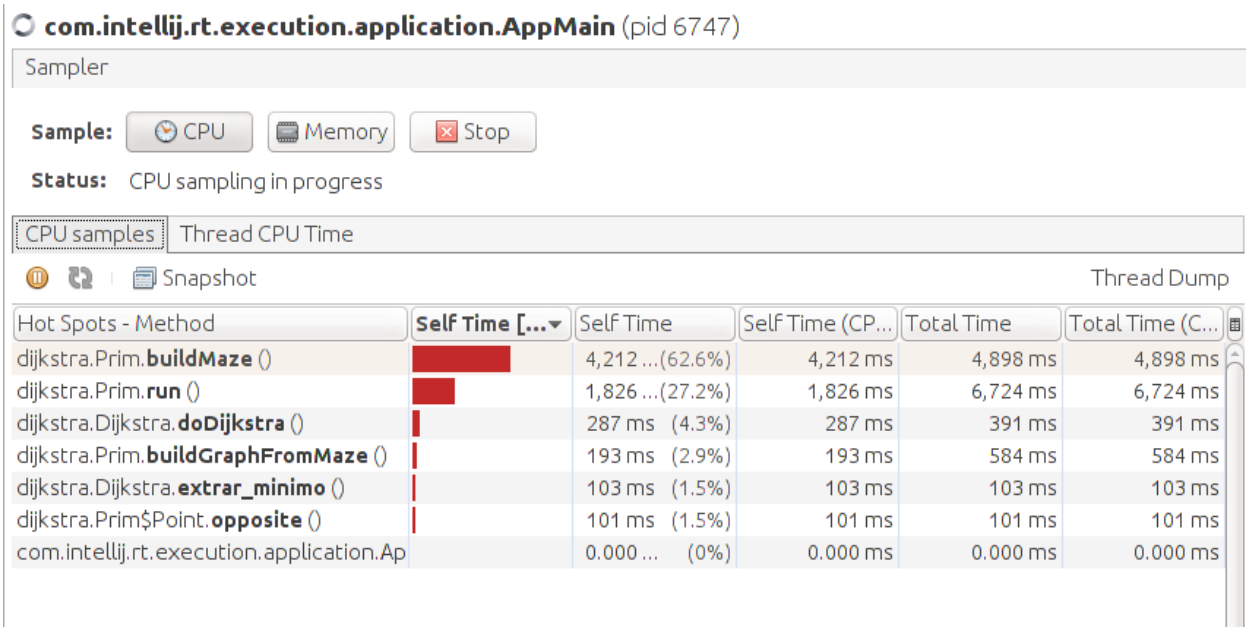


Ilustración 21: Tiempo que ocupa cada método en CPU

Todos los datos de VisualVM se pueden ver en tiempo gracias a la herramienta jstat. El software hace uso de la herramienta para extraer los datos raw y mostrarlos en las distintas gráficas que se han mostrado.

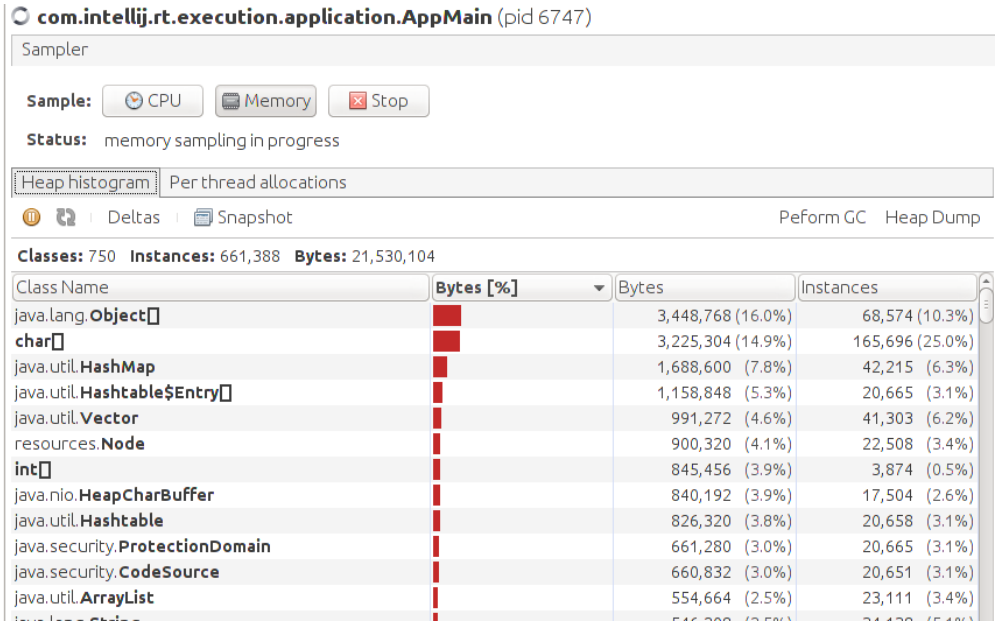


Ilustración 22: Histograma de allocations

4. Aplicaciones desarrolladas para el estudio del GC

En total se han desarrollado tres aplicaciones distintas basándose en algoritmos de búsqueda:

1. BFS
2. DFS
3. Dijkstra

Se han escogido estos tres algoritmos para comprobar si la implementación iterativa o recursiva se refleja de alguna manera en el análisis posterior de memoria mediante herramientas que muestran el uso de CPU o en el proceso del GC.

Para que cualquier de las aplicaciones funcione, en sus clases *Main* se ha de especificar el día en el que se realice la ejecución y la hora. La clase *Timer* controla cuando y a que hora empieza su ejecución.

4.1 BFS (Breadth First Search)

La aplicación BFS desarrollada en lenguaje Java se basa en obtención de un árbol binario a partir de imágenes en formato binario.

El proceso que realiza es el siguiente:

1. Lee una imagen del disco, la transforma a formato binario y la vuelve a guardar bajo un nuevo nombre.
2. Se lee ahora la imagen en binario. Cada pixel de la imagen será considerado un nodo que se añadirá a un árbol binario. La lectura de la imagen se realiza por filas, de izquierda a derecha.
3. Se hace uso de una cola de prioridad mediante una solución iterativa.

4.2 DFS (Deep First Search)

El desarrollo de DFS es idéntico en la obtención de imágenes binarias respecto a BFS. DFS realiza dos procedimiento; iterativo y recursivo.

1. Iterativo
   - Hace uso de una pila FIFO para construir el árbol binario a partir de los nodos, que son pixeles de la imagen en blanco o negro.
   - A medida que se obtienen los nodos, el método de la clase Tree, addNode() se encarga de construir el árbol binario.
   - Finalmente, cuando se tiene el árbol construido, se invoca al método procedureDFSIterative(nodeRoot) para buscar todos los nodos.

4.3 Dijkstra

La aplicación *Dijkstra* no utiliza imágenes como las dos aplicaciones anteriores, sino que crea laberintos aleatorios en cada ejecución. Dicho laberinto es un grafo formado por nodos, de los cuales dos son los responsables de indicar el nodo inicial y el nodo final.

Para crear un laberinto aleatorio, primero se crea una matriz con un tamaño N x M (por defecto es 5x5). Las paredes se consideran '*', no se puede pasar de un nodo a otro siempre y cuando sea '*' '*', o '*' y '-'. El algoritmo añadirá a una ArrayList de <Point> el nodo inicial 'S', e irá añadiendo iterativamente sus vecinos.

Una vez este proceso ha finalizado, abriremos un camino por el laberinto hasta quedarnos sin nodos que visitar.

El algoritmo *Dijkstra* empieza mirando los vecinos del nodo inicial según el peso de la arista que une cada nodo. Dado que *Dijkstra* es un algoritmo que siempre busca el camino más corto. Este cálculo lo realiza mediante el método extraer_mínimo, que devuelve el nodo con la distanci más corta de los vecinos del nodo. A medida que va moviéndose por los nodos y actualizando el peso de su movimiento encontrará siempre y cuando haya un camino abierto, el nodo final con el peso más pequeño.

Dijkstra no visita nodos ya visitados anteriormente durante el proceso de búsqueda.

5. Análisis de datos

Antes de realizar las comparaciones entre datos e intentar alcanzar alguna conclusión sobre los distintos algoritmos, su impacto en memoria y como detectar, a partir de los resultados mostrados, posibles *leaks* de memoria, se presentan datos y gráficas con su debida explicación para aclarar que hay en cada Ilustración.

El primer objetivo que se quiere comprobar es si realmente el GC realiza todas las tareas explicadas en los apartados anteriores e intentar confirmar el correcto funcionamiento.

La obtención de datos se ha llevado a cabo en el ordenador del Departamento de Matemáticas y Informática de la Universidad de Barcelona bajo el sistema operativo Open Suse de 64 bits, a través de una conexión *ssh* desde un ordenador particular corriendo Ubuntu 16.14 LTS.

Se han utilizado las herramientas mencionadas con anterioridad, en concreto el listado siguiente:

1. **JSTAT**
   - gcutil: aporta información sobre el porcentaje capacidad de S0, S1 o ES, entre otras.
   - gnew: recolta información sobre TT, DSS, o el tiempo total de los YGC.
   - gcoldcapacity: opción que se centra más en la información de OG.
   - gcold: aporta información sobre la PG, OG y los tiempos de GC en YG y FC.
   - gc: opción que permite extraer la mayoría de datos menos los tiempos de FGC y FGCT.
2. **HPROF**
   - java -agentlib heap = dump. Comando que genera el archivo heap.bin compatible con VisualVM. Se explica en la página 26, apartado 3.2.1.
3. **JMAP**
   - jmap -heap:format=b <**pid>** Se cubre en la página 32.
   - jmap -histo <pid> Explicación en la página 33. Se usa para los análisis en los apartados 5.1.1,5.2.2 y 5.4.2.
   - jmap -histo:live <pid> Comparte explicación con el anterior comando aunque solo muestra las clases alived. Analizado en los apartados 5.1.1.2, 5.2.2.1 y 5.4.2.1.
4. **VisualVM**

El análisis se estructura de la siguiente manera:

1. Se realiza una comparación entre los histogramas obtenidos en cada explicación. En los apartados 5.1.1., 5.2.2 y 5.4.2.1
2. Se comparan las generaciones YG, OG y PG.
3. Se comparan otros parámetros que aporten información relevante.
4. Se observa el tiempo de CPU con VisualVM.
5. Finalmente se resumen los datos obtenido para sacar una conclusión.

5.1.1. Histogramas BFS

5.1.1.1. Histograma 64

Los siguientes datos muestran el resultado de ejecutar el siguiente comando, jmap -histo D64, solo funcional para sistema operativos de 64 bits. Se observa entre las cuatro muestras repartidas en el tiempo (400, 1700, 2900, y 3500 min) que las clases que más se han referenciado han sido:

1. [B: array de bytes.
2. [I: array de enteros.
3. [C: array de chars.

La muestra a los 3500 min de ejecución arroja un resultado distinto a los anteriores, ya que observamos 31702 instancias de la clase sun.java2d.cmm.lcms.LCMSImageLayout debido al uso exhaustivo que se realiza de la clase LCMS. Dicho proceso ha ocupado durante el tiempo total de ejecución 1,77MB que se podrían haber disminuido de haber transformado los archivos una vez. Sin embargo, se ha elegido esta vía para analizar como afecta a la memoria.

5.1.1.2. Histograma Live

El histograma live se diferencia respecto al anterior en un factor, solo muestra clases alived, es decir, que mantienen todavía alguna referencia en memoria.

Se muestran a continuación el resultado de las muestras obtenidas:

1a Muestra: 400 minutos

Instances	Bytes	Class Name
7899	322863864	[I
11876	188116736	[B
7472	418432	sun.java2d.cmm.lcms.LCMSImageLayout

Tabla 12 Histo live

2a Muestra: 1700 min

Instances	Bytes	Class Name
343	500304	[I
4500	348304	[C
624	355552	[B

Tabla 13 Histo live

3a Muestra: 2900 min

Instances	Bytes	Class Name
366	1015672	[I
4505	348624	[C
846	291696	[B

Tabla 14 Histo live

4a Muestra: 3500 min

Instances	Bytes	Class Name
10918	98564032	[B
6238	42717304	[I
4638	355552	[C

Tabla 15 Histo live

A medida que la aplicación ha pasado más tiempo en ejecución se observa un aumento en la cantidad de instancias *alived,* hecho totalmente lógico.

A partir de estos datos podemos llegar a las siguientes observaciones:

1. [I,[C,[B son las clases que más instancias realizan.
2. Se observa que a los 3500 min no hay ninguna instancia de LCMSImageLayout. Por lo que se puede sospechar que el Garbage Collector ha realizado, en alguno de sus eventos GC, una limpieza y ha retirado de memoria dicha clase.
3. Se observa una aumento del 1290% respecto la 3a muestra y la 4a en la clase [B. Dicho aumento se debe a un muestro anterior a un evento GC.

5.1.2. Diferencia ES respecto a OG

Se han tomado medidas sobre el uso de Eden Space y Old Generation en estas pruebas y se obtiene lo siguiente:

	Eden Space	**Old Generation**
1700 minutos	91% (Capacidad 200MB)	70% (704MB)
2900 minutos	66% (161MB)	80% (677MB)
3500 minutos	92% (146MB)	73% (693MB)

Tabla 16

Más tarde se mostrarán gráficas y como han evolucionado el uso de estas generaciones, pero en esta primera aproximación se puede observar como OG se mantiene en un uso medio-alto de su capacidad, 700MB mientras que ES usa casi la máxima capcidad disponible. Esto refleja una gran acumulación de objetos nuevos o a punto de ser copiados a S0 y S1.

5.1.3. Observación BFS con VisualVM

VisualVM es capaz de mostrar cuales son los métodos con más procesamiento de cualquier aplicación o los conocidos HotSpots que se ha comentado con anterioridad. En este caso, *addNode()*, método privado de la clase Tree.Node, es el encargado de crear el árbol binario formado a partir de los píxeles de cada imagen. Se observa que hasta un 88.1% del tiempo de CPU lo ocupa éste método, 1314% más que doStuff(), encargado de transformar las imágenes a binario y guardarlas en disco.

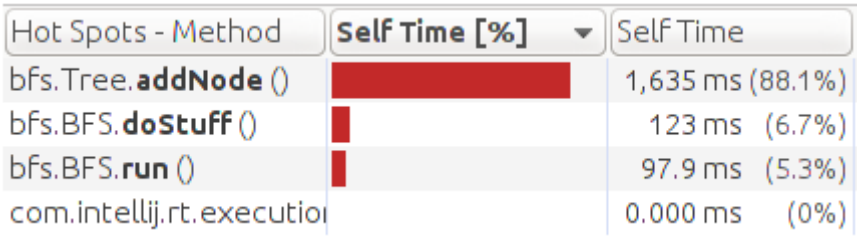


Ilustración 23: Método y su tiempo de CPU. VisualVM

5.1.4. Eden Space

Como se ha comentado con anterioridad, Eden Spce es el espacio donde se crean la mayoría de objetos que se consideran de tamaño aceptable, en caso de pesar demasiado se alojaría en Old Generation. Si Survivor 0 se llena, los objetos vivos de Eden o Survivor 1 que no se han copiado son desechados. En la Ilustración 24 se observa en color azul el espacio usado de la generación y en rojo la capacidad total. El eje x de la Ilustración refleja que cada instante de muestro equivale a 1000ms. Si se ampliara la Ilustración 24, se observarían bajadas de ES que indican MinorGG.

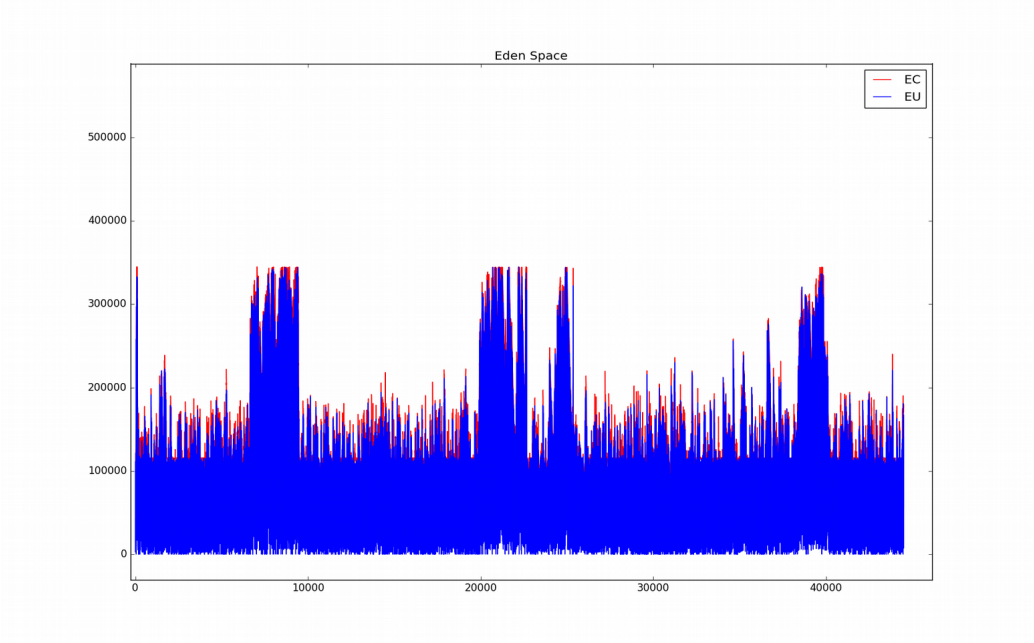


Ilustración 24: Eden Space BFS

5.1.5. Survivor 0

La ilustración 25 refleja S0, una de las dos zonas supervivientes de YG. Su uso baja a mínimos, mientras que Eden Space aumenta su porcentaje de uso. Esta situación se abarca en los siguientes puntos.

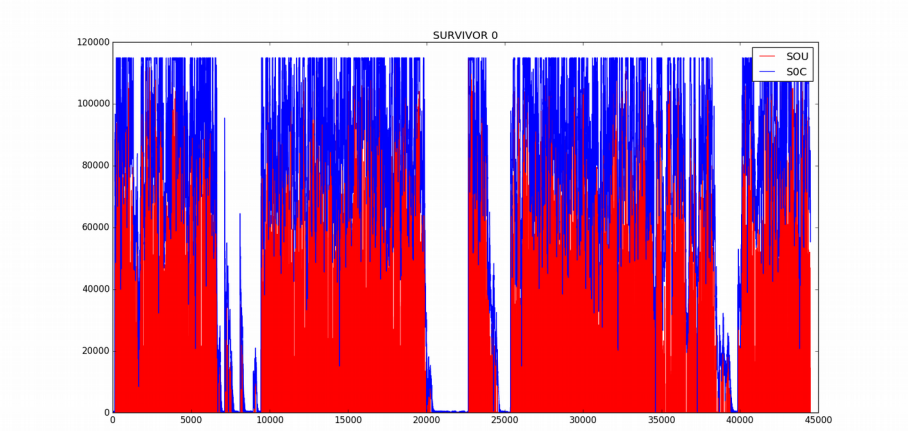


Ilustración 25: Survivor 0 S0U-SOC

5.1.6. Survivor 1

Se aprecian cuatro mínimos en la Ilustración 26 en el uso del espacio reservado para Survivor 1 que se deben a una mayor cantidad de Major GC y Minor GC.

Para entender mejor los resultados que se reflejan en las gráficas anteriores, Ilustración 24, 25, 26, se ha comparado analíticamente la cantidad de YGC y FGC en 750 minutos.

En concreto, entre los minutos 100 y 166, rango en el que se aprecia la bajada en el uso de Eden Space (Ilustración 26) y el minuto 200 y 250. En este lapso de tiempo el número de YGC aumenta como es de esperar, pero en cambio el número de FGC es inferior, lo que conlleva una consulta a otra generación para comprobar que es lo que ha provocado esto.

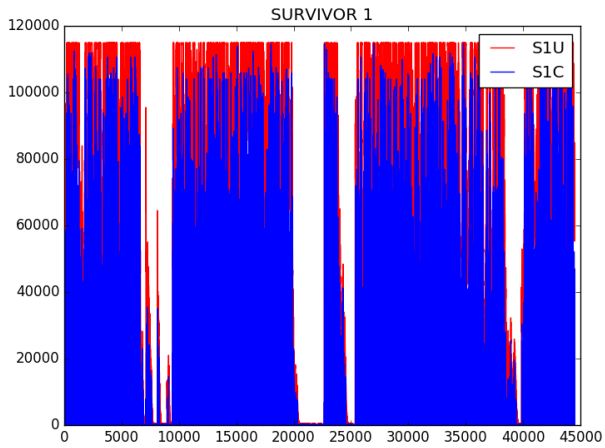


Ilustración 26: Survivor 1 S1U-S1C

A pesar de que parezca de que ambas zonas están siendo usadas a la vez, si se amplia la misma zona en la Ilustración 26, se observaría que en el cualquier momento de la ejecución, cuando From(S0) está ocupada, To (S1) está vacía. Véase la tabla 17. Por lo tanto, la teoría es cierta, siempre hay un espacio Survivor que está vacío mientras que la otra está en uso. En el caso de existir datos en el mismo instante de tiempo en ambas zonas, se consideraría un problema en la gestión de memoria y habría que investigar con más profundidad ya que es síntoma de un *leak* de memoria. En la Ilustración 27 se amplia una sección de la Ilustración 26 para contrastar lo dicho, cuando S0 tiene datos, S1 esta vacía.

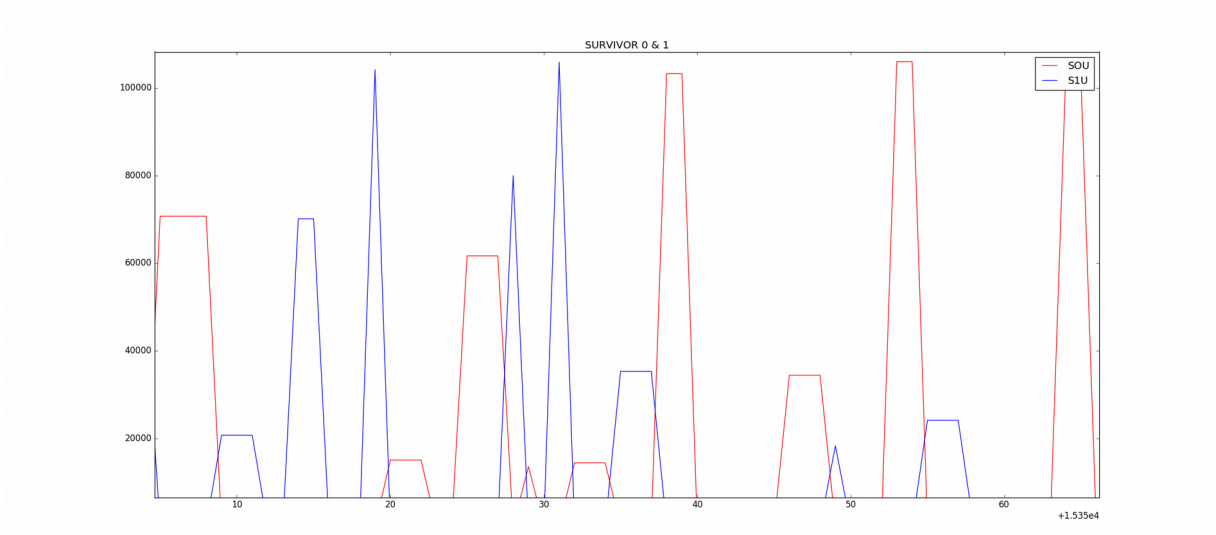


Ilustración 27: Survivor 0 y Survivor 1 ampliada

Minutos/Nº eventos GC	YGC	FGC
100-166	1797	262
200-250	2506	143

Tabla 17 Comparación eventos GC

5.1.7. Old Generation

Ha sido necesario la consulta a esta generación para resolver el problema que se había detectado en ES y S0 y S1. La causa por la cual se han detectado mínimos en los gráficas de ES, S1 y S0 es la siguiente:

- Algunas veces la OG esta demasiado llena como para aceptar todos los objetos que deberían ser trasladados a la OG, a no ser de haber vacíado YG previamente. En ese caso, para todos los tipos de recolectores, el algoritmo de la YG en concreto no se ejecuta. Sin embargo, el algoritmo de OG si que se ejecuta en todo el heap. Es por eso que observamos como S0 y S1 bajan al mínimo y Eden Space aumenta su capacidad y uso.

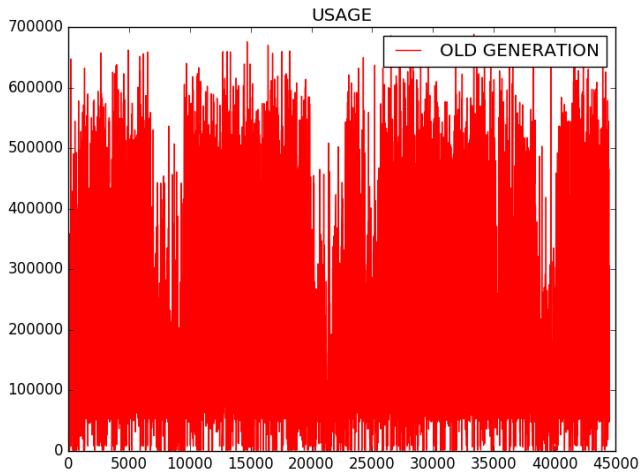


Ilustración 28: Uso de OG

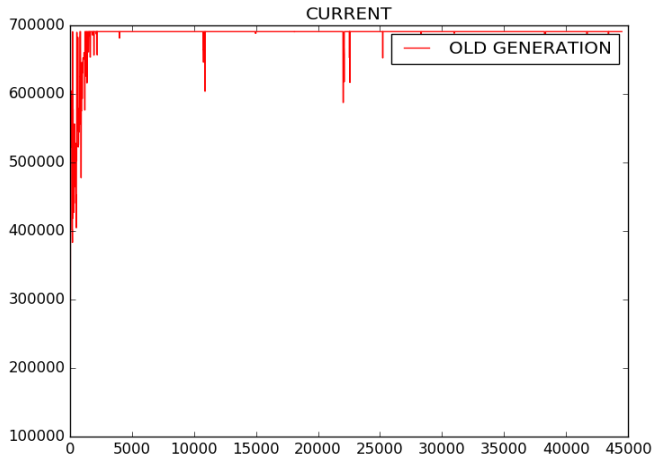


Ilustración 29: Capacidad total OG

5.1.8. S1 y DSS

No hay errores de memoria, S1U nunca sobrepasa DSS. Se puede observar en la Ilustración 31 aunque los datos estén comprimidos, se aprecian bajadas drásticas en el uso. Esto se debe a Full GC.

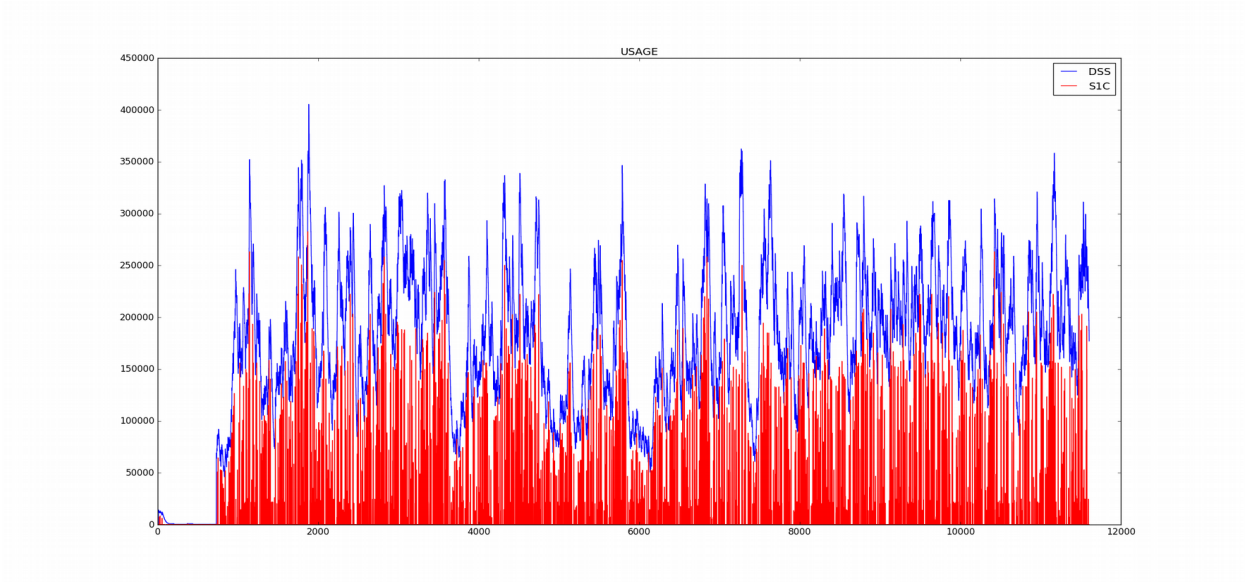


Ilustración 30: Uso de Survivor 1 y DSS

DSS o Desired Survivor Space indica que la máxima capacidad deseada, es decir, el espacio ocupado por los objetos vivos no debería pasar ese límite. Concepto que va ligado con TT. Ambos controlan la edad de los objetos y así se aseguran de controlar el uso de esta. El uso medio de S0U oscila entre los 100Mb y los 150Mb.

Este concepto se refleja en la Ilustración 30. En ningún momento el uso máximo sobrepasa el límite establecido por DSS.

5.1.9. S1 y S0U

En la Ilustración 31 se presentan los espacio S1U, DSS y S0 por separado. En la Ilustración se aprecia como DSS va oscilando a medida que el espacio usado en S1U y S0U aumenta o disminuye. Esto se debe también a que TT no es estático, su valor aumenta o disminuye dinámicamente a medida que los objetos sobreviven a los distintos eventos GC. Cuando llegan al límite, GC ejecuta su tarea y los cambia de generación, disminuyendo así el valor de TT. Este hecho se ve reflejado en la Ilustración. Otro aspecto importante es el margen que hay entre DSS y el tamaño usado de cada zona superviviente S0 y S1. Rara vez se observa DSS y S1 o S0 al mismo nivel, ya que DSS siempre mantiene una proporción respecto a la capacidad de S0 y S1.

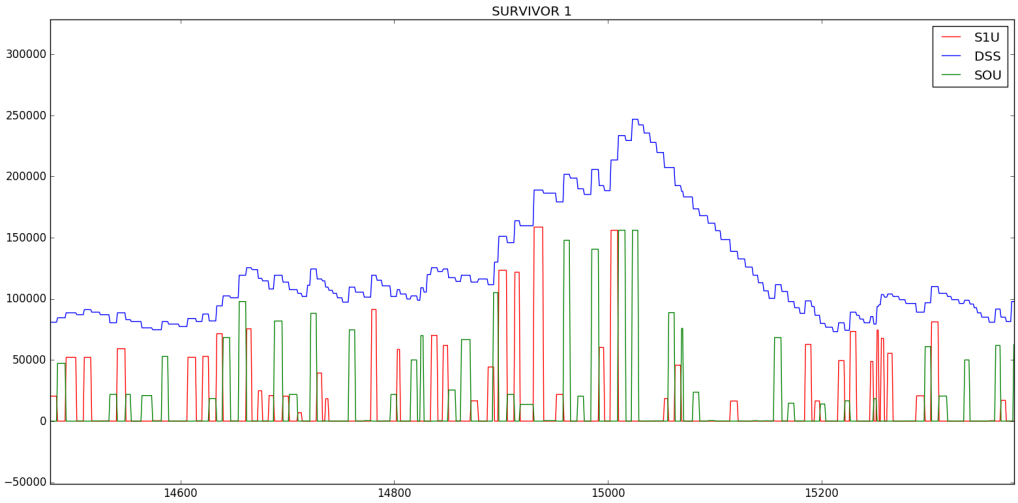


Ilustración 31: Uso de S1, S0 y DSS

5.1.10. TT y S0

En la Ilustración 32 se puede apreciar mucho mejor el impacto que tiene TT en el uso del espacio. Más o menos en el minuto 1 (valor 50 en el eje x de la Ilustración 31), se observa un uso altísimo de la S0U y como TT reduce su valor para disminuir esos niveles. Se debe a una acumulación de objetos en YG, por lo tanto, múltiples Minor GC ocurren, disminuyendo así TT y el espacio usado de S0U. Muchos objetos son considerados traspasables hacia OG, a consecuencia de que han sobrevivido a muchas GC.

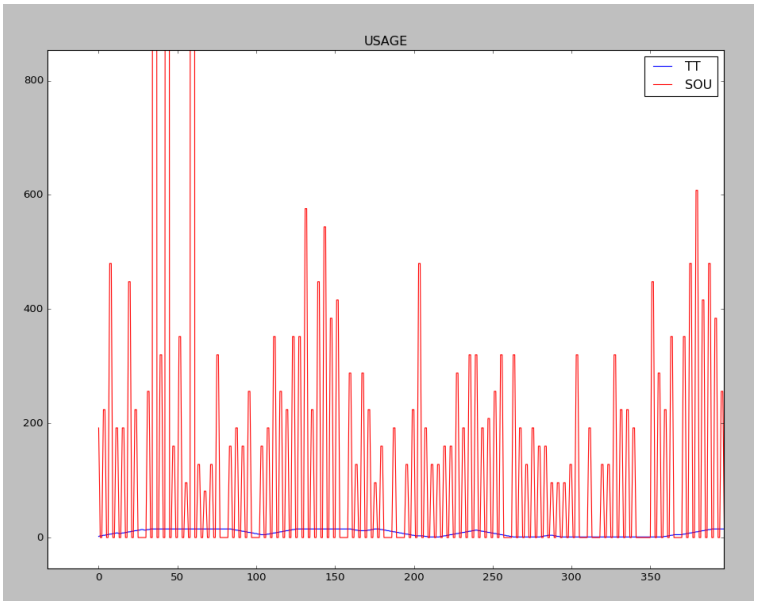


Ilustración 32: Tenuring Threshold y Survivor 0

5.1.11. Permanent Generation

Esta generación es la menos importante del heap la JVM ya que lo único que almacena es Bytecode, los métodos y nombres de las clases o objetos internos de la JVM. Se puede apreciar un aumento inicial del uso hasta los 4,7MB. A medida que el programa siga ejecutándose, este espacio aumenta muy lentamente.

Pueden ocurrir GC en esta generación, aunque no son muy habituales. Los objetos almacenados en PG no llegan a ocupar demasiada memoria, aunque son vitales para el GC.

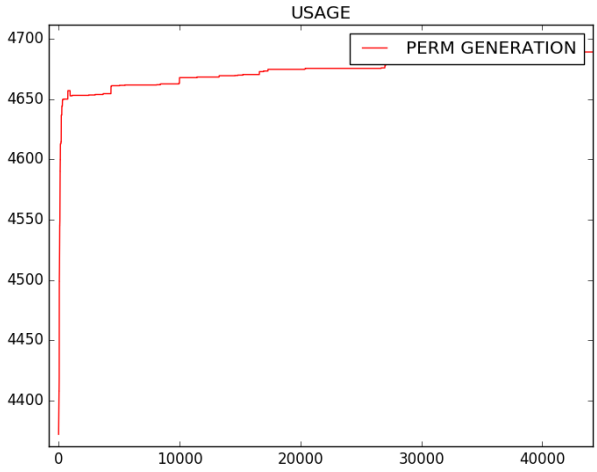


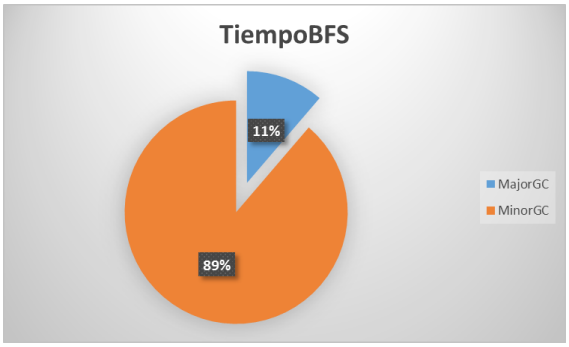
Ilustración 33: Permanent Generation BFS

5.1.12. Tiempos del Garbage Collector

El tiempo total de eventos GC en Young Generation (Minor GC) es de 337 segundos, frente a los 42 segundos de Full GC.

En total, 42 segundos han acumulado 1753 eventos (entre Major GC y Full GC) en 28 horas de ejecución. Estos datos se han extraído con jstat y confirman que los eventos Full GC ocurren con menos frecuencia.

Estos datos será necesarios para comparar el funcionamiento del GC frente a DFS.



5.2. DFS

Tras 15 minutos de ejecución, se aprecia que otra vez, tal y como se ha visto en BFS, que doStuff() es el método que más *CPU* consume, seguido de la implementación recursiva e iterativa.

Comparando la implementación recursiva e iterativa, el método recursivo consume un 300% más de CPU que la metodología iterativa.

5.2.1. Observaciones DFS conVisualVM

Hot Spots - Method	Self Time [%]	Self Time
dfs.DFS.**doStuff** ()		1,115,011 ms (99.1%)
dfs.RecursiveDFS.**procedureDFRecursive2** ()		7,043 ms (0.6%)
dfs.IterativeDFS.**procedureDFSIterative** ()		2,329 ms (0.2%)
dfs.DFS.**run** ()		199 ms (0%)
com.intellij.rt.execution.application.AppMain$		0.000 ms (0%)

Ilustración 34: Tiempo de CPU para cada Thread DFS

5.2.2. Histogramas DFS

Se repite el procedimiento realizado anteriormente con BFS y se comparan ambos histogramas, el responsable de mostrar las instancias *alived* y el histograma que muestra el total de instancias.

Se han realizado muestras en dos tiempos:

- 4437 minutos de ejecución
- 7000 minutos de ejecución

Se ha decidido extraer los datos en los tiempos que se especifica anteriormente con el objetivo de intentar detectar cambios drásticos, ya sea en el uso de CPU, en la cantidad de bytes ocupados por alguna clase en concreto y observar además si alguna clase era invocada más veces que el resto.

La tabla 19 confirma los datos vistos en BFS. Las 3 tres clases que aparecen en la tabla suelen ser las que mas instancias realizan. Queda claro pues una invariación en el número de instancias realizadas, siendo [I , [B y [C las clases que más *allocations* han realizado. TreeNode se encuentra en la 7a posición, clase pilar del programa para realizar árboles binarios a partir de una imagen.

5.2.2.1. Histograma Live

Num	Instances	Bytes	Class Name
1	745	24059976	[B
2	363	488328	[I
3	4648	355936	[C

Tabla 18 Clases alived

5.2.2.2.Histo D64 (SO de 64 bits)

Num	Instances	Bytes	Class Name
1	1070	319837984	[B
2	8711	285846992	[I
3	4651	356264	[C

Tabla 19 Clases totales

5.2.3. Survivor 0 & 1

Se observa un uso medio en las capacidades de S0U y S1U, lo que indica una estabilización en el uso de ambas zonas. Los máximos que se observan en la Ilustración 35 indican acumulaciones de objetos en YG, segundo más tarde se observan mínimos provocados por Minor GC.

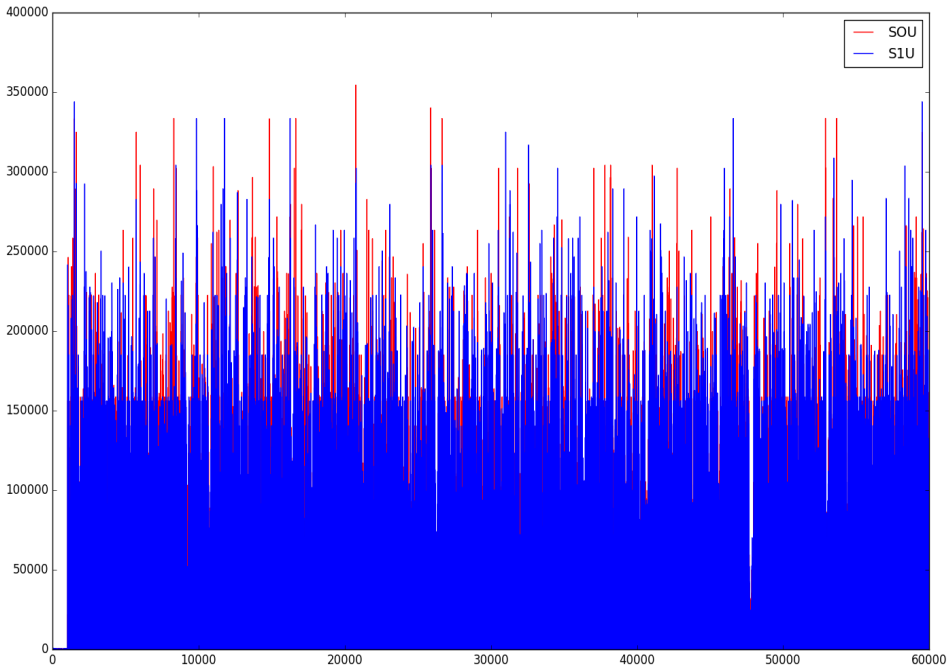


Ilustración 35: Survivor Uso y Capacidad

La ilustración 36 es una amplicación de Ilustración 35, porción {10,50}.

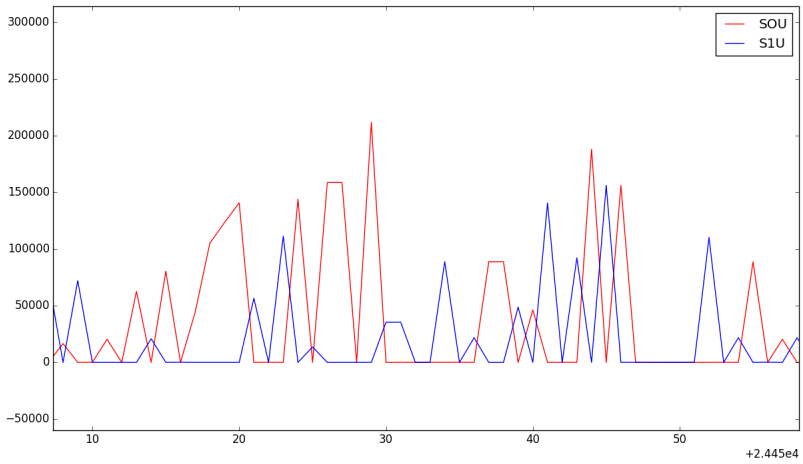


Ilustración 36: Ilustración 45 ampliada

5.2.4. Eden Space

En la Ilustración 37 se observan bajada en el uso de la Eden Space, señal de que el uso de esta zona ha llegado al límite y varios eventos Minor GC han ocurrido, disminuyendo así el EC y el EU. Esto repercute en el uso de memoria S0 y S1, ya que muchos objetos han pasado de ES a S0 o S1. Posteriormente, según la edad de los objetos en estas zonas, serían copiados a la OG.

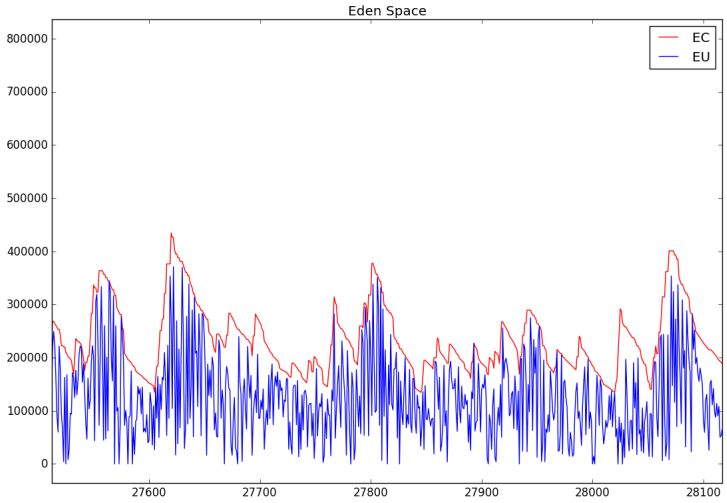


Ilustración 37: Eden Space DFS

5.2.5. Old Generation

Para realizar la extracción de los datos mostrado en la Ilustración 38, se utiliza el comando jstat -gcold, se guardan los datos en un archivo .csv y posteriormente en Python se realiza la Ilustración en cuestión.

En esta Ilustración 38 se puede apreciar una característica que se relaciona con el código y como funciona el proceso DFS. Las estructuras de datos como ArrayList o Maps, son las que llegarán antes a la OG debido a que acumulan más datos y ocupan más espacio. En la gráfica se observa como de repente el uso de OG desciende hasta estar vacío. Esto se debe a que en ciertos momentos de la ejecución ya no existe ninguna referencia a esas estructuras, por lo tanto, pierden su referencia a memoria y el GC reclama esos espacios de memoria.

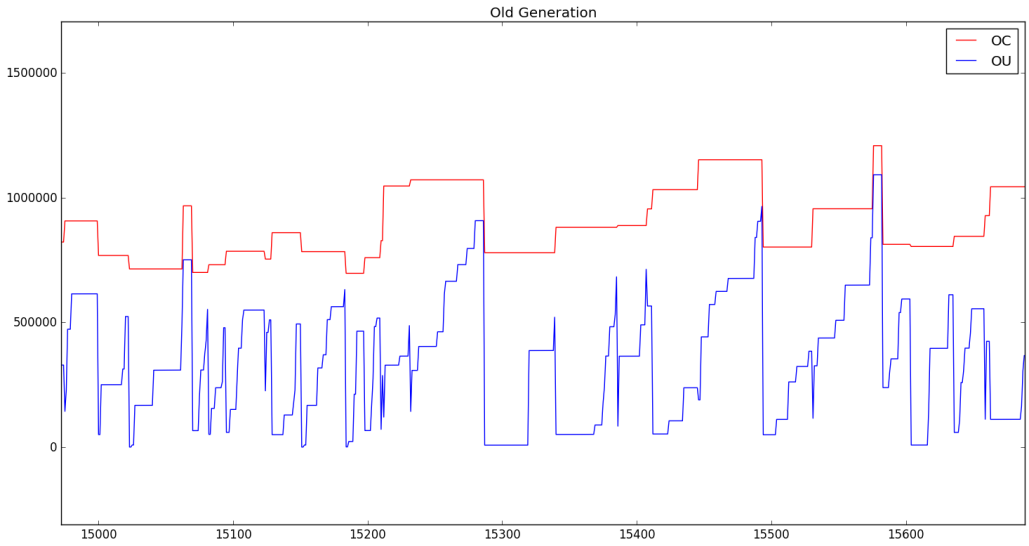


Ilustración 38: Old Generation DFS ampliada

5.2.6 S0U, DSS y S1U

En la siguiente Ilustración 39 que se presenta, se aprecia como DSS muestra cual sería el espacio deseado máximo del espacio disponible. No significa que se tenga que ocupar el 100%, sino que hay un margen de más.

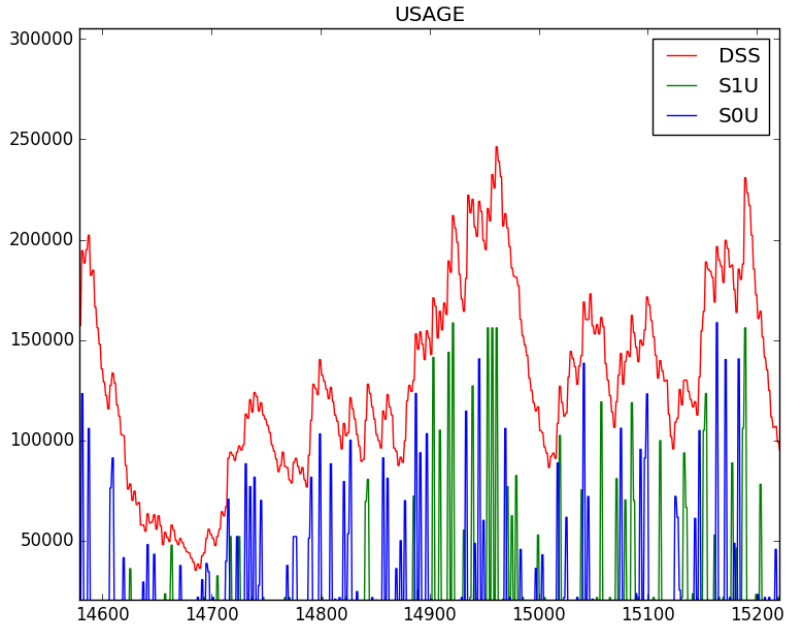


Ilustración 39: Uso DSS, S1U y S0U en DFS

5.2.7. TT y DSS

La relación entre en TT y DSS es más importante de lo que se piensa, ya que puede indicar e momento exacto de un Minor GC.

S0C	S1C	S0U	S1U	TT	MTT	DSS	EC	EU	YGC	YGCT
3072	3072	0	0	**8**	15	**3072**	**740864**	**562587**	**21**	**0,141**
3072	3072	0	0	**7**	15	**3072**	**818688**	**263334**	**22**	**0,167**

Tabla 20 Comparación TT y DSS.

Extracto de un porción de datos de la Ilustración 48 memoria en el minuto 2 de ejecución, se observan varias cosas:

- Survivor 0 y 1 no están en uso.
- TT indica un evento YGC.

- El Minor GC ha ocurrido en ES. Como se puede apreciar en la tabla 20, el uso del espacio se ha reducido un 213%.

Sin embargo, en la siguiente tabla se observa un comportamiento opuesto de TT. En vez de disminuir, aumenta su valor en 1.

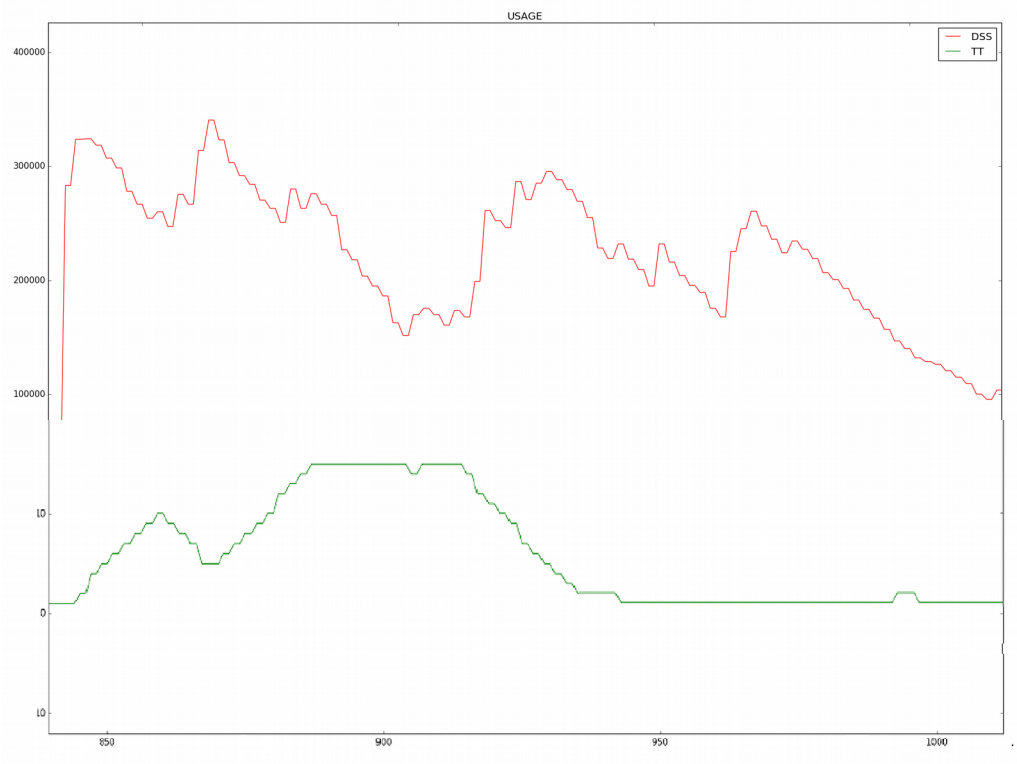


Ilustración 40: Relación TT y DSS en DFS

El valor de TT ha aumentado, por una simple razón, a pesar de que EU haya disminuido su uso, muchos objetos han sobrevivido al Minor GC, como se aprecia en la columna EU, 169Kb de datos en uso. Eso significa que muchos de estos objetos han envejecido y en futuros Minor GC o FullGC serán copiados a otra generación.

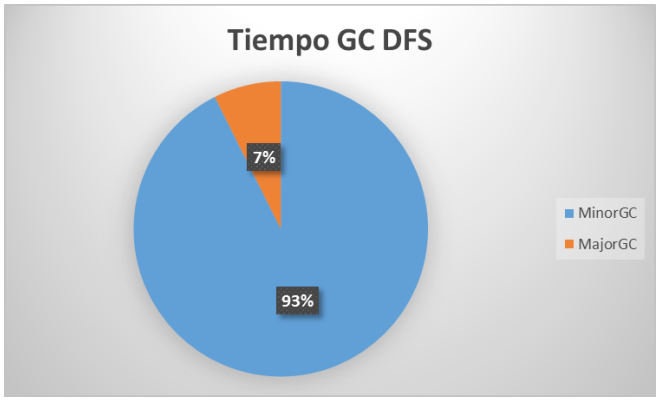
S0C	S1C	S0U	S1U	TT	MTT	DSS	EC	EU	YGC	YGCT
2560	3072	0	192	**7**	15	**2560**	**818688**	**794710**	**23**	**0,179**
1024	2560	544	0	**8**	15	**2560**	**970240**	**169615**	**24**	**0,182**

Tabla 21 Comparación TT y DSS.

5.2.8. Tiempos de eventos GC

Según los datos que de los que se disponen una vez finalizada la ejecución, se observa que los eventos de la Young Generation han acumulado 99136 eventos durante 28 horas, de los cuales ocupando 590 segundos.

Respecto al total de eventos FGC y su tiempo acumulado, se observa que solo 47 han sido ocupados por FGC, repartido en un total de 3343 eventos.
Se compararán estos datos en el apartado conclusiones entre las dos aplicaciones desarrolladas.



5.3. Conclusiones DFS frente BFS

La comparación entre los resultados obtenidos reflejan que DFS acumula de media un 57% de todos sus objetos vivos frente a un 23% de media en BFS. Esto se debe a que DFS utiliza ambos métodos iterativo y recursivo mientras que BFS solo realiza las búsquedas iterativamente, por lo que la cantidad de instancias realizas es mayor a cada nueva ejecución que se realiza. Esto afecta en la cantidad de objetos creados, aumentando considerablemente la cantidad de memoria utilizada.

Ambos Eden Space de BFS y DFS, muestran un resultado bastante dispar. Mientras que BFS tiene un uso de Eden Space bastante regular, DFS muestra un uso con muchos máximos, hecho que certifica el uso del 100% de ES. Tal comportamiento se justifica por la cantidad de procesos que realiza, 800 imágenes cuando resuelve el árbol binaro iterativamente más otras 800 cuando lo resuelve recursivamente. Esto provoca que ES alcance su máximo provocando así más cantidad de Minor GC reduciendo así el espacio usado de YG.

Las diferencias entre S0 y S1 entre BFS y DFS son abismales. En DFS se puede apreciar claramente una media en el uso de estos espacios S0 y S1, entre los 150Mb y los 200Mb, un 50% de la capacidad total. BFS sin embargo arroja unos datos distintos. Alrededor de un 60% de uso de S0 y S1 con picos superiores al 90%.

Lo primero que diferencia a OG entre BFS y DFS es la capacidad máxima. Sin duda en DFS la previsión de usar más espacio es mayor, y se puede contrastar. En algunos momentos el uso de OG asciende hasta los 900Mb mientras que el máximo uso de OG en BFS es de 700Mb aproximadadamente. Sin duda se puede contastar que DFS realiza más operaciones que BFS.

Por último, la frecuencia de eventos Minor GC entre DFS y BFS es la siguiente:

- BFS sufre un Minor GC cada 0’02 segundos mientras que en DFS detectamos Minor GC cada 0’0059 segundos. Esto demuestra el alto grado de creación de objetos en DFS frente BFS.
- Sin embargo, los tiempos de Full GC son idénticos, cada 0’014 segundos.

Esto evidencia el alto grado de impacto en el Garbage Collector que ejerce DFS. Realizar por cada ejecución, un proceso iterativo y recursivo es lo que marca la diferencia. Sin embargo, encontrar un tiempo de Full GC igual demuestra una gran eficiencia en el algoritmo utilizado por Garbage Collector.

5.4. Dijsktra

5.4.1. VisualVM

Se ha observado el uso de CPU de lo principales métodos de la aplicación Dijkstra y se observa lo siguiente:

1. *buildMace*, que se encarga de construir la mazmorra aleatoria es el método que más tiempo de CPU acumula, llegando en algunos momentos al 69%.
2. A pesar de que la aplicación Dijkstra no es multi-threading, si que consideramos cada ejecución nueva de Dijkstra un *task* distinto, por esa misma razón run() acumula el 30% de tiempo de CPU. Si estimamos que alrededor durante 1,008e+8ms se han ejecutado 40320 veces el mismo programa, es normal observar que run acumula tanto tiempo de CPU.
3. El tercer método, doDijkstra(), que realiza toda la lógica algorítmica de Dijkstra, acumula un 10,54% de tiempo de CPU, mostrando así que consume menos tiempo que CPU que construir un laberinto.

5.4.2. Histogramas

Con tal de comparar el número de instancias creadas en Dijkstra, junto al número de bytes ocupados, se ha tomado una muestra a las 36 horas de ejecución. Los resultados muestran que en el top 3 de clases con más instancias son:

5.4.2.1. Histo Live

Num	Instances	Bytes	Class Name
1	3276	195832	[C
2	388	121952	[B
3	3265	78360	java.lang.String

Tabla 21

5.4.2.2. Histo D64

Num	Instances	Bytes	Class Name
1	528188	13117392	[C
2	82296	3950208	Node
3	49044	2770688	[Ljava.lang.Object;

Tabla 22

5.4.3. Survivor 0 y Survivor 1

Tras extraer los datos de S0U y S1U, se observa que el uso de ambas zonas en Dijsktra es muy bajo respecto a DFS y BFS.

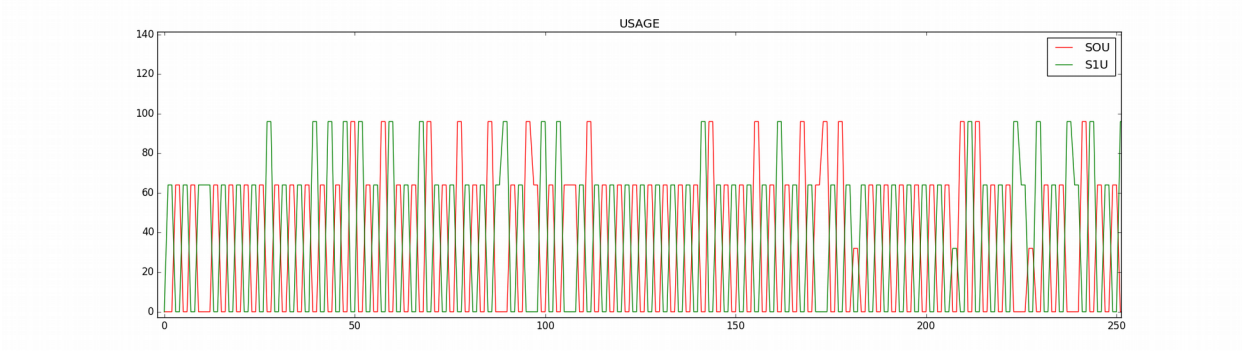


Ilustración 41: Uso de Survivor 0 y Survivor 1 en DFS

5.4.4. Eden Space

Analizando lo datos que arroja Eden Space se puede concluir que a diferencia de BFS que carga y guarda imágenes, crear laberintos aleatorios y buscar su salida haciendo uso de Dijkstra no muestra un gran impacto en memoria, ni de creación de objetos en Eden Space. De hecho solo se ve al inicio de la ejecución un uso superior de espacio que se reduce en los primeros minutos.

A pesar de que el programa realizado con Dijkstra es más liviano, se observa que durante los primeros 42 minutos de ejecución, el uso de ES es bastante exhaustivo.

Lo que ocurre en los primeros 41 minutos es lo siguiente:

- Los valores iniciales de TT son 15 y su MTT es de 15 también, pero se observa que los objetos no son transferidos a Old Generation, ya que se observa que el uso de OG es del 0'39%, por lo que la única explicación a este caso es que los objetos pierden su referencia y rápidamente son null. En ese caso GC reclama el espacio ocupado por estos. Sin embargo, curiosamente, Permanent Generation es la única que muestra un uso alto de su espacio, un 66%.

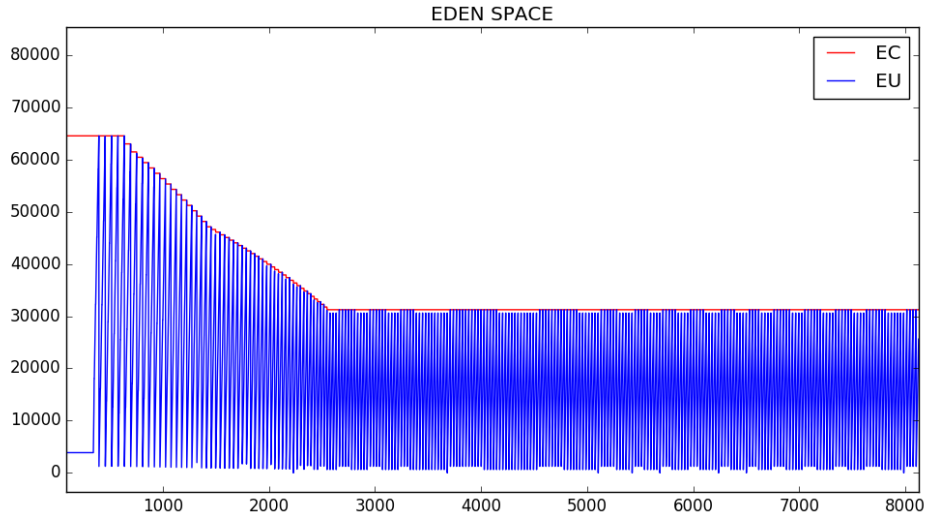


Ilustración 42: Eden Space DFS

5.4.5. Old Generation

Como se ha comentado con anterioridad, OG solo recibe objetos con un tamaño considerable u objetos cuyo edad ha aumentado demasiado para estar en ES. Alrededor de los 700ms, se observa un aumento del 0,298% en el uso de esta generación.

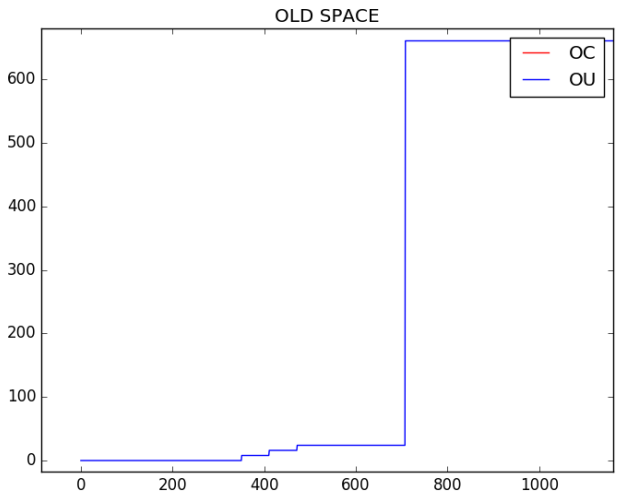


Ilustración 43: Old Generation DFS

5.4.6. TT y DSS

Es importante subrayar que TT es el parámetro que limita el número de veces que un objeto sobrevive un evento GC (su edad generacional). Observando la gráfica se concluye que solo al inicio de la ejecución TT desciende de 15 a 1 para mantenerse en ese nivel durante toda la ejecución, es decir, los objetos no alcanzan nunca el máximo de espacio establecido y por eso no tiene necesidad de aumentar. En otra palabras, nada más crearse los objetos, pierden cualquier referencia pasan a ser null.

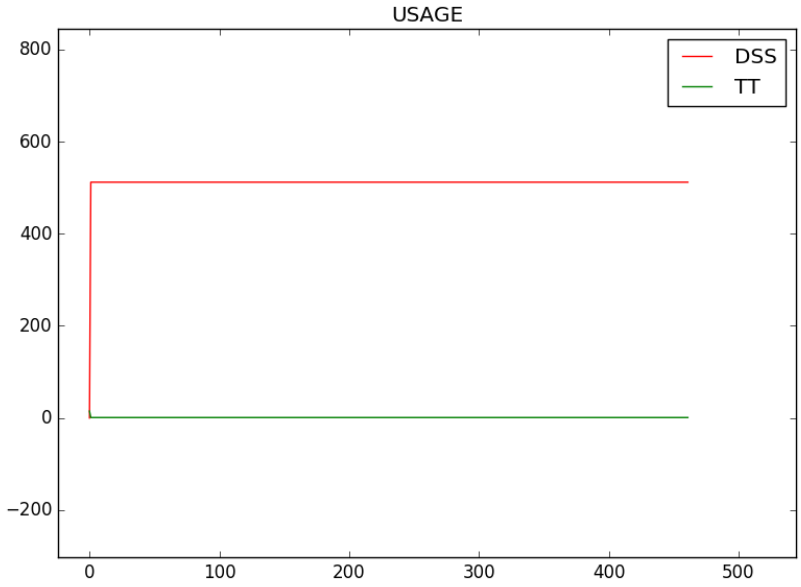


Ilustración 44: TT y DSS en DFS

5.4.7. S0U, TT y DSS

Es interesante comentar el comportamiento aquí de DSS. Los primeros 10 minutos detecta que el uso de S0U aumenta y decide ampliar todavía más el parámetro DSS, pronosticando que hasta los 10MB de capacidad se podrían ocupar. A medida que avanza la ejecución el uso de S0U se estabiliza y reduce su tamaño hasta bajar a los 1000Kb de capacidad. Una vez más se confirma que DSS es un parámetro que dinámico que trabaja en función de las necesidades de la memoria, y te indica cual sería la capacidad máxima/deseada de objetos superviviente.

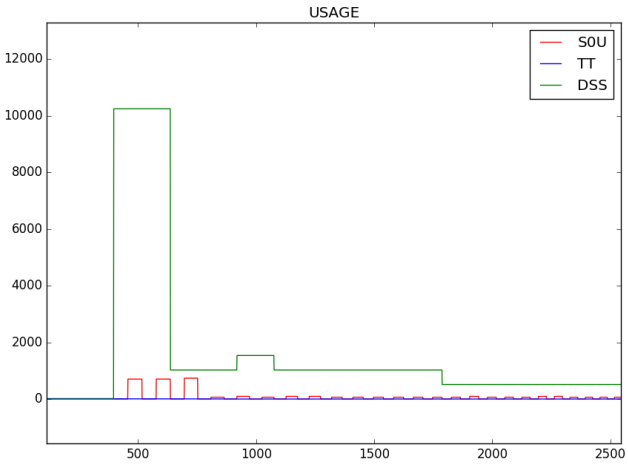


Ilustración 45: S0U, TT y DSS en DFS

5.4.8. TT y S0U

Si ampliamos la Ilustración 45 observamos lo siguiente:

- El valor de TT disminuye hasta alcanzar su valor mínimo 1. Esto se debe al tipo de implementación presente en la aplicación Dijkstra. Acabar.

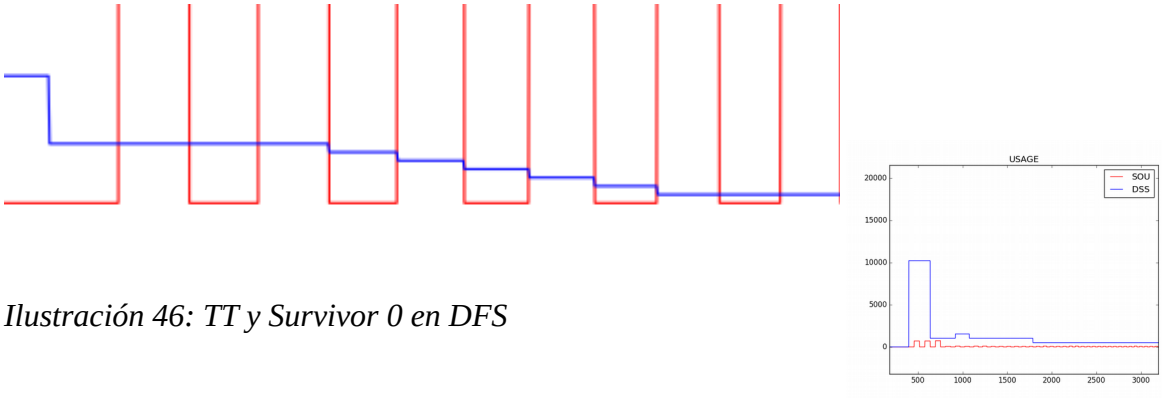
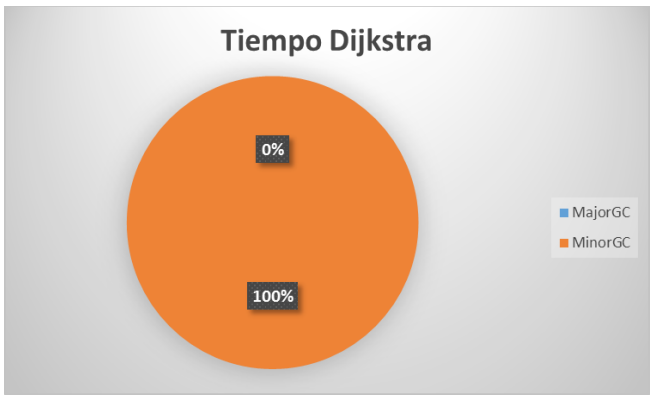


Ilustración 46: TT y Survivor 0 en DFS

5.4.9. Tiempos del GC

Tras 1700 minutos de ejecución, los datos de acumulación en lo que a tiempo respecta refleja lo siguiente:

- Según los datos que proporciona gcnew, el número de total de eventos en la YG asciende a 1542, con un total de 591 segundos.
- Mientras que los datos respecto a FGC, muestran que no se ha activado ningún evento en OG.



5.4.10. Conclusiones Dijkstra

El análisis realizado sobre la aplicación Dijkstra demuestra que es la aplicación, que en relación con las otras aplicaciones, es la que más evento Minor GC realiza.

Debido a la corta vida de sus objetos, OG nunca realizará Major GC. El hecho de cargar imágenes en las anteriore aplicaciones marca la diferencia entre los resultados.

Se confirma con Dijkstra que ES siempre usará el máximo de capacidad.

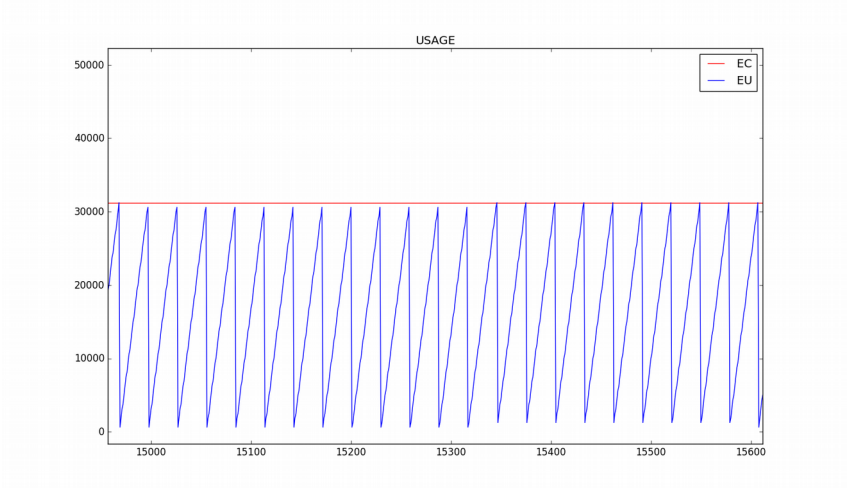


Ilustración 47: Uso ES

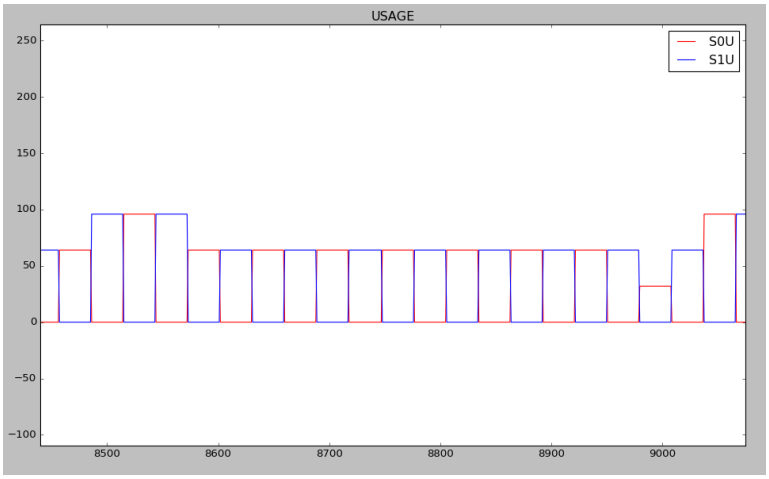


Ilustración 48: Uso S0 y S1

Sin embargo, no existe una relación entre el uso intensivo del GC con el tamaño de datos que pueda manejar la aplicación. Dado que se ha observado este comportamiento, se ha realizado una prueba con cada GC disponible, ya que Dijkstra no necesite del Parallel Collector.

Efectivamente, si utilizamos SerialCollector, el tiempo de ejecución disminuye un 161%. Por lo tanto, se puede confirmar que usar un GC u otro puede aportar una ganancia en tiempo de ejecución muy importante para aplicaciones cuyo set de datos sea bajo. Esto reafirma lo que dice la teoria, SerialCollector está enfocado para aplicaciones con un tamaño de datos reducido.

Garbage Collector	Tiempo (ms)
SerialCollector	312,561
ParallelGC	505,331
UseConcMarkSweepGC	419,596

Tabla 23 Pruebas con distintos GC

6. El GC es optimizable

En cualquier Garbage Collector, es posible ejecutar cualquier aplicación con unas medidas del heap personalizadas al tipo de programa que llevará a cabo. Alterando parámetros como el tamaño del heap o de la YG, es posible ganar unas milésimas en el tiempo de ejecución. Se considera que hay dos aspectos importantes a la hora de realizar estos cambios:

1. Rendimiento: porcentaje de tiempo gastado en realizar el proceso del Garbage Collector.
2. Tiempos de parada: tiempo que la aplicación ha permanecido parada debido al proceso del Garbage Collector.

Por ejemplo, se pueden reducir las paradas en la YG usando un tamaño reducido de dicha generación. Sin embargo, esta reducción, en teoría, no debe afectar al proceso del GC en otras generaciones tales como OG o PG.

6.1 Elección óptima del Garbage Collector

Es posible encontrarse ante un sistema con unas características mínimas, como puede ser 1 procesador monocore, aspecto que limitará bastante el rendimiento de la aplicación. Normalmente la JVM instaladas escogería SerialCollector, recomendado para sistemas como el descrito, aunque es posible hacer uso de Parallel Collector.

Los siguientes pasos se deberían realizar para obtener un buen rendimiento:

- Si la aplicación tiene un data set de alrededor 100MB, escoger SerialGC.
- Si la aplicación se ejecuta en un procesador monocore y no hay requisitos sobre los tiempos de parada, dejar a JVM escoger un GC o directamente escoger Serial GC.
- Si se prioriza el rendimiento sobre todo y no hay requisitos a lo que tiempos se refiere, utilizar ParallelGC, y opcionalmente ParallelOldGc.
- Si el tiempo se prioriza antes el rendimiento, utilizar MarkSweepGC.

6.2. Redimensionar las generaciones

Si el Garbage Collector resulta ser un cuello de botella en términos de procesado, es mejor que el programador ajuste los parámetros del tamaño máximo del heap y las generaciones por separado.

JVM proporciona el siguiente listado de parámetros para que cualquier usuario pueda alterar los valores default del tamaño del heap. Dichos parámetros pueden aumentar o disminuir el tamaño del heap.

Option	Default	Descripción
-Xms	512(MB)	Tamaño inicial del heap en bytes
-Xmx	512(MB)	Maximo tamaño del heap en bytes
-NewRatio	2	Espacio relativo entre Old Generation y Young Generation.
-NewSize	87MB (87031808 bytes)	Capacidad inical de Young Generation
-MaxNewSize	4177MB (4177526784 bytes)	Nueva capacidad de Young Generation
-SurvivorRatio	8	Parámetro que estable un ratio entre S0 y S1
-XX:MinHeapFreeRatio	0	Mínimo ratio entre objetos vivos y espacio libre
-XX:MaxHeapFreeRatio	100	Máximo ratio entre objetos vivos y espacio libre
XX:YoungGenerationSizeIncrement	20	Porcentaje de incremento de YG
XX:TenuredGenerationSizeIncrement	20	Porcentaje de incremento de OG
XX:AdaptiveSizeDecrementScaleFactor (D)	4	Si el crecimiento es X, el decrecimiento es X/D

Tabla 23 Parámetros del heap y su descripción

Como se ha explicado con anterioridad, *Xms* y *Xmx* fijan el tamaño del heap. Por defecto, *NewRatio* para servidor JVM tiene un valor de 2, es decir, OG ocupa ⅔ del heap mientras que la nueva generación o YG ocupa ⅓.

Los parámetros *NewSize* y *MaxNewSize* controlan el tamaño mínimo y máximo de YG, aspecto que se puede regular el tamaño de YG a través de estos parámetros, solamente tienen que valer igual. A mayor tamaño de la nueva YG, menos Minor GC ocurren.

El tamaño de la YG en comparación con la OG es controlado por NewRatio. Por ejemplo, configurando NewRatio = 3 significaría que entre OG y YG la proporción es de 1:3, con la consecuencia de que el tamaño combinado entre ES y las respectivas S0 y S1 ocuparán 1:4 del Heap. Si aumenta todavía más el tamaño de YG, más objetos con menor edad se pueden almacenar, reduciendo la necesidad de Major GC.

6.2.1. Pruebas realizadas

Se presenta a continuación la Tabla 24 con los valore por detecto que tiene la máquina donde se han realizado los distintos estudios del Garbage Collector. Como se puede apreciar, el máximo tamaño del Heap es de 4GB, a pesar de que actualmente la máquina en cuestión, dispone de 16GB de memoria RAM disponible.

A continuación se presentan 5 experimentos que intentarán alcanzar un objetivo básico, que parámetro es el que más afecta al rendimiento de GC.
En concreto se han testado 5 parámetros que estructuran el Heap de JVM:

1. *Xmx y Xms*
2. *NewRatio*
3. *SurvivorRatio*
4. *MinHeapFreeRatio*
5. *MaxHeapFreeRatio*

Como se ha comentado anteriormente, los parámetros expuestos en la Tabla 23 se pueden alterar. La siguiente tabla, la 25 realizará los mismo experimentos pero sin igualar los valores Xmx y Xms.

Al iniciarse la JVM, el espacio entero para el heap se reserva, Dicho tamaño se reserva según el rango que indiquen estos dos parámetros Xm (mín y max). Si Xms es más pequeño que Xmx, no todo el espacio se ocupará inmediatamente. La diferencia entre ambos parámetros se considera espacio virtual y no se usa a no ser que las generaciones aumenten su tamaño (Tabla 25), usando finalmente ese espacio virtual. Véase la Ilustración 6.

Se ha realizado primero una muestra con cada GC disponible para tener unos tiempos de referencia para después disponer de los datos en el apartado de las conclusiones.

Garbage Collector	Tiempo (ms)
SerialGC	129061
ParallelGC	127398
ConcMarkSweepGC	128545
ParallelGC ParallelOldGC	127567

Tabla 24 Tiempo con parámetros default con cada GC

6.2.2. Xms y Xmx

La tabla 25 muestra una ganancia de 1 segundo respecto a un valor igual para Xms. Tal y como se ha explicado, tener valores de Xms < Xmx es una ventaja para el rendimiento del GC. Este rendimiento es inversamente proporcional a la cantidad de memoria disponible, por lo tanto, si el margen fuera mayor entre ambos parámetros, se podría mejorar algo más.

Xms|Xmx (iguales)	Tiempo (ms)
256	java.lang.OutOfMemoryError: Java heap space
512	java.lang.OutOfMemoryError: Java heap space
768	129157,3
1024	128954,9
1280	128885,1

Tabla 25 Tests variando el valor de Xms y Xmx

La tabla 26 presenta los resultados con valores de Xms y Xmx diferentes.

Xms (Mb)	Xmx(Mb)	Tiempo (ms)
128	512	OutOfMemorryError
512	768	OutOfMemorryError
768	1024	128223

Tabla 26

6.2.3. NewRatio

Este parámetro controla el tamaño de YG. Si el parámetro vale 3, significa que la relación entre YG y OG es de 1:3. Es decir, el tamaño combinado de YG, S0 y S1 es 1:4 del tamaño total del Heap. Se observa que a mayor valor de NewRatio, la proporción entre OG y YG.

NewRatio	Tiempo (ms)
3	130185,4
4	129006,1
5	128769,5
6	128409,9

Tabla 27

6.2.4. Min y Max Heap Free Ratio

La tabla 28 analiza la mejora del tiempo de ejecución alterando el valor de MinHeapFreeRatio y MaxHeapFreeRatio. Se observa que a medida que aumentamos ambos valores, el tiempo de ejecución disminuye. Consideramos la primera fila de la tabla 28 el peor caso, frente al mejor caso, última fila. La diferencia entre ambos casos es de 2'7 segundos, confirmando que ambos parámetros pueden influir en el proceso del GC, mejorando indirectamente el tiempo de ejecución de la aplicación . Este dato puede parecer irrisorio, pero si se ejecutara el mismo programa 10000 veces, y considerando que de media se tardará 131541ms, en el peor caso, tardaríamos 21,92 minutos, frente a los 21.46 minutos en el mejor caso (128809ms por ejecución).

MinHeapFreeRatio	MaxHeapFreeRatio	Tiempo (ms)
0	25	131541
25	50	129740
50	75	129032
75	100	128407
0	100	128809

Tabla 28 Min y Max HeapFreeRatio

6.3 Conclusiones

Los tests se han llevado a cabo variando solo un parámetro para obtener los resultados más precisos posibles y observar así el impacto que conlleva cada uno de ellos. El valor de *Xms* y *Xmx* puede rebajar en 1 segundo el tiempo de ejecución, mientras que *NewRatio* gana 2 segundos.

Sin embargo, el parámetro que mejores resultados ha ofrecido ha sido Min y Max Heap Free Ratio, tal y como se observa en la tabla 28. Se ha considerado como mejor caso el rango de 0-100, valor por defecto, y el peor caso 0-25. Considerando lo expuesto, el mejor caso consigue bajar el tiempo de ejecución 3 segundos.

7. Conclusiones

El objetivo de este proyecto es el aprendizaje del funcionamiento interno del Garbage Collector de Java. Con los resultados obtenidos en las diferentes aplicaciones desarrolladas en Java, se han podido analizar los distintos resultados, y contrastar que realmente el GC realiza lo estipulado. Se ha realizado un análisis final para intentar optimizar el Garbage Collector, que directamente tiene una influencia en el tiempo de ejecución y de la aplicación.

La contribución más importante de este proyecto es la utilización de la herramienta jstat para extraer datos de las distintas generaciones, ya que consigue recopilar todos los movimentos del GC, los eventos GC, el tiempo que tarda o el uso y capacidad de cada una. Esta investigación permitirá que en futuros desarrollos de software, gracias a la realización de este TFG, se pueda detectar con más facilidad los posibles leaks que puedan ocurrir y la optimización del tiempo de ejecución.

Por otra parte, ha quedado pendiente el estudio de otros parámetros que indirectamente tienen relación con el GC, como son *YoungGenerationSizeIncrement* o *TenuredGenerationSize Increment*, parámetros que regulan el crecimiento de la YG. También se ha dejado atrás el mismo estudio realizado pero en otros sistemas operativos.

El estudio realizado sólo se centra en el sistema operativo Open Suse basado en Linux de 64 bits. Un posible añadido hubiera sido la comparativa de resultados entre SO para comprobar que realmente la JVM es independiente respecto al hardware y al software. Sería interesante comprobar como cada SO gestiona internamente los tiempos de CPU.

8. Referencias

Visitas realizadas en otoño.

[1] VisualVM All-in-One Java Troubleshooting Tool - https://visualvm.github.io/

[2] Java Virtual Machine Specification Oracle https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-1.html#jvms-1.1

[3] Java Virtual Machine Statistics Monitoring Tool, Java SE Documentation http://docs.oracle.com/javase/7/docs/technotes/tools/share/jstat.html

[4] HPROF: A Heap/CPU Profiling Tool, Java SE Documentation

http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html

[5] HPROF: A Heap/CPU Profiling Tool, Java SE Documentation

http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html

[6] Sun Java System Application Server Enterprise Edition 8.2 Performance Tuning Guide, Oracle, https://docs.oracle.com/cd/E19900-01/819-4742/abeik/index.html

[7] Java HotSpot VM Options, Oracle http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html

[8] Standard Edition HotSpot Virtual Machine Gargabe Collector Collection Tuning Guide https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/

9. Anexo

9.1 Consideraciones al ejecutar aplicaciones

Es importante conocer el uso de la aplicación desarrollada ya que eso puede ofrecer pistas sobre como evitar paradas en aplicaciones con un alto nivel de respuesta.

Por ejemplo, en Web Containers o Java Server Pages (una forma alternativa de crear servlets) es importante configurar el tamaño de la PG debido a la alta cantidad de clases que se cargaran. En el peor de los casos se pueden provocar excepciones por falta de tamaño. Por ejemplo *Exception in thread thread_name java.lang.OutOfMemoryError: PermGen space*

Otros aspectos a tener en cuenta cuando se ejecuten aplicaciones complejas son:

- El tamaño del heap inicial. Por defecto el tamaño es de 64MB, demasiado pequeño para aplicaciones servidor.
- A mayor cantidad de procesadores, aumentar la memoria disponible.
- Old Generation no debe ser extensa en tamaño ya que sino provocará paradas en la ejecución.

9.2 Execution Engine

El Java Bytecode que se asigna en tiempo de ejecución a las áreas de datos de la JVM a través del class loader se ejecutan por el motor de ejecución. El EE lee el *Java Bytecode* en la unidad de instrucción, similar a la *CPU* ejecutando un comando tras otros en la máquina. Cada comando del *Bytecode* consiste en un *OpCode* de 1 byte y operando adicional. La ejecución coge un *OpCode* y ejecuta la tarea con el *Operando*, a continuación ejecuta el siguiente *OpCode*.

Java Bytecode es más en un lenguaje interpretable por el humano que no un lenguaje que la máquina pueda ejecutar. Por esa misma razón el *EE* (*Execution Engine*) debe transformar bytecode al lenguaje que pueda ser ejecutado por la JVM. Existen dos maneras de realizar esta traducción:

1. *Interpreter*: este programa lee, interpreta y ejecuta las instrucciones de Java Bytecode una por una. Mientras interpreta y ejecuta la instrucciones, puede interpretar rápidamente un bytecode, sin embargo, ejecuta el resultado lentamente. Es una desventaja del interpretador.

2. *JIT (Just-In-Time)*: JIT, compilador de Bytecode o instrucciones de la VM. Tiene acceso a información dinámicamente mientras que Interpreter es incapaz. Realiza mejor uso de métodos usado frecuentemente, por ende, mejora la optimización. Compensa así el problema de rendimiento que sufre Interpreter.

EE funciona como un interpretador primero, pero en un determinado momento, el compilador JIT compila el bytecode entero para cambiarlo a código nativo.

Después de realizar este tipo de *parser (*traducción*),* la ejecución no interpreta nada más, pero ejecuta directamente usando el código nativo obtenido anteriormente. La ejecución en código nativo es mucho más rápida que interpretar el bytecode por bloques uno a uno.
El código compilado puede ser ejecutado rápidamente debido a que el código nativo está almacenado en caché.

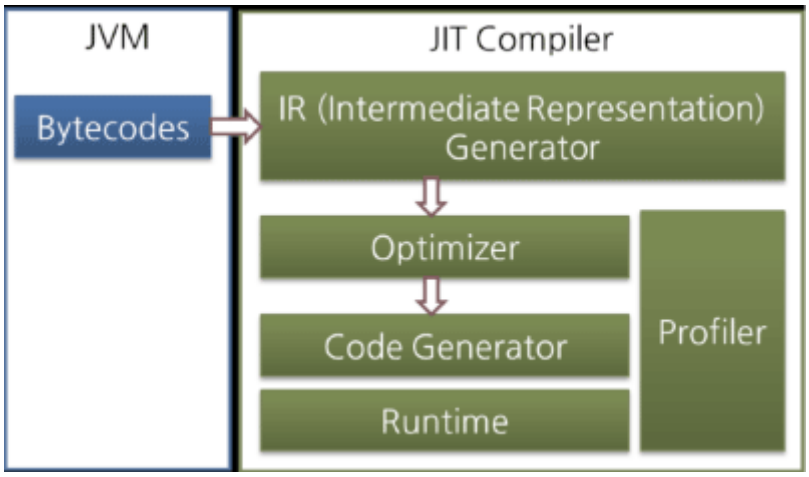


Ilustración 49: Compilador JIT

Tras esta conversión, el compilador JIT convierte el bytecode a IR (Intermediate Representation), y ejecuta una optimización. A continuación convierte la expresión a código nativo.

En concreto *Oracle Hotspot* utiliza un JIT llamado *Hotspot Compiler*. Recibe esta nomenclatura ya que busca métodos que estén utilizando más recursos (*hotspot*), y que requiera una compilación de máxima prioridad. Compila dicho punto a código nativo. Si el método que tiene el *bytecode* compilado no es frecuentemente invocado, en otras palabras, si el método no se considera *hotspot*, HostpotVM retira el código nativo de la caché y vuelve a ejecutar en modo interprete.

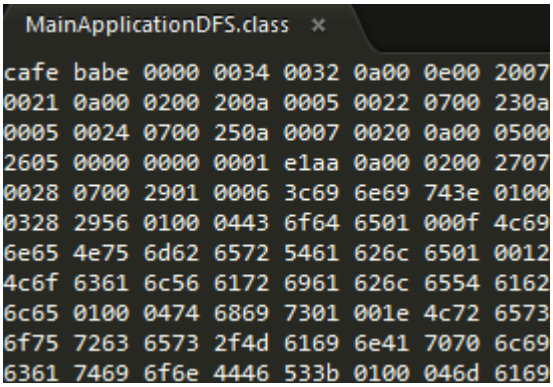


Ilustración 50: Archivo .class

Como se puede observar en la Ilustración 4, el archivo .class es un archivo binario que no puede ser interpretado por un humano. Para poder interpretarlo, JVM ofrece *javap*, un desensamblador. La siguiente Ilustración muestra un ejemplo:

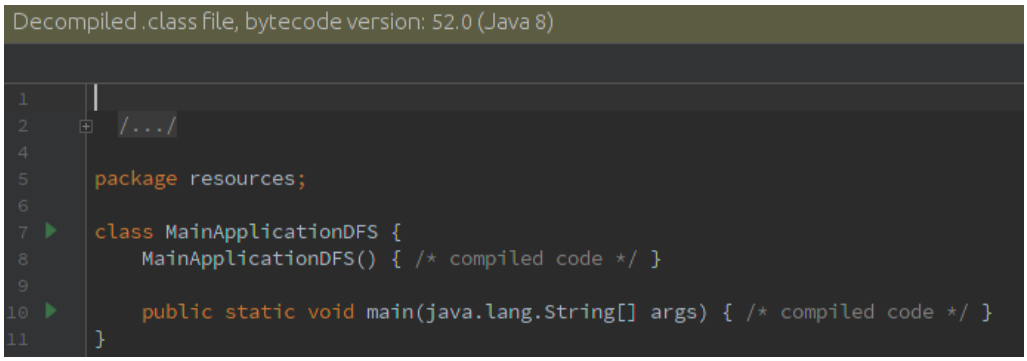


Ilustración 51: Archivo .class interpetrable

Tras compilar el código como se observa en la Ilustracion 5, el código Java Bytecode quedaría de la siguiente manera.

```
 class MainApplicationDFS {
 MainApplicationDFS();
   Code:
      0: aload_0
      1: invokespecial #1                  // Method java/lang/Object."<init>":()V
      4: return
 public static void main(java.lang.String[]);
   Code:
      0: new           #2                  // class java/util/Timer
      3: dup
      4: invokespecial #3                  // Method java/util/Timer."<init>":()V
      7: astore_1
      8: invokestatic  #4                  // Method java/util/Calendar.getInstance:
()Ljava/util/Calendar;
     11: astore_2
     12: aload_2
     13: bipush        7
     15: iconst_2
     16: invokevirtual #6                  // Method java/util/Calendar.set:(II)V
     19: aload_2
     20: bipush        10
     22: iconst_0
     23: invokevirtual #6                  // Method java/util/Calendar.set:(II)V
     26: aload_2
     27: bipush        12
     29: bipush        29
     31: invokevirtual #6                  // Method java/util/Calendar.set:(II)V
     34: aload_2
     35: bipush        13
     37: iconst_0
     38: invokevirtual #6                  // Method java/util/Calendar.set:(II)V
     41: aload₂ 11
     42: bipush        14
     44: iconst_0
     45: invokevirtual #6                  // Method java/util/Calendar.set:(II)V
     48: aload_1
     49: new           #7                  // class DFS
     52: dup
     53: invokespecial #8                  // Method DFS."<init>":()V
     56: aload_2
     57: invokevirtual #9                  // Method java/util/Calendar.getTime:()Ljava/util/Date;
     60: ldc2_w        #10                 // long 123306l
     63: invokevirtual #12                 // Method java/util/Timer.schedule:
(Ljava/util/TimerTask;Ljava/util/Date;J)V
     66: return
}
```

9.3Expresiones en Java Bytecode

Algunas de estas expresiones son presentes en el estudio del *Garbage Collector* en posteriores apartados.

JAVA BYTECODE	TYPE	DESCRIPTION
B	byte	Signed byte
C	char	Unicode character
D	double	Double-precision floating-point value
F	float	Single-precision floating-point value
I	int	Integer
J	long	Long integer
L<classname >	reference	An instance of class <classname>
S	short	Signed short
Z	boolean	True or false
[	reference	One array dimension

Tabla 1: Expresiones Bytecode

10. Acrónimos

Páginas donde poder encontrar los siguientes acrónimos.

GC Garbage Collector {6}

FGC Full Garbage Collector {21, 22, 23, 24, 35, 39, 40, 41, 51, 55, 56}

YGCT Young Garbage Collector Time {22, 23, 24, 49, 50}

YG Young Generation {5, 11, 12, 13, 22, 23, 24, 35, 36, 39, 40, 41, 43, 46, 49, 50, 51, 55, 58, 59, 61, 63}

OG Old Generation {5, 8, 10, 11, 13, 23, 35, 35, 37, 38, 41}

JVM Java Virtual Machine {2, 1, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 20, 24, 25, 26, 27, 44, 58, 59, 60, 63, 65, 67}

DFS Deep First Search {33, 44, 45, 48, 51, 52, 53, 68}

BFS Breadth First Search {33, 36, 38, 45, 51, 52, 53}

CPU Central Processing Unit {1,2,3,4,8,11,16,17,19,20,29,31,32,33,36,38,45,52,63}

DSS Desired Survivor Space {42, 43, 49, 50, 54, 55}

TT Tenured Threshold {10, 23, 35, 42, 43, 49, 50, 54, 55}

MTT Maxim Tenured Threshold {10, 23, 49, 50, 44}

GCT Gargabe Collector Time {22, 23, 24, 35}

S0C Survivor 0 Current Space in Kb {22, 23, 49, 50}

S0U Survivor 0 Usage {23,42,43,46,48,39,49,55}

S1C Survivor 1 Current Usage in Kb {22, 23, 40, 49, 50}

S1U Survivor 1 Usage {22, 23, 40, 41, 42, 46, 49, 48, 50, 53}

API Application Programming Interface {4}

JIT Just In Time {66}

JDK Java Development Kit {1}

IR Intermediate Representation {66}

EE Execution Engine {65, 66}

Xms Mínimo tamaño del Heap {5, 59, 60}

Xmx Máximo tamaño del Heap {61, 62}

SWE Stop the World Event {13}

FGCT Full Garbage Collector Time { 22, 23, 24, 35}

MinHeapFreeRatio {60, 62}

MaxHeapFreeRatio {62}