UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

OPTIMIZACIÓN DE PARÁMETROS DE LA MÁQUINA VIRTUAL DE JAVA CON ALGORITMO GENÉTICO

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

FELIPE ANDRÉS CANALES CARREÑO

PROFESOR GUÍA:
ALEXANDRE BERGEL

MIEMBROS DE LA COMISIÓN:
GEOFFREY HECHT
JOSÉ MIGUEL PIQUER GARDNER
FEDERICO OLMEDO BERON

SANTIAGO DE CHILE
2021

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN
POR: FELIPE ANDRÉS CANALES CARREÑO
FECHA: 2021
PROF. GUÍA: ALEXANDRE BERGEL

OPTIMIZACIÓN DE PARÁMETROS DE LA MÁQUINA VIRTUAL DE JAVA CON ALGORITMO GENÉTICO

Optimizar las opciones de la Java Virtual Machine (JVM) o máquina virtual de Java para poder obtener el mejor desempeño posible de un programa para uso productivo es una tarea desafiante y que requiere una cantidad significativa de tiempo. HotSpot, la implementación de código abierto de la JVM realizada por Oracle, ofrece más de 500 opciones, que reciben el nombre de *flags*, para ajustar los aspectos de compiladores, recolectores de basura o *garbage collectors* (GC), tamaño del *heap* y mucho más. Además de su gran número, estos *flags* se encuentran a veces pobremente documentados y crean la necesidad de realizar pruebas para asegurar que sus valores asociados entreguen mejor rendimiento y estabilidad con un programa específico.

Se han propuesto métodos de ajustamiento automático (*Auto-tuning* en inglés) para reducir esta carga. No obstante, pese a la aparición de técnicas de búsqueda cada vez más sofisticadas que permiten optimizaciones más efectivas, estos métodos no toman en cuenta las complejidades subyacentes de las *flags* de la JVM. Esto se debe a que existen dependencias e incompatibilidades entre *flags* que no son fáciles de expresar, las cuales de no ser consideradas pueden conducir a configuraciones inválidas o absurdas que no deben ser entregadas por el optimizador.

Es por esto que se propone un novedoso modelamiento inspirado por el *feature model* usado en *Software Product Line* que permite reflejar la complejidad de las *flags* de la JVM. También se demuestra la utilidad de este modelo, usándolo como input de un algoritmo genético para optimizar las pruebas de rendimiento de DaCapo.

A mi madre, que estuvo conmigo en los momentos más difíciles, y a mis amigos Rodrigo, Leonardo y Sebastián, siempre dispuestos a escuchar mis problemas.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

El trabajo realizado trata sobre la optimización de la máquina virtual de Java. Debido a que el usuario común de esta se limita a usarla, sin si quiera saber qué hace, es necesario describir qué es y mencionar brevemente su funcionamiento interno. Luego de esto, se procede a indicar el problema que se quiere abordar y porque es de interés abordarlo. Finalmente, se resume el acercamiento utilizado, el cual es explicado con mayor detalle en el capítulo correspondiente.

1.1. La Máquina Virtual de Java

Entre los lenguajes de programación de uso común, Java es uno de los más populares [1]. Una de las características principales que posee es ser un lenguaje basado en la familia de C, compartiendo notorias similitudes en sintaxis, necesidad de compilar el código y otros elementos. No obstante y a diferencia de otros lenguajes de la familia, utiliza casi exclusivamente el paradigma de programación orientada a objetos basada en clases, siendo la introducción de elementos de programación funcional algo relativamente reciente en la vida del lenguaje.

Si bien Java posee varios atributos que influyen en su popularidad, es de especial mención su portabilidad, la cual es presentada junto con la frase *Write once, run anywhere*. Esta frase hace referencia a la posibilidad de correr el mismo programa en distintas plataformas y arquitecturas, sin la necesidad de realizar cambios en el código. Esto se logra porque proceso de compilación de Java no crea un ejecutable (que sería dependiente de la plataforma), sino un archivo de *bytecode*, el cual puede ser interpretado por una máquina virtual. La máquina virtual de Java, o JVM por sus siglas en inglés, es la aplicación que permite correr código escrito en el lenguaje. Existen versiones de esta realizadas por distintas empresas, para diversas plataformas y arquitecturas, sin embargo, todas son capaces de ejecutar un programa compilado a *bytecode*. La máquina virtual que se utiliza en este trabajo es Hotspot de Oracle.

No obstante, el trabajo de la JVM no se limita solamente a interpretar código compilado, pues esto significaría que la ejecución de programas fuera notablemente más lenta a otros lenguajes compilados. Junto con la interpretación, la máquina virtual realiza un compilación dinámica de fragmentos del *bytecode* que recibe el nombre de compilación justo-a-tiempo (JIT por sus siglas en inglés). Este es de hecho, el trabajo más costoso que hace la JVM.

La forma en que opera la compilación JIT es llevando un registro de la frecuencia con la cual se llaman todos los métodos. Al cumplir cierto umbral de llamadas, un método es compilado a instrucciones nativas de la plataforma, optimizando los caminos más utilizados. Por ejemplo, si los registros de un método indican que en un `if` nunca se toma el camino en el cual la condición es verdadera, se omite esa condición en las instrucciones compiladas. También existe un mecanismo para corregir aquellas optimizaciones que no sean correctas. La razón de no compilar todo el código es evitar el costo de la compilación para segmentos que no sean ejecutados o que tengan muy poco uso, en cuyo caso el costo de interpretación va a ser menor a lo largo de la ejecución del programa.

Otra tarea importante de la JVM es la administración de memoria. La máquina virtual posee cierto espacio de memoria que se utiliza para guardar los objetos que se encuentren en uso, usualmente llamado *heap*. Este puede crecer si es insuficiente o disminuir si el uso de memoria es relativamente bajo. Para evitar que objetos en desuso sigan ocupando espacio, se utilizan algoritmos de recolección de basura o *garbage collection* (GC). Estos exploran la memoria buscando objetos que hayan sido derreferenciados y marcan el espacio que ocupan como libre. Es por esto que en Java no se necesita reservar o liberar memoria como se hace en C, pues es la máquina virtual la encargada de esto.

1.2. Problema

La versión de la Hotspot JVM de Oracle, distribuida con Java SE 15, provee una lista de 528 *flags* considerados como producto (en otras palabras, excluyendo grupos de parámetros experimentales, de diagnóstico y otros) al ejecutar el comando `java -XX:+PrintFlagsFinal -version`. Cientos de estas opciones pueden ser usadas con el fin de optimizar el rendimiento de la JVM mediante el ajuste de varios procesos internos. Entre estos se encuentran la administración de memoria, la recolección de basura y el comportamiento de los compiladores. Como ejemplos, el *flag* `-Xmx2G` configura el tamaño máximo del `heap` como 2 GB de RAM, mientras que el parámetro `-XX:+UseParallelGC` indica que la JVM debe usar el recolector de basura paralelo.

Aunque en general, los parámetros por defecto de la JVM son suficientes para correr una aplicación, la tarea se vuelve más compleja si un usuario avanzado desea optimizar el desempeño de un programa. En efecto, la gran cantidad de `flags` pueden tomar una inmensidad de valores entre booleanos, enteros y strings, los que deben ser adaptados a las necesidades de la aplicación y el sistema anfitrión que la ejecuta. Se necesita, por lo tanto, de cierta experiencia para diseñar una configuración que mejore el rendimiento del programa, especialmente porque no todas las opciones se encuentran debidamente descritas en la documentación oficial, a veces requiriendo revisar el código fuente de la JVM [2] u otros sitios expertos para entender sus efectos apropiadamente.

Un ejemplo de este tipo de configuraciones conseguidas manualmente, se puede encontrar en la figura 1.1. Esta popular configuración para un servidor de Minecraft busca optimizar el uso de memoria y el desempeño del recolector de basura. De acuerdo a su autor, la encontró después de muchas semanas de estudiar la JVM, *flags* y probar varias combinaciones, siendo el resultado de un montón de esfuerzo y de verlo en producción en varios tamaños de servidor, listas de *plugin* y tipos de servidor [3].

Una suposición ingenua acerca de este problema es que existen ciertas combinaciones de parámetros que son óptimas en la mayoría de los casos. No obstante, cada programa es distinto, lo que se traduce en diferentes patrones en el uso de memoria y compilación de código, y por lo tanto, diferentes configuraciones óptimas. Por ejemplo, se podría intentar aplicar la configuración de la figura 1.1 a otra aplicación y efectivamente conseguir un incremento en el rendimiento observado, sin embargo, no se tiene garantía que este sea el desempeño óptimo.

Ya se han propuesto métodos para automatizar este proceso de optimizaciones como se explica en el capítulo 2. No obstante, los mecanismos usados para manejar dependencias e incompatibilidades de los diversos *flags* son limitados. Como consecuencia, el espacio de búsqueda de parámetros no es óptimo, lo que hace más difícil y largo el proceso de encontrar la mejor configuración de opciones. Ha de mencionarse también, que estos estudios [2, 4] presentan resultados para OpenJDK7 u OpenJDK8, que se encuentran actualmente sin soporte.

Figura 1.1: Ejemplo de configuración de parámetros recomendada para un servidor de Minecraft.

Se muestra en la figura 1.1 el resultado de un ajuste manual de *flags* de un servidor de Minecraft. Esta configuración de la JVM presenta varias particularidades, como por ejemplo, el parámetro `-XX:G1HeapRegionSize=8M` que solo es válido porque `-XX:+UseG1GC` también está presente en la misma configuración. Además, el *flag* `-XX:+UseParallelGC` (junto con todos aquellos que lo necesitan como requisito previo) no puede ser usado en esta configuración, pues entra en conflicto con `-XX:+UseG1GC`. Este tipo de restricciones no son evidentes e implican la necesidad de tener conocimientos profundos y experiencia de un experto en rendimiento en Java.

1.3. Objetivos

El objetivo general de este trabajo es crear una herramienta capaz de optimizar los parámetros de la JVM para un programa arbitrario en Java. El uso de esta herramienta no debe requerir conocimientos profundos de la máquina virtual o los *flags* disponibles. De esta forma, se permite extender los beneficios de este ejercicio a un mayor número de personas.

Con esta meta en mente, aparecen una serie de pasos necesarios para alcanzarla. Primero, se debe reunir información sobre los parámetros de la JVM, sus interrelaciones y valores aceptados. Segundo, se debe modelar y dar formato a las opciones de la máquina virtual junto con las restricciones encontradas para poder utilizar este modelo como entrada de un optimizador. Tercero, se debe crear un programa que realice el trabajo de optimización de la JVM sobre un aplicación Java, entregando la configuración óptima obtenida. Finalmente, se debe validar el trabajo realizado, confirmando que para una aplicación se puede alcanzar un mejor desempeño que al realizar una ejecución con parámetros por defecto.

1.4. Solución propuesta

Para resolver este problema, se organizan los *flags* de la JVM de la misma forma que se hace con características en un *feature model*. A continuación, se usa este modelo como input de un algoritmo genético capaz de interpretar sus particularidades. Se aplica este enfoque en varias pruebas del conjunto de pruebas de rendimiento DaCapo [5] y otros programas del mismo tipo. El resultado esperado es obtener una configuración que maximice el rendimiento del programa, por ejemplo minimizando el tiempo de ejecución.

Feature model es el nombre que recibe una metodología para el modelamiento de características definida por Kang [6] en su trabajo sobre el análisis de dominio de sistemas de software. Este modelo se utiliza para crear una representación gráfica de las características de una familia de sistemas en el dominio y las relaciones entre ellas. Si bien, el objetivo original de este diagrama es el utilizarlo como medio de comunicación entre usuarios y desarrolladores en el desarrollo de un producto, en el mismo paper se muestra su versatilidad al usarlo para representar las características de un automóvil.

La razón de usar un *feature model* es que al modelar las opciones como características, se pueden establecer las dependencias e incompatibilidades que existen entre opciones. El modelo resultante contiene restricciones que definen todas las posibles configuraciones válidas. En consecuencia, al utilizar este como entrada del algoritmo genético, se reduce el espacio de búsqueda, mejorando así el proceso de optimización.

A su vez, un algoritmo genético se ajusta al problema, pues se puede considerar cada *flag* como un gen. Teniendo esto en cuenta, el sujeto es equivalente a una configuración de opciones y la reproducción entre sujetos es mezclar parámetros entre dos configuraciones. Para la validación de esta solución, se utiliza el resultado de una prueba de rendimiento como función objetivo. Para un programa común y corriente, se podría aceptar alguna métrica que entregue o su tiempo de ejecución.

El uso de pruebas de rendimiento o *benchmarks* resulta evidente, pues el objetivo de este tipo de programas es medir el rendimiento de una configuración de hardware y/o software consistentemente. En consecuencia, si se logra que una prueba de rendimiento reporte un mejor resultado, se puede tener cierta confianza que se obtiene una optimización de la máquina virtual. Qué se considera un mejor resultado es algo que se abarca en el capítulo 5.

En lugar de utilizar las dos herramientas ya descritas, se podría haber creado una solución en base a *frameworks* ya existentes, como OpenTuner [7] o ParamILS [8]. No obstante y

como ya se ha mencionado, trabajos anteriores [2, 4] basados en estas tecnologías presentan ciertas falencias en relación a configuraciones incorrectas, tal como se describe en el capítulo 2. Además, en la solución propuesta, solo el *feature model* es dependiente de la versión de la JVM usada, por lo que trasladar el trabajo a otra versión consiste en simplemente cambiar el modelo para ajustarlo a los nuevos parámetros disponibles.

Finalmente, se ha de destacar que una implementación parcial de esta solución ha sido publicada en la conferencia internacional sobre ingeniería de rendimiento ACM/SPEC [9].

Capítulo 2

Estado del Arte

Se han propuesto varios métodos de optimización automática o *auto-tuning* para diversos dominios entre los cuales se encuentran compiladores [10], sistemas de *runtime* [11], o *deep-learning* [12]. Incluso hay frameworks como ParamILS [8] u OpenTuner [7] que permiten construir *auto-tuners* personalizados para realizar optimizaciones en un dominio específico.

Existen varios estudios que se enfocan en la optimización de las *flags* de la JVM, concentrándose en un subconjunto de opciones especializadas y cuidadosamente seleccionadas [13–18]. De esta forma, se evita la necesidad de modelar las dependencias e incompatibilidades entre *flags*. A continuación, se mencionan dos trabajos [2, 4] que se relacionan en alcance con lo propuesto en este informe, puesto que abarcan un gran número de *flags*.

2.1. Conceptos importantes

Antes de describir los dos trabajos previos más relevantes a lo que se presenta en este documento, se deben introducir algunos conceptos que son necesarios para entender el resto del cápitulo.

2.1.1. Espacio de búsqueda

Este es el nombre que recibe el dominio de un problema de optimización. Este dominio se compone de un conjunto o continuo (dependiendo del problema) de posibles soluciones. Sobre estas se realiza una búsqueda de una o más soluciones candidatas utilizando uno de los muchos algoritmos existentes. En el problema presentado, el espacio de búsqueda corresponde a todas las posibles combinaciones de parámetros y la mejor solución es aquella que optimiza el rendimiento de un programa.

2.1.2. Algoritmo genético y mutación avara

Este algoritmo resuelve problemas de optimización mediante la evolución de una población de soluciones candidatas, que reciben el nombre de individuos. En la primera generación, se crean soluciones aleatorias dentro del espacio de búsqueda. Las generaciones subsecuentes, se obtienen a partir de los mejores individuos y sus combinaciones. Existen varias técnicas para realizar la combinación de dos soluciones. La más común es, para dos padres a y b, definir el hijo x con un punto de corte k, tal que $x_i = a_i$ si $i < k$ o $x_i = b_i$ en otro caso. Luego, estos

nuevos individuos son mutados o alterados aleatoriamente. Una explicación más profunda del algoritmo se encuentra en la seccion 4.2.

La implementación de algoritmo genético utilizada por OpenTuner [7] (llamada mutación avara) tiene características poco comunes. No se realiza una combinación de dos soluciones, sino que se clona a los mejores. Luego, sobre estos clones se realizan mutaciones para obtener nuevos individuos.

2.1.3. Evolución diferencial

Esta técnica de optimización propuesta por Rainer Storn y Kenneth Price [19] obtiene un conjunto de vectores aleatorios que representa a las soluciones candidatas, intentando mejorarlos iterativamente. En cada iteración se combina vectores para obtener nuevas posibles soluciones, que se comparan con los vectores existentes. Estas combinaciones consisten en tomar un vector v y alterarlo en una cantidad aleatoria de dimensiones, garantizando al menos un cambio. La alteración se calcula con la fórmula $n_i = x_i + F \times (y_i - z_i)$, donde n es el nuevo vector, x, y y z son vectores distintos a v y F una variable del algoritmo. Finalmente, la nueva solución n reemplaza v si entrega mejores resultados.

2.1.4. Hill climbing

Este es un algoritmo de optimización que considera solo un vector como solución candidata. Iterativamente se realizan cambios sobre un valor del vector, guardando la nueva solución si da un mejor resultado. Por ejemplo, en un problema de maximización se tiene un vector $x = (3, 15, 8)$. El primer paso realizado es crear otro cambiando el segundo elemento $y = (3, 12, 8)$. Al comparar ambos vectores, se obtiene que $f(x) < f(y)$, por lo que y reemplaza a x como solución candidata.

2.2. Auto-tuning de parámetros de GC

Lengauer y Mössenböck [4] proponen un método, construido sobre ParamILS [8] y su algoritmo de *hill climbing*, para optimizar automáticamente todas las opciones the recolección de basura disponibles para OpenJDK8. Estos autores son capaces de mejorar de forma consistente el desempeño de las pruebas de rendimiento de DaCapo y SPECjbb 2005. También entregan un análisis detallado de sus resultados, siendo este una útil guía para los usuarios avanzados que deseen optimizar sus programas.

Para modelar las dependencias entre *flags*, se usa el mecanismo de parámetros condicionales provisto por ParamILS. Este permite que una opción B sólo pueda ser usada si una opción A posee ciertos valores (usualmente que sea verdadero). Sin embargo, los autores mencionan que este mecanismo no evita que ocasionalmente se entregaran configuraciones incorrectas (llamadas falsos positivos). Esto se debe a que con el método de *hill climbing* se puede establecer el valor de una opción B manteniéndose entre generaciones, pese a que la opción A, de la cual depende, haya cambiado a otro valor incompatible.

Para solucionar este inconveniente, ParamILS permite definir combinaciones de parámetros que no son válidos, no obstante, los autores no mencionan haber utilizado esta herramien-

ta. De todas formas, y pese a que esto permite efectivamente modelar todas las restricciones, el uso de esta característica resulta impráctico, debido a que se requiere listar todas las posibles combinaciones de valores no deseados (por ejemplo, `{flagA=valorA1,flagB=valorB1,...}`, `{flagA=valorA2,flagB=valorB2,...}`, etc). En otras palabras, si hipotéticamente existen 100 configuraciones que incluyen un par de opciones que son mutuamente exclusivas, entonces se debe agregar estas 100 configuraciones inválidas al modelo. Es evidente, que con un número muy alto de posibles combinaciones, y múltiples grupos complejos de *flags*, la cantidad de soluciones no deseadas a registrar puede ser muy grande.

2.3. HotSpot Auto-tuner

Jayasena *et al.* [2] proponen un método similar al que llamaron *HotSpot Auto-tuner*, aunque considerando todos los parámetros disponibles en OpenJDK7 y usando como base el *framework* OpenTuner [7]. Estos autores logran mejorar el desempeño de las pruebas de rendimiento de SPECjvm2008 y DaCapo. Además, encuentran que los mejores resultados se obtienen utilizando todos los tipos de opciones, en lugar de limitarse a solo usar un subconjunto de parámetros de GC o de compiladores. Curiosamente, con DaCapo el rendimiento disminuye al solo usar opciones de recolección de basura, lo que es contrario a lo que describen Lengauer y Mössenböck [4].

Por defecto, OpenTuner utiliza un método más sofisticado que ParamILS para explorar el espacio de búsqueda, dependiendo de una meta técnica *AUC Bandit* que combina algoritmos de mutación avara, evolución diferencial y dos *hill climbers*. No obstante, carece de los mecanismos para tratar con dependencias e incompatibilidades entre *flags*, lo que reduce la efectividad del proceso de optimización. Para resolver este problema, los autores proponen usar una jerarquía de opciones que agrupa aquellos parámetros que son mutuamente exclusivos, como `-XX:+UseParallelGC` y `-XX:+UseG1GC`. Si bien, esta idea permite explorar el espacio de búsqueda más rápidamente, todavía existe propensión a generar configuraciones de opciones que son incorrectas. En específico, no se puede expresar prerrequisitos entre *flags*, por ejemplo, `-XX:+UseAdaptiveSizePolicy` solo es válido con `-XX:+UseParallelGC` o `-XX:+UseG1GC`, pero no con los otros recolectores de basura. Esto se debe a que este tipo de restricciones no obedecen una jerarquía directa, sino que aparecen entre grupos no relacionados directamente.

Capítulo 3

Parámetros de la Máquina Virtual de Java

En este capítulo se abordan en términos generales los *flags* de la JVM, haciendo énfasis en los utilizados para realizar las optimizaciones. En primer lugar, se menciona información relacionada a todos los parámetros existentes y se justifica la razón de excluir algunos del ejercicio de modelado. Tras esto, se procede a entrar en detalle sobre las opciones consideradas, separándolas en varias categorías. El origen de las categorías escogidas es explicado en la sección Generalidades.

3.1. Generalidades

La versión de la HotSpot JVM de Oracle, distribuida con Java SE 15.0.2, entrega una lista de 562 *flags* al ejecutar el comando `java -XX:+PrintFlagsFinal`. Este no es el número total de opciones definidas en el código fuente de OpenJDK (en el cual Hotspot está basado), sino la cantidad de opciones categorizadas como `product`, `manageable` y `product x86`. Este último aparece porque la arquitectura usada para realizar el trabajo es x86-64.

Las principales agrupaciones de parámetros son las siguientes:

- `define_pd_global`: Constantes que dependen de la arquitectura.
- `develop`: Parámetros usados en desarrollo. Solo disponibles en la versión de depuración de Hotspot.
- `diagnostic`: Parámetros usados para depurar la JVM. Disponibles si son precedidos con `-XX:+UnlockDiagnosticVMOptions`.
- `experimental`: Parámetros experimentales. Disponibles si son precedidos con `-XX:+UnlockExperimentalVMOptions`.
- `experimental`: Parámetros exclusivos a la JVM de 64 bits.
- `manageable`: Parámetros que pueden ser cambiados dinámicamente durante ejecución [20].
- `notproduct`: Similares a `develop`.

• `product`: Parámetros disponibles por defecto.

Junto con los presentes en esta lista, hay tres grupos adicionales: `develop_pd`, `diagnostic_pd` y `product_pd`, que dependen de las constantes definidas por `define_pd_global`. Ha de mencionarse también, que existen parámetros que son exclusivos a una arquitectura o sistema operativo en específico. Todos estos *flags* presentan valores por defecto, que son los usados si no son configurados manualmente.

Con la idea de hacer un modelamiento de opciones que sea independiente de la plataforma, no se consideran ni los *flags* exclusivos, ni los que dependan de constantes de la arquitectura, para evitar configuraciones inestables o que causen errores. También se excluyen los parámetros de diagnóstico, por destinarse a entregar información del funcionamiento de la JVM y los experimentales, por ser inestables. Del conjunto de opciones restantes, se descartan algunas más bajo los criterios de no cambiar el desempeño (al menos positivamente) de la máquina virtual o encontrarse deprecados, por ejemplo, `-XX:+PrintCompilation` y `-XX:+UseBiasedLocking` respectivamente.

Tras establecer el grupo de parámetros sobre los cuales se ha de trabajar, se debe obtener información sobre los valores que aceptan, con que otras opciones se relacionan y más. Aquí es donde se presenta una de las grandes dificultades para una persona que quiera realizar una optimización manual, pues la información provista por documentación oficial es muy escasa. El documento que describe el comando `java` [21], detalla algunas de las opciones, pero no todas. También, incluye *flags* de distintos grupos sin diferenciarlas, como `-XX:+LogCompilation` que es una opción de diagnóstico y `-XX:RTMAbortRatio` que es exclusiva de la arquitectura x86. Finalmente, la mención de dependencias es inconsistente. Por ejemplo, en el caso del *flag* `-XX:RTMRetryCount` se indica que necesita que `-XX:+UseRTMLocking` sea verdadero, pero con las opciones `-XX:+UseCountedLoopSafepoints` y `-XX:LoopStripMiningIter` existe una mutua exclusividad que no se declara en el documento.

Es por estas razones que se debe recurrir a otras fuentes de información. Es de especial mención la página *VM Options Explorer* [22], pues contiene información de todos los *flags* con su categorización, valor por defecto y una pequeña descripción. Si bien, esta información ha sido extraída desde el código fuente de OpenJDK, el sitio provee de herramientas para buscar y filtrar bajo diversos criterios. Se utilizan también otras fuentes, pero estas se mencionan junto con la metodología usada para formar el *feature model* en el capítulo 4 sección 2.

A continuación se entra un poco en detalle acerca de los parámetros utilizados. Estos se separan en cuatro categorías establecidas por la misma JVM: GC, C1, C2 y Runtime.

3.2. Garbage Collector (GC)

En esta categoría se encuentran las opciones de la JVM que regulan la administración de memoria. Se pueden establecer tres tipos de parámetros, aquellos que regulan el cambio de tamaño y distribución del *heap*, los que configuran el algoritmo de recolección de basura y los que no entran en los dos grupos previos. Al observar la figura 3.1 se distinguen los dos primeros tipos como los relacionados a *GenSize* y *GarbageCollector*. Cabe mencionar que los diagramas solo indican los grupos jerárquicos, pues no sería factible mostrar una figura con

todas las opciones.

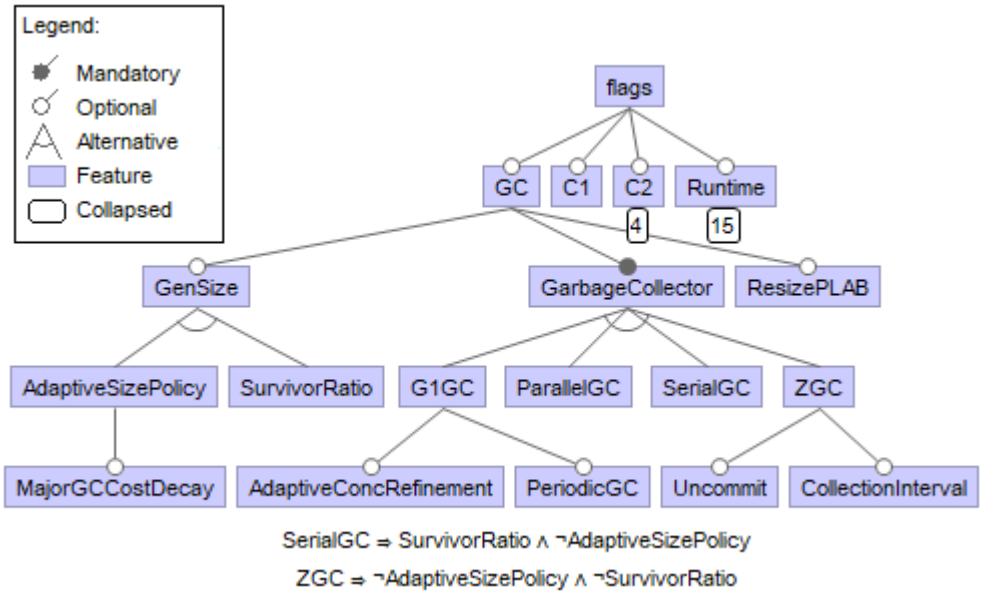

Figura 3.1: *Feature model* de las opciones GC.

La JVM separa su *heap* en varias generaciones (*eden*, *survivor* y *tenured*). Cuando se crea un objeto, este se coloca en *eden* y si es usado por un largo tiempo es movido en otras generaciones [23]. Este proceso se ilustra en la figura 3.2. Hay dos algoritmos, mutuamente exclusivos que se dedican a administrar la cantidad de memoria asignada a cada una de estas. Por un lado, se tiene *AdaptiveSizePolicy*, que cambia dinámicamente el tamaño de las generaciones según las necesidades del programa. Por otro lado, se encuentra *SurvivorRatio* que establece un objetivo fijo sobre la proporción del *heap* que *Survivor* debe ocupar.

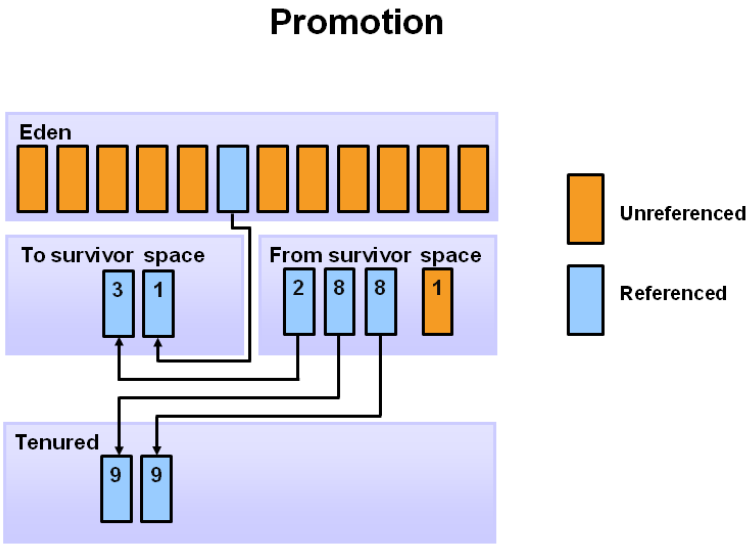

Figura 3.2: Generaciones del *heap* de la JVM [23]

Para el recolector de basura, se encuentran cuatro opciones: *G1GC*, *ParallelGC*, *SerialGC* y *ZGC*. De estos, *SerialGC* destaca por no utilizar paralelismo para la recolección de basura y no ser compatible con *AdaptiveSizePolicy*. El más reciente de estos cuatro es *ZGC*, un algoritmo que utiliza solo una generación [24], por lo que no es compatible con las opciones del tipo *GenSize*. Curiosamente, hay un parámetro para activar *ShenandoahGC*, siendo esta un quinta alternativa de GC, no obstante, este se encuentra en desarrollo (la mayor parte de sus *flags* son experimentales) y al intentar usarlo la JVM entrega el error "`Option -XX:+UseShenandoahGC not supported`" y termina. Cada uno de los recolectores ha sido diseñado con un objetivo o situación específica en mente.

La gran mayoría de optimizaciones de la JVM se centran en esta colección de opciones, lo que la hace un buen punto de partida. De hecho, el trabajo parcial que fue publicado [9], se centra en parámetros de esta categoría. Un detalle que se puede mencionar, es que en los trabajos previos que entregan una combinación de parámetros con un mejor rendimiento, esto es sobre una aplicación o librería en específico.

3.3. Compiladores (C1 y C2)

Para la compilación JIT que realiza la máquina virtual, se dispone de dos compiladores distintos conocidos históricamente como *client* y *server*, pero llamados internamente C1 y C2 respectivamente. Las opciones en estas categorías regulan condiciones y umbrales para la aplicaciones de varias operaciones de optimización de la compilación, como por ejemplo, *inlining*.

El compilador C1 se caracteriza por ser rápido. En términos de la compilación JIT, esto significa que no recopila una gran cantidad de datos sobre la ejecución del *bytecode* antes de empezar a compilarlo. El resultado es instrucciones nativas a la plataforma que se ejecutan más rápido que el *bytecode*, pero que se podrían haber compilado en algo aún más eficiente. Algo a destacar de las opciones relacionadas a este proceso, es la ausencia de dependencias. Todos los *flags* son independientes entre sí, aun cuando regulen la misma característica.

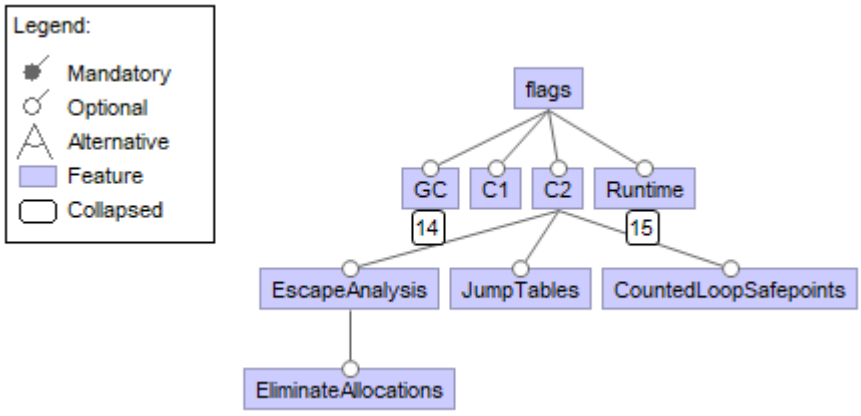

Figura 3.3: *Feature model* de las opciones C2.

El compilador C2, en contraste, reúne más información sobre los métodos antes de com-

pilarlos, consiguiendo así instrucciones nativas mejor optimizadas. La desventaja de este acercamiento, es que se interpreta el *bytecode* por más tiempo, lo que desencadena en que el programa sea notoriamente más lento en un inicio. He aquí la razón por la que este compilador es conocido como *server*, pues en un servidor las aplicaciones se ejecutan por un largo tiempo, tomando provecho de lo positivo y sin ser muy afectado por lo negativo. Aquí si se encuentran dependencias de opciones, como se puede ver en la figura 3.3, siendo lo más notable el grupo *EscapeAnalysis*. Esta optimización permite reemplazar la creación de un objeto, por la inserción de valores primitivos en el *stack* [25].

Ambos compiladores se usan paralelamente en lo que se llama compilación escalonada o *tiered compilation*. Esta técnica consiste en compilar código tan pronto como se pueda con C1 y seguir recolectando información de ejecución. En el momento que el C2 tiene suficientes datos, el código es recompilado, esta vez obteniendo instrucciones más optimizadas. De esta forma, se pueden obtener las ventajas de ambos compiladores, sin sufrir en demasía las desventajas. Las opciones que regulan este proceso pertenecen a la categoría Runtime.

Si bien, el número de *flags* en estas dos categorías es bajo, la información disponible acerca de estos es bastante escasa. El mejor recurso para entender las relaciones entre parámetros en este caso fue el código fuente de OpenJDK15, debido a que estos procesos se encuentran autocontenidos en paquetes particulares.

3.4. Runtime

Esta categoría es la más numerosa y presenta varios grupos e interdependencias. Sus parámetros, en general, configuran varias características de la ejecución de la JVM. De los grupos de parámetros que se pueden observar en la figura 3.4, el más relevantes es sin duda *UseCompiler*. Aquí se encuentran todas las opciones relativas al uso de la compilación JIT, presentando incluso la posibilidad de solo usar el intérprete. Dentro de este conjunto, se encuentra el subgrupo *CompilationType* que determina si se usa solo un compilador (y cuál se usa) o si se aplica la compilación escalonada. Este proceso es regulado por un gran número de umbrales que determinan, entre otras cosas, cuando se empieza a compilar en un escalón, cual es el número de llamadas que debe recibir un método para compilarlo, etc. La optimización de estas opciones es crítica, pues por resultados experimentales, una mala configuración puede empeorar significativamente el rendimiento de un programa.

Otros grupos a mencionar son características específicas a la plataforma. Aquí destaca *NUMA*, que configura el uso de una característica que poseen ciertos procesadores, entregando una advertencia cuando se activa en hardware no compatible. Es dentro de estas opciones que puede existir una mayor diferencia de la configuración óptima para distintas plataformas o combinaciones de hardware.

Finalmente, están los siguientes de grupos de interés: *AllocatePrefetch*, una optimización extremadamente técnica sobre los cachés del procesador que puede empeorar notablemente el rendimiento si los rangos de valores no son los adecuados y *StringDeduplication*, que tiene como restricción que el recolector de basura en uso sea *G1GC*.

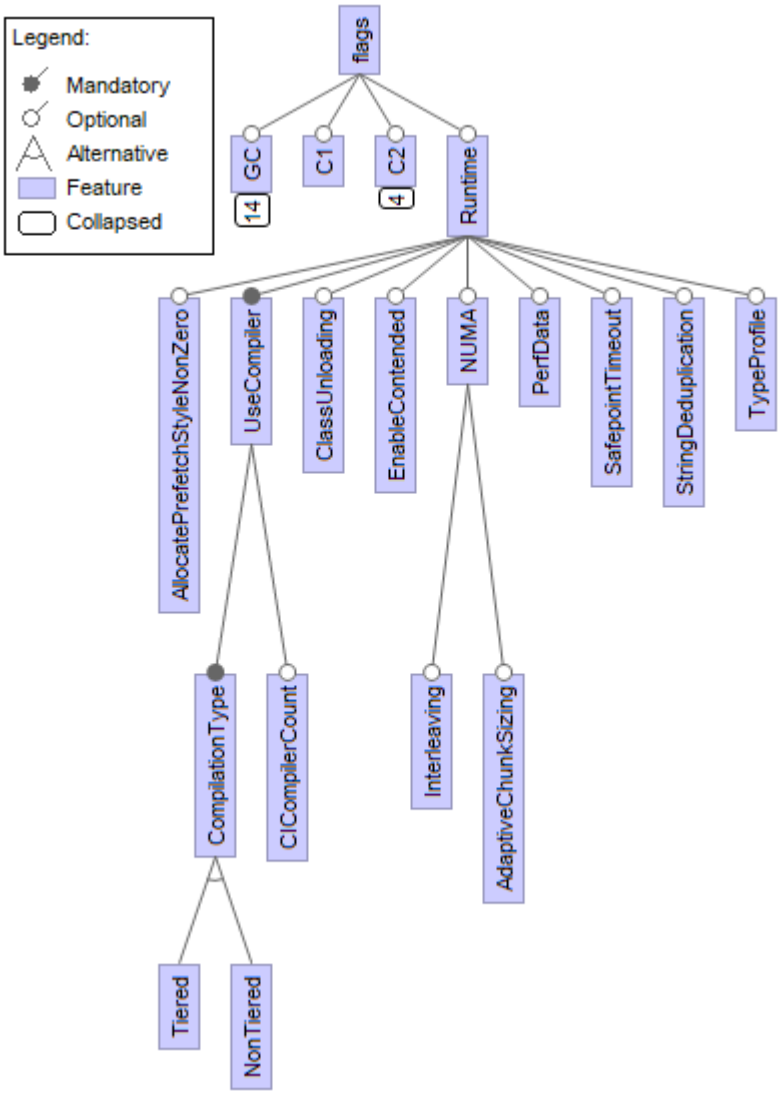

Figura 3.4: *Feature model* de las opciones Runtime.

Aunque se ha mencionado varias veces este problema, la documentación de gran parte de estos *flags* es prácticamente nula. En este caso, el código fuente deja de ser útil como referencia debido a lo dispersas que se encuentran estas opciones en el código, lo que hace más difícil seguir las dependencias. Muchos de los parámetros aquí definidos tienen efectos significativos en el rendimiento de una aplicación, y en ciertos casos, estos pueden ser positivos, como se describe en los resultados. Es por esto, que para obtener información sobre esta categoría fue más importante la experimentación manual con *flags*.

Capítulo 4

Solución

A continuación, se entra en detalle sobre el acercamiento usado para realizar la optimización de *flags* de la JVM. Este consiste, como primer paso, en el modelamiento de las opciones usando un *feature model*, el cual permite representar las dependencias e incompatibilidades que existen. El segundo paso es la implementación de un algoritmo genético, del cual se describen las características que posee para poder emplear el modelo creado como input. Finalmente, se describe el formato con que se definen los genes para lograr asociarlos a parámetros.

4.1. Construcción del Feature Model

Este es el fuerte del trabajo realizado para este informe. Debido a la gran cantidad de *flags* que se utilizan para la optimización, junto con la escasa información disponible en la documentación oficial, se diseña una metodología para poder comprender y organizar, las interrelaciones existentes. También se busca de esta forma los rangos de valores sensatos para cada opción.

En primer lugar, se separan las opciones en las cuatro categorías mencionadas en el capítulo 3. Cada una de estas se trabaja por separado, debido a que la mayor parte de las relaciones entre parámetros solo involucran un mismo grupo. Si bien, se podría trabajar todas juntas, esto aumenta significativamente el tiempo requerido para estudiar las interacciones entre parámetros. De todas formas, existen limitados casos de dependencias de opciones en grupos distintos, sin embargo, no se requiere agregar grupos adicionales para poder modelar estas restricciones.

Para comenzar la estructuración de los *flags* de una categoría, primero se listan con todos los datos provistos por la página *VM Options Explorer* [22], incluyendo los valores por defecto. Junto con esto, se busca información adicional en las referencias oficiales e internet en general. Usualmente, si la documentación es pobre, no se encuentra mucha información valiosa fuera de esta. De hecho, este suele ser el caso aun cuando existe información oficial. No obstante, se organizan los parámetros según dependencias e incompatibilidades descritas explícitamente, también dejando registro de aquellos grupos que parecen estar relacionados (por ejemplo, las opciones que regulan operaciones de *SuperWord*) o que presentan inconsistencias en la documentación disponible.

A continuación se revisa el código fuente de OpenJDK. El objetivo de este paso es comprobar la veracidad de las posibles dependencias o incompatibilidades anotadas en la etapa anterior. Si no se logra confirmar que dos opciones están relacionadas, se asume que son independientes. Es relevante mencionar que, debido al gran número de parámetros, el tiempo dedicado a averiguar sobre cada uno en el código fuente es limitado.

Luego, se hacen pruebas realizando ejecuciones con los diversos parámetros numéricos para determinar rangos de valores que sean apropiados a lo que hacen. Como punto de partida, se establecen los rangos que son especificados en la herramienta *Hotspot Auto-tuner* [2]. Estos son todos los valores entre 0,5 y 1,5 veces el valor por defecto. En varias ocasiones, se deben hacer correcciones, ya sea porque los límites no son enteros cuando el *flag* solo acepta este tipo de números (por ejemplo, `-XX:HeapSearchSteps`, porque tiene más sentido que los valores de la opción crezcan exponencialmente (parámetros que especifican una cantidad de memoria), o porque existe un rango especifico de valores que se aceptan (parámetros que indican porcentajes).

Finalmente se realiza una optimización con el algoritmo genético usando todos los *flags* de la categoría para comprobar que no existen irregularidades. Si bien, en esta etapa se podrían encontrar conflictos ya sea por errores o advertencias entregadas por la JVM, La gran mayoría de estas no son reportadas ni inhiben la ejecución. Por ejemplo, al usar el parámetro *-XX:-ZUncommit* para configurar el *ZGC* y a la vez especificando que se use *G1GC* como el recolector de basura, la JVM realiza su trabajo sin indicar que la configuración es errónea. No obstante, es en este paso donde se encuentra la mutua exclusividad no documentada entre `-XX:+UseCountedLoopSafepoints` y `-XX:LoopStripMiningIter`, siendo esta una de las pocas excepciones a este comportamiento.

El modelo resultante se puede observar de forma fragmentada en el capítulo 3 y en su completitud en la figura 4.1. Las características que se muestran, corresponden a grupos de *flags*. Esto se debe a que si se incluyeran todos, el diagrama sería ilegible. En el *feature model* se puede apreciar que existe una jerarquía entre opciones correspondiente a los distintos niveles del árbol (algo ya propuesto en un trabajo anterior [2]). A esto se añade los tipos y características de relaciones entre opciones definidas para este tipo de diagramas, las cuales son descritas a continuación:

- **Alternativas** entre *flags*, como por ejemplo, la elección de un único recolector de basura.
- **Combinaciones** de *flags*, como *Interleaving* y *AdaptiveChunkSizing* que comparten una dependencia a *NUMA*, pero representan grupos independientes entre sí.
- Parámetros **Obligatorios** u **Opcionales**, por ejemplo, se debe indicar el recolector de basura usado, pero es posible omitir las opciones relacionadas a *NUMA*.
- **Restricciones** entre grupos de opciones expresadas con operadores lógicos como $\wedge$ (*and*), $\vee$ (*or*), $\neg$ (*not*) e $\implies$ (*implica*). Por ejemplo, *ZGC* es incompatible con *AdaptiveSizePolicy* y *SurvivorRatio*, pues este recolector de basura usa una generación para todo el *heap*.

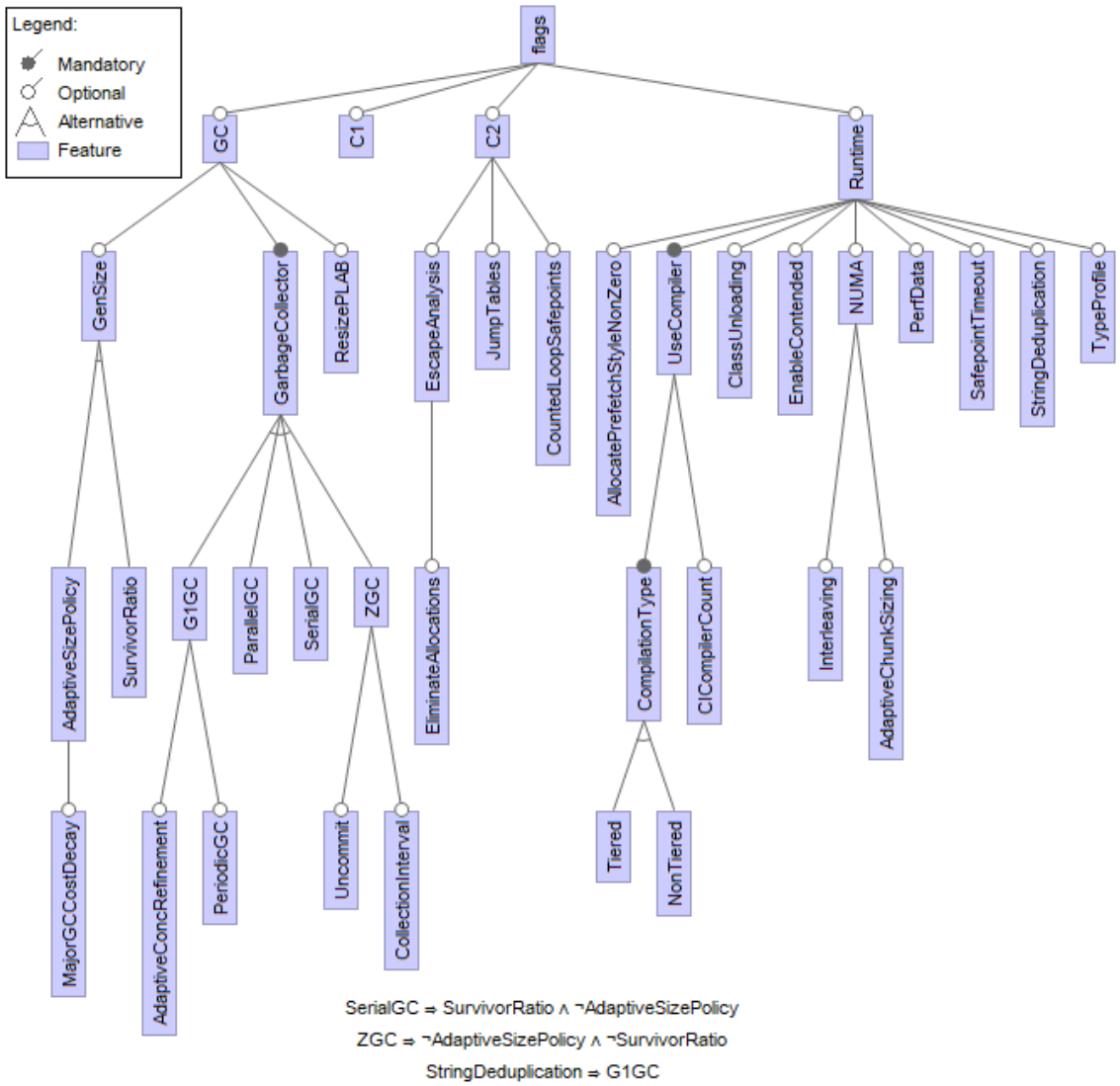

Figura 4.1: *Feature model* de las *flags* de la JVM.

Se puede inferir del procedimiento descrito, que existen dificultades para identificar todas las dependencias e incompatibilidades entre opciones, siendo posible que existan algunas las cuales hayan sido pasadas por alto. En efecto, ese problema es el enfrentado cuando una persona quiere realizar una optimización manual de la JVM para una aplicación específica. Lamentablemente para los alcances de este trabajo, ser más minucioso en el modelamiento de parámetros requeriría de más tiempo para familiarizarse a fondo con el funcionamiento de la JVM o tener ya un conocimiento sobre el tema. No obstante, el modelo obtenido expresa una cantidad importante de interrelaciones que ya disminuye en gran cantidad las configuraciones inválidas y puede ser también alterado fácilmente por una persona que posea los conocimientos apropiados.

4.2. Algoritmo Genético

Para la optimización de los parámetros se utiliza un algoritmo genético. Está técnica corresponde a una heurística para resolver problemas de optimización que está inspirada en la selección natural procedente de la biología. En su forma más simple, esta consiste en: genes que representan valores o características, sujetos que corresponden a una cadena de genes y generaciones que agrupan un número de sujetos.

Para iniciar la primera generación, se crean individuos a partir de la selección aleatoria de valores para cada gen. Luego, se evalúa el desempeño de cada sujeto mediante el cálculo de una función objetivo. Posteriormente, se descarta a los peores resultados de la generación y se obtiene la nueva rellenando las vacantes creadas con la cruza de los mejores individuos. En este proceso, se crean dos nuevas cadenas genéticas tomando de forma aleatoria los genes de los padres. Después de esto, se suele hacer una etapa de mutación sobre algunos de los nuevos individuos para añadir variabilidad a la población. Un resumen de lo descrito se encuentra ilustrado en la figura 4.2.

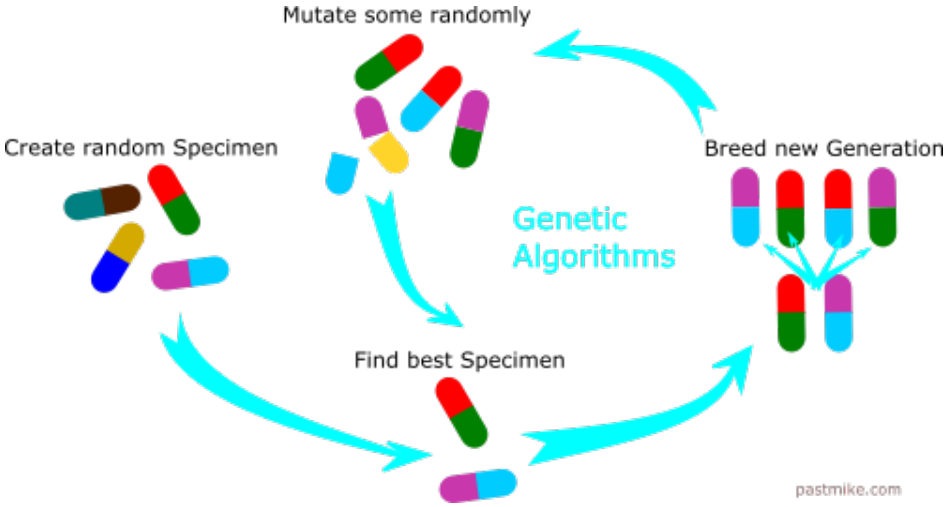

Figura 4.2: Ilustración de cómo funciona un algoritmo genético [26].

El algoritmo genético modela las diversas configuraciones de la siguiente forma: un gen representa un *flag* y el valor del gen representa el valor que el *flag* posee en una configuración (booleano, entero o decimal). A su vez, un individuo corresponde a un conjunto de valores ordenados de opciones. La función objetivo toma como argumento un individuo, lo transforma en una configuración de la JVM, ejecuta un programa utilizando esta configuración y entrega como resultado un valor que puede ser el tiempo de ejecución medido o una métrica reportada por una prueba de usuario. El cruce o *crossover* de individuos que se hace tras cada generación es el más utilizado en algoritmos genéticos, consistiendo en escoger un punto de corte k y luego creando un nuevo sujeto con los valores del padre 1 hasta k y del padre 2 desde k, como se puede ver en la figura 4.3.

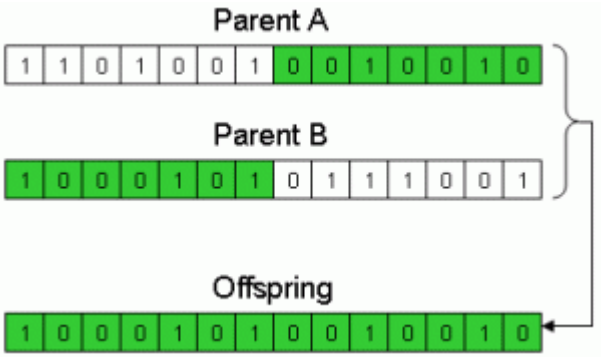

Figura 4.3: *Crossover* entre individuos de un algoritmo genético [27].

Junto con todos los elementos comunes de un algoritmo genético, se implementa también la capacidad de desactivar algunos *flags* mediante genes de control. Esta es una característica que también se da en la naturaleza: ciertos genes pueden regular la expresión de otros, proceso que se ilustra en la figura 4.4. De esta forma, se pueden crear grupos de genes que sean mutuamente exclusivos y dependan del valor de un gen de control, como puede ser una variante del recolector de basura. En otras palabras, es una forma de implementar las **alternativas** expresadas en el *feature model*. Esto también resulta útil para implementar las **restricciones**, pues al utilizar un *flag* se puede desactivar otros (ya sean regulares o de control) que sean incompatibles.

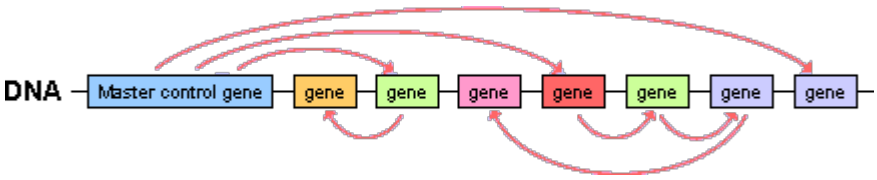

Figura 4.4: Relación entre genes de control y otros genes [28].

La implementación del algoritmo genético se realiza con el lenguaje de programación Python, pues este permite un rápido desarrollo del programa en cuestión. Además, la relativa lentitud de un lenguaje interpretado con un lenguaje compilado es irrelevante cuando la mayor parte del tiempo de ejecución cae en la máquina virtual de Java. Junto a esto, se decide no ocupar librerías existentes de algoritmos genéticos, debido a la necesidad de manejar las dependencias y relaciones entre parámetros. Esta es una dificultad inherente al problema abordado y que no es considerada en las librerías disponibles.

Por último, es interesante mencionar que esta técnica de optimización pertenece a una familia que recibe el nombre de algoritmos evolutivos. La característica principal de estos es poseer una o más soluciones que "evolucionan" o se ven alteradas entre generaciones. Dentro de esta categoría se encuentran otros métodos de optimización que han sido mencionados en el capítulo 2.

4.3. Estructura de los genes

El *feature model* obtenido es utilizado como entrada del algoritmo genético. Esto significa, que se debe traspasar las relaciones de dependencia provenientes del modelamiento de los

parámetros a un formato que contenga la información de cada opción. Con este objetivo se utiliza el lenguaje JSON, pues entrega gran flexibilidad para estructurar los datos. La estructura utilizada para describir un parámetro es la que se puede observar en la figura 4.5.

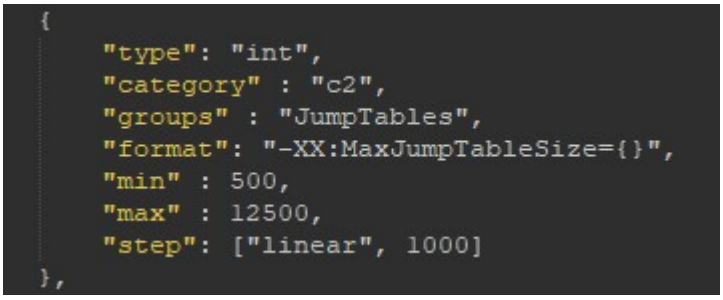

Figura 4.5: Estructura de una opción como gen.

Un gen usando este formato define un *flag* de la JVM, especificando los valores posibles, dependencias y más. A continuación se describen los distintos datos que se encuentran presentes:

- `type`: El tipo de valor que utiliza la opción (booleano, entero, etc).
- `category`: El grupo de parámetros al que pertenece según la documentación oficial.
- `groups`: Conjunto de características del modelo que son prerrequisitos. Si uno de estos grupos se encuentra inactivo, este gen no se expresa en la configuración.
- `format`: Como se expresa esta opción como argumento de la JVM.
- `min`: El valor mínimo de un parámetro numérico.
- `max`: El valor máximo de un parámetro numérico.
- `step`: La discretización del rango de un parámetro numérico. Puede ser `linear` (lineal) o `power` (geométrica), siendo el segundo valor la diferencia o razón respectivamente entre valores consecutivos.

Un detalle que puede causar confusión es el hecho de que el rango de valores posibles se encuentra discretizado por *pasos*. Algunos parámetros requieren que los valores usados sigan ciertas condiciones, por ejemplo `-XX:ContendedPaddingWidth` solo acepta múltiplos de 8. De forma más general, este es una decisión consiente para disminuir el tiempo de convergencia del algoritmo mediante la reducción el espacio de búsqueda, pues el proceso de optimización toma una gran cantidad de tiempo. A cambio, se sacrifica la capacidad de llegar a un óptimo, si este se encuentra entre dos de los valores posibles, existiendo espacio para mejorar la optimización.

Además, se debe profundizar un poco en las particularidades de la variable `groups`. Para determinar si un gen está activo, se realiza un AND lógico entre todos los grupos expresados. Actualmente no se puede manejar una restricción que requiera un OR lógico, no obstante, no

existen restricciones de este tipo en el modelo. En consecuencia, hacer cambios para permitir estas interacciones es innecesario.

La desactivación de grupos de parámetros se hace con genes de control que siguen la estructura ilustrada en la figura 4.6. Cada gen de control corresponde a una característica presente los diagramas del *feature model*. En este caso, el gen que se muestra corresponde a la característica *AdaptiveChunkSizing* del grupo *NUMA* de la categoría Runtime. En un sujeto del algoritmo genético, se define el valor de este gen como uno de los objetos de la variable `variants`, la cual, indica cómo se expresa el gen y que grupos marca como inactivos. En este caso, una de las variantes no posee formato, porque se encuentra activa por defecto.

```
{
    "type": "control",
    "category" : "runtime",
    "groups" : "NUMA",
    "variants" : [
        {
            "format" : "-XX:-UseAdaptiveNUMAChunkSizing",
            "deactivate": ["AdaptiveChunkSizing"]
        },
        {
            "format" : "",
            "deactivate": []
        }
    ]
},
```

Figura 4.6: Gen que representa la característica *AdaptiveChunkSizing* del modelo.

4.4. Experimentos

La solución descrita se evalúa mediante la optimización de diversas pruebas de rendimiento con el objetivo de probar si este acercamiento es capaz de incrementar el desempeño de estos programas. Entre estas se incluyen algunas de las pruebas provistas por DaCapo [5] versión 9.12, la cual es usualmente utilizada para obtener métricas de desempeño sobre Java con especial mención a trabajos relacionados con este [2, 4]. No obstante, solo se usan las pruebas *avrora*, *fop*, *jython*, *luindex*, *lusearch*, *lusearch-fix*, *pmd* y *xalan*. Aquellas que no son mencionadas, presentan problemas de estabilidad que hacen imposible la optimización. Además, se complementan estos resultados con los obtenidos de una prueba de rendimiento sobre la librería de interfaz gráfica JavaFX diseñada por Nikita Prokopov [29].

Para entender un poco que es lo que está midiendo con cada *benchmark* de DaCapo, se incluye a continuación las descripciones correspondientes. Más información sobre estas se encuentra en la documentación oficial.

- `avrora`: Simula un conjunto de programas corriendo en una red de microcontroladores AVR.
- `fop`: Parsea y formatea un archivo XSL-FO, generando un archivo PDF.

- `jython`: Interpreta la prueba de rendimiento `pybench` de Python.
- `luindex`: Se usa la librería Apache Lucene para indexar un conjunto de documentos.
- `lusearch`: Se usa la librería Apache Lucene para realizar una búsqueda de texto de palabras claves sobre un conjunto de documentos.
- `pmd`: Analiza un conjunto de clases de Java para encontrar una serie de problemas en el código fuente.
- `xalan`: Transforma documentos XML en HTML.

También se realiza una optimización sobre una de las pruebas de rendimiento de SPECjvm2008. Debido a que este programa solo funciona con Java 7, se modifica de forma ingenua el *feature model*, simplemente descartando los *flags* que han sido introducidos en las nuevas versiones de la JVM. El objetivo de este ejercicio es probar la facilidad de modificar el trabajo realizado para compatibilizarlo con otras versiones de la máquina virtual.

El algoritmo genético es ejecutado un total de cinco veces por prueba de rendimiento. Cada optimización se realiza con un conjunto distinto de opciones: solo GC; C1 y C2; GC, C1 y C2; solo Runtime; y todos. En los casos de solo GC; C1 y C2; y solo Runtime, se utiliza una población de 32 individuos. En los restantes, el número de sujetos se incrementa a 60 por la mayor cantidad de parámetros. En todas las optimizaciones, el algoritmo genético se ejecuta por 10 generaciones. Al terminar una generación, se descarta la mitad de la población (seleccionada por un algoritmo de torneo) y se rellena con la cruza de los sobrevivientes. Además, entre los genes activos existe un 30 % de posibilidad de no expresarse, utilizando en cambio el valor por defecto que se encuentra preconfigurado en la JVM.

Durante la ejecución del algoritmo genético, cada individuo es evaluado dos veces, dígase en otras palabras, la prueba de rendimiento se ejecuta dos veces por cada configuración. El puntaje utilizado para ordenar los individuos es el promedio de los dos resultados. La razón de esto, es que el tiempo de ejecución de un programa posee cierta variabilidad que se debe al *scheduling* que realiza el sistema operativo. Efecto que se busca aminorar de esta forma.

Al final de las 10 generaciones, se considera al sujeto con el mejor desempeño como la configuración optimizada de la JVM. Para obtener una comparación rigurosa entre el resultado del algoritmo genético y la máquina virtual con los parámetros por defecto, se ejecutan ambos conjuntos de opciones un total de 30 veces tras cierto precalentamiento, tal como se recomienda en un estudio previo [30].

Si bien, el número de generaciones escogido es probablemente no el más adecuado para el problema, es difícil establecer la cantidad apropiada. Se usa como base optimizaciones tentativas sobre las pruebas de rendimiento DaCapo, para las cuales, este número es más que suficiente. Sin embargo y como se discute en el siguiente capítulo, las generaciones necesarias para el algoritmo dependen del programa que se esté ejecutando.

La máquina utilizada para realizar los experimentos posee la siguientes características: Hotspot de Java SE 15.0.2 en Windows 10 Professional (64 bits) con un procesador Ryzen

2600X 3.6 GHz (6 núcleos), 16 GB de RAM DDR4 a 3200 Mhz y una tarjeta gráfica AMD RX580 con 8 GB de VRAM.

Capítulo 5

Resultados Obtenidos

En este capítulo se encuentran los resultados obtenidos de realizar los experimentos descritos en la sección 4.4. Se describen los datos producidos por la optimización de DaCapo, realizando observaciones sobre los detalles interesantes. Luego, se hacen comparaciones con la optimización de la prueba de rendimiento de JavaFX. A continuación, se comentan los resultados de ejecutar SPECjvm2008. Finalmente, se analiza en conjunto toda la información presentada. Los archivos que contienen los resultados en bruto se encuentran en el repositorio del trabajo [31].

5.1. DaCapo

En la tabla 5.1 se puede ver como porcentajes, la reducción de los tiempos de ejecución de las pruebas de rendimiento de DaCapo. Entre las primeras observaciones que se pueden hacer, se encuentran el hecho de que hay valores negativos. Esto indica que en ciertos casos el desempeño del programa es peor con la configuración obtenida por el algoritmo genético, que con los parámetros por defecto. Esto ocurre usualmente con los *flags* de los compiladores en todas las pruebas, con la excepción de *fop* y *jython*.

Sobre el resto de grupos, la optimización de GC siempre arroja una mejora en el rendimiento. La configuración de las opciones Runtime, produce la peor pérdida de desempeño sobre *fop* y la segunda mejora más grande con *jython*. Solo en el caso de *lusearch*, la combinación de GC, C1 y C2 obtuvo un mejor rendimiento que optimizando exclusivamente GC. Finalmente, las configuraciones que contienen todos los *flags*, consiguen excelentes mejoras en *pmd*, *lusearch* y *jython*, siendo este último caso el incremento de desempeño más grande al reducir el tiempo de ejecución a la mitad.

Otra forma de ver los resultados es en el gráfico de la figura 5.1. Aquí se pueden observar más fácilmente otros patrones, como que los tres mejores resultados con todas las opciones se correlacionan con mejoras notables de Runtime. También es notorio que los resultados con solo GC y con GC, C1 y C2 suelen ser algo similares. Por último, se ve que las diversas pruebas son afectadas de forma distinta ante las cinco optimizaciones, con la excepción de *luindex* y *avrora* que de todas formas presentan cambios menores.

Tabla 5.1: Mejoras obtenidas al optimizar *benchmarks* de DaCapo.

Benchmark	GC	C1+C2	Runtime	GC+C1+C2	All
avrora	1,90 %	-1,59 %	-1,76 %	1,17 %	-0,80 %
fop	26,55 %	2,41 %	-10,76 %	20,67 %	-0,43 %
jython	3,56 %	1,31 %	40,98 %	1,07 %	52,47 %
luindex	3,24 %	-1,41 %	-4,10 %	2,78 %	-1,23 %
lusearch	7,23 %	-3,11 %	4,93 %	9,12 %	13,33 %
lusearch-fix	7,41 %	-5,31 %	2,04 %	4,42 %	1,31 %
pmd	5,72 %	-2,98 %	13,50 %	3,66 %	18,31 %
xalan	6,22 %	-3,06 %	-5,07 %	4,65 %	3,32 %

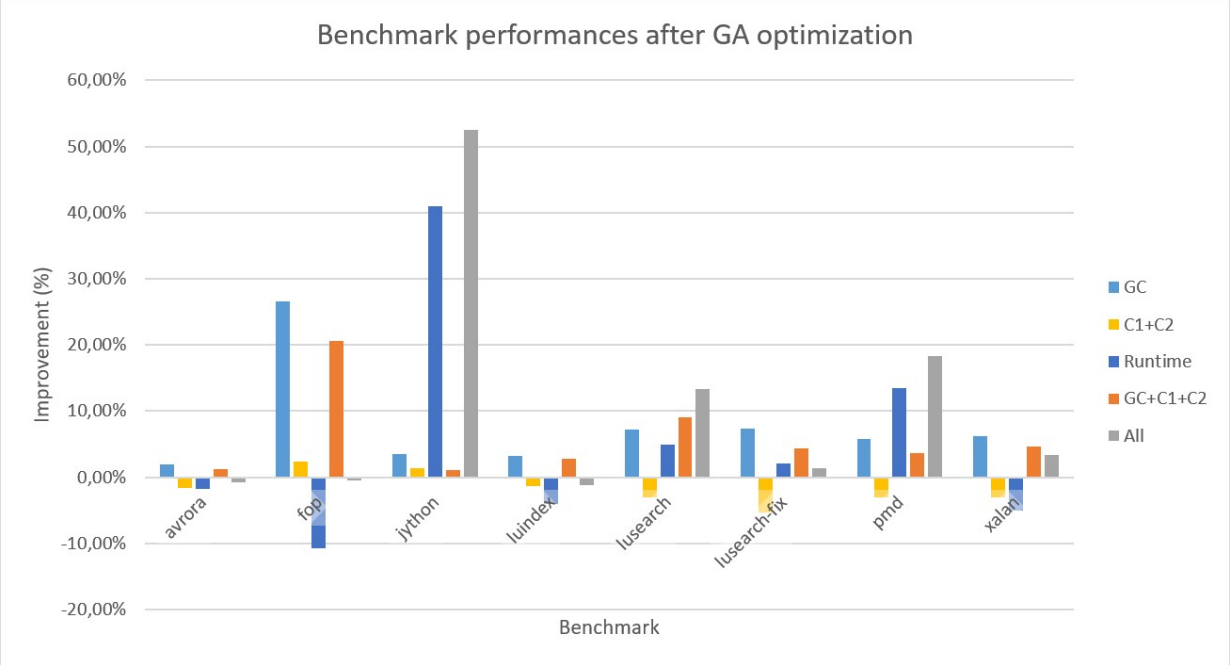

Figura 5.1: Gráfico de mejoras obtenidas al optimizar *benchmarks* de DaCapo.

Un detalle a considerar es, tal como se menciona en la sección 4.4, que los tiempos de ejecución de un programa no son valores exactos, sino que poseen cierta variación. Una duda obvia es por tanto, qué tan importantes son las mejoras o pérdidas de rendimiento. Para medir esto, se utiliza un índice llamado "d de Cohen".

Esta métrica estadística tiene como función indicar el tamaño del efecto, o en otras palabras, mide la magnitud del cambio entre dos muestras estadísticas. En específico, el d de Cohen mide el tamaño del efecto en relación a las desviaciones estándar de las muestras a comparar. El número entregado por el índice se puede discretizar en las siguientes categorías: muy pequeño (vs), pequeño (s), mediano (m), grande (l), muy grande (vl) y gigante (h) [32]. Al realizar este cálculo sobre los resultados obtenidos, se consigue la tabla 5.2.

Tabla 5.2: D de Cohen de los resultados de optimizar *benchmarks* de DaCapo.

Benchmark	**GC**	**C1+C2**	**Runtime**	**GC+C1+C2**	**All**
avrora	5,34 (h)	3,31 (h)	4,35 (h)	3,60 (h)	2,34 (h)
fop	6,75 (h)	0,61 (m)	3,36 (h)	5,86 (h)	0,14 (vs)
jython	5,15 (h)	1,60 (vl)	67,44 (h)	1,42 (vl)	93,20 (h)
luindex	2,01 (h)	0,78 (m)	2,66 (h)	1,75 (vl)	0,63 (m)
lusearch	2,32 (h)	0,90 (l)	1,62 (vl)	2,87 (h)	4,49 (h)
lusearch-fix	2,56 (h)	1,63 (vl)	0,55 (m)	1,46 (vl)	0,37 (s)
pmd	3,55 (h)	1,92 (vl)	9,79 (h)	2,73 (h)	9,98 (h)
xalan	1,19 (l)	0,52 (m)	1,03 (l)	0,97 (l)	0,68 (m)

En general, los resultados de GC y Runtime demuestran poseer los cambios de mayor relevancia, siendo seguidos de cerca por el grupo que combina GC, C1 y C2. Por otra parte, se ve que la optimización con todos los parámetros sobre *fop* y *lusearch-fix* tiene un tamaño de efecto pequeño, lo cual hace algo difícil justificar que las diferencias sean reales. Fuera de estos dos casos, los valores de los índices d de Cohen son lo suficientemente altos para concluir que existen mejoras o pérdidas de rendimiento.

También se observa que *avrora* presenta cambios consistentes pese a que no tengan gran magnitud porcentual. En contraste, los tamaños de efecto *lusearch-fix* son menores, indicando que las diferencias resultan menos relevantes en comparación, aun cuando son porcentualmente más grandes.

Otro detalle digno de mención, es que algunos *flags* aparecen recurrentemente en las configuraciones óptimas. En GC, suelen escogerse *ParallelGC* junto a *AdaptiveSizePolicy* y *ZGC*. Si bien, *AdaptiveSizePolicy* es una opción por defecto, el recolector de basura predeterminado es *G1GC*. En Runtime se encuentran dos opciones que llaman la atención. Primero, se encuentra la opción `-XX:AllocatePrefetchStyle=0`. Si este parámetro y los dependientes están mal configurados, pueden reducir severamente el desempeño del programa. Segundo, está `-XX:CompilationMode`, que se posee los valores *high-only-quick-internal* o *default*. La categoría C2 también posee parámetros que se repiten, pero debido a que esta entrega resultados mayormente negativo, no hay gran mérito en mencionarlos.

Algo más que se puede mencionar de DaCapo, es que el desempeño del mejor individuo casi no mejora a medida que avanzan las generaciones. En la figura 5.2 se muestra el cambio más significativo que se observa entre la primera generación y la última. No es muy difícil notar que la selección y reproducción de individuos afecta muy poco el tiempo de ejecución del mejor individuo. En este caso, la optimización se podría haber detenido en la generación 5 y la configuración obtenida sería tan buena como la conseguida en la generación 10. Como se ha dicho, en el resto de optimizaciones la mejora intergeneracional es aún menor. En algunos casos, el algoritmo genético se comporta como la selección de una configuración en la población inicial.

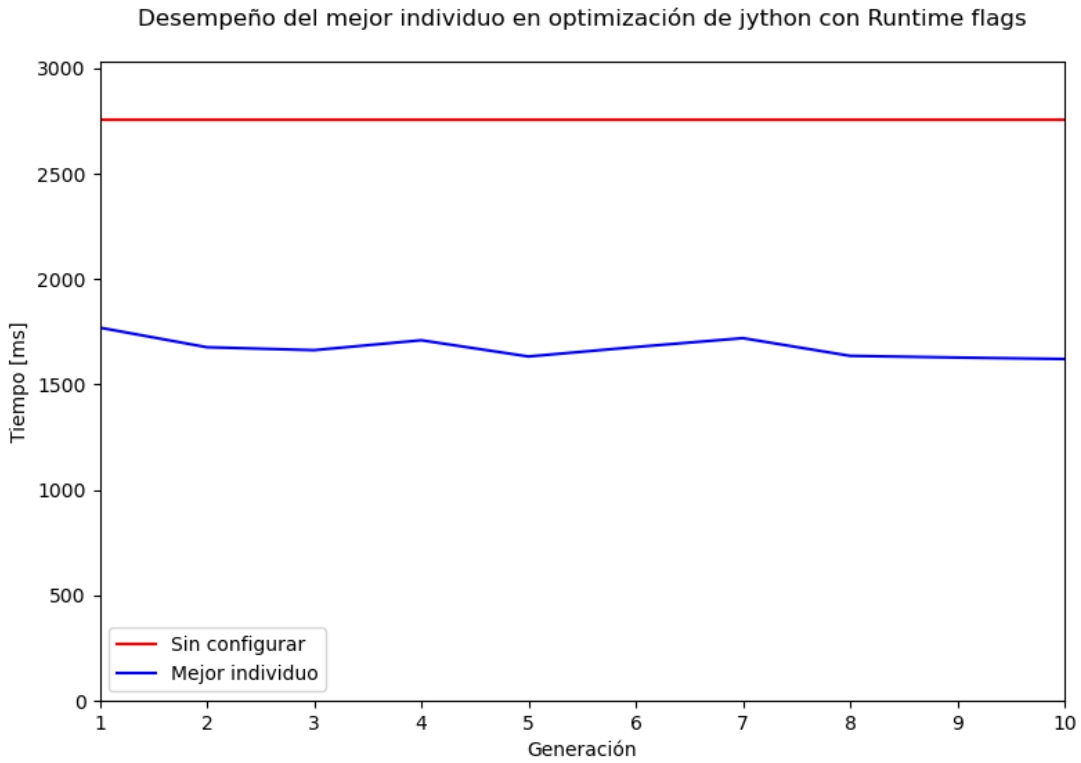

Figura 5.2: Gráfico de la mejora entre generaciones más notoria entre las optimizaciones de DaCapo.

5.2. Prueba de rendimiento sobre JavaFX

En esta prueba, lo que se optimiza no es el tiempo de ejecución, sino la cantidad de cuadros por segundo. Los resultados se encuentran en la tabla 5.3. Ya a simple vista, es posible darse cuenta que los resultados no son muy distintos a los encontrados con DaCapo, siendo la excepción notable la optimización con todas las opciones. No obstante, si se presenta un aspecto muy distinto a lo visto anteriormente. En la figura 5.3, se observa que el algoritmo genético se comporta como es esperado, algo contrario a lo observado con DaCapo.

Tabla 5.3: Mejoras obtenidas al optimizar el *benchmark* de JavaFX por Nikita Prokopov.

GC	**C1+C2**	**Runtime**	**GC+C1+C2**	**All**
2,73 %	0,13 %	-2,59 %	2,14 %	-18,33 %

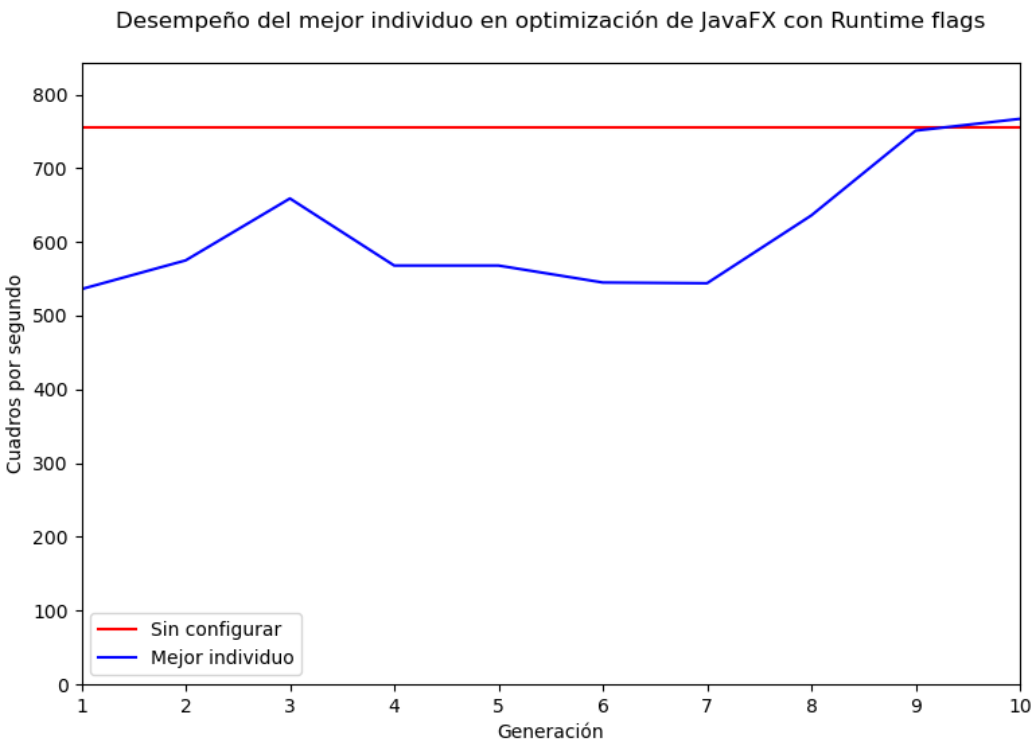

Figura 5.3: Gráfico de la mejora entre generaciones de una optimización del *benchmark* de JavaFX.

5.3. SPECjvm2008

En la tabla 5.4, se muestran los resultados de ejecutar la prueba de rendimiento *compress* de SPECjvm2008, la cual, comprime datos usando una modificación del método Lempel-Ziv (comúnmente abreviado como LZW). Es notable que se haya logrado incrementos tan significativos en el desempeño, pues como este programa solo está disponible para Java 7, la optimización se realizó con la intersección de parámetros disponibles en esta JVM y los modelados. Tampoco se tomó en cuenta posibles cambios en los valores por defecto, que podrían alterar los rangos utilizados para crear los individuos. Cabe destacar que todos los cambios necesarios fueron hechos en el archivo JSON que contiene los genes.

Tabla 5.4: Mejoras obtenidas al optimizar el *benchmark compress* de SPECjvm2008.

Benchmark	**GC**	**C1+C2**	**Runtime**	**GC+C1+C2**	**All**
compress	0,44 %	3,75 %	8,15 %	3,93 %	5,40 %

Algo esperable en estos resultados es que GC sea el menos optimizado, pues entre las versiones 7 y 15 se han removido algunos recolectores de basura e introducido otros. De cualquier modo, no se puede considerar una optimización de SPEC como representativa del trabajo efectuado, porque depende de una versión distinta de la máquina virtual a la

considerada en la modelación de *flags*.

5.4. Análisis

De los resultados obtenidos se pueden hacer algunas observaciones respecto al trabajo realizado. En primer lugar, se nota que el espacio de búsqueda es extremadamente dependiente de la aplicación a ejecutar. Esto queda demostrado al ver la diferencia de comportamiento del algoritmo genético entre DaCapo (figura 5.2) y JavaFX (figura 5.3). Es posible incluso, que los parámetros del algoritmo genético, dígase generaciones y población, deban ajustarse al programa a optimizar. Esto último requeriría un estudio enfocado a responder esta incógnita.

Aun así, es posible especular que un algoritmo genético no es el método más apropiado para realizar optimizaciones sobre la JVM. Entre los argumentos a favor de esta postura, se encuentra la dificultad en algunos casos que presenta el algoritmo para encontrar un óptimo mejor que los parámetros por defecto. Esto tampoco es algo que se pueda solucionar simplemente aumentando el número de generaciones, pues cuando se obtiene resultados negativos, la función objetivo ya se ha estabilizado. También se ha de mencionar los largos tiempos requeridos para hacer la optimización, pues para reunir los resultados de la tabla 5.1 fue necesaria alrededor de una semana de ejecución de *benchmarks* día y noche.

Es por esto que llama la atención lo buenos que son los resultados de la prueba de rendimiento *compress* de SPECjvm2008. Es posible que la razón se deba a que se utiliza Java 7 y que este posea valores por defecto que den amplio espacio para optimizar. Otra causa, podría ser que el *benchmark* mismo favorezca un configuración un específico. Lamentablemente, como no es posible ejecutar SPEC en versiones más recientes de la JVM, no se puede determinar la veracidad de estas teorías. En contraste, si se puede determinar que el *feature model* creado resulta útil, pues con cambios menores se pudo aplicar a otra JVM con buenos resultados.

En términos generales, se puede concluir que en muchos de los casos presentados se logra mejorar el desempeño de la JVM. Entre los distintos grupos de opciones utilizadas para la optimización, GC resulta ser una apuesta segura, produciendo mejoras al rendimiento de forma consistente. Por otro lado, no es muy llamativo incluir las opciones de compilación, ya que solo ocasionalmente logran efectos positivos. Esto tiene sentido cuando uno sabe que la mayor parte de *flags* en estas categorías desactivan operaciones de optimización de la compilación JIT.

Finalmente, para algunos programas los parámetros Runtime pueden incrementar considerablemente el rendimiento. Sin embargo, otras aplicaciones se ven perjudicadas cuando se intenta optimizar estas opciones. Esto destaca la importancia de entender y modelar correctamente esta categoría.

Capítulo 6

Conclusiones

La solución propuesta es exitosamente aplicada a la JVM para optimizar diversas pruebas de rendimiento, logrando en el mejor de los casos un incremento del 52 % del rendimiento con respecto a una ejecución normal. En consecuencia, se obtiene una herramienta que se puede ocupar para conseguir mejor desempeño de una aplicación Java. En otras palabras, se logra recrear el trabajo realizado con Hotspot Auto-tuner [2], aunque usando métodos de optimización distintos.

Es una pena, pero no se puede hacer una comparación directa con este estudio o con el de Auto-tuning de parámetros de GC [4], debido a que esto requeriría estudiar el funcionamiento de las herramientas correspondientes y modificarlas para que sean compatibles con Java 15. Tampoco se puede evaluar el desempeño de la solución sobre SPECjvm2008, porque estas pruebas de rendimiento no están disponibles para versiones recientes de Java. De no encontrarse estas dificultades, se podrían presentar resultados mucho más robustos. De todas maneras, el hecho de haber publicado un paper con resultados preliminares de lo conseguido [9] muestra el potencial del acercamiento utilizado.

Otro aspecto positivo logrado, es demostrar la utilidad de un *feature model* para representar las interrelaciones entre parámetros. Con esta herramienta se es capaz de eliminar configuraciones inválidas, algo con lo que estudios anteriores [2, 4], presentan complicaciones. También se ha de mencionar, que el modelo diseñado es una buena base para replicar este trabajo en otras versiones de la JVM. Esto queda evidenciado al aplicar el modelo sobre Java 7, realizando cambios menores subóptimos.

Un detalle que se debe mencionar es que, debido a la omisión de las opciones deprecadas, el modelo presentado es completamente compatible con Java 16. Si bien, no se encuentran las nuevas opciones introducidas en esta versión, añadirlas constituye una tarea considerablemente menor a la descrita en este informe. Se hubiera querido esta versión de la JVM para realizar optimizaciones, sin embargo y tristemente, su lanzamiento ocurre a mediados del desarrollo de este trabajo [33].

En lo que respecta a los objetivos planteados, estos generalmente se alcanzan de forma satisfactoria. La herramienta creada no requiere de conocimientos profundos de la JVM para obtener una optimización. Además, los capítulos 3 y 4 son muestra que se ha recopilado y modelado información suficiente para realizar la optimización. La validación de la solución se

logra parcialmente, pues si bien, se logra en algunos casos un mejor rendimiento que el por defecto, en otros casos el desempeño es peor.

Pese a todo lo mencionado, existe espacio para extender y mejorar lo descrito en este informe. En primer lugar, es posible que el *feature model* creado no sea completamente certero, pasando por alto algunas dependencias e incompatibilidades. Es necesario investigar más profundamente la JVM o recurrir a personas que posean conocimientos sobre esta para asegurarse de la correctitud del modelo. Adicionalmente, se puede efectuar la actualización de la herramienta a la última versión de Java o incluir más *benchmarks* a evaluar.

Bibliografía

[1] "Top programming languages of 2021." https://www.codingdojo.com/blog/top-7-programming-languages. Visitada: 2021-07-19.

[2] S. Jayasena, M. Fernando, T. Rusira, C. Perera, and C. Philips, "Auto-tuning the java virtual machine," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pp. 1261–1270, 2015.

[3] "Tuning the jvm – g1gc garbage collector flags for minecraft." https://aikar.co/2018/07/02/tuning-the-jvm-g1gc-garbage-collector-flags-for-minecraft/. Visitada: 2021-07-18.

[4] P. Lengauer and H. Mössenböck, "The taming of the shrew: Increasing performance by automatic parameter tuning for java garbage collectors," in *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, (New York, NY, USA), p. 111–122, Association for Computing Machinery, 2014.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, (New York, NY, USA), pp. 169–190, ACM Press, Oct. 2006.

[6] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (foda) feasibility study," 01 1990.

[7] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *International Conference on Parallel Architectures and Compilation Techniques*, (Edmonton, Canada), Aug 2014.

[8] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "Paramils: An automatic algorithm configuration framework," *J. Artif. Int. Res.*, vol. 36, p. 267–306, Sept. 2009.

[9] F. Canales, G. Hecht, and A. Bergel, "Optimization of java virtual machine flags using feature model and genetic algorithm," in *Companion of the ACM/SPEC International Conference on Performance Engineering*, ICPE '21, (New York, NY, USA), p. 183–186, Association for Computing Machinery, 2021.

[10] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch, "A multi-objective auto-tuning framework for parallel codes," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2012.

[11] C. Tapus, I.-H. Chung, and J. Hollingsworth, "Active harmony: Towards automated performance tuning," in *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pp. 44–44, 2002.

[12] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: end-to-end optimization stack for deep learning," *CoRR*, vol. abs/1802.04799, 2018.

[13] T. Brecht, E. Arjomandi, C. Li, and H. Pham, "Controlling garbage collection and heap growth to reduce the execution time of java applications," *SIGPLAN Not.*, vol. 36, p. 353–366, Oct. 2001.

[14] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. Irwin, and M. Wolczko, "Tuning garbage collection in an embedded java environment," in *Proceedings Eighth International Symposium on High Performance Computer Architecture*, pp. 92–103, 2002.

[15] I. H. Kazi, H. H. Chen, B. Stanley, and D. J. Lilja, "Techniques for obtaining high performance in java programs," *ACM Comput. Surv.*, vol. 32, p. 213–240, Sept. 2000.

[16] J. Singer, G. Brown, I. Watson, and J. Cavazos, "Intelligent selection of application-specific garbage collectors," in *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, (New York, NY, USA), p. 91–102, Association for Computing Machinery, 2007.

[17] J. Singer, G. Kovoor, G. Brown, and M. Luján, "Garbage collection auto-tuning for java mapreduce on multi-cores," in *Proceedings of the International Symposium on Memory Management*, ISMM '11, (New York, NY, USA), p. 109–118, Association for Computing Machinery, 2011.

[18] K. Taht, I. Mitić, A. Barth, E. D. Vecchio, S. Agarwal, R. Balasubramonian, and R. Stutsman, "Dynajet: Dynamic java efficiency tuning," 2020.

[19] R. Storn and K. Price, "Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces," *Journal of Global Optimization*, vol. 11, pp. 341–359, 01 1997.

[20] "Java hotspot vm options." www.oracle.com/java/technologies/javase/vmoptions-jsp.html. Visitada: 2021-07-19.

[21] "The java command." https://docs.oracle.com/en/java/javase/15/docs/specs/man/java.html. Visitada: 2021-07-19.

[22] "Vm options explorer." https://chriswhocodes.com/hotspot_options_openjdk15.html. Visitada: 2021-07-19.

[23] "Java garbage collection basics." https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html. Visitada: 2021-07-19.

[24] S. K. Per Liden, "Jep 333: Zgc: A scalable low-latency garbage collector (experimental)," February 2018.

[25] "Escape analysis in java." https://www.beyondjava.net/escape-analysis-java. Visitada: 2021-07-19.

[26] "What is a genetic algorithm?." https://pastmike.com/what-is-a-genetic-algorithm/. Visitada: 2021-07-19.

[27] "genetic algorithms in php code example of (evolutionary programming)." http://www.abrandao.com/2015/01/simple-php-genetic-algorithm/. Visitada: 2021-07-20.

[28] "Understanding complexity." https://evolution.berkeley.edu/evolibrary/article/0_0_0/evodevo_05. Visitada: 2021-07-20.

[29] "Java graphics benchmark." https://github.com/tonsky/java-graphics-benchmark. Visitada: 2021-07-20.

[30] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," *ACM SIGPLAN NOTICES*, vol. 42, no. 10, pp. 57–76, 2007.

[31] "optimizador-jvm-algoritmo-genetico." https://github.com/felipe-canales/optimizador-jvm-algoritmo-genetico. Visitada: 2021-07-19.

[32] S. Sawilowsky, "New effect size rules of thumb," *Journal of Modern Applied Statistical Methods*, vol. 8, pp. 597–599, 11 2009.

[33] "Oracle announces java 16." https://www.oracle.com/news/announcement/oracle-announces-java-16-031621.html. Visitada: 2021-07-21.