Implementación de un modelo de memoria basado en regiones en una máquina virtual a gran escala.

Taboada, Alejandro Darío

Directores:
Garbervetsky, Diego
Yovine, Sergio

5 de octubre de 2009

Resumen

La mayoría de los lenguajes de programación modernos, como Java, suelen liberar al programador de las preocupaciones relacionadas con la administración de memoria, siendo esta tarea derivada en un agente que se ocupa de la recolección de los objetos que ya no son utilizados (*Garbage Collector -GC-*).

En los últimos tiempos la utilización de estos lenguajes ha crecido y se ha propagado a diversas áreas, por ejemplo a dispositivos celulares, PDAs, y sistemas embebidos. Todos ellos tienen la particularidad de que disponen de una memoria y una capacidad de procesamiento limitada, por lo cual es importante tener especial cuidado en el uso tanto del procesador como de la memoria. En estos ambientes, la utilización de los *GC* tradicionales comprometen las garantías de predictibilidad espacial y temporal requeridas, ya que no es posible conocer en qué momento del programa se llevará a cabo el proceso de recolección, ni tampoco estimar el consumo de memoria en algún momento de la ejecución.

Para mitigar estos problemas surgen diversas formas alternativas de administrar la memoria. En este trabajo se tomará en particular el modelo de administración de memoria basado en regiones asociadas a métodos, que consiste en agrupar conjuntos de objetos de acuerdo a su ciclo de vida. Todos los objetos asignados a una región serán eliminados en conjunto cuando el método, correspondiente a dicha región, finalice su ejecución. Teniendo control sobre el ciclo de vida de las regiones, se puede evitar el uso del GC.

Una de las dificultades inherentes a este enfoque es determinar el modo en que las regiones son constituídas (a qué métodos son asociadas, qué objetos contendrá cada una de ellas, y en algunos casos obtener una estimación del tamaño que ocupará en memoria). Existen diversos trabajos en el área que proveen diferentes soluciones a esta problemática, disponibilizando herramientas para la generación de esta información. De esta manera, dado un programa, es posible obtener la información de regiones asociada, que podría ser utilizada al momento de su ejecución por un administrador de regiones.

El propósito de este trabajo es implementar los mecanismos para soportar regiones de memoria asociadas a métodos sobre la máquina virtual Java Jikes RVM, brindando una interfaz de alto nivel mediante el uso de anotaciones Java, para poder incorporar la información generada por estos procesos.

El poseer un entorno de prueba real tendrá como beneficio la posibilidad de poder evaluar el modelo de memoria contrastando los resultados con las estimaciones de modelos teóricos, o simplemente con otros modelos de administración de memoria. Además, al realizar evaluaciones empíricas se podrá determinar sobre qué programas o patrones algorítmicos la utilización de esta técnica és o no, una buena opción.

Índice general

Agradecimientos

A mis padres Adriana y Miguel por brindarme toda la educación que recibí y alentarme a obtener este título. A Connie por ser mi compañera de vida y aguantarme todo este tiempo quemándole la cabeza. A mi amigo y compañero de discusión Martín, porque ha sido una gran ayuda para poder avanzar en la carrera, y también por haber brindado sus aportes a lo largo de este trabajo. A mis directores de tesis Diego y Sergio, porque hicieron posible esta tesis brindando toda la ayuda y predisposición necesaria. A Osmar Pan porque siguiendo sus consejos finalmente aprobé todas las materias "logarritmos" I II y III. A mis amigos profanos Gero, José y Juan por postergar infinitos asados. Y finalmente a todos los que vienen escuchando la frase "cuando termine la tesis".

Capítulo 1

Introducción

La administración de memoria siempre ha jugado un rol fundamental en los lenguajes de programación orientados a objetos como Java, Smalltalk o .Net. Para facilitar el desarrollo, la administración de la memoria se lleva a cabo en forma automática, desvinculando al programador de la necesidad de liberar manualmente los recursos que ya no son utilizados. Es por ello que estos lenguajes carecen de instrucciones para el manejo explícito de la memoria, delegando dicha responsabilidad a procesos automáticos que deberán garantizar que el programa se ejecute correctamente, no agote los recursos del sistema, y no genere pérdida de memoria.

Una de las políticas más utilizadas es la de recolección de objetos en desuso (*Garbage Collection -GC-*)[16], que se basa en la ejecución de un proceso que libera la memoria tomada que ya no es utilizada, permitiendo su reutilización. Por ejemplo si tenemos un lenguaje orientado a objetos, el recolector liberará la memoria ocupada por aquellos objetos que no están siendo utilizados, o mejor dicho, los que no son alcanzados directa o indirectamente desde algún objeto activo del programa.[1]

Debido al gran crecimiento de las comunidades que utilizan este tipo de lenguajes y su consecuente evolución, muchos ámbitos han volcado su interés hacia ellos, como es el caso de los sistemas embebidos y de tiempo real. No obstante, se presentan los siguientes problemas para sistemas basados en estas características: uno de ellos es el tiempo de ejecución no determinado de estas rutinas de administración de memoria, y la irrupción de las mismas en cualquier punto del programa. Otro, es el uso de un mayor espacio de memoria. Dado que una rutina de recolección es costosa en tiempo, no es conveniente aplicarla regularmente, por lo que es necesario disponer de más recursos de memoria para albergar todos los objetos que se vayan creando hasta su futura liberación.

En sistemas de tiempo real, donde el tiempo de ejecución de las rutinas es crucial, no se podría utilizar esta metodología. Al menos se necesitaría asegurar un tiempo determinado para los procesos de recolección, así como saber el momento en que se ejecutarán. Es por eso que en este tipo de ambientes está muy controlado o directamente prohibido el uso de los recolectores.

Por otro lado, en sistemas con escasa memoria o limitada, como dispositivos celu-

[1] Los objetos activos del programa son aquellos que pueden ser alcanzados reflexiva o transitivamente desde los objetos raíces (estáticos o que viven en el pila de llamadas a métodos -*stack frame*-).

lares o *PDAs*, estas técnicas tampoco son una buena opción, ya que no proporcionan una forma de cuantificar el consumo de memoria, y en consecuencia, no es posible definir una cota de consumo para una aplicación dada.

Existen alternativas que combinan los beneficios del *GC*, y proveen soluciones para la predictibilidad espacial y temporal. Una de ellas es la administración de memoria basada en regiones, que consiste en agrupar a los objetos con similar ciclo de vida en el mismo espacio o región de memoria. Aquí los objetos pertenecientes a una región son creados de forma independiente consecuentemente con el flujo normal del programa, pero son eliminados al mismo tiempo, cuando se elimina la región que los contiene. Si se tiene el conocimiento del tamaño de una región, y en qué momento se crea y se elimina, es posible tener un control sobre la administración de la memoria evitando el uso del *GC*.

Existen muchas variantes para este tipo de manejo de memoria, algunas de ellas no permiten dependencias entre regiones, o sea que los objetos albergados en una, no pueden mantener referencias hacia otros objetos de regiones diferentes. En otros casos, sí se pueden dar este tipo de relaciones, siempre teniendo en cuenta el ciclo de vida de los objetos albergados en ellas. Por ejemplo no podría pasar que un objeto siga siendo referenciado tras haberse eliminado la región que lo contenía.

Otra característica a tener en cuenta en un modelo de regiones es la posibilidad de permitir que éstas sean expansibles, o que mantengan un tamaño fijo. Las primeras tienen la ventaja que no hace falta conocer sus tamaños con anterioridad, mientras que utilizando regiones de tamaño fijo, se puede definir una administración de memoria más eficiente debido a la no fragmentación del espacio requerido para cada una de ellas.

Algunas de las dificultades de utilizar una administración de memoria basada en regiones son:

- **Determinar cómo construirlas y definir las políticas de recolección de manera que el programa se ejecute correctamente: la configuración de una región debe ser tal que sólo podrá ser eliminada cuando ninguno de los objetos contenidos esté en uso. Para ello, se debe tener en cuenta cuáles objetos albergará cada una de ellas, y en qué momento podrán ser eliminadas.**

- **En el caso de utilizar regiones con tamaños predefinidos, determinar una cota del consumo de memoria para cada región (según la cantidad/tipo de objetos que albergue), previamente a su creación.**

Otorgar al programador la posibilidad de administrar las regiones de forma directa es una solución propensa a errores. Algunas soluciones como [6], se basan en asignar como "dueño" de la región a alguna unidad de cómputo del programa, como podría ser un método, un *thread*, o una instancia de objeto. De esta forma el ciclo de vida de la región es determinado por el de su dueño, proveyendo más seguridad durante la ejecución y agilizando el modo de programar. Por ejemplo, si tomamos un caso particular en donde la unidad de cómputo es el "método", se podrían definir regiones asociadas a éstos, de modo enque al iniciarse su ejecución, se cree una región. Ésta contendrá a todos los

objetos que viven a lo sumo durante el tiempo de vida del método, o sea los que no son retornados ni están vinculados por referencias desde otras unidades de cómputo cuyo ciclo de vida sea mayor. Y al finalizar la ejecución del método, se procede a la liberación de la región con todos los objetos albergados en ella.

En este trabajo en particular se implementó el modelo de administración de memoria basada en regiones que se propone en [24], que básicamente trata de asociar el ciclo de vida de una región a un método, y asigna los objetos en la región que corresponda según su ciclo de vida (*scope*).

En este modelo, es posible que no todos los métodos del programa tengan asociada una región. Éstas son determinadas dependiendo de cómo se agrupen los objetos (según su *scope*). Así, un método en el que todos los objetos creados a partir de su ejecución, lo sobreviven (escapan), no tendrá una región asociada.

Existen diversas investigaciones como [7, 21, 9] que desarrollaron herramientas sobre este modelo de memoria, las cuales permiten detectar qué métodos serán los que tendrán regiones asociadas, y discriminar qué objetos vivirán en ellas. Estas herramientas utilizan un proceso de análisis de "escape", que consiste en detectar estáticamente sobre el código fuente el alcance de los objetos respecto a los métodos en donde son utilizados, pudiendo luego agruparlos y confeccionar las regiones. En el caso de [7], donde las regiones utilizadas deben tener un tamaño predefinido al momento de su creación, adicionalmente, se presenta una técnica (todavía no del todo automatizada) para poder acotar el tamaño que consumirán, que está determinado por una expresión en función de los parámetros del método asociado.

1.1. Motivación

Si bien estos modelos nos proveen una forma de obtener información sobre las regiones a crear, y en algunos casos las expresiones de consumo asociadas a las mismas, el mecanismo para calcularlas es costoso; los procesos involucrados en el análisis provienen de herramientas heterogéneas y constan de múltiples pasos.

El no poseer un entorno de prueba real hace que sea muy difícil poder evaluar este modelo teórico de memoria y contrastar el comportamiento con otras implementaciones tanto en consumo de memoria como en tiempos. Estos son algunos de los beneficios que se obtendrían al poseer una implementación real:

- Verificar los resultados reales con los obtenidos en las herramientas manuales.
- Obtener información de consumo real. En el modelo teórico se estima una cota de consumo para las regiones, pero no se conoce el consumo real. Tener datos reales servirá para poder saber qué tan preciso es el método de medición.
- En el modelo teórico muchas veces no es posible obtener información de consumo para todo tipo de algoritmo. Por ejemplo en casos de ciclos donde no es posible inferir la cantidad de iteraciones, o en recursiones en las cuales no se puede predecir el nivel de anidación. Realizar

pruebas empíricas podría ser de gran ayuda para poder manejar estos casos. Las mediciones y estadísticas pueden proporcionar información sobre el tamaño consumido, o la cantidad de objetos controlados por región.

- Obtener estadísticas sobre tiempos. Para determinar la conveniencia de utilizar o no regiones, además del consumo de memoria, es importante que los procesos de administración de memoria no degraden la performance de la aplicación.
- Contrastar con otras VMs. Comparar mediciones contra otras máquinas virtuales que manejen administraciones de memoria diferentes.
- Determinar empíricamente patrones algorítmicos para los cuales la administración basada en regiones sea o no, una buena opción (menor consumo de memoria y tiempos de ejecución razonables).
- Y finalmente poseer un entorno de prueba sobre un lenguaje de programación moderno, lo que permite analizar cualquier biblioteca actual disponible.

1.2. Implementación

Los puntos más importantes a destacar para tener en cuenta en la implementación de este modelo son:

- Tener el conocimiento previo a la ejecución de qué métodos van a ser los que generen regiones ante su invocación (métodos dueños de regiones).
- Si es posible conocer cuál es la expresión de tamaño máximo para cada una de ellas.
- Saber cuáles son las sentencias '*new*' que se deben asignar a cada región (*creation sites*).

Se analizaron varias máquinas virtuales Java *Open Source*, eligiendo a Jikes RVM (*Research Virtual Machine*)[15] como base para el desarrollo de esta tesis.

La idea fue introducir el nuevo plan de memoria de la manera más transparente posible al programador, de manera que el programa sufra la menor cantidad de modificaciones. Para ello se optó por el uso de anotaciones, que si bien es una modificación en el lenguaje Java para incorporar la nueva funcionalidad, se provee compatibilidad hacia atrás (se pueden ejecutar programas Java convencionales), el código dentro de cada método se mantiene intacto, y no se requiere modificar el mecanismo de compilación a código bytecode.

1.2.1. Compilación y ejecución de un programa Java

En la siguiente figura se puede visualizar el proceso de compilación/ejecución, desde el desarrollo del programa Java hasta su ejecución en la máquina virtual Jikes RVM.

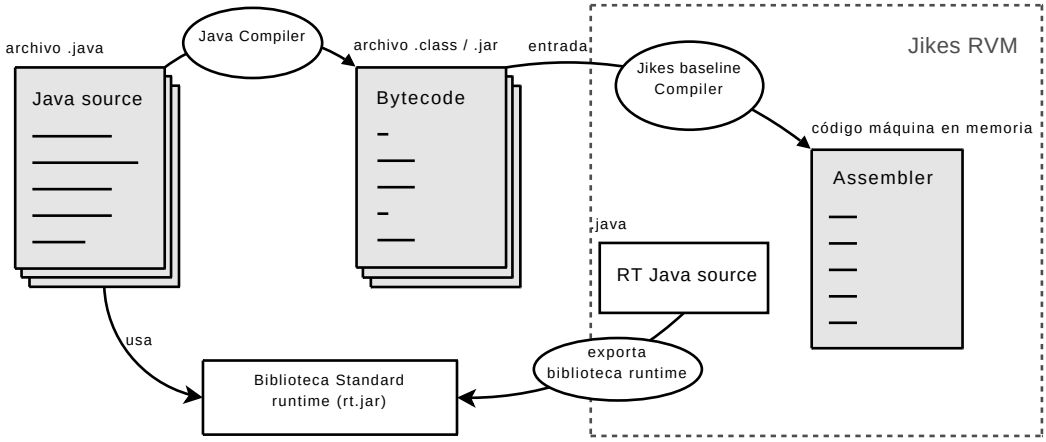

Figura 1.1: Esquema de desarrollo, compilación y ejecución de un programa Java en Jikes RVM.

En el proceso de desarrollo de una aplicación, el programador escribe el código fuente (Java) en archivos cuya extensión es *.java*. En este paso el programador puede utilizar bibliotecas y en particular, las básicas que responden a la implementación de la especificación del lenguaje (*runtime* environment).

En el proceso de compilación Java se genera el archivo *.class*, que es la representación en sintaxis *bytecode* del código fuente. Este paso es necesario debido a que Java fue pensado para ser un lenguaje interpretado, y el código *bytecode* unifica el fuente Java a un conjunto de instrucciones bien conocidas que cualquier intérprete del lenguaje debe saber ejecutar. Los archivos *.class* pueden ser comprimidos y agrupados en bibliotecas llamadas *.jar*.

Hasta este punto la máquina virtual Jikes RVM no interviene en el proceso. Sólo recién al ejecutar el programa (conjunto de .class y jars), ésta deberá compilar el *bytecode* a código máquina y proceder consecuentemente a su ejecución siguiendo el flujo del programa. Este proceso, a diferencia de los anteriores (que generan archivos), se realiza en memoria.

El trabajo realizado en esta tesis está enfocado principalmente a los procesos que intervienen dentro de Jikes RVM:

- **Compilación del bytecode a código máquina.**
- **Definición de la estructura de memoria de la *VM* y su administración durante la ejecución para soportar el uso de regiones.**
- **Incorporación de las clases necesarias en la biblioteca *Java Runtime Environment* para brindar el soporte necesario para la utilización de regiones en el código fuente.**

1.3. Contribuciones

Se desarrollaron tres máquinas virtuales con políticas de regiones diferentes:

- La primera combina un espacio de memoria para regiones y un espacio inmortal (sin GC) que sirve para albergar a todos los objetos que no se alojan en ninguna región.
- La segunda implementación es similar a la anterior con el agregado de un GC sobre el *heap*. El *heap* será el área de memoria que no utiliza regiones y tendrá una administración de memoria Java convencional (con GC).
- Por último se realizó un modelo puro de regiones con un solo espacio. La memoria inmortal es implementada mediante una región más que existe durante toda la ejecución. En este caso se utilizó el modelo de regiones con tamaño variable.

Al tener que implementar diferentes modelos de memoria que deben compartir la mayor parte de sus componentes, se trató de realizar un diseño modular de modo que sea posible agregar o combinarlos así pudiendo reutilizar la mayor parte de ellos. De esta manera el modelo de regiones asociadas a métodos sería sólo una posible implementación sobre la plataforma de desarrollo, permitiendo en un futuro reaprovechar los módulos disponibles.

Una vez realizadas las tres implementaciones se escogió un conjunto de pruebas para realizar *benchmarks* y obtener estadísticas sobre el consumo de memoria en cada una de las VM implementadas. En los resultados se pueden ver casos para los que las regiones son (o no) una buena opción, y otros casos en donde sólo se aplican a una parte del programa.

Se espera que esta implementación se utilice como herramienta para la corroboración de resultados teóricos en el modelo descripto y sea la base para futuros trabajos sobre modelos de memoria basados en regiones.

Capítulo 2

Administración de memoria basada en regiones

2.1. Estado del arte

Diversos trabajos apuntan a resolver la problemática de la predictibilidad espacial y temporal mencionada anteriormente. Básicamente, se apunta a reemplazar el uso de recolectores de basura tradicionales por procesos de recolección más controlados, que provean un mejor aprovechamiento de la memoria. Algunas soluciones están basadas en la utilización de recolectores de tiempo real, como en los proyectos Metronome [1] y JamaicaVM [22]. Los GC de tiempo real utilizan un modelo estadístico basado en parámetros dependientes del programa, como la tasa de creación de objetos (*allocation rate*) y la tasa de generación de basura (*garbage generation rate*), que sirven como información para segregar la memoria en forma generacional[1], agilizando así los procesos de recolección que resultan reducidos en tiempo y son ejecutados intermitentemente para evitar el agotamiento de la memoria. Esta metodología propone una solución interesante, pero la tasa de asociación y de generación de residuos generada durante la ejecución del programa es difícil de predecir.

Otra alternativa consiste en utilizar regiones de memoria [24]. La técnica consiste en cambiar el modelo organizacional de los objetos en memoria agrupándolos según su ciclo de vida. De esta manera los objetos que posean similar ciclo de vida pertenecerán a la misma región. En este modelo los objetos se crean individualmente, como sucede durante la ejecución de un programa convencional, pero serán removidos en conjunto tras eliminar la región que los contiene. En un esquema de este tipo es posible mejorar ampliamente la predictibilidad temporal al eliminar los objetos, ya que el borrado de una región no requiere realizar ningún trazado sobre el heap. Dado que la utilización de áreas de memoria es un concepto general, hay muchas maneras de definirlas con sus pros y contras. Por ejemplo la utilización de regiones de tamaño fijo permite definir una

[1] La segregación generacional consiste en detectar y agrupar los objetos con similar tiempo de vida (pueden tener ciclos de vida diferentes) en áreas de memoria con políticas de recolección independientes. Generalmente se provee un sistema adaptativo basado en estadísticas donde los objetos son promovidos entre las diferentes áreas generacionales.

administración de memoria simple y eficaz, pero trae mucha complejidad al tener que calcular el tamaño de las regiones previamente a la ejecución. En cambio un modelo de regiones cuyos tamaños pueden variar no requiere correr ningún proceso previo para el cálculo del consumo de memoria, pero la administración resulta compleja y menos eficiente.

En [13] y [23] se propone agregar a un lenguaje existente soporte para la utilización de regiones en forma manual, de esta manera el programador puede crear, asignar objetos en las regiones existentes y determinar cuándo eliminarlas. Estos modelos permiten flexibilidad, pero hacen más compleja la programación, tornando explícito el manejo de memoria.

Una de las primeras propuestas en optar la utilización de regiones es la RTSJ (Real Time Specification for Java)[5], que propone la creación de *threads* especializados para ejecuciones en tiempo real, y permite crear espacios de memoria en donde no interfiere el recolector de basura. Una de las mayores contras de RTSJ es el agregado de mucha complejidad al programar, haciendo visible un modelo de memoria difícil de administrar que omite los beneficios originales del lenguaje. Otra de las contras, es la incapacidad de poder especificar el comportamiento de las regiones de memoria para las bibliotecas incluidas en el lenguaje (por ejemplo la *Standard Library*); de esta manera se hace imposible aprovecharlas, teniendo que reescribir gran parte de ellas.

Otros trabajos como [9, 21, 12] han apuntado a automatizar el desarrollo implementando herramientas para la inferencia de las regiones a partir del código fuente mediante el uso de analizadores estáticos, y finalmente poder incorporarlas instrumentando el código o directamente aplicarlas durante la ejecución del programa.

Estos procesos de automatización apuntan a asignar una región a un elemento de cómputo del programa de forma que tengan el mismo ciclo de vida. Los elementos pueden ser por ejemplo objetos, métodos, threads. Los objetos que son albergados en las regiones deben tener alguna relación con el elemento, y en particular su ciclo de vida debe estar contenido en el ciclo de vida de la región/elemento a la que pertenece.

En [6] se propone un modelo general del concepto "dueño de región" permitiendo que varios objetos puedan tener control sobre sus regiones, en cambio [9, 21, 7] presentan variaciones de modelos de regiones vinculadas a métodos. Estos últimos brindan herramientas semi automáticas basadas en el análisis estático de “escape”, que se basa en determinar el *scope* de un objeto dentro de la pila de métodos en tiempo de ejecución (dependiendo del grafo de referencias generadas posibles), asociando finalmente su ciclo de vida a dicho método-región [25, 10, 3, 4].

En [9] se utiliza la información generada y se reproduce como salida un programa Java equivalente con instrucciones para el manejo de memoria. Estas instrucciones proveen una interfaz para la administración de las regiones de memoria y son una extensión de la sintaxis Java, por lo que requiere de un compilador especializado que sepa interpretar dichas instrucciones. En cambio en [21], se determinan “tribus” de objetos donde cada una de ellas está conformada por los objetos que se referencian de alguna manera. Aquí no se genera un nuevo programa Java, sino que la información es interpretada para condicionar la utilización de regiones en *runtime*.

En [7] se utiliza un modelo de regiones determinado por un análisis de escape más refinado. El fuerte de éste modelo es la capacidad de obtener cotas de consumo para cada región-método. El

poder contar con esta información hace que sea posible desarrollar una administración de memoria muy eficiente pudiendo diagramarla en forma contigua (por thread), ya que se conoce el tamaño de cada región, y esto trae beneficios como la facilidad de remoción de las regiones y asegurar la defragmentación de la memoria.

La dificultad de este tipo de modelos es obtener cotas de consumo precisas para las regiones generadas asegurando que el programa no se quede sin memoria ni genere regiones excesivamente grandes.

2.2. Modelo de memoria elegido

Para el desarrollo de este trabajo se tomó como base la idea del modelo de regiones asociadas a métodos. Cada método puede tener una región asociada y los objetos contenidos en ella no deberán sobrevivir al método.

Para poder cumplir con esta condición es preciso definir reglas a la hora de determinar qué objetos vivirán en cada región de memoria. Los algoritmos de análisis de escape mencionados anteriormente proveen soluciones seguras de modo que los objetos del programa nunca referenciarán a objetos eliminados por el mal uso de las regiones.

En algunos procesos de análisis de escape se obtienen resultados más precisos que en otros: se llegan a detectar más regiones permitiendo una mejor administración de los recursos de memoria (se evita tener en la región objetos que podrían ser inalcanzables). Dado que encontrar una solución con un grado muy preciso de refinamiento es costoso, los modelos que intentan conseguirlo concluyen en herramientas cuyos procesos son ineficientes o incompletos, y no es factible aplicarlos en la práctica. Otras, en cambio, plantean un análisis más básico (y más rápido); proveen soluciones válidas, pero no tan precisas, ya que posiblemente exista una mejor forma de asociar objetos a las regiones de manera que la ejecución de la aplicación resulte más efectiva en tiempo y espacio.

El modelo de memoria planteado se basa fuertemente en la adquisición de la información de escape (independientemente de su origen) para que pueda ser utilizada a la hora de definir un programa manualmente, o en una futura implementación, automáticamente.

En el ejemplo 2.1 se presenta un caso sencillo para poder apreciar el comportamiento de un administrador de memoria basado en regiones. Se supondrá que los tamaños de las regiones son conocidos al momento de su creación de modo que los objetos creados cabrán en ellas. El comportamiento de las regiones se asemejará al modo en que se ejecutan los métodos, concluyendo en un modelo de pila de regiones-métodos en memoria.

```
public void m0(){
   A a  = m1();          (m0.1)
   B b = new B();        (m0.2)
    ...
}

public A m1(){
   B b = new B();        (m1.1)
   m2(b);                (m1.2)
   return b.a;           (m1.3)
}

public void m2(B b)){
   b.a = new A();        (m2.1)
}
```

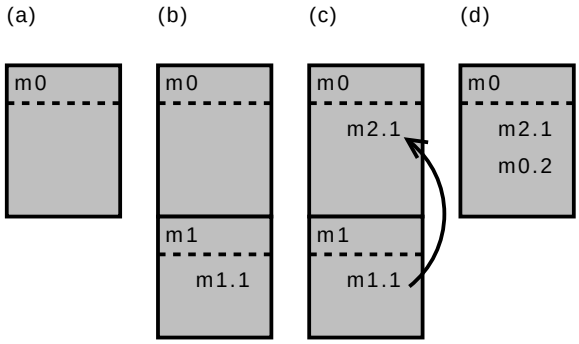

Figura 2.1: Ejemplo del comportamiento del modelo de regiones asociadas a métodos.

Si la ejecución comienza en *m0*, una solución trivial para la utilización de regiones de memoria sería hacer que todos los objetos creados durante el programa se creen en la región de *m0*, de esta forma nos aseguramos de no cometer errores en la asignación de regiones ya que todos los objetos vivirán hasta que finalice *m0*. Pero esta solución no trae ninguno de los beneficios mencionados anteriormente ya que el comportamiento sería similar al de una memoria inmortal. Una mejor alternativa es segregar los objetos según su ciclo de vida y asociarlos a los métodos (regiones) correspondientes. En el programa se pueden distinguir los siguientes lugares de creación de objetos:

- En *m0*: línea 2 (se denominará *m0.2* para abreviar la notación).
- En *m1*: *m1.1*
- En *m2*: *m2.1*

Dado que la ejecución comienza en *m0*, el objeto creado en *m0.2* deberá ser asignado a una región para este método. En *m1* se crea *m1.1*, pero se retorna solo *b.a* (que no contiene referencias a *m1.1*), por lo tanto puede definirse una región para *m1* que contenga a *m1.1*. En *m2* el objeto *m2.1* es asignado a *b.a* y escapa del ciclo de vida de *m2* porque el objeto *b* es pasado por parámetro desde *m1*, y también escapa de *m1* porque es retornado quedando finalmente asignado a la región de *m0*.

En la figura se puede apreciar la simulación del comportamiento de las regiones durante la ejecución del programa.

(a) Cuando se invoca a *m0* se crea su región, que todavía no contiene objetos ya que la primer instrucción es la invocación al método *m1*.

(b) Al iniciarse la ejecución de *m1* deberá crearse su región asociada, que almacenará a *m1.1*

(c) En *m1.2* se invoca a *m2* (no tiene región asociada). En *m2.1* se crea el objeto *a* que será almacenado en la región *m0* quedando referenciado por el objeto *m1.1*

(d) Al finalizar la ejecución de *m2* y luego *m1*, se procede a la eliminación de la región de *m1* con todos los objetos que contiene (en este caso *m1.1*) y finalmente se crea en *m0* el objeto *m0.2*. Al finalizar *m0* deberá removerse la región volviendo la pila de regiones al estado inicial.

Una restricción importante a mencionar es la dependencia entre las regiones, definida a través de las relaciones entre los objetos: los objetos de una región sólo pueden referenciar a objetos de una región creada anteriormente en la pila de regiones (como en el caso de *m1.1* que referencia a *m2.1*). Esta condición es necesaria para asegurar que al eliminar una región no queden referencias rotas. Además, como las regiones simulan de alguna manera el *stack frame* de ejecución, es intuitivo pensarlo de esta manera.

Capítulo 3

Jikes RVM

Para poder implementar el modelo de administración de memoria se evaluaron diferentes máquinas virtuales Java de código abierto, entre ellas Jikes RVM[15], Kafee[17], JamVM[19], Cacao[18].

Entre las máquinas virtuales Java evaluadas Jikes RVM y Kaffe presentan una arquitectura modular accesible y los respalda una comunidad de desarrolladores con artículos publicados en la materia, por lo que serían las candidatas para el desarrollo. Se optó por elegir Jikes RVM debido principalmente a que está programada casi íntegramente en Java, a diferencia de Kaffe que está escrita en C, y emplea componentes reutilizables que facilitan el desarrollo, sobre todo en las tareas de administración de memoria.

Jikes RVM es una máquina virtual de código abierto que corre programas en bytecode Java. A diferencia de otras JVM, está programada en Java y responde al estilo de implementación meta-circular donde el código fuente es evaluado por el mismo lenguaje.

Para poder tener esta facilidad JikesRVM requiere un proceso de inicialización (*bootstrap*), que está escrito en C y tiene la responsabilidad de cargar y correr una imagen de inicialización.

En esta sección se describirá el funcionamiento básico de esta VM para poder entender el resto del trabajo.

Historia

1998	JikesRVM inicialmente fue un proyecto interno de IBM con el propósito de investigación (Jalapeño JVM)
2000	Se publican papers sobre características de Jikes y el código fuente es liberado a algunas universidades.
2001	Se libera la versión 2.0 como un proyecto de código abierto bajo la licencia *Common Public License*. Esta versión posee soporte para PowerPC y arquitecturas Intel. Se disponen de diferentes algoritmos de garbage collectors.
2002	Se libera la versión 2.2 con implementaciones de Garbage Collectors implementados sobre MMTk (Memory Management Toolkit)[2].

2004 Mejoras en la estabilidad y performance con la versión 2.4.

2007 Versión 2.9 Soporte para Java 5.0.

En este trabajo se utilizó la versión 2.9 de la RVM. Veremos algunos conceptos necesarios para entender los detalles de implementación y la toma de algunas decisiones.

3.1. Startup y mapeo de memoria

Inicialización de la máquina virtual Al correr el bootstrap (programa de inicialización en C), se mapean los datos y algoritmos necesarios en memoria para el funcionamiento de la VM Java, y finalmente se cambia el control de la ejecución a la JVM. La inicialización consta de los siguientes pasos: la registración de los administradores de señales (*signal handlers*) para el manejo de errores de hardware generados por la RVM, la generación del mapa de memoria inicial, mapeo de los archivos de imagen, mapeo de las direcciones de las funciones C invocadas por *Runtime* para interactuar con el sistema operativo, la inicialización de las tablas de objetos en memoria (*JTOC*), y finalmente la inicialización del *thread* de bootstrap Java, quien se encargará de crear los *threads runtime* de Java (compilación, optimización de bytecode y Main). Luego terminar la ejecución de la imagen del *bootstrap*.

Mapa de memoria de la VM Jikes divide la memoria en varios segmentos que contienen código, datos, o combinación de ambos:

Segmento Boot Ocupa la parte de memoria más baja y aloja la imagen de inicialización, y algunos datos y códigos generados dinámicamente durante la inicialización.

Segmento imagen RVM Esta área es creada por el programa de inicialización y contiene los datos estáticos iniciales, las instancias y métodos compilados requeridos para poder ejecutar la *VM*. Los datos son cargados desde un archivo de imagen creado *offline* llamado (*boot image writer*). Al volcar esta imagen en memoria se dispondrá de la siguiente estructura inicial:

- JTOC: Tabla de objetos. Contiene referencia a métodos estáticos, constantes y constructores de clases.
- TIBs (Type Information Blocks): Contienen información sobre los tipos y clases del sistema. Se almacenará la información dinámica generada, como referencias a métodos instancia compilados.
- Arrays de métodos estáticos y campos estáticos referenciados desde JTOC.
- Arrays de código de los métodos dinámicos referenciados indirectamente de TIBS.
- Clases internas de los *Classloaders* e instancias de métodos también referenciadas por TIBS.

- Estructuras auxiliares de estas clases y métodos como arrays de *bytecode*, registros de compilación, mapas de los *garbage collectors*, etc.
- Instancia del proceso de inicialización para comenzar la ejecución de Java.
- Instancia de *thread* Java donde comenzará la ejecución.
- Pila de *threads* utilizada por el *thread bootstrap*.
- Registro maestro que contiene las referencias a las anteriores partes de la imagen en memoria (como el *JTOC*, el *thread bootstrap*, etc).

RVM Heap Segment Este segmento provee espacio para almacenar el código y los datos creados durante la ejecución del programa Java. La *VM* puede ser configurada para emplear diferentes administradores de memoria a través de *MMT*k, de esta manera es posible estructurar el heap dependiendo de qué tipos de objetos se deban almacenar.

3.2. Componentes que conforman JikesRVM

Memory Management Toolkit (MMTk) Memory Management Toolkit (MMTk) es un componente externo a JikesRVM programado en Java, que incluye un conjunto de recolectores de basura, y resuelve la paralelización en la creación/recolección de objetos controlando la sincronización entre componentes locales y globales.

Biblioteca de clases Java (*classpath*). Las bibliotecas Java soportadas para Jikes son Apache Harmony o el classpath de GNU.

Compilers Jikes RVM posee dos modos de compilación: la primera se denomina "baseline" y en el proceso de compilación se genera código máquina para una arquitectura particular a partir del bytecode. El otro tipo de compilador es el optimizado, que recompila el código en diferentes niveles de optimización analizando el bytecode y optimizando las instrucciones y porciones de código utilizado. Para ello utiliza diferentes representaciones intermedias. Este proceso de compilación se basa en un modelo de análisis de costo-beneficio para determinar cuando recompilar y qué nivel de optimización utilizar.

Runtime El entorno runtime es el encargado de llevar a cabo la ejecución del código máquina compilado y brinda soporte para los mecanismos de bloqueo, *scheduling* colaborativo y manejo de excepciones.

3.3. Administración de memoria en JikesRVM

Se brindará una sección especial para el módulo de administración de memoria que utiliza Jikes RVM ya que será la base para la mayor parte de implementación del trabajo.

3.3.1. Conceptos preliminares

En esta sección se explicarán algunos conceptos básicos sobre administración de memoria indispensables para entender las decisiones y el funcionamiento de los algoritmos utilizados durante la implementación.

Asignación de memoria Las asignaciones en memoria o (*allocations*) de las máquinas virtuales o intérpretes de lenguajes pueden ser de tres tipos: estáticas (*static*), de pila (*stack*) o de heap. Las asociaciones estáticas suceden en tiempo de compilación y son utilizadas para datos como variables globales y constantes. Las de pila se dan en tiempo de ejecución y se utilizan sobre variables locales o de *threads* en ejecución (*thread local*), parámetros de los métodos, y valores de retorno de los mismos. La asignación en el heap se refiere a la creación dinámica de los objetos cuyo ciclo de vida puede escapar al ciclo de vida de los procesos que los crean, o aquellos cuyo tamaño no se puede determinar en la fase de compilación. **Este trabajo se focaliza en la administración de memoria del heap**. Existen diferentes maneras en las que se puede llevar a cabo el proceso de asignación de memoria en dicho espacio, en la mayoría de los casos se utilizan estructuras de tipo *bump-pointers* o *free-lists (ver siguiente).*

3.3.1.1. Allocators

La función principal de un *allocator* es asignar objetos a un rango de memoria disponible. Son los encargados de administrar los recursos de memoria del sistema reservándolos y liberándolos cuando sea necesario, y de estructurar la memoria de la manera más conveniente para soportar el funcionamiento del *GC*. Existen varios tipos de allocators que difieren en el modo de utilizar los espacios de memoria, y en cómo disponen de ellos. En esta subsección se describirán dos tipos de allocators: *bump-pointer* (lista de pedazos - *chunks* - contiguos de memoria) y *free-list* (lista de espacios libres).

Bump-Pointer Este tipo de allocator utiliza la memoria en forma creciente y mantiene un puntero para marcar la dirección en donde sucederá la asignación (*alloc*). Cada vez que esto ocurre el puntero es incrementado (*bumped*) hacia la posición de memoria donde finaliza la asignación. El resultado es una serie de asignaciones contiguas con direcciones de memoria crecientes. El allocator dispone de un número finito de páginas pedidas al administrador de memoria virtual (administrador de páginas), que constituyen un *chunk*. Inicialmente el *bump-pointer* es seteado al comienzo del *chunk* y es incrementado en el tamaño del objeto en cada asignación. Cuando un *chunk* es consumido, uno nuevo es obtenido. Si el heap está completo, se lanza el *GC* para liberar la memoria.

Este algoritmo de asignación es uno de los más simples y rápidos. Cuando una asignación es realizada en el *chunk* de memoria actual (memoria local reservada para el thread de cada procesador) se la suele llamar *fast-path allocation*, sólo se ejecutan unas pocas instrucciones incluyendo el chequeo para verificar el espacio disponible en el *chunk*, incrementar el puntero y retornar la dirección de asignación.

El *slow-path allocation* sucede cuando el *chunk* de memoria actual está completo, por lo tanto se debe verificar si se debe lanzar el *GC*, reservar nuevas páginas de memoria (requiere sincronización de threads) e inicializar el *chunk* con la estructura del bump-pointer (seteo de *headers* y puntero). Dado que el tamaño de memoria pedida es muy chico en relación a las páginas reservadas, las asociaciones de tipo *fast-path* tendrían que ser el caso más común haciendo más eficiente la asignación.

Este algoritmo es utilizado en conjunto con recolectores de basura *semi-space*: divide el heap en dos partes iguales llamadas *from space* y *to space* (inicialmente vacío). Cuando *from space* se llena se dispara el *GC,* que compacta el heap copiando los objetos que están vivos al *to space,* liberando el *free space*.

Una de las desventajas de este allocator es que requiere mover objetos en cada ejecución del GC.

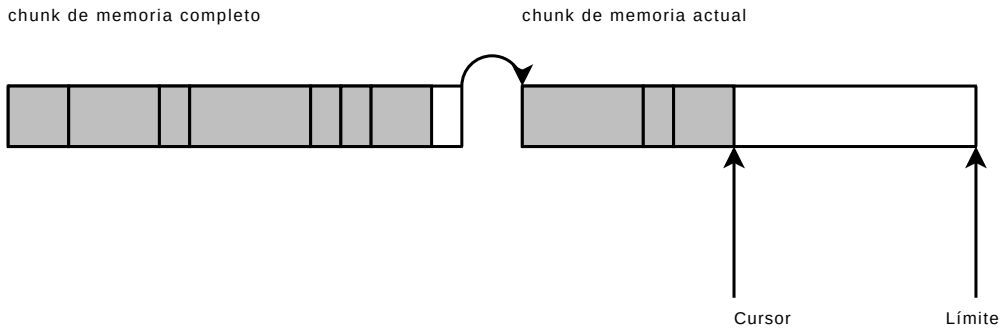

Figura 3.1: Allocator bump-pointer.

Free-List El modo de funcionamiento de una *free-list* se basa en la unión de áreas libres de memoria en una lista encadenada. Los nodos de la lista pueden ser de diferente tamaño. En el proceso de asignación, el algoritmo busca un nodo de la lista lo suficientemente grande para que quepa el objeto. Cuando el nodo es encontrado, se elimina de la lista y la dirección de memoria del mismo es retornada. El proceso de GC es disparado cuando no puede ser realizado un pedido de asignación, y la memoria liberada es devuelta a la *free-list*.

La principal ventaja de las *free-list* es que en el GC no es necesario mover objetos. De todas formas, en cada asignación es necesario recorrer la lista en busca de un nodo (lo que agrega *overhead* y deteriora la *performace*). Además, esta estrategia no asigna los objetos de forma contigua, lo que dificulta el aprovechamiento óptimo del espacio. Al igual que en el caso de bump-pointer, la asignación de memoria puede ser por *fast-path* o *slow-path*, dependiendo del espacio ocupado en la lista

Free-List Segregadas Las listas segregadas son una extensión de las *free-list*. La diferencia es que eśtas mantienen varias listas para cada tamaño de clase diferente en el sistema. En el proceso de asignación se obtiene un nodo de la lista correspondiente al tamaño de la clase del objeto a crear. El uso de este tipo de listas elimina el *overhead* de buscar un nodo con el tamaño adecuado

para asignar el objeto. Las listas segregadas son utilizadas en espacios de memoria donde el GC no mueve objetos como por ejemplo *reference counter* y *mark&sweep GC*.

3.3.2. MMTk (Memory Management Toolkit)

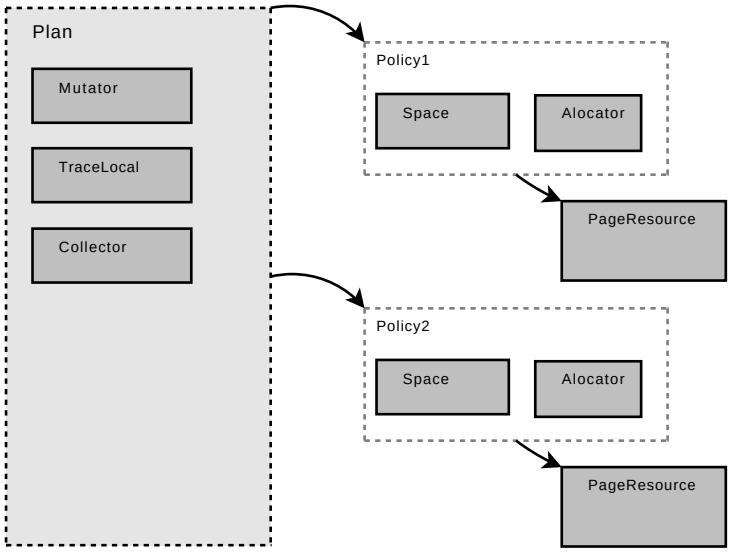

Figura 3.2: MMTk. Plan de memoria.

En la sección anterior mencionamos que *MMTk* es una herramienta que permite implementar la mayor parte de los procesos vinculados con la memoria, dispone de un conjunto de funcionalidades comunes que permiten abstraerse del manejo de los recursos de memoria a bajo nivel, como también implementaciones de diferentes tipos de *GC* y allocators. Está diseñado para el desempeño en entornos con multiprocesadores contemplando la sincronización y la coherencia de la cache.

MMTk plantea un modelo de jerarquía de clases a implementar (ver figura 3.2) para facilitar el desarrollo de un nuevo administrador de memoria, lo que contemplaría la creación o especificación de allocators, *garbage collectors* y el *layout* de la memoria virtual. De esta forma implementar una nueva administración de memoria se realizaría extendiendo las clases provistas, agregando y sobreescribiendo la funcionalidad provista por defecto.

La entidad de mayor nivel en esta jerarquía de clases es el *Plan*. Un administrador de memoria está definido por una única instancia de *Plan* en donde se definen el conjunto de espacios/políticas de memoria (ver 3.3.2.2) que intervienen en la asignación de los objetos y en el comportamiento de *garbage collector*.

Componentes del plan:

- `Mutator`: define las operaciones de asignación por thread de procesador. El componente global que sincroniza es el Plan. La idea es que el mutator utilice el allocator como componente local cuando los allocs se realizan por *fast-path*, y cuando no puede llevarse a cabo la asignación, deberá realizarse un nuevo pedido *slow-path* al componente global sincronizado.

- `Collector`: define los métodos para la recolección de basura en la VM. Hay tantas instancias de collectors como procesadores. Al igual que en el mutator, el Plan es el componente global que sincroniza la recolección.

- `TraceLocal`: Es el componente local que implementa el *core* de la clausura transitiva a través del grafo de objetos en el heap.

Los tres componentes principales actúan sobre todos los objetos del heap y en ellos se debe discriminar a qué espacio corresponden para delegar las tareas a las correspondientes políticas que intervienen en el plan. En la figura 3.2 puede observarse un diagrama de los componentes que conforman un plan de memoria.

Un espacio de memoria es la representación virtual de una porción de memoria física. Cada espacio es manejado bajo una política de asignación/recolección independiente, pero sus rutinas están dirigidas por el mismo plan de memoria.

Jikes RVM tiene los siguientes espacios:

• Immortal. Asignación de objetos mediante un *bump-pointer*. No posee *garbage collect*or.

• Mark-sweep. Asignación de objetos administrados por una lista segregada (*free-list*). *Mark-sweep collector*.

• Large object space. Manejo de memoria para objetos grandes (arreglos y matrices).

• Copy space. Asignación de objetos mediante un bump-pointer. Copying collector.

• VM space. Es el espacio donde vive la máquina virtual. No se producen *allocs*, pero la MMTk puede escanear objetos allí, ya que pueden existir estructuras que apuntan al heap.

• Raw memory space. MMTk crea sus propias estructuras de datos en esta área, se guarda la metadata del *garbage collector*.

En el plan base de la *VM* se definen cuatro espacios: el espacio nativo donde vive la máquina (*VM space*), un pequeño espacio inmortal, un espacio Raw para el *GC* y un espacio *Large*. La implementación del plan debe definir nuevos espacios y sobrecargar los métodos correspondientes para administrar los objetos del heap.

3.3.2.1. Garbage Collectors en Jikes RVM

El proceso de recolección de basura es inicializado mediante la invocación del método *collect* del plan local. Es responsabilidad de la máquina virtual asegurar que este método sea llamado en cada procesador de forma paralela. Hay tres mecanismos principales en el proceso de GC:

• El *polling*: los allocators deben periódicamente verificar el estado del consumo de la memoria que administran para habilitar al plan a decidir cuándo realizar el proceso de recolección.

• La recolección (incluye múltiples fases): implementadas por el método collectionPhase de los planes local y global.

• Traza del heap (clausura transitiva): implementado en Trace y TraceLocal.

El proceso de recolección puede ser iniciado por una de las siguientes razones: el mutator puede especificar explícitamente su ejecución (por ejemplo ejecutando la sentencia java *System.gc()*), o el administrador de memoria puede determinar la necesidad de ejecución ante la incapacidad

de satisfacer un pedido de asignación por agotamiento del espacio. Esta validación se lleva a cabo cuando el allocator realiza un pedido de memoria por *slow-path* de acuerdo a la frecuencia especificada en la creación del espacio. Cuando surge la necesidad de lanzar el GC, la VM se prepara a sí misma para la recolección e invoca al método *collect* del plan local para cada CPU virtual. Una vez finalizado el proceso se ejecutan tareas de post colección y se intenta satisfacer el pedido de memoria original.

Fases de la recolección La recolección de basura generalmente sucede de forma estructurada y está dividida en diferentes fases. En *MMTk* las fases de la colección son ejecutadas mediante el método *collectionPhase()* definido en las instancias de Plan local y global. Este método toma como parámetro un entero que identifica la fase, que es definida en el plan de memoria, y cada una de ellas es ejecutada en paralelo en los threads de recolección.

Ej.: Fases de Mark & Sweep GC: Este recolector consta de 2 fases principales, una de traza del heap en donde se marcan (*mark*) los objetos alcanzables desde las raíces (objetos accesibles desde el programa en ejecución), y una fase de liberación (*sweep*) de los objetos no marcados en la fase anterior.

Traza del heap En algún punto del proceso de colección es necesario descubrir qué objetos están vivos. Generalmente esta tarea se lleva a cabo realizando una clausura transitiva sobre los objetos vivos. Éste es el proceso más crítico en la performance de los GC.

La traza está encapsulada en el plan de MMTk y se realiza de la siguiente manera:

• Enumeración de raíces: se agregan en una cola de referencias a objetos raíz (objetos referenciados directamente mediante los stack frame de los métodos activos en ejecución o estáticos).

• Iteración sobre la cola de raíces marcando los objetos alcanzables y agregándolos en una nueva cola para ser trazados.

• Iteración sobre la cola de objetos rastreados marcando los nuevos alcanzados y agregando el conteo de referencias que será de utilidad para el proceso de recolección. Los nuevos objetos serán agregados a la cola y el proceso se iterará hasta que la cola esté vacía.

3.3.2.2. Política

Una política en MMTk define a una región virtual de memoria con un método de asignación y un mecanismo de recolección de basura. Está determinada por un componente global sincronizado y otro local no sincronizado denominados Space y Allocator respectivamente.

Espacios Los espacios de memoria organizan el heap, lo dividen en unidades que pueden funcionar independientemente entre sí definiendo sus propios procesos de asignación y recolección. Cada espacio administra un rango de memoria virtual de acuerdo a un allocator y un recolector determinado. Al crear un espacio se puede especificar el tamaño del mismo, si es contiguo o no, y hasta su posicionamiento en el rango de memoria disponible. Por ejemplo si se quieren definir los espacios de memoria para poder implementar un garbage collector *semi-space*, se podría definir

uno de ellos en las posiciones de memoria más bajas disponibles, para el *from-space,* y otro en las posiciones más altas, para el *to-space*.

Algunas de las responsabilidades que tiene un espacio son:

- Control de las páginas tomadas: El espacio debe poder contabilizar las páginas que tiene en uso y las reservadas. Esta información es de utilidad para los procesos que se encargan de lanzar el *GC*.
- Definir el trazado de los objetos en el espacio: cada espacio conoce la estructura de sus objetos y la metadata para podes marcarlos o desmarcarlos en la clausura transitiva del heap, según la etapa del *GC* en que se encuentren.
- Preparar el espacio para la ejecución garbage collector (fase Prepare del GC).
- Ejecutar tareas luego del Garbage collector (fase Release del GC).
- Determinar si un objeto (referencia a un objeto) está vivo.

Generalmente sólo hace falta crear los espacios de memoria correspondientes para la administración de los objetos en el heap. Jikes provee los espacios específicos para administrar los objetos que maneja internamente (espacio de la VM).

Allocators Los allocators son los que llevan a cabo el proceso de asignación de memoria para albergar los objetos en el espacio, para ello diagraman la memoria disponible en lo posible garantizando una buena organización para facilitar las tareas del GC. Los allocators se ejecutan por procesador y mantienen un conjunto de páginas de memoria disponible. El espacio es el componente que los sincroniza y esto sólo sucede cuando se liberan o se requieren más páginas de memoria que las disponibles (*slow-path*).

3.3.2.3. Administrador de páginas

El administrador de páginas de memoria (*PagesResource*) es el encargado de tener el control de las páginas de la memoria virtual pedidas al sistema para un espacio dado. Es la administración de memoria de más bajo nivel y centraliza los pedidos de los allocators manteniendo la sincronización entre ellos. Al inicializarla deben especificarse atributos del mismo como la contigüidad de los *chunks* de memoria a reservar y la capacidad o no de expansión. Posee dos métodos principales: *reserve* y *release*, que se utilizan para pedir páginas de memoria al sistema y liberarlas respectivamente.

Capítulo 4

Análisis del problema

4.1. Decisiones generales

El desarrollo del administrador de regiones sobre la máquina virtual Jikes RVM deberá reflejar el funcionamiento del modelo teórico elegido y utilizar la información generada por las herramientas disponibles. Para ello se implementarán los componentes necesarios para administrar regiones, respetando las definiciones teóricas del modelo. Se deberá realizar una implementación modular y extensible, para luego poder incorporar fácilmente nuevas funcionalidades y/o mejoras sobre la VM.

Al escoger una VM en particular, muchas de las decisiones a tomar van a estar ligadas a la misma. No obstante, hay varias alternativas en las formas de encarar el desarrollo de un modelo de memoria, que serán explicadas a lo largo de este trabajo.

Las herramientas disponibles para llevar a cabo la implementación son: el modelo teórico de regiones, la máquina virtual Java Jikes RVM, y un conjunto de herramientas para determinar las regiones de un programa y la estimación del consumo de memoria las mismas.

Existen diferentes maneras de implementar un modelo de memoria basado en regiones. El caso ideal sería proponer una máquina virtual que tome un programa Java convencional y lo ejecute detectando las regiones de memoria a utilizar con sus tamaños automáticamente, es decir incorporar el proceso de inferencia de regiones en tiempo de ejecución - compilación a código máquina.

Si se implementa el modelo de pila de regiones utilizando sólo las herramientas provistas por [7] no se podría obtener la información completa de regiones/consumo para cualquier caso. Por ejemplo en muchos tipos de recursión o ciclos en donde no es factible determinar un invariante de consumo (y hasta a veces es complejo detectar las regiones), las herramientas automáticas no suelen calcular una buena solución y por lo tanto se debe dejar la posibilidad de permitir al programador la especificación de regiones manualmente, y para ello es necesario definir una sintaxis. Esta es una de las primeras tareas a definir y en el capítulo 6 se analizarán algunas alternativas.

En principio la implementación constaría de dos etapas: la definición de un componente interno para el manejo de regiones en memoria, y la definición de una sintaxis Java para la administración

de las mismas, con todos los componentes intermedios para la interacción entre ambos. Si se desea poder incorporar la especificación de regiones en forma automática, también habría que implementar los mecanismos necesarios para poder introducirla a la VM, manteniendo el mismo manejador de memoria.

En esta tesis la idea es implementar el administrador de regiones de bajo nivel y brindar una interfaz Java para utilizarla en cualquier programa Java convencional, dejando para un trabajo a futuro la incorporación de procesos automáticos que faciliten la tarea.

La implementación actual de Jikes RVM provee un conjunto de GCs a utilizar. En este trabajo utilizaremos como base para el desarrollo: Jikes RVM con Mark&Sweep (dada las sencillez de su funcionamiento). La misma consta de un sólo espacio de memoria destinado al heap, con el proceso de recolección mencionado.

Para soportar regiones, se deberá convertir (o adicionar) un espacio de memoria destinado a tal fin, con los mecanismos necesarios para su administración; lo que involucra la determinación del tipo de administrador de páginas y allocator a utilizar, la estructuración de la memoria en dicho espacio, deshabilitar o integrar el GC Mark&Sweep (según se utilicen o no ambos espacios) y proveer una interfaz de uso en runtime (para los procesos internos a Jikes), y desde el lenguaje Java, para el desarrollo de programas. Para ello se deberán incorporar al lenguaje las sentencias requeridas para administrar este espacio manualmente, pero dejando abierta la posibilidad de incorporar (en un futuro) los procesos automáticos.

Estos objetivos serán de importancia al definir la implementación de los componentes, resultando en una arquitectura separada en capas que interactúan a diferentes niveles. Por ejemplo si en vez de definir un modelo de regiones ligadas a métodos, se quisiera implementar un modelo más general donde el usuario indicara de manera explícita la creación, eliminación, o lugar de asignación de los objetos, sólo habría que redefinir la sintaxis Java correspondiente (con su proceso de compilación), pero la interfaz de memoria de bajo nivel sería exactamente la misma.

Para proveer flexibilidad en la implementación de diferentes VM sobre el mismo desarrollo se implementarán un conjunto de componentes individuales como espacios virtuales de memoria (Spaces), allocators y planes de memoria (componentes de MMTk), de manera que se puedan construir "VMs a medida" con sólo combinarlos. Para determinar qué componentes crear se analizarán los trabajos previos relacionados.

Si se toma como ejemplo las características de la RTSJ [5], que especifica la utilización de *threads* de tiempo real y regiones de memoria acotadas que pueden ser utilizadas con el resto del programa (que no es real time), estamos en un modelo de memoria que debe soportar el comportamiento habitual del heap con GC, y definir una nueva administración/estructura para el modelo de tiempo real.

Si se quisiera implementar en Jikes un área para soportar la utilización de regiones determinadas por el programador o una herramienta de instrumentación, manteniendo el funcionamiento habitual para el resto del programa, sucedería algo similar: se debe partir de al menos dos áreas de memoria

para poder administrar los objetos pertenecientes a regiones en una, y la otra dejarla para el resto de los objetos.

Si en cambio se quisiera soportar un modelo completo de regiones (sin el área para objetos que no están en regiones), en primer lugar habría que tener en cuenta que todo objeto debería tener asociada una región (que no siempre es factible). Para ello haría falta un análisis sobre las asignaciones de memoria de todos los métodos teniendo en cuenta el *classpath* entero de la aplicación a correr y tener control sobre todas las asignaciones internas de la VM en todos sus *threads* internos (no threads de aplicación).

Las herramientas provistas en los analizadores de [21, 7, 9] podrían generar la información necesaria para la especificación de las regiones que intervienen en el programa con los lugares de creación de los objetos que pertenecen a las mismas, pero como se mencionó anteriormente, a veces no es posible obtenerla, concluyendo en una implementación no del todo eficiente para un entorno de tiempo real. Como complicación adicional, la máquina virtual a utilizar puede crear objetos en el heap para uso interno de la VM (adicionalmente a los que se crean en el programa a ejecutar), y no es posible conocer a priori todas las formas en que pueden llevarse a cabo. Por ejemplo en Jikes RVM algunos objetos son creados desde threads internos como el sistema adaptativo de compilación, las entradas y salidas (I/O) de bajo nivel, o creación de objetos en el entorno de *runtime* que permiten la ejecución del programa. Todas ellas no pueden ser procesadas en un analizador genérico de código fuente Java como los propuestos.

Por estas razones se definirán dos áreas de memoria: *regions-heap* para soportar la creación de objetos en regiones, y *common-heap* para albergar a aquellos objetos que no tienen alguna región asignada. Para administrar estas áreas de memoria es necesario definir cómo serán organizadas: mientras que el área *regions-heap* debe responder a una diagramación de la memoria para el soporte de regiones, en *common-heap* es necesario definir alguna política de administración de memoria convencional.

La manera más simple de organizar el área *common-heap* es definirla inmortal: todos los objetos albergados vivirán hasta que concluya la ejecución del programa y se destruya la VM, independientemente de que los objetos no sean alcanzables. Si bien esta solución es la más sencilla de implementar y no tiene los costos de *overhead* de un GC, sólo sería conveniente utilizarla si la cantidad de objetos en ella fuera pequeña y el tamaño de la misma no dependiera de la parametrización de la ejecución del programa. De otra forma no se podrá asegurar que la máquina virtual pueda ser ejecutada en el espacio de memoria esperado.

Una manera de tener control sobre el tamaño del área *common-heap* es agregar una política de recolección de basura sobre la misma. De esta manera, además de poder correr la VM con una cantidad de memoria acotada, se obtendrá un modelo híbrido de memoria, lo que permitiría realizar pruebas de performance sobre la incorporación de objetos en ambas áreas de memoria y poder correr aplicaciones total o parcialmente orientadas a regiones.

Para implementar *regions-heap*, también existen varias posibilidades: una buena opción para éste modelo en particular sería crear los objetos que no escapan en el *stack frame* de cada método [20, 14], pero esta decisión no permitiría en un futuro implementar otros modelos diferentes donde

las regiones no estén ligadas directamente a los métodos y se perdería la arquitectura modular propuesta, de modo que se ha optado por crear un área en memoria dedicada exclusivamente a la asignación de objetos a regiones.

4.2. Plan del desarrollo

Para poder abarcar un gran espectro de las posibilidades planteadas se tomó la decisión de crear las siguientes máquinas virtuales:

- La primera de ellas presenta un plan de memoria comprendido por dos espacios: uno correspondiente al área de memoria *common-heap*, y el otro a *regions-heap*. El espacio *common-heap* es inmortal, y el de *regions-heap* es administrado por un espacio de regiones de tamaños fijos.
- La segunda implementación es una extensión de la primera y consiste en incorporar un GC sobre el espacio correspondiente a *common-heap*.
- En la tercera, se define un modelo completo de regiones. Sólo existe un espacio de memoria destinado a la administración de regiones. Para soportar el área de memoria *common-heap* se debe considerar una región base que funcione como inmortal, y el resto de las regiones serían las correspondintes al área *regions-heap*. La región base sería similar a tener el espacio de memoria inmortal de la primer VM planteada, pero permitiría evaluar el sistema con una única administración de memoria por regiones. Al inicializarse la VM debería crearse la región *common-heap,* y sólo se eliminaría cuando termina la ejecución del programa. Al no poder predecir el tamaño de esta región y estar en la base de la pila, el modelo debería permitir la expansión de las regiones definiendo una política diferente para este espacio, que en las VMs anteriores.

A grandes rasgos cada una de las implementaciones de las *VMs* consta de las siguientes partes:

1. Definición del Plan de memoria MMTk:
   - Definición de espacios con políticas de memoria e integración con otras políticas nativas de *JikesRVM*. La creación del plan contempla la estructuración de la memoria virtual por medio de la definición de espacios de memoria (Spaces de MMTk) con sus políticas de asignación y recolección de objetos asociada. A este nivel de la implementación se deben brindar los mecanismos básicos para permitir la asignación de los objetos del programa en los espacios de memoria correspondientes.
   - Administración de alto nivel (interno a la VM) de las regiones (*API* Plan de memoria). Se debe definir la interfaz que permite la creación y eliminación de regiones, y la asignación de los objetos dentro de ellas.
2. Definición de la nueva sintaxis y compilación
   - Incorporación e interpretación de sentencias Java al lenguaje para la definición y manejo de las regiones.

- Definición de la estructura de datos y métodos del compilador para guardar la información de regiones.
- Modificación de compilación de métodos para instrumentar con las llamadas a la interfaz de regiones en runtime (3) cuando sea necesario.

3. *Runtime*

   - Creación de *entrypoints* para las invocaciones desde el código compilado, definición de *API* de regiones a nivel runtime.
   - Integración con interfaz de memoria (Plan MMTk).

4. Implementación de memoria de bajo nivel.

   - Administrador de páginas de memoria para soportar el esquema de pila de asignación.
   - Administrador de regiones y allocator de objetos: se implementarán dos allocators de regiones/objetos: el primero soporta la creación de regiones de tamaño fijo, permitiendo contigüidad tanto en la asignación de regiones, como en los objetos dentro de ellas.
     El segundo tipo de allocator soporta crear regiones de tamaño dinámico. Ante la falta de espacio en alguna de ellas, se procede a su expansión por medio de una nueva "parte de región". Este último allocator contempla particionamiento de regiones, y reaprovechamiento de espacios libres.

5. Estadísticas: creación de los módulos de estadísticas para los componentes desarrollados.

Capítulo 5

Políticas y planes de memoria

5.1. Planes de memoria

El plan de memoria es el componente de mayor jerarquía en MMTk. En cada Máquina virtual sólo puede haber un plan de memoria activo. Si bien se pueden tener diferentes planes a disposición, al momento de compilar la máquina virtual, se debe escoger sólo uno.

Se implementaron tres planes de memoria diferentes y en consecuencia tres máquinas virtuales. Como se ha mencionado anteriormente, estos planes de memoria difieren en la especificación de sus políticas y en la utilización de los mecanismos de *garbage collection*. Sin embargo todas tienen como componente en común un espacio/política de asignación de memoria llamado *RegionsSpace*, que será el encargado de proporcionar la nueva administración de memoria (ver 5.2).

Los componentes principales del plan a desarrollar son las clases del módulo MMTk: *Mutator* y *Collector*. En *Mutator* deben crearse los métodos de administración de regiones que son invocados en runtime desde la interfaz de memoria de la VM *(MM_Interface)*, que a su vez llaman a sus correspondientes funciones en el allocator seleccionado. En *Collector*, dependiendo del la configuración del plan, se deberán lanzar excepciones en el caso de que plan no soporte garbage collection. Ej: un caso común sería la llamada `System.gc();` desde el código Java, que deberá retornar una excepción, ya que no estará permitido en la configuración del plan.

5.1.1. Regiones - Inmortal

El primer plan de memoria combina un espacio para la administración de regiones cuyo tamaño es fijo, y la utilización de un espacio de memoria inmortal implementado mediante un *bump-pointer*. La implementación del espacio inmortal es provista por Jikes RVM.

Esta configuración de memoria sería la ideal si todos los objetos del sistema pudieran estar ligados a una región de memoria, y se conociera con anticipación el consumo de la misma. De esta manera el espacio de regiones sería lo que es el heap en una máquina virtual convencional, quedando un pequeño espacio para los objetos inmortales. Los allocators de ambos espacios son eficientes ya que están basados en la idea de *bump-pointer*, que no requiere mantenimiento alguno.

Las únicas operaciones importantes de memoria son las de creación y eliminación de regiones, que se realizan de manera sencilla por poseer un tamaño fijo y poder disponerlas físicamente en una pila.

5.1.1.1. Detalles de la implementación

Como en esta implementación no se requiere la utilización de un recolector de basura, se extenderá simplemente de la clase Plan de MMTk, que no involucra los mecanismos de recolección. Para desarrollar un plan de memoria es necesario crear los componentes principales: La clase *Plan* en sí misma, el *Mutator*, *Collector* y *TraceLocal*.

En el plan se definen los espacios a utilizar:

```
public static final RegionsSpace regionsSpace =
      new RegionsSpace("regions", DEFAULT_POLL_FREQUENCY, VMRequest.create(0.60f));
public static final ImmortalSpace msSpace =
      new ImmortalSpace("ms", DEFAULT_POLL_FREQUENCY, VMRequest.create());
```

En la creación de cada espacio se especifica un descriptor, el período de verificación para la ejecución del GC (omitido en este modelo), y un pedido de reserva de espacio especificado porcentualmente. Si no se especifica este valor, se tomará todo el espacio disponible.

Por defecto Jikes RVM intentará correr el GC cuando la memoria esté llena, y para evitarlo, se deberá sobreescribir el método *collectionRequired* para que siempre retorne el valor *false*.

También, en la clase *Collector*, se deben agregar las excepciones correspondientes en la implementación de sus métodos, evitando de esta manera la invocación de los mismos.

5.1.2. Regiones - Mark&Sweep

Este plan de memoria divide también el heap en dos espacios: uno destinado al uso de regiones *RegionsSpace*, y otro espacio convencional con un sistema de recolección de GC en mark&sweep. Este último utiliza como allocator una lista de espacios libres segregada, dado que en este tipo de procesos de recolección no es necesario mover objetos.

En este plan surge la necesidad de contemplar un espacio para almacenar objetos que no poseen una región identificada. Si se utilizara *Regiones - Inmortal*, y se dispusiera de una gran cantidad de objetos huérfanos de región, el heap crecería rápidamente agotando los recursos de memoria del sistema. Si bien este modelo sigue dependiendo del uso del GC, sirve como entorno de prueba para poder verificar el agregado de regiones a programas convencionales, y poder limitar el espacio total utilizado en el caso en que muchos objetos no tengan regiones.

5.1.2.1. Detalles de la implementación

Definición de espacios:

```
public static final MarkSweepSpace msSpace =
      new MarkSweepSpace("ms", DEFAULT_POLL_FREQUENCY, VMRequest.create(0.3f));
```

```
public static final RegionsSpace regionsSpace =
      new RegionsSpace("regions", DEFAULT_POLL_FREQUENCY, VMRequest.create(0.3f));
```

Al combinar dos espacios heterogéneos donde uno no utiliza recolección de basura y el otro sí, es necesario integrarlos correctamente para lograr un correcto funcionamiento. Esta tarea involucra utilizar las clases provistas por Jikes RVM para habilitar el funcionamiento del garbage collector sobre el espacio Mark&Sweep teniendo en cuenta los objetos en regiones (extendiendo de las clases del plan que soportan GC en vez de las básicas).

Al habilitar el GC en ambos espacios de memoria, se debe modificar el funcionamiento del mismo para hacer inocua la fase de recolección en el espacio de regiones, pero dejando sin alterar la fase en donde se realiza la traza de los objetos. Esto es necesario ya que puede haber dependencias entre los espacios, y de no contemplar las raíces de objetos pertenecientes a "regiones", se estarían eliminando objetos alcanzables del heap que son apuntados desde este espacio. Para ello en la clase del plan *TraceLocal* se debe discriminar el espacio al cual pertenece el objeto e invocar al Trace adecuado a cada espacio, que es el que conoce el header de sus objetos y puede verificar su alcanzabilidad.

5.1.3. Regiones - Puro

Esta implementación responde a un modelo puro de regiones. Su buen funcionamiento depende de la correcta asignación de los objetos a regiones y sólo tiene sentido su utilización cuando todos los objetos poseen una región asociada. Para contemplar la asociación de objetos inmortales se define la primer región como inmortal, de modo que al iniciarse la VM se crea la primer región y sólo al apagarse mediante la interrupción *shutdown*, ésta será eliminada.

Como no es posible asegurar el tamaño de esta región inmortal, el espacio de memoria funciona con el allocator de regiones expansibles (8.2.4), que ante la falta de espacio en cualquier región generará las extensiones necesarias. Si bien la administración de memoria de este modelo es poco eficiente debido al allocator utilizado, si los tamaños son bien ajustados, el comportamiento se asemejaría al modelo *Regiones - Inmortal* 5.1.1.

Este plan fue realizado como prueba de que una VM puede funcionar correctamente solo con un espacio de memoria administrado por regiones. Es un paso previo para la implementación de una VM que utilice información de regiones provista por herramientas externas donde no se posea el conocimiento del tamaño de todas o algunas de las regiones que conforman el programa.

5.1.3.1. Detalles de la implementación

Al no contemplar un recolector de basura, la implementación de este plan es muy similar al de *Regiones - Inmortal*, con la diferencia que el Plan sólo crea el espacio *RegionsSpace* y la política de asociación utiliza el allocator *ResizeRegionsAllocation*. Como tarea adicional, se deben agregar los "*stubs*" en la clase principal *VM*, para crear y eliminar la primer región inmortal al iniciar y finalizar la máquina virtual.

Definición del espacio de regiones:

```
public static final RegionsSpace regionsSpace =
      new RegionsSpace("regions", DEFAULT_POLL_FREQUENCY, VMRequest.create());
```

5.2. Espacios y Políticas

Como se explicó en la subsección 3.3.2.2, una política de memoria está compuesta por dos componentes principales: el espacio de memoria, y un allocator que administra los objetos en memoria devolviendo una dirección virtual por cada *alloc*. En un espacio de memoria se define el modelo de administración de páginas (*PageResource)* y el comportamiento general del área de memoria que comprende. Algunas de sus responsabilidades son: definir la estructura del header de los objetos que alberga, la alcanzabilidad de los mismos, y el comportamiento del GC en las fases *prepare* y *release*.

5.2.1. Espacio RegionsSpace

Dado que Jikes RVM provee varias políticas a utilizar, en particular las ligadas a la utilización de espacios con diferentes GC, sólo hace falta crear una política de memoria basada en regiones. Se definirá el espacio de regiones *RegionsSpace* con las siguientes características:

- Debe utilizar *RegionsPageResource 8.2.1* como administrador de páginas. *RegionsPageResource* es el administrador de páginas de memoria diseñado especialmente para la utilización eficiente de regiones en forma de pila.
- No se debe ejecutar el proceso de recolección de basura. Sólo se llevará a cabo la primer fase (*prepare*), para poder realizar la traza del heap.
- Los objetos se definen alcanzables sólo si están en la clausura del trazado desde las raíces. El determinar qué objetos son alcanzables mejora la performance del GC en un ambiente con espacios mixtos, ya que los objetos pertenecientes a *RegionsSpace* deben tenerse en cuenta en la clausura de la totalidad del heap.

Para poder lograr este comportamiento, una solución es agregar un *flag* en el encabezado de cada objeto del espacio y marcarlo en cada traza. Como solamente se debe saber si el objeto es alcanzable, alcanza con disponer de este indicador booleano, y su valor se irá alternando en cada traza. De esta manera, con una sola pasada puede verificarse la alcanzabilidad del objeto, dado que no es necesario volverlo al estado inicial en la fase de *release*.

5.2.1.1. Implementación

Para desarrollarlo se deben sobreescribir los métodos *prepare* y *release*, que son invocados en las fases de los recolectores provistos por Jikes. En el método *prepare* de debe setear el entorno para una nueva llamada al GC, que en este caso sólo se debe invertir el valor del indicador. El método *release* se ejecuta luego de la pasada del GC, pero en él no se debe realizar ninguna operación ya que el espacio no sufre modificaciones.

Para entender en detalle el funcionamiento de marcado de objetos, en el algoritmo 5.1 se define la clase *RegionsSpace* con el detalle de sus métodos en forma simplificada.

Algorithm 5.1 RegionsSpace.

```
public final class RegionsSpace extends Space{

      static final Word GC_MARK_BIT_MASK = Word.one();
      private Word markState = Word.zero(); // valor inicial del flag
      public RegionsSpace(String name, int pageBudget, VMRequest vmRequest) {
           super(name, false, true, vmRequest);
           // se utiliza el administrador de páginas de regiones
           pr = new RegionsPageResource(pageBudget, this, META_DATA_PAGES_PER_REGION);
      }
      // Inicialización del encabezado del objeto. Seteo markState. En este caso sería lo mismo que hacer
      // VM.objectModel.writeAvailableBitsWord(object, markState) ya que la mascara es 1.
      public void initializeHeader(ObjectReference object) {
           Word oldValue = VM.objectModel.readAvailableBitsWord(object);
           Word newValue = oldValue.and(GC_MARK_BIT_MASK.not()).or(markState);
           VM.objectModel.writeAvailableBitsWord(object, newValue);
      }
      // Utilizado durante la traza. Devuelve true si los valores son distintos.
      // Tiene en cuenta la sincronización de escritura con otros traces. Solo cambia
      // el valor si el valor obtenido es el mismo que al momento de su seteo.
      @Inline
      private static boolean testAndMark(ObjectReference object, Word value) {
           Word oldValue;
           do {
                oldValue = VM.objectModel.prepareAvailableBits(object);
                Word markBit = oldValue.and(GC_MARK_BIT_MASK);
                if (markBit.EQ(value))
                   return false;
           } while (!VM.objectModel.attemptAvailableBits(object, oldValue, oldValue.xor(GC_MARK_BIT_MASK)));
           return true;
      }
      @Inline
      public ObjectReference traceObject(TransitiveClosure trace, ObjectReference object) {
           if (testAndMark(object, markState)) {
                // Si el flag es distinto enconlo al objeto como alcanzable
                trace.processNode(object);
           } return object;
      }
      //Preparación para una nueva recolección. Sólo se invierte el flag entre diferentes recolecciones.
      public void prepare() {
           markState = GC_MARK_BIT_MASK.minus(markState);
      }
      // no se debe hacer nada.
      public void release() { }
      @Inline
      //libera páginas a partir de la dirección de memoria start.
      public void release(Address start) {
           ((RegionsPageResource) pr).releaseTopRegionPages(start);
      }
      @Inline
      // Todos los objetos están vivos. Asi se evita la recolección de los
      //mismos en caso de que actúe un GC sobre el espacio.
      public boolean isLive(ObjectReference object) {
           return true;
      }
      // Testeo alcanzabilidad del objeto comparando el flag con markState
      public boolean isReachable(ObjectReference object) {

           return (VM.objectModel.readAvailableBitsWord(object).and(GC_MARK_BIT_MASK).EQ(markState));
      }

}
```

5.2.2. Políticas implementadas

Se desarrollaron dos políticas de memoria para contemplar la implementación de las diferentes máquinas virtuales. Ambas utilizan *RegionsSpace*, pero se diferencian en el allocator designado. Para la implementación de las VMs que utilizan espacio de regiones fijos se utilizó como allocator a *RegionsAllocator* (ver 8.2.3), mientras que en la implementación de regiones extensibles, *RegionsResizeAllocator (ver* 8.2.4).

Volviendo al esquema de memoria de MMTk, en la siguiente figura se presentan las políticas utilizadas por el Plan en el caso de utilizar la combinación de espacios/allocators *mark&sweep* y regiones de tamaño fijo.

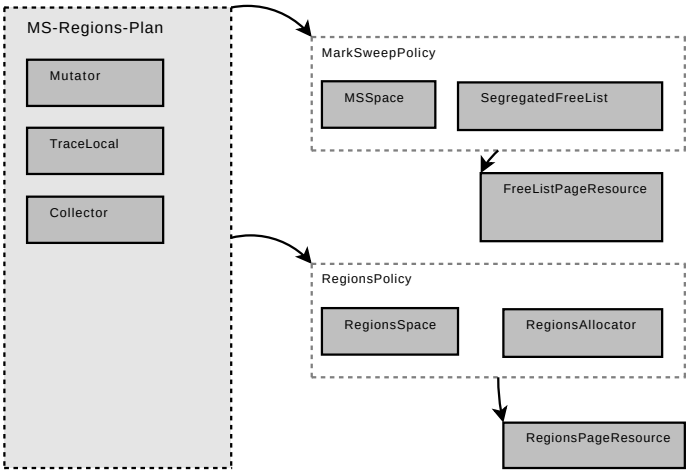

Figura 5.1: Diseño del plan de memoria utilizando *Regiones - Mark&Sweep*.

Capítulo 6

API Java

6.1. Alternativas para sintaxis de regiones

Brindar una sintaxis Java proporciona al programador una herramienta para poder definir regiones y determinar en cuál de ellas se creará cada objeto.

Si bien existen analizadores basados en análisis de escape para identificar las regiones y determinar los objetos que las componen según su ciclo de vida, por lo que se podría realizar el manejo de regiones implícitamente de manera transparente al programador incorporando dicha información en runtime (o precompilándola), sucede que en muchos casos es complejo obtener toda la información necesaria para que el proceso sea completamente automatizado. Un ejemplo son las recursiones, o en bucles donde no es posible determinar su finalización. Por eso surge la necesidad de crear una interfaz de alto nivel para proveer al programador de las herramientas necesarias para ajustar los valores necesarios para definir las regiones.

En este trabajo sólo se dispondrá al programador de las bibliotecas Java necesarias para poder hacer uso de regiones de memoria, dejando para un trabajo a futuro el desarrollo de los procesos para la incorporación de la información provista por herramientas externas.

Uno de los requerimientos es que la sintaxis se modifique lo menos posible, y que los programas Java que no utilizan una administración de memoria por regiones también puedan correr sobre la máquina virtual. Para ello se debe incorporar la información de regiones, en lo posible sin alterar el código fuente, o que ésta sea inocua a la ejecución del programa original.

Existen diversas posibilidades a la hora de definir la sintaxis. La idea es poder definir una región vinculándola a un método, y a su vez poder definir que objetos se deben crear en ella.

Identificación de creación de objetos en el código fuente La creación de un objeto en Java se identifica mediante el keyword *new*. Los mismos pueden ser arreglos, o instancias de una clase. Los tipos básicos como *int*, *char*, *float* no son objetos y no requieren asignación en el heap, mientras que un arreglo de alguno de ellos, sí.

Ej. de objetos a identificar:

```
new int[10]
new UnaClase()
new UnaClase[10]
new UnaClase[4][3] //(multiarreglo)
```

6.1.1. Tipos de sintaxis evaluadas

Dado que Jikes RVM no compila el código fuente, sino que toma como entrada los archivos *.class* que corresponden al bytecode de las clases del programa y luego genera el código máquina, no se podría optar por ejemplo por modificar la gramática del lenguaje incorporando nuevos *keywords* como *newRegion* para poder crear regiones de memoria, ya que no sería interpretada por un compilador Java convencional. En este caso sería necesario también implementar un compilador Java que traduzca a las instrucciones bytecode apropiadas.

Existen maneras más simples que surgen de utilizar la gramática Java (sin alterar) y aprovechar la *standard library* disponible.

Se analizaron dos prototipos de modelos de sintaxis para describir las operaciones con regiones:

1. Agregar al classpath las clases que proporcionen los métodos para poder crear, borrar y crear objetos en regiones (utilización de métodos estáticos).

2. Incorporar declaraciones de regiones y lugares de creación mediante anotaciones.

La primera opción permite definir en forma explícita la administración de regiones y objetos dentro de ellas, mientras que mediante el uso de anotaciones, se provee un mecanismo semiautomático basado en el tiempo de vida del método anotado. Aunque ninguna estas sintaxis asegura una correcta ejecución del programa (las regiones pueden estar mal definidas), la primera opción es muy propensa a errores ya que es posible crear una región en un método y eliminarla dentro de otro, y asignar objetos desde cualquier parte del programa mientras la región viva (y las referencias estén bien).

Con las anotaciones, al limitar el scope de la región a un método, sólo hace falta validar las dependencias entre los objetos, y para eso se obtendrá el soporte de los analizadores de escape. Adicionalmente, el código Java no es aterado, y también es posible aplicarlo directamente al bytecode (mediante procesos automáticos externos).

6.1.1.1. Administración explícita

Si tomamos la primera opción como modo a implementar, por ejemplo se podría definir la clase *Region* con los métodos *create*, *remove*, *new* y *newArray* que crean, eliminan, y crean objetos o arrays en regiones respectivamente (ver clase en el siguiente algoritmo).

Algorithm 6.1 API de regiones con Clases Java.

```
public final class Region{
    public static void create(int id){
        // unimplemented
    }
    public static void remove(int id){
        //unimplemented
    }
    public static Object new(int[] regionId, Class clase, Object[] parameters){
        //unimplemented
    }
    public static Object newArray(int[] regionId, Class clase, int size){
        //unimplemented
    }
}
```

Aunque la implementación de dichos métodos esté vacía, ya que no se dispone de una interfaz de la VM de bajo nivel para administrar memoria en este contexto, los métodos se podrían implementar directamente en el proceso de compilación identificándolos por la declaración de la clase, su nombre y el descriptor.

Notar que los métodos *new* y *newArray* toman un arreglo como primer parámetro para identificar las regiones donde podría crearse el objeto. Dado que el método que contiene la sentencia *new* puede ser invocado desde diferentes métodos (o con configuraciones de pila de llamadas diferentes), la región en donde se creará el objeto podría variar durante la ejecución. En el momento de la creación del objeto sólo una región de las pasadas por parámetro será válida: la última región creada que responda a alguno de los identificadores de región especificados.

Ejemplo: En el algoritmo 6.2 puede verse que *m2* es invocado primero desde *m0* y luego desde *m1*. La instancia de *Clase1* creada por *m2*, cuando es llamada desde *m1* será asociada a la región 2. Cuando se invoque desde *m0*, a la región 1.

Algorithm 6.2 API de regiones con Clases Java: ejemplo de uso.

```
public void m0(){
    Region.create(1);
    m1();
    Clase1 b = m2();
    Region.remove(1);
}
public void m1(){
    Region.create(2);
    Clase1 c1 = m2();
    ...
    Region.remove(2);
}
public Clase1 m2(){
    Object parameters = new Object[]{};
    return Region.new([1,2],Clase1.class,parameters);
}
```

Si bien esta notación serviría para poder crear regiones de memoria (asociadas o no a métodos), asignar objetos dentro de las mismas y eliminarlas, la sintaxis no permite evitar la generación

de errores durante la ejecución, ya que la eliminación de un región es explícita y depende del programador el buen uso de las instrucciones. Una solución podría ser eliminar el método *remove*, y dejar la responsabilidad de eliminarla a la *VM* (al terminar la ejecución del método). De esta manera el programador sólo debería crearla para asignarle un identificador.

Otro problema común en este esquema, es hacer referencia a identificadores de regiones que no son válidos al momento en que se crea el objeto aludido. No es posible detectar este comportamiento mediante un análisis estático del código fuente, ya que las regiones existentes al momento de crear el objeto, dependerán del flujo del programa. Por ejemplo puede haber una asignación a una región que ya ha sido eliminada o todavía no se haya creado. Este tipo de anomalías son sencillas de identificar en runtime lanzando una excepción, o si hay un plan de contingencia, como crear dicho objeto en el área de memoria inmortal, mostrar una advertencia.

Otra forma de evitar estos casos es implementar a las regiones como objetos Java, lo que permitiría tratarlas como cualquier objeto del lenguaje con todos los beneficios que ello implica: no se generarían errores en tiempo de ejecución por la especificación errónea de las regiones, se soportaría el pasaje por parámetro y retorno de regiones, la anidación de regiones (regiones que contienen regiones), manejo dinámico de relaciones entre regiones y asociación de objetos, etc). La contra de este modelo tan flexible es que no hace transparente el modelo de memoria al programador. Sería como proponer un nuevo lenguaje donde las regiones pasarían a tener un rol escencial, de hecho, el modo de programar sería bastante complejo y diferente a Java. Otra contra de esa sintaxis, es la generación de objetos adicionales para la creación de la región. Por ejemplo en el algoritmo 6.2, en el método *m2* puede verse la creación de un array de *Object* (que son los parámetros para el constructor de la instancia a crear). Este arreglo es una instancia siempre local a *m2*, pero no se la puede diferenciar de una instrucción Java convencional, por lo tanto también se necesita de un espacio de memoria inmortal o un pequeño heap para poder administrar y eliminar estos objetos temporales, o brindar los mecanismos de bajo nivel para poder diferenciarlos de un objeto Java convencional.

6.1.1.2. Administración mediante anotaciones

La otra alternativa analizada, que será la elegida para la implementación, es el uso de anotaciones, disponibles a partir de *Java 5.* Las anotaciones pueden ser utilizadas sobre elementos del lenguaje, como métodos, clases, campos, etc. A diferencia de las sintáxis explícitas planteadas anteriormente, las anotaciones definen regiones asociadas al elemento anotado. Por ejemplo, si se anota un método definiéndole una región, la misma "vivirá" durante su ejecución. Este hecho facilita el uso de las regiones, pero también quita flexibilidad. Por ejemplo, en los casos anteriores era posible crear una región dentro de una estructura de decisión "if", mientras que con esta sintaxis esto no es factible (al menos si la región está asociada a métodos).

Dado que las anotaciones en el código fuente pueden ser leídas de igual manera desde el bytecode (no se compila, permanece inmutable), esto nos permite:

- poder definirlas en el código fuente Java sin la necesidad de alterar el compilador de bytecode.

- utilizarlas tanto en el código Java como en el bytecode (en caso de instrumentar directamente).

Adicionalmente, se soluciona el problema de la creación de objetos temporales para el uso del administrador de regiones y no se altera el código fuente dentro del método.

Características de las anotaciones Existen diversas maneras de definir anotaciones dependiendo del *target*, que es el elemento del lenguaje al cuál se le aplica la anotación, y la política de retención, que define el alcance de la misma en las diferentes etapas de la ejecución del programa.

Target Pueden ser aplicadas a múltiples objetos del lenguaje como: *TYPE* (clase, interfaz, enum o una anotación), *FIELD* (objetos declarados a nivel de clase), *METHOD* (se aplica sólo a métodos), *PARAMETER* (parámetros del método en su definición), *CONSTRUCTOR* (constructor de una clase), *LOCAL_VARIABLE* (variables locales dentro de un método o bloque de código), *ANNOTATION_TYPE* (aplicada a otras anotaciones), *PACKAGE* (paquete).

Retención Las políticas de retención determinan hasta qué punto del proceso de compilación/ejecución será conservada la anotación. Hay tres alternativas: *SOURCE* (sólo permanecerán en el código fuente), *CLASS*(son accesibles desde el archivo .class -bytecode-), *RUNTIME* (son accesibles en tiempo de ejecución).

Como en el modelo de administración de regiones sólo es necesario poder ligar las regiones a métodos, se utilizarán anotaciones cuyo target sean *METHOD* y CONSTRUCTOR, y con una política de retención de tipo *RUNTIME* ya que será necesario acceder a las anotaciones en el momento de la ejecución.

Finalmente optando por el uso de anotaciones, se realizaron dos implementaciones diferentes de definiciones de regiones: las basadas en referencias a regiones, y las basadas en referencias a lugares de asignación.

Implementación	objetos temporales	propensa a errores en runtime	simplicidad	prolijidad	altera código fuente	facilidad para instrumentar
Nueva gramática	No	Sí	Sí	Sí	Sí	No
Como objetos Java	Sí (śolo objetos Región)	No	No	No	Sí	No
Métodos estáticos	Sí	Sí	No	No	Sí	No
Anotaciones	No	Sí	Sí	Sí	(sólo incorpora anotaciones)	Sí

Cuadro 6.1: Comparación de diferentes alternativas de sintaxis para el soporte de regiones.

6.2. Anotaciones basadas en referencias a regiones

Permiten definir regiones mediante un identificador proporcionado por el programador, y definir los lugares de creación relativos al método anotado, asociándolos a los identificadores de las regiones establecidas. La sintaxis está conformada por dos tipos de anotaciones:

- `java.lang.regions.Region`: Identifica las regiones a crear al inicio del método definiendo su identificador y su tamaño en bytes. Finalizada la ejecución del método se procederá a la eliminación de la región con todos los objetos albergados en ella.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(value={ElementType.METHOD,ElementType.CONSTRUCTOR})
public @interface Region {
     String name();
     int size();
}
```

  La definición de cada región está conformada por una etiqueta que la identifica y su respectivo tamaño en bytes, que puede ser obtenido mediante herramientas de estimación de tamaño de regiones. Ejemplo de uso:

```
@Region(name="my-region", size=1024)
public void method1(){...}
```

  Al ejecutarse *method1* se deberá crear la región cuyo identificador será *my-region* con un tamaño de 1024 bytes.

- `java.lang.regions.RegionAllocationSites`: Permite ligar lugares de creación a regiones, los cuales son identificados por medio del número de línea relativa al método a la cuál se aplica la anotación.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(value={ElementType.METHOD,ElementType.CONSTRUCTOR})
public @interface RegionAllocationSites {
     String[] allocationSites();
}
```

  Cada definición de *allocation site* está compuesta por la región a la que pertenece y su ubicación, que se identifica por el número de línea relativa al método y opcionalmente se puede indicar su posicionamiento en la misma (si hay más de un *new* por línea). En el caso de definir más de un lugar de creación por región, éstos pueden ser separados por coma. Ejemplo de uso:

```
@Region(regions={"my-region:1024"})
@RegionAllocationSites(allocationSites={"other-region:1,3","my-region:2"})
public Object[] myMethod(){
     Object o1 = new Object(); //line 1
     Object local = new Object(); //line 2
     Object[] values = new Object[10]; //line 3
     values[0]=o1;
     return values;
```

```
}
```

Al ejecutar *myMethod* se creará la región *my-region* y se supondrá que ya está creada con anterioridad *other-region*. La definición de la anotación *RegionAllocationSites* determina la ubicación de los objetos del método en las regiones *other-region* para los objetos creados en la línea 1 y 3 (*o1* y *values*), y en *my-region* para la asignación de la segunda línea (*local*). En el caso que se hallasen dos o más *new* en la misma línea, pueden ser distinguidos por su posición (base en 0).

Definición de posicionamiento de los lugares de creación:

```
{numero_linea}[.posicion][,{numero_linea}[.posicion]]*
```

Las llaves indican uso obligatorio mientras que los corchetes no, y el * indica repetición de 0 o más veces.

Ej.: si la quinta instrucción del método es `new A(new B( new C()));` y se quiere identificar el objeto `new C()`, la notación sería: *5.2*. Si se utiliza sólo el número de línea: *5* o *5.0*, se estará referenciando al primer *new*.

En el siguiente algoritmo puede verse el resultado de aplicar este tipo de anotaciones a un pequeño programa.

Algorithm 6.3 Ejemplo de anotaciones con referencias a regiones.

```
@Region(name="m0", size=232)
B[] m0(int mc) {

      m1(mc);
      B[] m2Arr = m2(2 * mc);
      return m2Arr;

}
@Region(name="m1", size=112)
@RegionAllocationSites(allocationSites={"m1:2"})
void m1(int k) {

      for (int i = 1; i <= k; i++) {
            A a = new A(); // línea 2
            B[] dummyArr = m2(i);
      }

}
@Region(name="m2", size=232)
@RegionAllocationSites(allocationSites={"m1:1,3","m0:1,3","m2:4"})
B[] m2(int n) {

      B[] arrB = new B[n]; // línea 1
      for (int j = 1; j <= n; j++) {
            arrB[j - 1] = new B(); // línea 3
            C c = new C(); // línea 4
            c.value = arrB[j - 1];
      }
      return arrB;

}
```

Si el programa a ejecutar comienza con la invocación del método *m0* con el parámetro entero 10, primero se creará la región "*m0*" con tamaño de 232 bytes. Luego "*m1*" (112 bytes) tras la ejecución de la primer línea de *m0*. En *m1* todas las instancias creadas en la línea 2 pertenecerán a

la región "*m1*" según la definición de la anotación. En la siguiente línea, *m2* es invocado quedando la siguiente configuración en la pila de llamadas a métodos: *m0(línea 1) -> m1(línea 3) -> m2.*

En *m2* las asignaciones de las líneas 1 y 3 hacen referencia a las regiones "*m0*" y "*m1*", pero como el método invocador más arriba en la pila de llamados es *m1*, los objetos se almacenarán en su región *("m1")*. El objeto local creado en la línea 4 será almacenado en la región *"m2",* ya que no escapa al método. Luego, cuando *m2* es invocado desde *m0,* la configuración de la pila será: *m0 (línea 2) -> m2.* Y en esta oportunidad los objetos creados en las líneas 1 y 3 se alojarán en *m0.*

6.3. Anotaciones basadas en referencias a lugares de creación

Permiten definir regiones asociadas a métodos, pero la diferencia con la precedente radica en que cada región define localmente su contenido referenciando los lugares de creación. Esta sintaxis está compuesta por una sola anotación:

- `java.lang.regions.RegionDefinition`: Identifica a la región asociada al método. Sólo se especifica el tamaño en bytes de la región y un array de lugares de creación. No hace falta definir el identificador de la región ya que no va a ser referenciada desde ningún lugar del programa.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(value={ElementType.METHOD,ElementType.CONSTRUCTOR})
public @interface RegionDefinition {
     int size();
     String[] allocations();
}
```

Cada lugar de creación está conformado por un *string* que identifica el método, número de línea, y de ser necesario, la posición de la asignación dentro de esta. La sintaxis para identificar al método está basada en el descriptor de método del bytecode.

Definición de lugar o lugares de creación:

```
[[paquete_clase:]nombre_metodo:]numero_linea[.posicion][,numero_linea[.posicion]*]}
```

El único componente obligatorio es el posicionamiento. Si no se indica la descripción de la clase (*paquete_clase*) se supondrá que el método (nombre_metodo) pertenece a la misma clase a la que pertenece el método anotado. Siguiendo el mismo razonamiento, si sólo se especifican los posicionamientos, los lugares de creación pertenerán al método anotado. Estas abreviaciones permiten simplificar las definiciones de los lugares de creación. Al igual que en la definición de posicionamiento de las anotaciones anteriores, se soporta identificar múltiples lugares de asignación en el mismo método, separando su posicionamiento por coma.

Ejemplo de lugares de creación válidos La siguiente sentencia identifica los lugares de creación definidos en las líneas 6 y 8 del método *String substring(int from, int to)* de la clase java.lang.String. La sintaxis del nombre del método es la utilizada por Java como descriptor del mismo.

```
"java.lang.String:substring(II)Ljava/lang/String;:6,8"
```

Si se quiere referenciar a un lugar de creación de la misma clase, puede utilizarse la siguiente sintaxis. En ella se identifican los lugares de creación definidos en las líneas 6 y 8 del método *String substring(int from, int to)* de la clase del método anotado. Si el método anotado pertenece a java.lang.String, la definición es similar a la anterior.

```
"substring(II)Ljava/lang/String;:6,8"
```

La siguiente y última, es utilizada para referenciar a los lugares de creación definidos en las líneas 6 y 8 del método anotado. Para que esta definición sea similar a las anteriores, el método en cuestión debería ser *String substring(int from, int to)* de la clase java.lang.String.

```
"6,8"
```

Si tomamos el primer ejemplo de la subsección anterior, para obtener los mismos resultados a la hora de la ejecución habría que definir 2 anotaciones. La anotación en *myMethod* crea la región denominada anteriormente como *my-region*, mientras que la de *otherMethod* crearía la región *other-region*. Cada una especifica qué objetos albergará.

```
@RegionDefinition(size=1024, allocation={"2"})
public Object[] myMethod(){

      Object o1 = new Object(); //line 1
      Object local = new Object(); //line 2
      Object[] values = new Object[10]; //line 3
      values[0]=o1;
      return values;

}
@RegionDefinition(size=1024, allocation={"2","myMethod()[Ljava/lang/Object;:1,3"})
public Object[] otherMethod(){

      Object[] array = this.myMethod();

      ....

}
```

En el algoritmo 6.4 se puede ver el resultado de aplicar este tipo de anotaciones al ejemplo presentado.

6.4. Ventajas y desventajas de los dos tipos de sintaxis

Las dos sintaxis tienen el mismo poder expresivo. La primera toma como unidad principal los lugares de creación del método, y a partir de allí se define a qué regiones pertenecen, mientras que en la segunda sintaxis la unidad principal es la región, y en ella se definen los diferentes lugares de creación.

La utilización de la primer alternativa facilita ubicar las asignaciones de un método en diferentes regiones y permite detectar objetos huérfanos de regiones tan sólo verificando el método anotado. Otra ventaja es que la forma de identificar a una región es mucho más intuitiva que identificar un

Algorithm 6.4 Ejemplo de anotaciones con referencias a lugares de creación.

```
@RegionDefinition(size=232,allocations={"m2(I)[Lismm/B;:1,3"})
B[] m0(int mc) {

      m1(mc);
      B[] m2Arr = m2(2 * mc);
      return m2Arr;

}
@RegionDefinition(size=112,allocations={"2","m2(I)[Lismm/B;:1,3"})
void m1(int k) {

      for (int i = 1; i <= k; i++) {
           A a = new A(); // línea 2
           B[] dummyArr = m2(i);
      }

}
@RegionDefinition(size=232,allocations={"m2(I)[Lismm/B;:4"})
B[] m2(int n) {

      B[] arrB = new B[n]; // línea 1
      for (int j = 1; j <= n; j++) {
           arrB[j - 1] = new B(); // línea 3
           C c = new C(); // línea 4
           c.value = arrB[j - 1];
      }
      return arrB;

}
```

lugar de creación mediante la descripción de la clase/método. La principal desventaja es que no se posee un conocimiento global de los objetos que pertenecen a una región, y lo más importante es que las bibliotecas de uso común, como por ejemplo *String*, deberían estar anotadas para todas las implementaciones que la utilicen asociando las creaciones de objetos internas a las nuevas regiones. Esto es prácticamente imposible, sobre todo si las bibliotecas están compiladas en un .jar, que es el caso más común.

En la segunda alternativa, al proponer un modelo centralizado en la región, se tiene conocimiento de todos los objetos que se albergarán y a qué método pertenecen. Sobre este esquema, es posible referenciar objetos de bibliotecas comunes con sólo conocer en qué lugar se crean los objetos (clase/método/número de línea). Ej.: Si se utiliza el método *substring* de la clase *String* y en dicho método se crea una nueva instancia de *String* en la línea 6, se puede incluir esta asignación para que forme parte de la región del método que se está definiendo, sin alterar la clase *String*.

En el algoritmo 6.5 se aprecia la diferencia entre los dos modos de anotaciones en el caso de asociar el objeto creado por *substring*.

Algorithm 6.5 Comparación entre los tipos de anotaciones.

Anotaciones basadas en referencias a regiones:

```
@Region(size=1024,name={"my-region"})
public void myMethod(){

      String newString = "un string".substring(2,4);
      ...

}
```

En este caso habría que anotar el método substring para agregar la región my-region. Además de todas las regiones en donde se utilice este método.

```
class String{

      ....
      @RegionAllocationSites(allocationSites={"my-region:1",.....})
      public String substring(String from, String to){
      }
      ....

}
```

Anotaciones basadas en referencias a lugares de creación: utilizando esta notación no es necesario modificar la clase String, pero es necesario conocer el posicionamiento de la instrucción 'new' dentro de String. Dado que el objetivo a futuro no es instrumentar manualmente el código fuente, sino anotarlo mediante un proceso automático, esta información puede generarse fácilmente a partir de las herramientas de análisis de escape.

```
@RegionDefinition(size=1024,allocations={"java.lang.String:substring(II)Ljava/lang/String;:6"})
public void myMethod(){

      ...
      String newString = "un string".substring(2,4);
      ...

}
```

Aunque es posible utilizar ambas anotaciones en forma simultánea, es recomendable utilizar sólo una de ellas para hacer más legible el código fuente y evitar la creación de múltiples regiones para un determinado método (no está restringido ya que el modelo de bajo nivel lo soporta).

6.5. Anotaciones de soporte para facilitar el desarrollo

Dado que a veces es muy complejo identificar los lugares de creación, en particular el número de línea relativo al método y la nomenclatura que propone el bytecode, se creó la anotación java.lang.regions.PrintRegionInfo con la intención de proporcionar información sobre la creación de objetos y estadísticas dinámicas sobre el uso de las regiones durante la ejecución.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(value={ElementType.METHOD,ElementType.CONSTRUCTOR})
public @interface PrintRegionInfo {

      boolean news() default true;
      boolean statistics() default false;

}
```

Al anotar un método con *java.lang.regions.PrintRegionInfo*, se deben especificar dos propiedades booleanas: news y statistics. La primera identifica la creación de objetos en el método y si tiene una región definida previamente compilada. Statistics muestra por pantalla estadísticas sobre la región al finalizar la ejecución del método anotado. En la sección 9 se explicará con detalle qué valores se almacenan y su significado.

Propiedad *news* Si news está habilitada, al compilarse el método se listarán por *standard output* todas las asignaciones presentes implícitas o explícitas. Ej. m1:

```
@PrintRegionInfo(news=true,statistics=false)
public void m1(){

      int[] intArray = new int[4];
      String hola = new String("hola ");
      String mundo = new String("mundo");
      String holamundo = hola + mundo;
      A miObjeto = new A(new B());

}
```

El método m1 contiene 6 asignaciones, 5 explícitas (precedidas por el keyword *new*) y 1 implícita resultado de convertir *hola + mundo;* en la sentencia `new StringBuilder(“hola”).append(“mundo”);` durante la compilación del archivo *bytecode class*.

Al ejecutarlo se listarán en pantalla las siguientes líneas:

```
resolved_newarray: regions.EjemplosAPI:m1()V absolute line: 9, relative line: 1
resolved_new: regions.EjemplosAPI:m1()V absolute line: 10, relative line: 2
resolved_new: regions.EjemplosAPI:m1()V absolute line: 11, relative line: 3
resolved_new: regions.EjemplosAPI:m1()V absolute line: 12, relative line: 4
unresolved_new: regions.EjemplosAPI:m1()V absolute line: 13, relative lin: 5
unresolved_new: regions.EjemplosAPI:m1()V absolute line: 13, relative line: 5.1
```

Con esta información se puede obtener la definición del método y el número de línea relativo. Por ejemplo se desea crear el objeto *miObjeto* en una región (conformado por las asignaciones 5 y 5.1), se podría definir la anotación `@RegionDefinition(size=100,allocations={"m1()V:5,5.1"})` en el método que corresponda, y al ejecutar nuevamente el método se obtendría el siguiente resultado por pantalla:

```
resolved_newarray: regions.EjemplosAPI:m1()V absolute line: 14, relative line: 1
resolved_new: regions.EjemplosAPI:m1()V absolute line: 15, relative line: 2
resolved_new: regions.EjemplosAPI:m1()V absolute line: 16, relative line: 3
resolved_new: regions.EjemplosAPI:m1()V absolute line: 17, relative line: 4
unresolved_new: regions.EjemplosAPI:m1()V absolute line: 18, relative line: 5
Found!
unresolved_new: regions.EjemplosAPI:m1()V absolute line: 18, relative line: 5.1
Found!
```

En esta sección se ha descripto la sintaxis Java proporcionada al programador para interactuar directamente con el administrador de memoria, pero no se ha explicado cómo es interpretada por la máquina virtual. En la siguiente sección se detallará el proceso de compilación del bytecode, que utilizará la información provista por las anotaciones para generar las rutinas de administración de memoria correspondientes.

Capítulo 7

Compilación

El proceso de compilación (Figura 7.1) consta de las siguientes fases: la compilación a bytecode (archivo .class), que se deberá realizar por un compilador Java externo a Jikes RVM, y la compilación del bytecode a código máquina, que se realizará en Jikes RVM y es en donde se tendrán que interpretar las anotaciones de regiones. No se debe prestar particular atención a la compilación del código Java a bytecode ya que las anotaciones no son alteradas (dado que su target es *RUNTIME*) por lo que puede utilizarse cualquier compilador Java. En esta sección se detallará el proceso de compilación del bytecode a código máquina para soportar ambos tipos de anotaciones, lo que involucra el análisis y almacenamiento de la información que cada método provee, y el uso de ésta para la generación del código máquina apropiado.

7.1. Compilación base de un método

El proceso de compilacíon en Jikes tiene como centro el método y consiste en interpretar cada instrucción del bytecode y generar el código máquina correspondiente. Finalizada la compilación se graba el código generado en memoria para su futura ejecución en runtime. **La principal tarea a realizar es añadir la interpretación de la nueva sintaxis Java, en este caso las anotaciones propuestas, y almacenar dicha información en memoria para su utilización en tiempo de ejecución.**

Se analizaron dos alternativas para llevar a cabo este proceso. La primera consiste en almacenar la información de regiones (regiones por método y lugares de creación) en un área especial del heap, que podrá ser leída por los procesos en runtime, en particular por el *Mutator* de regiones. Recorriendo esa tabla se podrá determinar en qué región se debe crear un objeto cuando se invoca el método asignador de memoria *alloc*, o cuándo se debe crear una región. En este caso tarea del compilador sólo sería generar el código máquina para incorporar la información interpretada (proveniente de las anotaciones) a dicha tabla. La ventaja de utilizar esta técnica es que no se deben modificar las instrucciones máquina correspondientes los diferentes tipos de "new" del bytecode, sino sólo incorporar como dato adicional el posicionamiento del objeto a crear (el número de línea y el orden para la misma). Luego, en tiempo de ejecución, por cada *alloc* se deberá recorrer la

tabla para verificar la existencia de regiones asociadas al lugar de creación, y determinar cuál de ellas es válida para la configuración actual de la pila de regiones.

La segunda opción consiste en mantener la información de regiones en la memoria del compilador (no accesible por objetos de runtime) y modificar cada instrucción "new" bytecode del método a compilar para incorporar las regiones posibles, de modo que ya estén predefinidas al momento de la ejecución. La diferencia con la técnica anterior radica en compilar las regiones posibles en vez de analizarlas en la tabla para cada *alloc*. Es posible fijar las regiones posibles para un lugar de creación porque los métodos no cambian durante toda la ejecución (a menos que se recargue una clase, que el método es recompilado). La ventaja de este modelo consiste en la facilidad para determinar en runtime en qué región se debe albergar el objeto sólo con acceder a la pila de regiones, lo que provee mayor performance en el allocator. La desventaja es que en la sintaxis de anotaciones orientadas a referencias a lugares de creación, se puede dar el caso en donde haya lugares de creación ya compilados que pueden ser referenciados desde métodos/regiones todavía no compiladas. Al compilar estos métodos, habrá que volver a instrumentar el "new" correspondiente al lugar de creación del método previamente compilado, agregándole la nueva región como posible. Es por ello que un mismo método puede ser recompilado reiteradas veces hasta incorporar todas las posibles regiones para un lugar de creacíon. A pesar del *overhead* de compilación, esto se efectuará sólo hasta que todos los métodos que todavía no estén compilados, y mantienen anotaciones de regiones que referencien a algún lugar de creación ya compilado, también sean compilados.

Se optó por implementar la segunda opción priorizando la performance del allocator. Además, si se implementara la tabla de regiones en el heap, el desarrollo sería mucho más complejo debido la ausencia de estructuras auxiliares provistas por Jikes RVM.

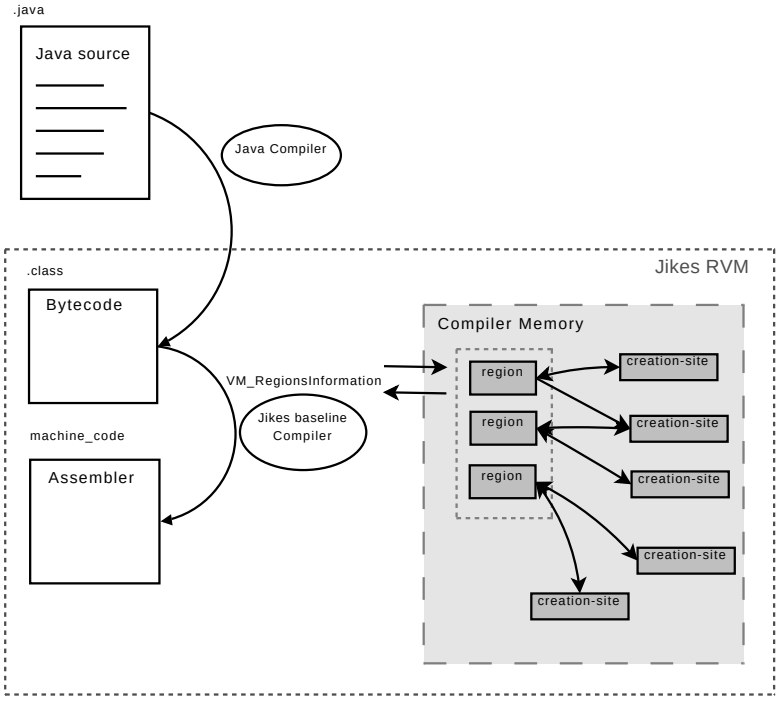

Figura 7.1: Proceso de compilación.

7.2. Definiciones

Sea m el método de un programa a compilar.

Definiremos como r_m a la región asociada al método m, $cs_{m,p}$ al lugar de creación (*creation site*) ubicado en la posición p del método m, y a la relación de asociación entre una región y un lugar de creación.

$$CS = \{cs_{m,p} / m \text{ es el método donde se encuentra } cs \text{ y } p \text{ su posicionamiento dentro el mismo}\}$$

CS es el conjunto de todos los lugares de creación del programa.

$$R = \{r_m / m \text{ es un método del programa}\}$$

R es el conjunto de todas las regiones del programa.

$$CS_{r_m} = \{cs_{m',p} / cs_{m',p} \sim r_m \text{ tal que } cs_{m',p} \in CS \wedge \ r_m \in R\}$$

CS_{r_m} representa el conjunto de todos los lugares de creación asociados a r_m. Un mismo lugar de creación puede pertenecer a diferentes $CS_{r_{m'}}$.

$$r_m =< id_r, size_r, id_m, name_r, CS_{r_m} >$$

Una región r_m está conformada por una tupla que incluye un identificador interno para el uso a bajo nivel (que será otorgado por el compilador), el tamaño en bytes de la región, el id interno del método asociado, un nombre de región (en caso de utilizar anotaciones con referencias a regiones se utiliza el label especificado), y el conjunto de lugares de creación asociados a la región.

$$cs_{m,p} =< m_{Id}, Name_m, Descriptor_m, p, R_{cs_{m,p}} >$$

Un lugar de creación $cs_{m,p}$ está compuesto por una tupla que contiene el identificador del método (en caso que se haya compilado con anterioridad), o un nombre y descriptor (que responden a la forma explicada en la sección anterior), la posición conformada por el número de línea relativo al método en donde se encuentra y la posición para especificar el orden en caso de que haya más de una asignación en la misma línea de código, y un conjunto de regiones asociadas a las que puede pertenecer en algún momento de la ejecución.

$$R_{cs_{m,p}} = \{r_{m'}/r_{m'} \sim cs_{m,p} \text{ tal que } r_{m'} \in R \ \wedge \ cs_{m,p} \in CS\}$$

$R_{cs_{m,p}}$representa el conjunto de regiones asociadas a $cs_{m,p}$.

Al compilar un método la idea es almacenar los CS_{r_m} en una estructura estática para que puedan ser accedidos y utilizados durante la compilación para instrumentar el código máquina. Si por ejemplo ya se ha compilado previamente el método m, y la información actual compilada incluye a $CS_{r_m} = \{...., cs_{m',p}, ...\}$, luego al compilar m' se detectará que ya hay información para la creación del objeto de la de posición p, y que éste puede ser creado en las regiones especificadas por $cs_{m',p}$, es decir en alguna de las regiones que pertenecen a $R_{cs_{m',p}}$.

7.3. Entorno de compilación Jikes

Jikes RVM soporta dos tipos de compilación: la compilación base (baseline compiler) en donde el bytecode del método a compilar es transformado a código máquina según la arquitectura utilizada, y la optimizada, que funciona en conjunto con el sistema adaptativo, y en base a estadísticas se toma la decisión de recompilar métodos con diferentes niveles de optimización, obteniendo código intermedio con optimizaciones en el uso de registros y operaciones.

En este trabajo se utilizará el compilador base para simplificar y evitar posibles modificaciones del código máquina generado, que podrían interferir en la performance de la VM, no pudiendo asilar el comportamiento del nuevo administrador de memoria.

Antes de ejecutar un método, éste debe estar previamente compilado. El proceso de compilación tiene como objetivo traducir el bytecode a código máquina, para ello Jikes RVM dispone de las clases Java necesarias que representan los objetos que van a intervenir durante el proceso:

- `VM_Method`: representación interna de un método del programa. Contiene la información propia de un método, como la clase a la que pertenece, las anotaciones, y en particular el bytecode asociado.

- `VM_CompiledMethod`: encapsulador del método que agrega información de la compilación.
- `ASM`: abstracción de lenguaje assembler con las instrucciones correspondientes implementadas como métodos de una clase, cuya finalidad reside en que el código generado responde a la arquitectura subyacente.
- `VM_Compiler` (*Compiler Framework*): es el encargado de generar el código a ejecutar, llevando a cabo la transformación del bytecode del método a código máquina utilizando la interfaz *ASM*. Finalmente el código máquina es encodeado y mapeado a memoria.

En tiempo de ejecución Jikes RVM contiene una tabla de contenido en memoria llamada *jTOC,* donde se almacenan estructuras de uso global como los literales, enteros, constantes, variables y métodos estáticos, y los punteros a los bloques de información para cada clase Java (*Type Information Block -TIB-*).

La compilación tiene como eje el "método", que está representado por la clase *VM_Method*. Cuando éste es compilado se obtiene una instancia de *VM_CompiledMethod* que contendrá el *VM_Method* original y se incorporará información obtenida durante el proceso de compilación.

El proceso de compilación se realiza por demanda a medida que los métodos son invocados por medio de los bytecodes *invokespecial,* para métodos estáticos y constructores, e *invokevirtual* para los métodos instancia. Si el método todavía no fue ejecutado con anterioridad es compilado y el resultado, que es un arreglo de instrucciones en código máquina, será almacenado en *jTOC* o *TIB* según el tipo de método.

La compilación aporta información que será utilizada para la instrumentación de la creación de objetos del mismo u otros métodos. Para poder almacenarla se crearán estructuras de datos estáticas en un espacio de memoria inmortal, que podrán ser accedidas durante el proceso de compilación de cualquier método. Se deberán guardar todas las regiones y lugares de creación, y se dispondrá de utilidades que provean acceso rápido a la información para un método determinado.

Las funciones en runtime que deberán realizar las acciones de administración de memoria, como crear una región, eliminarla, o crear un objeto en alguna de ellas, fueron referenciadas de manera estática en la clase *VM_Entrypoints*, por lo que son accesibles desdes *jTOC*. Durante la compilación, estos métodos internos de la VM serán incorporados al código máquina agregando las llamadas assembler correspondientes que referencien al *jTOC* por su *offset* (puntero al método de la API Runtime). En el capítulo 8 se detallará la interfaz provista.

7.4. Compilación de anotaciones

En el proceso de compilación de un método en particular se podrá tener acceso a las estructuras estáticas que almacenan los r_m y los CS_{r_m} para todo todo m, y se podrá ingresar la nueva información compilada para el método actual en compilación.

Para lograr el correcto funcionamiento de la aplicación según la especificación de regiones, al ejecutarse un método m (previa compilación del mismo), éste deberá tener disponible (precompilada) toda la información que refiere a la creación de regiones asociadas a él (r_m), y poseer el

conocimiento de las regiones asociadas a los objetos que se crearán durante su entorno actual de ejecución (subconjunto de $R_{cs_{m,p}}$ que comprende a los $r_{m'}$ tal que m' ya ha sido compilado). Esto sucede porque la información es incorporada por demanda, lo que implica que la información restante no es de interés ya que los métodos que la proveen no han sido incorporados al programa, y en consecuencia no interfieren en el estado actual de la ejecución. Este caso puede darse comunmente mediante la utilización de anotaciones basadas en referencias a lugares de creación. Un ejemplo básico es la referenciación de lugares de creación en bibliotecas de uso común como *String*, que se compila antes que las bibliotecas del programa a ejecutar, y hasta ese momento los lugares de creación no poseen regiones asociadas.

En cambio, con anotaciones basadas en referencias a regiones, el proceso es más simple ya que al compilar un método se podrá decidir en ese momento si se debe crear o no una región (al ejecutar dicho método), y se podrá tener acceso a la información para cada uno de sus lugares de creación y saber en qué región se almacenarán. Lo que puede suceder, es que haya regiones que no estén registradas todavía en el compilador, lo que indica que los métodos dueños todavía no se han compilado y ejecutado, y por lo tanto tampoco son regiones válidas hasta el momento.

7.4.1. Implementación

7.4.1.1. Definición del modelo de clases

Se definieron las siguientes clases que representan una región y un lugar de creación, y proveen los métodos de acceso para obtener o incorporar datos.

- `VM_RegionData`: representa una región (r_m) en particular. En el algoritmo 7.1 se distinguen los atributos protegidos, que definen la información descripta en la tupla de r_m, y luego se presentan los métodos para crear los $cs_{m,p}$ (que definirán una relación bidireccional), y los modificadores de propiedades.

Algorithm 7.1 VM_RegionData. Representa una región.

```
package org.jikesrvm.compilers.common.regions;
public class VM_RegionData {

      protected String name;
      protected int size;
      protected int owner;
      protected int identifier;
      protected VM_Set<VM_RegionAllocationSite> allocationSites = new VM_Set<VM_RegionAllocationSite>();
      public VM_RegionAllocationSite addAllocationSite(int methodId, int lineNumber);
      public VM_RegionAllocationSite addAllocationSite(int methodId, int lineNumber,
                                                       int positionInLineNumber);
      public VM_RegionAllocationSite addAllocationSite(String methodName,
                                          String methodDescriptor, int lineNumber);
      public VM_RegionAllocationSite addAllocationSite(String methodName,
                  String methodDescriptor, int lineNumber, int positionInLineNumber);
      public void addAllocationSite(VM_RegionAllocationSite allocationSite);
      public VM_HashSet<VM_RegionAllocationSite> getAllocationSites();
      public String getName();
      public void setName(String name);
      public int getSize();
      public void setSize(int size);
      public int getOwner();
      public void setOwner(int owner);
      public int getIdentifier();
      public void setIdentifier(int identifier);

}
```

- `VM_RegionAllocationSite`: representa un lugar de creación ($cs_{m,p}$). Siempre está ligado a un *VM_ RegionData*. En el algoritmo 7.2 pueden verse los atributos que define la tupla $cs_{m,p}$ y el resto de las funciones con los modificadores de dichas propiedades.

Algorithm 7.2 VM_RegionAllocationSite. Representa un lugar de creación.

```
package org.jikesrvm.compilers.common.regions;
public class VM_RegionAllocationSite {
     protected int methodId=-1;
     protected String methodName;
     protected String methodDescriptor;
     protected int lineNumber;
     protected int positionInLineNumber = 0;
     protected VM_HashSet<VM_RegionData> regions=new VM_HashSet<VM_RegionData>();
     public VM_RegionAllocationSite(int methodId, int lineNumber) {
          this.methodId = methodId;
          this.lineNumber = lineNumber;
     }
     public VM_RegionAllocationSite(int methodId, int lineNumber,
                                              int positionInLineNumber) {
          this.methodId = methodId;
          this.lineNumber = lineNumber;
          this.positionInLineNumber = positionInLineNumber;
     }
     public VM_RegionAllocationSite(String methodName, String methodDescriptor,
                                              int lineNumber) {
          this.methodName = methodName;
          this.methodDescriptor = methodDescriptor;
          this.lineNumber = lineNumber;
     }
     public VM_RegionAllocationSite(String methodName, String methodDescriptor,
                                  int lineNumber, int positionInLineNumber) {
          this.methodName = methodName;
          this.methodDescriptor = methodDescriptor;
          this.lineNumber = lineNumber;
          this.positionInLineNumber = positionInLineNumber;
     }
     public int getMethodId();
     public int getLineNumber();
     public void setLineNumber(int lineNumber);
     public int getPositionInLineNumber();
     public void setPositionInLineNumber(int positionInLineNumber);
     public VM_HashSet<VM_RegionData> getRegions();
     public void setRegions(VM_LinkedList<VM_RegionData> regions);
     public void setMethodId(int methodId);
     public String getMethodDescriptor();
     public void setMethodDescriptor(String methodDescriptor);
     public String getMethodName();
     public void setMethodName(String methodName);
}
```

- `VM_RegionsInformation`: es el punto de entrada del compilador para poder recopilar la información de regiones que poseen las anotaciones del método actual en compilación. Almacena estáticamente los $cs_{m,p}$ y posee las funciones para obtener el r_m del método (para saber si se debe crear una región), y las regiones pertenecientes a $R_{cs_{m,p}}$(para instrumentar el *new*). En el algoritmo 7.3 se presenta una versión acotada de la clase con las estructuras de almacenamiento y las funciones principales.

Algorithm 7.3 VM_RegionInformation. Mantiene la información de las anotaciones de todos los métodos compilados.

```
package org.jikesrvm.compilers.common.regions;
public class VM_RegionInformation {

      protected static VM_HashMap<String, VM_RegionData> regionsPerName =
                                new VM_HashMap<String, VM_RegionData>();
      protected static VM_HashMap<Integer, VM_RegionData> regionsPerMethodId =
                                new VM_HashMap<Integer, VM_RegionData>();
      protected static VM_HashMap<Integer, VM_HashSet<VM_RegionAllocationSite>>
                allocationSitesPerMethodId = new VM_HashMap<Integer, VM_HashSet<VM_RegionAllocationSite>>();
      protected static VM_HashMap<String, VM_HashSet<VM_RegionAllocationSite>>
                allocationSitesPerMethodString = new VM_HashMap<String, VM_HashSet<VM_RegionAllocationSite>>();
      protected static int identifier = 0;
      public static VM_RegionData getRegionData(VM_Method method);
      public static VM_HashSet<VM_RegionData> getAvailableRegionsData(VM_Method method,
                                          int lineNumber, int positionInLineNumber);

}
```

7.4.1.2. Estructuras de datos:

- `regionsPerName`: mapa que asocia los *VM_RegionData* identificados por nombre. Se utiliza para almacenar las regiones utilizadas con las notaciones basadas en referencias a regiones.
- `regionsPerMethodId`: mapa que asocia el identificador del método compilado con su *VM_RegionData*. Lo utilizan ambas anotaciones.
- `allocationSitesPerMethodId`: mapa que asocia el id del método con el conjunto de sus lugares de creación. Se utiliza con anotaciones basadas en referencias a regiones.
- `allocationSitesPerMethodString`: mapa que asocia el nombre del método conformado por el nombre de la clase, nombre del método y descriptor, con el conjunto de sus lugares de creación. Es utilizado por las anotaciones basadas en referencias a lugares de creación.

7.4.1.3. Funciones:

- `getRegionData`: toma un *VM_Method* por parámetro y analiza sus anotaciones. En el caso en que el método posea una anotación de tipo *java.lang.regions.@Region* se creará una instancia de *VM_RegionData* que será referenciada desde *regionsPerName* (es la manera de referenciarla desde otros métodos que utilizan anotaciones con referencias a regiones), y desde *regionsPerMethodId,* para ser identificada de ahora en más desde el entorno de compilación sin tener que levantar las anotaciones.
  Si en cambio la anotación es *java.lang.regions@RegionDefinition*, en este paso se deben crear, además de *VM_RegionData*, todas los *VM_RegionAllocationSite* que pertenecen a dicha región. Ésta será referenciada desde *regionsPerMethodId* porque en la anotación no se define un nombre identificatorio. Al agregar los lugares de creación se pueden dar dos casos: si el método al cual pertenece el lugar de creación todavía no ha sido compilado, se ingresa el lugar de creación a *allocationSitesPerMethodString*. Si fue previamente compilado, además de registrarlo en *allocationSitesPerMethodString* se lo vuelve a recompilar para modificar los $R_{cs_{m',p}}$ registrando la nueva región. Como pueden generarse ciclos al tener que compilar

métodos que posean información de regiones mutuamente referenciadas por sus lugares de creación, si el método ya había sido previamente compilado no se realiza la verificación de recompilación para los métodos que referencian sus lugares de creación (ver algoritmo 7.4). En la creación de un *VM_RegionData* se genera un id único para la región, independientemente del tipo anotación utilizada. Este id va a servir para identificar a la región en la administración de bajo nivel (allocators).

- `getAvailableRegionsData`: recibe un *VM_Method*, el número de línea en el código fuente relativo al método, y su posicionamiento en la línea. Este método devuelve el conjunto de *VM_RegionData* que contienen como lugar de creación el *VM_RegionAllocationSite* del método especificado por parámetro, en la línea *lineNumber* y posición *positionInLineNumber*. Este resultado se corresponde con la definición $R_{cs_{vmmethod,<lineNumber,position>}}$, que son las posibles regiones donde puede albergarse el objeto.

Algorithm 7.4 Compilación anidada.

```
@RegionDefinition(allocationSites={"m0:1"})
public void main(int n){

      m0(n);

}
@RegionDefinition(allocationSites={"m1:1"})
public A m0(int n){

      A a  = new A();
      if(n>0){
           B b = m1(n-1);

           ...
      }
      return a;

}
@RegionDefinition(allocationSites={"m0:1"})
public B m1(int n){

      new B();
      if(n>0){
           A a = m0(n-1);

           ...
      }
      return b;

}
```

En este tipo de algoritmos se presenta la situación en donde los lugares de creación de diferentes métodos se referencian mutuamente. Al compilar por primera vez *m0* se creará r_{m0} y se creará el lugar de creación $cs_{m1,<1,0>}$ perteneciente a la región r_{m0}. El método *m1* recién es compilado al tratar de ejecutarlo mediante el bytecode *invokevirtual* desde *m0*. Al intentar crear $cs_{m0,<1,0>}$, como *m0* ya está compilado sin tener en cuenta este lugar de creación, se procede a recompilarlo. Al recompilar r_{m0}, se presenta el mismo caso, pero no se debe volver a compilar *m1* ya que no se agrega información adicional, y además se entraría en un *deadlock*. Para evitar este comportamiento sólo se verificará la recompilación si el método actual en compilación no fue previamente compilado.

7.5. Compilación de bytecode

La generación del código de un método está definida por las siguientes fases:

- Generación del prólogo: el prólogo es el código máquina que se debe ejecutar antes de lanzar el método. Es el encargado de guardar el contexto actual: almacenar el puntero a la próxima instrucción del método llamador, guardar los registros de propósito general, guardar y actualizar el puntero al *stack frame*, etc. También se realizan las verificaciones de overflow para expandir el tamaño del *thread stack* si es necesario.

- Compilación del bytecode: es un proceso de iteración sobre cada instrucción del bytecode donde se realiza la generación del código máquina. Cada instrucción del bytecode se debe mapear al código máquina correspondiente mediante la utilización de *ASM*, registros, y haciendo uso de la información almacenada en el *stack frame* y el *jTOC*.

- Generación del epílogo: es código máquina que se ejecuta luego de haber finalizado el método. Se restaura el puntero al stack frame del método llamador, los registros de uso general, y se realiza un branch a la dirección de retorno del método llamador.

De esta forma, al ejecutar un método, primero se ejecutará el prólogo, luego el código propio del método, y finalmente el epílogo.

Siguiendo el modelo de regiones propuesto, al iniciarse la ejecución de un método que esté vinculado a una región, ésta debería crearse antes de ejecutar alguna instrucción *new* que la referencie. Una manera de asegurarlo, es crear la región luego de ejecutar el prólogo. De la misma manera, como al finalizar la ejecución del método la región asociada debería eliminarse, se puede ejecutar la rutina de eliminación de región antes de generar el epílogo.

Faltarían identificar las instrucciones *new* dentro del bytecode para instrumentarlas incorporando las regiones asociadas a cada una de ellas. Este proceso se introduce en la etapa de mapeo de instrucciones.

El proceso de compilación quedaría de la siguiente manera:

- Generación del prologo.
- **Verificación de región de memoria asignada.**
- **Compilación del bytecode (mapeo de instrucciones).**
- **Verificación de eliminación de región de memoria asignada.**
- Generación del epilogo.

7.5.1. Implementación

7.5.1.1. Verificación de creación de región

Se agregó al proceso de compilación una fase más que verifica la existencia de alguna región asociada al método. Para ello se creó el evento `emit_verify_region_start` que es invocado luego de ejecutarse el prólogo.

En esta etapa se debe utilizar *VM_RegionInformation.getRegionData(method)* que devolverá una instancia de *VM_RegionData* (en el caso que el método tenga una región asignada), y

además incorporará al compilador toda la información que aportan las anotaciones. En el caso que exista dicha región, se generará el llamado al método encargado de crearla en runtime: `VM_Entrypoints.createNewRegionMethod`.

Algorithm 7.5 Verificación de creación de región.

```
protected final void emit_verify_region_start(){
    VM_RegionData regionData=VM_RegionInformation.getRegionData(method);
    if(regionData!=null){
        asm.emitPUSH_Imm(regionData.getOwner());
        asm.emitPUSH_Imm(regionData.getIdentifier());
        asm.emitPUSH_Imm(regionData.getSize());
        // 3 words pasados por parámetro
        genParameterRegisterLoad(3);
        //JTOC = EDI
        asm.emitCALL_RegDisp(JTOC, VM_Entrypoints.createNewRegionMethod.getOffset());
    }
}
```

7.5.1.2. Verificación de eliminación de región

En esta fase, al igual que en la de verificación de creación de región, se verifica la existencia de región asociada para ser eliminada antes ejecutar el epílogo del método. Se creó el evento `emit_verify_region_end` que obtiene la región asociada también mediante el método *VM_RegionInformation.getRegionData(method)*. Como el método ya tiene compilada la información de regiones, esta vez se devuelve la información almacenada en las estructuras estáticas en vez de parsear nuevamente las anotaciones.

Algorithm 7.6 Verificación de eliminación de región.

```
protected final void emit_verify_region_end(){
    if(VM_RegionInformation.getRegionData(method)!=null){
        asm.emitPUSH_Imm(method.getId());
        genParameterRegisterLoad(1);
        asm.emitCALL_RegDisp(JTOC, VM_Entrypoints.removeRegionMethod.getOffset());
    }
}
```

7.5.1.3. Determinación de región de asignación

Hasta el momento está resuelta la creación y eliminación de la región asociada a un método. Faltaría determinar en qué región crear los nuevos objetos teniendo en cuenta los lugares de creación. Para ello se deben identificar las instrucciones del bytecode correspondientes a la creación de un objeto.

Supongamos el siguiente código Java con las posibles formas de crear un objeto, y su correspondiente bytecode de creación:

Java:

```
public static void main(String[] args) {
    int v1 = 1; // ICONST_1
    int[] v2 = new int[2]; // NEWARRAY T_INT
    A v3 = new A(); // NEW regions/A
```

```
        A[] v4 = new A[2]; // ANEWARRAY regions/A
        A[][] v5 = new A[2][3]; // MULTIANEWARRAY [[Lregions/A; 2
        String test= new String("test"); // NEW java/lang/String
    }
```

Las instrucciones bytecode de creación de objetos son NEW, NEWARRAY, y MULTINEWARRAY. Notar que la asignación del entero 1 en la primer línea de código no genera un nuevo objeto. Jikes RVM al interpretar el bytecode y encontrar alguna de estas instrucciones de creación discrimina según el tipo de instancia a crear e invoca los siguientes eventos de compilación:

- `resolved_new`: Bytecode *NEW* de una clase que forma parte de la VM. Ej.: `new String("test");`. El evento recibe como parámetro el tipo de objeto a crear.
- `unresolved_new`: Bytecode *NEW* de una clase creada dinámicamente en tiempo de compilación. Ej.: `new MiClase();`. Este evento recibe como parámetro la referencia al tipo de objeto a crear, que deberá ser resuelto en runtime.
- `resolved_newarray`: Bytecode *NEWARRAY* de una clase interna. Se especifica el tipo de objetos que contendrá el array y su dimensión.
- `unresolved_newarray`: Bytecode *NEWARRAY* de una clase creada dinámicamente.
- `multianewarray`: Bytecode *MULTINEWARRAY*. Define una matriz. Se debe especificar el tipo de objeto, la cantidad de dimensiones y la referencia (*stack frame*) al tamaño de sus dimensiones.

En la implementación de cada una de estas instrucciones se debe verificar la existencia de lugares de creación correspondientes al método en compilación, el número de línea dentro del método y su posicionamiento. Como la información del método ya la tenemos a disposición, faltaría tener conocimiento del número de línea y la posición. El número de línea es posible calcularlo a partir del índice de bytecode y el mapa de números de línea, y la posición se obtiene con un contador local en el compilador que cuenta cantidad de new en la misma línea.

A diferencia de la creación y eliminación de una región, en este caso los métodos de runtime ya están creados (son los que utilizan el allocator convencional que no maneja regiones), sólo hace falta modificar las llamadas invocando a los métodos runtime que asignan objetos en regiones.

Ej: Si tenemos el siguiente método de compilación para el bytecode *resolved_new*, se deben obtener las regiones posibles en donde se debe crear este objeto mediante:

```
VM_RegionInformation.getAvailableRegionsData(method,newLineNumber,positionInLineNumber);
```

que devuelve el conjunto de regiones $R_{cs_{m,p}}$, que deberá ser incorporado en el *stack frame*. Finalmente se reemplaza la función de runtime por la función alternativa que asigna objetos en regiones. Esta función será explicada en el capítulo 8.

Método original

```
protected final void emit_resolved_new(VM_Class typeRef) {

      int instanceSize = typeRef.getInstanceSize();
      Offset tibOffset = typeRef.getTibOffset();
      int whichAllocator = MM_Interface.pickAllocator(typeRef, method);ç
      int align = VM_ObjectModel.getAlignment(typeRef);
      int offset = VM_ObjectModel.getOffsetForAlignment(typeRef);
      int site = MM_Interface.getAllocationSite(true);
      asm.emitPUSH_Imm(instanceSize);
      asm.emitPUSH_RegDisp(JTOC, tibOffset);
      asm.emitPUSH_Imm(typeRef.hasFinalizer() ? 1 : 0);
      asm.emitPUSH_Imm(whichAllocator);
      asm.emitPUSH_Imm(align);
      asm.emitPUSH_Imm(offset);
      asm.emitPUSH_Imm(site);
      genParameterRegisterLoad(7);
      asm.emitCALL_RegDisp(JTOC, VM_Entrypoints.resolvedNewScalarMethod.getOffset());
      asm.emitPUSH_Reg(T0);

}
```

Algorithm 7.7 Instrumentación de lugares de creación.

```
protected final void emit_resolved_new(VM_Class typeRef) {

      VM_HashSet<VM_RegionData> regionsData =
      VM_RegionInformation.getAvailableRegionsData(method,newLineNumber,cantNewsInLineNumber);
      if(regionsData!=null){
           for(VM_RegionData r:regionsData){
                // incorporo regiones en el stack
                asm.emitPUSH_Imm(r.getIdentifier());
           }
      }
      final int PARAMETERS = 9;
      //20 es el offset de Rcs_{method,<newLineNumber,cantNewsInLineNumber>} a partir
      //de los parámetros, OFFSET_WORDS es el desplazamiento de los
      //parámetros relativos al stack frame
      final int OFFSET_WORDS = PARAMETERS+20;
      int instanceSize = typeRef.getInstanceSize();
      Offset tibOffset = typeRef.getTibOffset();
      int whichAllocator = MM_Interface.pickAllocator(typeRef, method);
      int align = VM_ObjectModel.getAlignment(typeRef);
      int offset = VM_ObjectModel.getOffsetForAlignment(typeRef);
      int site = MM_Interface.getAllocationSite(true);
      asm.emitPUSH_Imm(instanceSize);
      asm.emitPUSH_RegDisp(JTOC, tibOffset);
      asm.emitPUSH_Imm(typeRef.hasFinalizer() ? 1 : 0);
      asm.emitPUSH_Imm(whichAllocator);
      asm.emitPUSH_Imm(align);
      asm.emitPUSH_Imm(offset);
      asm.emitPUSH_Imm(site);
      // parametros de regiones
      asm.emitPUSH_Imm(regionsData.size());
      asm.emitPUSH_Imm((regionsSize + OFFSET_WORDS) << LG_WORDSIZE);
      genParameterRegisterLoad(PARAMETERS);
      asm.emitCALL_RegDisp(JTOC, VM_Entrypoints.resolvedNewScalarRegionMethod.getOffset());
      // borro regiones del stack
      if(regionsData!=null){
           for (VM_RegionData r:regionsData) {
                asm.emitPOP_Reg(S0); // S0 = ECX scratch registry
           }
      }
      asm.emitPUSH_Reg(T0);

}
```

7.6. Compilación de anotación @PrintRegionInfo

El proceso de compilación de esta anotación es similar al de las anotaciones de regiones. Cuando la propiedad *news* es verdadera se deberán instrumentar todos los métodos *emit new...* , para agregar el código máquina necesario para imprimir por pantalla la información del lugar de creación.

Si la propiedad *statistics* es verdadera, en el método *emit_verify_region_end* se debe agregar una llamada al método *printRegionInfo* de runtime que imprimirá los datos estadísticos actuales del método recientemente ejecutado.

Capítulo 8

Runtime

En el capítulo 7, Compilación, se ha detallado el proceso por el cual la información de regiones es almacenada y utilizada para generar el código máquina correspondiente con las acciones de administración de regiones. En este capítulo, de detallará cómo se ejecuta el programa compilado, haciendo incapié en los procesos de administración de memoria.

Como al compilar un método se ha garantizado la creación y eliminación de la región asociada (dentro de su tiempo de vida) y la ejecución de los métodos conforman una pila de llamadas, la estructura de regiones resultante también posee el mismo comportamiento. Esto permite diseñar una administración de memoria eficiente siempre y cuando se trate de un solo hilo de ejecución (*thread*). En el caso en que se tuvieran en cuenta múltiples *threads*, la generación de regiones no tendría forma de pila, más bien podrían ser múltiples pilas. **Es por eso que en esta implementación se limitará el funcionamiento de las regiones a un solo *thread*. Todos los métodos creados deberán validar que el *thread* en ejecución sea el *Main*, y en caso contrario, las operaciones de creación y eliminación de regiones serán omitidas y los *allocs* se llevarán a cabo en el heap o en la región inmortal (en el caso de un modelo puro de regiones). En un trabajo futuro se detallarán ideas para contemplar una implementación *multithread* sin desechar el modelo implementado.**

Un caso particular donde se soporta la utilización de múltiples threads es utilizando la herramienta de análisis de escape de [21]. Las regiones generadas forman tribus de objetos que no se relacionan entre sí, por lo que objetos compartidos por diferentes *threads* pertenecerán a la región del método cuyo *thread* posea un tiempo de vida más largo.

8.1. VM_Runtime

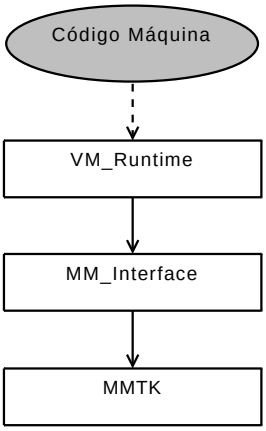

Figura 8.1: Interfaces de Runtime.

La ejecución del programa es guiada por el mismo código máquina generado. Es responsabilidad del compilador generar la correcta "plantilla" de código para cada instrucción bytecode, de modo que se asegure la continuidad del programa. Eventualmente, pueden presentarse llamadas (*asm calls*) a los métodos definidos en la tabla *jTOC* (*entrypoints*). Estos métodos están escritos en Java y sólo pueden hacer uso de componentes internos de la VM que cumplen ciertas condiciones (en particular no deben ser generadores de nuevos objetos Java).

Como interfaz principal para *runtime Jikes RVM* provee la clase *VM_Runtime*, que incluye estos métodos especiales, y en particular los de administración de regiones, que fueron referenciados desde el compilador. Los mismos harán uso de la interfaz de memoria *MM_Interface*, que es donde se definirán los métodos que utiliza el *Mutator* del plan de memoria de *MMTk* con los nuevos componentes implementados.

8.1.1. Interfaz (sólo métodos de regiones)

- `createNewRegion`: recibe como parámetro el identificador del owner (el método), el identificador asignado por el compilador (*VM_RegionInformation*) y su tamaño en bytes. Crea una región de memoria en el tope de la pila de regiones.
- `removeRegion`: remueve la última región creada (tope de la pila).
- `resolvedNewScalarRegion`: recibe como parámetro el *TIB* (*Type Information Block*), que contiene la información sobre el tipo de objeto a crear, el tipo de allocator a utilizar, la alineación, el desplazamiento y la información para acceder a las regiones disponibles (cantidad de regiones y offset en el *stack frame*). Este método intenta crear un nuevo objeto en la región que corresponda según las disponibles que fueron registradas en el *stack frame*. Como este método es utilizado para toda asignación de memoria (utilizando regiones o no), el allocator

determinará finalmente cómo se efectuará la asignación. En el caso que las regiones disponibles sean nulas o ninguna sea válida, el objeto deberá crearse en el espacio por defecto según el plan.

- `unresolvedNewScalarRegion`: recibe como parámetro la referencia al *TIB*, y las regiones disponibles. Primero se resuelve la clase del objeto a crear, la alineación, el desplazamiento, y se invoca a *resolvedNewScalarRegion*.

- `resolvedNewArrayRegion`: recibe por parámetro la longitud del arreglo, la estructura que representa la información del arreglo (*VM_Array*), y el conjunto de regiones disponibles. Al igual que en la creación de un objeto, se determina la región a crear el array en base a la configuración de regiones en memoria.

- `unresolvedNewArrayRegion`: recibe por parámetro el tamaño del array, la referencia a *VM_Array*, y las regiones disponibles. Se resuelve la referencia y se invoca a *resolvedNewArrayRegion*.

- `newArrayArrayRegion`: Recibe por parámetro el TIB, la dimensión, el offset de los tamaños de las dimensiones en el *stack frame*, y la información de regiones disponibles (igual que en los métodos anteriores). Es invocado para crear una matriz n dimensional. El método debe llamar a *resolvedNewArrayRegion* según la dimensión del array pasada por parámetro. Por ejemplo si la dimensión es 2, primero se crea un array con el tamaño de la primer dimensión, y luego por cada elemento, un array con el tamaño de la segunda dimensión. En cada invocación se deben pasar por parámetro las regiones disponibles.

- `printRegionInfo`: método de soporte para debug y estadísticas de conteo en regiones. Toma un identificador de región por parámetro y genera un dump del estado de una región.

8.1.1.1. Selección de región de asignación

Dado que el lugar de creación en un método puede tener múltiples regiones definidas en compilación, pero sólo una al momento de la asignación, en los métodos citados anteriormente se deberá determinar en qué región crear el objeto. Para poder acceder a las regiones disponibles que fueron dispuestas en el *stack frame* se recibe por parámetro la cantidad de regiones y el desplazamiento de estos dentro del *stack frame*. En el siguiente algoritmo se puede ver el proceso de selección de región para un lugar de creación teniendo en cuenta la configuración de la pila de regiones dispuesta por el allocator.

Si por ejemplo se tiene un lugar de creación $cs_{m,p}$ con su conjunto de regiones correspondientes $R_{cs_{m,p}}$, el $r_{m'} \in R_{cs_{m,p}}$ escogido será el creado más recientemente en la pila del allocator de memoria. De no existir $r_{m'}$ el objeto se crea en el espacio de memoria por defecto (inmortal o el heap para la VM con espacio con GC) dependiendo de la configuración del plan de memoria (ver sección 5.1).

Algorithm 8.1 Algoritmo de selección de región en tiempo de ejecución.

```
private static int getActiveRegion(int regionsSize, int argOffset) {

      VM.disableGC();
      // Obtengo el puntero al stack frame
      Address firstArgp = VM_Magic.getFramePointer() + argOffset;
      int result = INVALID_REGION_ID;
      // Obtengo la región del tope de la pila
      int topId = MM_Interface.getTopRegionId();
      // recorro la pila hasta encontrar una región que concuerde con alguna de las disponibles
      while(topId!=INVALID_REGION_ID){
            Address argp = firstArgp;
            for (int i = 0; i < regionsSize; ++i) {
                  // resto 4 bytes para acceder al próximo entero
                  argp = argp - 4;
                  if(topId==argp.loadInt()){
                     result = topId;
                     break;
                  }
            }
            if(result!=INVALID_REGION_ID){
                  break;
            }
            topId = MM_Interface.getPreviousRegionId(topId);
      }
      VM.enableGC();
      return result;

}
```

Orden algorítmico: $\mathcal{O}$(regiones en la pila * regiones disponibles)

Este preceso, en realidad se podría hacer sólo una vez para cada configuración de la pila de llamadas a métodos (stack trace) manteniendo una cache de lugares de creación según pila de métodos en ejecución. Lo que invalidaría la cache para una entrada podría ser:

- la carga/recarga dinámica de alguna biblioteca (recompilación de algún método involucrado)
- la modificación por *reflection* de algún método involucrado.

Si se implementara en un futuro esta optimización, se bajaría el orden de cómputo a $\mathcal{O}(1)$ una vez conocido el lugar de creación.

Ejemplo Si tomamos el programa Java definido en el algoritmo 6.4 una posible configuración de pila en el momento de asignar memoria para $cs_{m2,3}$ sería:

$[r_{m0}, r_{m1}, r_{m2}, r_{m2}, r_{m2}]$

Se puede observar que la región asociada a *m2* se repite varias veces en la pila por cada invocación del método. Cuando sucede este caso se toman como regiones válidas (de la pila) las últimas de cada método, es decir r_{m0}, r_{m1}, y la del top de la pila de r_{m2}. Para elegir dónde crear $cs_{m2,3}$ se deben tomar las regiones posibles $R_{cs_{m2,3}} = \{r_{m0}, r_{m1}\}$ y obtener la primera ocurrencia de alguna de ellas en la pila de regiones. Como r_{m1} está más cerca del tope de la pila que r_{m0}, será allí el sitio donde se llevará a cabo la operación. Este lugar de creación tiene sentido ya que es la que pertenece al método que invocó a *m2*. Si se desea crear $cs_{m2,4}$ como $R_{cs_{m2,4}} = \{r_{m2}\}$ se tomará la última región de la pila.

8.2. Administración de memoria de bajo nivel (MMTk)

En esta sección se detallarán las estructuras y algoritmos utilizados para diagramar la memoria de forma que sea posible administrar regiones. Primero se detallará la estructura de memoria más básica correspondiente al administrador de páginas, y luego los allocators que lo utilizan para finalmente poder realizar las asignaciones de memoria para regiones y objetos. La interfaz *VM_Runtime* de la sección anterior se comunicará con la interfaz de memoria *MM_Interface* que hará uso de estos allocators, por medio del plan de memoria (clases *Plan*, *Mutator*, *Collector* y *Trace*).

8.2.1. RegionsPageResource. Administración de páginas de memoria.

El administrador de páginas es el componente del *Espacio* que cumple la función de reservar y liberar chunks de memoria del sistema para satisfacer los pedidos de los allocators.

La versión disponible de Jikes RVM incluye las siguientes implementaciones:

- `MonotonePageResource`: Implementada como *bump-pointer*. Reserva chunks contiguos de memoria en cada pedido de asignación de páginas. Los chunks pueden ser o no contiguos entre sí (entre distintos pedidos). No posee la posibilidad de liberar las páginas tomadas. Esta implementación se utiliza para implementar espacios de memoria inmortales. La proceso de asignación es rápido y sencillo.

- `FreeListPageResource`: Está implementada mediante una *free-list*. Las páginas sólo pueden ser contiguas si se especifica el rango de direcciones de memoria antes de ser creada, lo que no permitiría ampliar la memoria en un futuro. Es el administrador de páginas por defecto de los espacios que trabajan sobre el heap y utilizan *GC*.

Para que las operaciones de administración de memoria sobre las regiones sean eficientes es esperado que el espacio dedicado a cada una de ellas sea contiguo, de otra manera este modelo de administración de memoria sería muy complejo con tiempos de administración poco favorables. *FreeListPageResource* permite realizar asociaciones contiguas, pero no permite extender el espacio de memoria si no alcanza con lo especificado al iniciarse el espacio, por lo tanto no es una buena opción ya que no es posible determinar un tamaño fijo de consumo para cualquier ejecución de cualquier programa, y tampoco sabemos si el hardware a utilizar posee los suficientes recursos de memoria disponibles.

MonotonePageResource parecería ser la opción adecuada para la implementación, ya que en cada pedido de asignación de n páginas es posible asegurar contigüidad, y soporta la expansión del *budget* de páginas si lo reservado no alcanzara para cubrir el pedido. Si se piensa que cada pedido de reserva de páginas es destinado a la disposición de una nueva región se obtendrán espacios de regiones contiguas, y lo único que haría falta implementar es la liberación de las mismas al eliminar una región. En otras palabras, habría que transformar la memoria inmortal en una memoria de pila.

El administrador de páginas será denominado *RegionsPageResource* y posee la siguiente interfaz.

- `Address allocPages(int requestPages)`: reserva *requestPages* páginas y retorna la dirección de memoria inicial del pedido.
- `releaseTopRegionPages(Address from)`: libera todos los *chunks* de memoria reservados a partir de la posición de memoria *from*. Si la posición *from* no fue devuelta en un pedido de reserva anterior se lanza una excepción.
- `int getReservedPages()`: devuelve la cantidad de páginas de memoria reservadas.

Para entender mejor el funcionamiento de este componente se presenta el siguiente ejemplo: se deben crear tres regiones *r1*, *r2* y *r3*. Sus tamaños son 120000 bytes, 43600 bytes y 20000 bytes.

Inicialmente no hay memoria reservada como puede verse en la figura 8.2 (a). Hay un solo chunk de memoria no disponible que está siendo reservado previamente por alguna otra aplicación.

Al querer crear *r1, RegionsAllocator* hará un pedido al administrador de páginas para reservar $ceil(120000/(pagesize))$ siendo *pagesize* el tamaño en bytes de una página. Si tomamos *pagesize* como 4096 bytes entonces el pedido de asignación será de 30 páginas.

Como todavía no se han reservado *chunks* de memoria, se procede a pedir $ceil(30/(pagesPerChunk))$ chunks de memoria contigua, lo que se corresponde con 4 chunks si $pagesPerChunk = 8$ (b). Para este entonces se reservan las 30 páginas e internamente se mantienen disponibles para una futura asignación las 2 páginas restantes del chunk.

Para crear *r2* (43600 bytes) se necesitan 11 páginas (2 chunks) contiguas. Como no es posible realizarlo en las 2 páginas reservadas restantes se procede a hacer un nuevo pedido (c). Quedarían 5 páginas sin utilizar del último chunk que serán utilizados por *r3* en (d). En este caso no se realiza ningún pedido de memoria al sistema ya que habían quedado reservados al crear *r2*.

Luego se procede a eliminar las regiones en el siguiente orden: *r3*, *r2*, *r1* y recién al eliminar *r2* se liberarán los 2 chunks tomados (e). Finalmente cuando *r1* es eliminado se retorna al mismo estado que en (a).

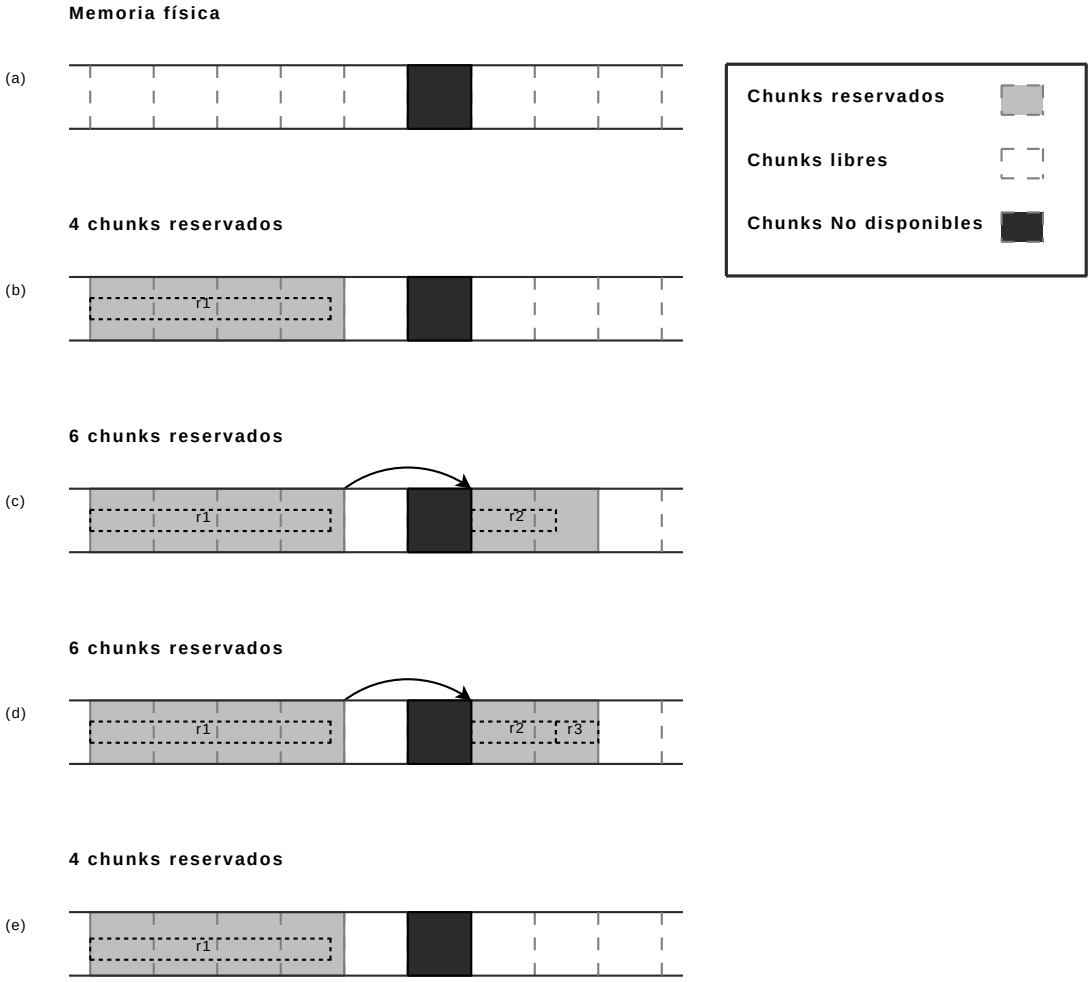

Figura 8.2: RegionsPageResource.

Nota: si las regiones a crear hubieran tenido un tamaño menor al de una página, el pedido sólo hubiera llegado al administrador de páginas cuando se haya agotado el espacio disponible en dicha página.

8.2.2. Allocators

La implementación de los allocators es una de las tareas más importantes en este trabajo. En ellos se define la estructura de datos que representan a las regiones y como serán almacenadas, se determina el modo en que se crean o eliminan, y cómo se guardan los objetos dentro de ellas. También se mantiene el control de los recursos de memoria pedidos al administrador de páginas (se debe determinar cuándo pedir más memoria y cuándo liberarla).

Para desarrollar un allocator se debe tener en cuenta que a este nivel de la implementación todavía no es posible crear objetos Java (dado que el *heap* todavía no ha sido creado). En el allocator mismo se definirá la manera en que se disponen los objetos para el espacio que administre. Es por eso que en esta capa del desarrollo sólo se pueden utilizar los tipos básicos y otros, nativos de la

VM, diseñados especialmente para el desarrollo de los componentes internos.

Lo primero que se debe realizar es definir la estructura del heap, determinar la forma en que se asignarán los pedidos (*allocs*) y la dinámica de liberación de los objetos. Si bien en un *heap* con GC los recolectores de objetos se encargan de ello, lo hacen invocando a métodos del allocator, que tienen el control local sobre las distintas fases del proceso de recolección de sus objetos. Por ejemplo en el GC semi-space, el allocator de memoria de cada espacio debe volver a inicializar la memoria del espacio *from space* al finalizar el proceso de recolección, y se deben devolver todas las páginas reservadas reseteando el espacio. De manera similar, en un espacio de regiones, el allocator debe devolver o poner en cero las zonas de memoria tomadas.

Se implementaron dos allocators de memoria para la administración de regiones, *RegionsAllocator* y *ResizeRegionsAllocator*. La primera implementación permite la utilización de regiones con tamaños predefinidos mientras que la segunda soporta la expansión del tamaño de las regiones por medio de punteros internos que permiten su particionamiento. Este último allocator, a diferencia del primero, pude ser utilizado como único espacio del plan de memoria. Esto se debe a que es posible establecer una región por defecto (en la base de la pila), que será utilizada para albergar a todos los objetos "huérfanos" de región, sin conocer previamente el tamaño que ocupará. Si bien el comportamiento es similar al de tener un espacio inmortal, disponer de esta región inmortal permite evaluar una *VM* con un único administrador de memoria que esté basado en regiones. Otra ventaja de este allocator es la posibilidad de no tener que especificar el tamaño de todas las regiones.

En ambos allocators las regiones son representadas mediante una lista encadenada donde cada región conoce a su predecesora. Las operaciones sobre esta lista son efectuadas conforme a una pila de regiones donde la creación y eliminación se realizan en el tope (final de la lista). Dado que hay una mayor actividad sobre tope de la pila, es conveniente que las regiones conozcan a su región anterior. Por ejemplo, además de la eliminación de regiones, en las operaciones de creado de objetos es más probable que se realicen sobre regiones creadas recientemente que sobre regiones cercanas a la base de la pila.

Los algoritmos propuestos utilizarán la interfaz de manejo de memoria que provee Jikes RVM, en particular las clases que representan una dirección de memoria, el desplazamiento y los métodos para almacenar o levantar un objeto de ella:

- `Address`: representa a una dirección de memoria.
- `Offset`: es el desplazamiento relativo a una dirección de memoria. Offset funciona como una abstracción de un tamaño de memoria, puede ser especificado en *bytes*, *words*, *extents*.

Address posee principalmente las siguientes responsabilidades:

- `store`: guarda el objeto pasado por parámetro. Este puede ser: `byte, char, float, Address, int, long, double, float, Word, ObjectReference`. Ej. de llamada: `myAddress.store(10)` // (10 es de tipo int)
- `load{type}`: levanta el objeto de tipo *type*. A diferencia del método *store*, en este caso hace falta especificar `Type` ya que *Address* desconoce el tipo de objeto que contiene. Ej. de

llamada: `myAddress.loadInt()` // en este caso se levantarán de memoria los bytes necesarios que representan un *int*.

- operadores `(+, -) offset`: Permiten sumar o restar desplazamientos a una dirección de memoria. Los operadores retornan la dirección de memoria resultante de desplazar la dirección en *offset* unidades. Ej: `Address resultAddress = myAddress + myOffset.`
- operadores `(+, -) bytes`: retorna la dirección de memoria resultante de sumar *bytes*. Este método es una variante del anterior.
- comparadores `(<, <=, ==, >, >=, !=)`: Soporte de operadores de comparación y orden.

8.2.3. RegionsAllocator

RegionsAllocator permite crear regiones de tamaño predefinido. Éstas estarán dispuestas de manera contigua en una pila, y sólo se podrá eliminar la región que esté en el tope (la última creada). Esta administración es compatible con la asociación método/región, y asegura la contigüidad en memoria.

8.2.3.1. Estructura

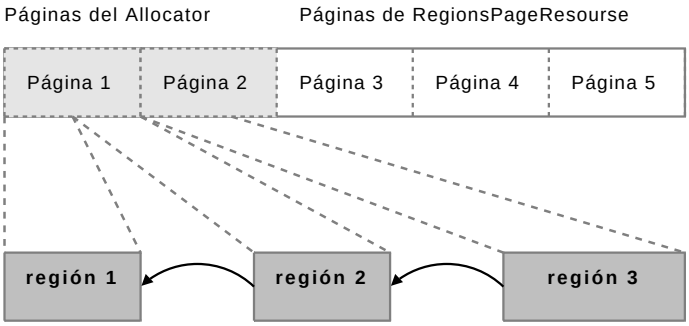

Figura 8.3: RegionsAllocator. Estructura.

La estructura generada en memoria es la de una pila de regiones $[r_1..r_i..r_n]$ donde r_i ocupa una posición de memoria más baja que r_k siendo $i < k$.

8.2.3.2. Interfaz

La interfaz presenta los siguientes métodos:

- `createNewMemoryRegion(int regionId, int owner, int memoryRegionSize)`: crea una región con identificador `regionId` en el tope de la pila con el tamaño especificado. Orden: $\mathcal{O}(1)$
- `removeTopMemoryRegion()`: elimina la región del tope de la pila. Orden: $\mathcal{O}(1)$
- `Address alloc(int size, int regionId, int align, int offset)`: Devuelve una dirección de memoria disponible en la región `regionId` para asignar un objeto de `size` bytes. Orden: $\mathcal{O}(n)$ (n : cantidad de regiones en la pila).

- `getTopRegionId()`: devuelve la región que está en el tope de la pila. Si no hay regiones devuelve la constante *INVALID_REGION_ID*.
- `getPreviousRegionId(int regionId)`: devuelve la región que precede en la pila a *regionId*. En el caso que sólo haya una región en la pila se retornará *INVALID_REGION_ID*.

8.2.3.3. Implementación

El allocator mantiene en memoria un puntero a la última región creada, asegurando tener el acceso a las demás, y los punteros correspondientes para administrar la reserva de memoria por medio de *RegionsPageResource*.

La dinámica del algoritmo de reserva de memoria funciona de manera similar a un *bump-pointer* en cuanto a la creación y eliminación de regiones (debido a la contigüidad de los recursos reservados), mientras que para la creación de los objetos en regiones la asignación se realiza de manera contigua dentro de la misma.

Para administrar las páginas reservadas se utilizan dos punteros denominados *cursor* y *limit* que indican la posición de escritura de la región y el límite de memoria disponible respectivamente.

Inicialmente, ambos punteros son asignados con la dirección de memoria *zero (Address.zero())*, que representa la posición de memoria nula (dispara la excepción *NullPointerException* en runtime).

Cuando se requiere crear una región de tamaño t, se verifica si $cursor + t <= limit$. Si se cumple dicha condición, se actualiza *cursor* aumentándolo en t bytes. Este tipo de asignación es denominada *fast-path allocation* y se efectúa en forma local al thread. En caso contrario se realiza un pedido de páginas a *RegionsPageResource*, que retornará un nuevo rango de memoria disponible que deberá ser incorporado actualizando *cursor* y *limit* (teniendo en cuenta la contigüidad del nuevo rango respecto a *limit*). Cuando sucede este caso se denomina *slow-path allocation* y se efectúa en el componente global sincronizado del plan de memoria. En el algoritmo de creación de una región se puede observar con más detalle este comportamiento.

Durante la creación de objetos en cualquier región los valores tanto de *cursor* como de *limit* no son alterados, recién al eliminar la última región, se actualizará *cursor* apuntándolo al final de la nueva última región y sólo se actualizará *limit* si dicha región fue creada mediante *slow-path allocation*. Este es el único caso en donde se deben devolver las páginas de memoria liberadas al administrador de páginas, debido a que *RegionsPageResource* registra únicamente los chunks pedidos por los allocators, y como se explicó anteriormente, los chunks tomados se deben administrar localmente en cada allocator para evitar la sincronización global de pequeños pedidos de memoria.

Estructura de una región en memoria Una región está compuesta por un encabezado de región (*header*), y un área contigua de datos. El header lo componen (ver figura 8.4):

- `RE` (*Region End Pointer*)*:* Dirección de memoria que indica el final de la región.
- `RID` (*Region ID*): número entero que identifica a la región.

- `OWN` (*Owner*): número entero que identifica al dueño de la región (puede ser nulo). Este identificador se agregó para soportar implementaciones de regiones asociadas a elementos de cómputos, en este trabajo se utilizó el identificador interno del método. Este valor todavía no es utilizado en esta tesis pero se incorporó para soportar ampliar la funcionalidad en trabajos futuros.
- `STA` (*Statistics*): está compuesta por dos enteros. Llevan el conteo de objetos albergados y el tamaño actual de la región (ver capítulo 9).
- `DW` (*Data Write Pointer*): apunta a la primer dirección de memoria libre en el área de datos. Al inicializarse una región, el puntero señala la siguiente posición de memoria que procede a su encabezado. Su cota está determinada por el puntero fin de región *RE*.
- `PR` (*Previous Region Pointer*): puntero a la primer dirección de memoria de la región anterior. Puede ser *zero* en caso de que la región en cuestión sea la primera (base de la pila).
- `PRI` (*Page Request indicator*): Indicador de *slow-path request*. Se registra la dirección de memoria devuelta por *RegionsPageResource* si la asignación fue realizada mediante *slow-path*. Esta información es útil a la hora de eliminar la región, para saber si se debe liberar el espacio al administrador de páginas.

El espacio de datos comprende desde la finalización del header hasta la posición de memoria apuntada por *RE*. No está estructurado, sino que los objetos son asignados contiguamente, actualizándose el puntero de escritura de datos *DW* en cada asignación.

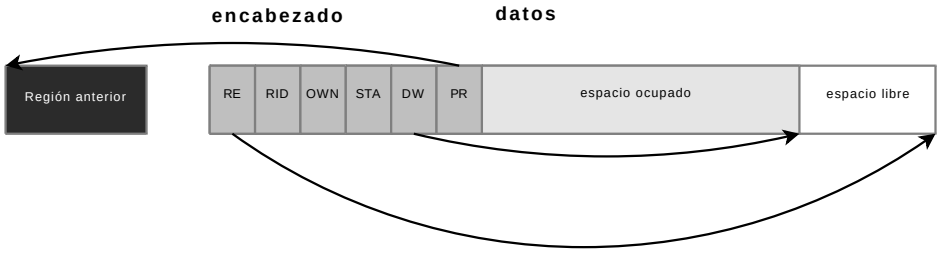

Figura 8.4: Estructura de región.

Algoritmos de creación y borrado de regiones, y de asignación de objetos A continuación se presentan en pseudocódigo y en forma simplificada los algoritmos para la creación y eliminación de regiones, y de asignación de objetos.

Constantes del allocator

- `Offset int RID_OFFSET`: desplazamiento del identificador de la región dentro del header.
- `Offset OWN_OFFSET`: desplazamiento del identificador owner dentro del header.
- `Offset PR_OFFSET`: desplazamiento del puntero a la región anterior dentro del header.

- `Offset RE_OFFSET`: desplazamiento del puntero fin de región dentro del header.
- `Offset DW_OFFSET`: desplazamiento del puntero de escritura de objetos dentro del header
- `Offset STA_OFFSET`: desplazamiento del área de estadísticas dentro del header
- `Offset PRI_OFFSET`: desplazamiento del indicador de *slow-path request* dentro del header.
- `Offset REGION_HEADER_SIZE`: indica el tamaño del header. Su valor es la suma de los desplazamientos anteriores.

Información que mantiene el allocator

- `RegionsPageResource regionsPageResource`: instancia de *RegionsPageResource*. Se utilizará para pedir y liberar páginas de memoria.
- `Address cursor`: puntero a escritura de región.
- `Address limit`: puntero centinela de memoria.
- `Address lastRegion`: puntero a la última región.

Algorithm 8.2 Creación de una región con RegionsAllocator.

```
createNewMemoryRegion(int regionID, int ownerID, int memoryRegionSize){
    assert(memoryRegionSize > 0, "invalid region size")
    //calculo tamaño de la región incluyendo el desplazamiento del header
    memoryRegionSize += REGION_HEADER_SIZE.toInt()
    Address start = cursor
    Address end = start.plus(memoryRegionSize)
    boolean slowPath = end > limit
    if (slowPath) {
        // Realizo pedido de páginas al RegionsPageResource
        int pages = Math.ceil(memoryRegionSize / PAGE_SIZE)
        Address temp =regionsPageResource.allocPages(pages)
        // verifico que el pedido fue exitoso
        assert(!start.isZero(),"Out of memory")
        if(start.NE(limit)){
           start = temp;
           end = start.plus(memoryRegionSize)
        }
        start.store(PRI_OFFSET,temp) PRI - Page Request Address
        // actualizo el limite
        limit = start + pages * PAGE_SIZE
    }
    // inicializo el header
    start.store(RID_OFFSET,regionID) // regionID
    start.store(OWN_OFFSET,ownerID) // ownerID
    start.store(PR_OFFSET,ultimaRegion) // previousRegion
    start.store(RE_OFFSET,end) // RegionEnd
    start.store(DW_OFFSET,start + REGION_HEADER_SIZE) // Data Writer
    start.store(STA_OFFSET,(0,0)) // Statistics
    // actualizo variables del asociador
    lastRegion = start;
    cursor = end
}
```

Orden algorítmico: $\mathcal{O}(1)$.

8.2.3.4. Se mostrará con un ejemplo el mecanismo para crear y eliminar regiones.

Creación de una región

- Crear $r_{regionId}$
  - Verificar el espacio reservado del allocator. Si no es suficiente reservar espacio de *RegionsPageResource*.
  - Inicializar header de la parte de región especificando si el pedido se realizó por *slow-path*.
  - Actualizar punteros *cursor* y *limit* (según corresponda).
  - Si no es la primer región apuntar a la región anterior.
- Apuntar *lastRegion* a $r_{regionId}$

Supongamos que se tienen que crear 3 regiones: *region1*, *region2* y *region3*. Sus tamaños en bytes son 2500 para *region1* y *region2*, y 1400 para *region3*. Al igual que en el ejemplo anterior tomaremos el tamaño de una página como 4096 bytes.

En la figura 8.5 (a), se puede observar el estado del allocator cuando no hay ninguna región creada. Los punteros *cursor* y *limit* están seteados en *zero* ya que todavía no se realizó un pedido al administrador de páginas.

Para crear *region1* se calculan las posiciones de memoria que abarcará la región (*desde start address* hasta *end* address) y se verifica si cabe en la memoria reservada. Esta condición se cumple si $end < limit$. Como todavía no se realizó una reserva de páginas, *end* es mayor a *zero* y por lo tanto se procede a realizar el pedido por *slow-path*: se calcula la cantidad de páginas que ocupará la región $ceil(2500\ bytes/4096\ bytes) = 1\ página$ y si *RegionsPageResource* posee recursos de memoria reservará internamente un chunk, y devolverá la posición de memoria inicial de la página pedida.

Como el nuevo pedido puede no ser contiguo con el anterior (la dirección retornada es distinta que *limit*), y en este caso como es la primera página pedida el valor de *limit* es *zero*, se debe volver a calcular el rango de la región a partir de la posición de memoria retornada. Luego debe registrarse en el header de la nueva región el pedido de *slow-path* (*PRI*) y actualizar *limit* con la última dirección de memoria disponible. Figura 8.5 (b)

Hasta este momento está resuelto el espacio para albergar la región, pero todavía no se ha guardado toda la información para que quede inicializada. Lo que resta es escribir el *header* de *region1* utilizando las constantes de *Offset* para cada uno de los datos a guardar, apuntar *lastRegion* al inicio de la región, y actualizar *cursor* indicando la posición de escritura para la siguiente asignación.

En (c), se muestra el resultado de crear *region2* (2500 bytes). Dado que el espacio restante en el allocator es menor que 2500 bytes: $limit - cursor < 2500$ ($1596\ bytes < 2500\ bytes$), al igual que en (a), se realiza un pedido de 1 página a *RegionsPageResource*, quien no tiene que reservar más chunks (un chunk contiene más de 2 páginas). La diferencia es que esta vez la página retornada es contigua a la anterior. Como la asignación se realizó por *slow-path*, también se procede a actualizar *limit*.

Un caso diferente sucede en (d) donde se requiere crear *region3* (1400 bytes). El espacio disponible es de 2 *páginas*-5000 *bytes* = (8192 *bytes* − 5000 *bytes*) = 3192 *bytes*, con lo cual alcanza para crear *region3* sin realizar un pedido. Como la asignación se realiza por *fast-path*, sólo debe actualizarse *cursor*, además de escribir el *header* para la región.

Notar que las asociaciones de tipo *fast-path* siempre son contiguas con la asignación de la región anterior; esta propiedad no es posible asegurarla con *slow-path*.

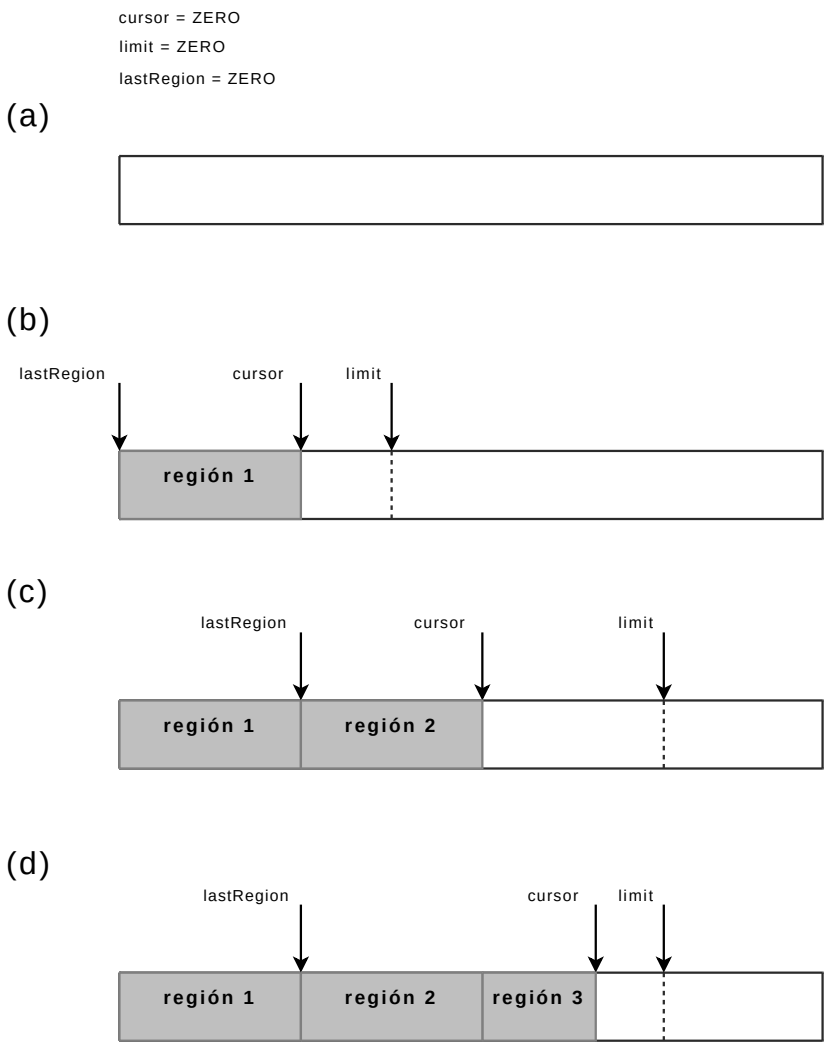

Figura 8.5: Dinámica de asignación y eliminación de regiones

Eliminación de una región El algoritmo para eliminar una región se comporta de manera inversa al de creación. Cuando se requiere remover una región, que siempre será la que está en el tope de la pila (debido a que éstas están ligadas a la ejecución de los métodos y es un ambiente *mono-thread*), debemos actualizar cursor a la dirección de memoria que apunta *lastRegion*, de modo

que las futuras creaciones de regiones se realicen sobre este espacio a liberar. Luego se debe apuntar como *lastRegion* a la región previa (indicada en el header como PR_OFFSET) y finalmente proceder a la liberación del espacio. En este paso hay dos caminos a seguir, si la región a eliminar fue creada mediante *slow-path*, se deben liberar las páginas tomadas de *RegionsPageResource,* si no sólo volver a *zero* el área de memoria.

- Eliminar $r_{lastRegionId}$
- Actualizar *cursor*.
- Si la región fue creada por *slow-path* liberar páginas y actualizar *limit.*
- Apuntar *lastRegion* a la región anterior.

Notar que el proceso de creación y borrado de regiones es simétrico, dado que se mantiene inalterado el estado del allocator luego de haber creado y eliminado regiones para cualquier punto de la pila de regiones. Es por eso que si recorremos el ejemplo anterior desde abajo hacia arriba (desde la figura 8.5 (d) hasta la (a)), se puede ver la eliminación progresiva de las regiones y la actualización de los punteros en cada caso.

Algorithm 8.3 Eliminación de una región con RegionsAllocator

```
public final void removeTopMemoryRegion() {
assert(countMemoryRegions != 0)

      Address toDelete = lastRegion
      cursor = toDelete
      //actualizo el puntero a última región
      lastRegion = (toDelete + MEMORY_REGION_PREVIOUS_OFFSET).loadAddress()
      // si la region anterior no es contigua y hubo un pedido al page resource,
      // debo liberar las páginas, sino la vuelvo a 0.
      requestAddress = (toDelete + MEMORY_REGION_REQUEST_OFFSET).loadAddress()
      if(cursor != lastRegion and requestAddress!=Address.zer()){
           regionsPageResource.release(requestAddress)
           limit = requestAddress
      }
      else{

           VM.memory.zeroFromTo(toDelete, (toDelete + MEMORY_REGION_END_OFFSET).loadAddress() - toDelete)
      }

}
```

Orden algorítmico: $\mathcal{O}(1)$.

Creación de objetos en regiones El algoritmo para crear un objeto en una región es relativamente simple, sólo hace falta indicar en qué región se llevará a cabo, el tamaño del objeto y el posicionamiento relativo a la dirección de memoria, indicado por medio de la alineación y el desplazamiento dentro de la misma. Los objetos son dispuestos contiguamente en el área de datos de la región. El espacio disponible comprende desde la posición de memoria donde finaliza el header, hasta la dirección apuntada por RE_OFFSET.

- Buscar $r_{regionId}$.
- Asignar objeto en región

- Verificar disponibilidad de espacio en la región. Si el objeto cabe, se lo asigna y se actualiza *DW*. Si no se lanza la excepción "Espacio agotado en la región *regionId*".

Si se observa el algoritmo, primero se obtiene la región especificada mediante el método *getRegionById*. Luego se obtiene el puntero de escritura de datos *DW_ OFFSET*, que es el lugar donde se llevará a cabo la asignación. Los parámetros *align* y *offset* determinarán la posición exacta de asignación del objeto, que deberá caber en el área de datos. Esto se verifica comparando *endAddress* <= *endData*, siendo *endAddress* la dirección donde finalizaría la asignación, y *endData* la dirección de memoria que determina el final de la región *RE_ OFFSET*. Si esta condición no se cumple se lanzará una excepción indicando la falta de espacio en la región. De lo contrario se procede a actualizar el cursor de escritura de datos *DW_ OFFSSET* y retornar la dirección inicial de la asignación.

Algorithm 8.4 Asignación de objetos en una región

```
public final Address allocBump(int memoryRegionId, int bytes, int align, int offset) {

      // obtengo la región
      Address startRegion = getMemoryRegionById(memoryRegionId)
      Address endAddress = (startRegion + MEMORY_REGION_END_OFFSET).loadAddress()
      Address writeCursor = (startRegion + MEMORY_REGION_BUMP_CURSOR_OFFSET).loadAddress()
      Address startData = alignAllocationNoFill(writeCursor, align, offset)
      Address endData = startData + bytes
      // si el tamaño no es suficiente se retorna una excepción y el programa termina
      if (endData.GT(endAddress)) {
           assertFail("out of memory in region")
      }
      fillAlignmentGap(writeCursor, startData)
      // actualizo punteros internos de la región
      (startRegion + MEMORY_REGION_BUMP_CURSOR_OFFSET).store(endData)
      (startRegion + MEMORY_REGION_STATISTICS_OFFSET).store((startRegion +
                                        MEMORY_REGION_STATISTICS_OFFSET).loadInt() + 1)
      return startData

}
private Address getMemoryRegionById(int regionId) {

      VM.assertions._assert(lastRegion.NE(Address.zero()))
      Address regionAddress = lastRegion
      while (regionAddress.NE(Address.zero())) {
           int id = (regionAddress + MEMORY_REGION_IDENTIFIER_OFFSET).loadInt()
           if (id == regionId)
                return regionAddress
           regionAddress = regionAddress.loadAddress(MEMORY_REGION_PREVIOUS_OFFSET)
      }
      assertFails("no existe la región pedida")
      return Address.zero()

}
```

Orden algorítmico: $\mathcal{O}(n)$. (cantidad de regiones en la pila)

8.2.3.5. Optimizaciones

Los algoritmos anteriores detallan el comportamiento del allocator de una manera simplificada, pero la implementación real propone algunas optimizaciones basadas en su comportamiento habitual, como por ejemplo la creación de varios objetos seguidos en una misma región, o la creación y eliminación iterativa sobre una misma región. Se desarrollaron las siguientes optimizaciones sobre el algoritmo planteado.

Uso de región activa Todas las operaciones sobre las regiones en memoria se podrían realizar sólo con la existencia del puntero *lastRegion*, pero para alivianar la lectura y escritura en memoria se definió el concepto de región activa, que consiste en mantener punteros directos desde el allocator hacia los datos necesarios para trabajar con una región dada sin tener que levantarlos y luego escribirlos en memoria en cada operación. El beneficio de poseer una región activa es el que el orden para tener acceso a dicha región es de $\mathcal{O}(1)$ en vez de $\mathcal{O}(n)$ (recorrer la pila de regiones) y se evitan múltiples escrituras en los headers de la región. Dado que no es posible acotar la cantidad de regiones durante la ejecución de cualquier programa, sólo se utilizará una región activa, y suponiendo que la mayor cantidad de asignaciones se realizan sobre la última región, se le asignará esta distinción. Al agregar esta optimización deberá verificarse en todas las acciones si la región con la cual se quiere operar es la activa. Si no lo es, se debe operar de manera convencional.

Nota: Si antes de ejecutar el programa se pudiera acotar la cantidad de regiones en n, se podrían mantener *las* n regiones activas, y sólo se guardarían en el espacio las asignaciones de los objetos del programa (como en un heap convencional), las regiones no tendrían header ya que dicha información se mantendría estáticamente en el allocator. Actualmente sólo es posible acotar la cantidad de regiones en los programas que no son recursivos, pero la detección de recursividad en un programa puede ser una tarea dificultosa ya que se deberían realizar análisis sensibles al contexto para tener en cuenta los condicionales.

Estructura de la región activa: Datos de la región activa que mantiene el allocator (llamada *currentRegion*):

- puntero a inicio de región actual (apunta al header de la región)
- puntero a fin de región actual (RE)
- Identificador de región (RID)
- Estadísticas de región (STA)
- Puntero de escritura de datos (DW)

Notar que los datos de la región activa son los mismos que se presentan en el header menos el puntero a la región anterior, ya que únicamente se utiliza para ubicar una región cuando esta no es activa.

Reteniendo esta información en el allocator, sólo hace falta volcarla en memoria cuando cambia la región activa, que es cuando se cargarán los datos de la nueva región. El caso más usual donde su utilización mejora ampliamente la performance es cuando suceden grandes cantidades de asignaciones consecutivas sobre una misma región. Si no existiera la región activa, en cada asignación habría que levantar los datos del header de la región, y luego escribir el puntero de escritura DW actualizado, y también actualizar las estadísticas. Teniendo la región activa sólo hace falta volcar los datos modificados a memoria cuando ésta deja de serla.

Creación de la región ante la primer asignación Muchas veces en procesos automáticos no sensibles al contexto es posible que se creen regiones de memoria y nunca se creen objetos dentro de ellas. Por ejemplo en el modelo de memoria implementado, donde asociamos regiones de memoria a métodos, muchas sentencias *news* están en estructuras de decisión, o en *catch* de excepciones. De modo que si creamos la región al iniciarse el método y luego ninguna sentencia *new* es alcanzada en ejecución, estaríamos desperdiciando memoria y tiempo, ya que la región es innecesaria.

La implementación de esta funcionalidad se basa en las siguientes reglas:

- Regla 1: al realizar un pedido de creación de región sólo se mantienen los datos de ésta en el allocator (id y tamaño y owner), y se habilita un flag de creación de región pendiente. No se guarda la región en memoria como se explicó en el algoritmo de creación de regiones.
- Regla 2: si se solicita una asignación de memoria en dicha región y está el flag activado, se crea la región en memoria y se deshabilita el flag.
- Regla 3: si se realiza un pedido de creación de región y está habilitado el flag, se crea la región pendiente en memoria y la nueva región pasa a estar pendiente por regla 1. (Hace falta crear la región previa dado que utilizamos asignación de memoria contigua y sólo disponemos de un solo *slot* para mantener regiones pendientes).
- Regla 4: al eliminar una región que estaba pendiente de creación sólo se deshabilita el flag de región pendiente.

Reinicialización de la región eliminada Así como en el caso de la creación de regiones, que a veces no es necesario llevarlas a cabo, lo mismo sucede con la eliminación. Supongamos el siguiente fragmento de algoritmo con los métodos *crearRegion* y *eliminarRegion*:

```
for(int i=0;i<n;i++){

      crearRegion("mi región",5000); // crea una región de 5000 bytes
      .....
      eliminarRegion("mi región");

}
```

Este algoritmo debe crear y eliminar n veces la región "mi región". Si suponemos que una página de memoria consta de 4096 bytes, para la creación y la eliminación de la región se estarían realizando pedidos y devolución de páginas de memoria a *RegionsPageResource* degradando la performance del administrador de memoria, ya que sólo es necesario realizar esta operación en la primera y última iteración del ciclo, sin la necesidad de sincronizar la memoria con otros allocators.

Se definen las siguientes reglas para resolver este problema:

- Regla 1: ante un pedido de eliminación de región se setea un flag de eliminación pendiente.
- Regla 2: si está el flag habilitado y se realiza una asignación en dicha región, se ignora como si la región no estuviese.

- Regla 3: si está el flag habilitado y se crea una región diferente, antes debe eliminarse la región pendiente y se deshabilita el flag. Esta regla es necesaria para mantener la relación de pila región por método permitiendo que la nueva región ocupe el espacio de la región a eliminar.

- Regla 4: si está el flag habilitado y se solicita crear la misma región, se reinicia la región pendiente pisando los datos del header y se elimina el área de datos de la región (se pone en *zero*). Finalmente se deshabilita el flag.

- Regla 5: si está el flag habilitado y se solicita eliminar alguna región, primero se deberá eliminar la región pendiente y marcar como pendiente la nueva región a eliminar por regla 1 (caso análogo a la regla 3 de la optimización de creación de región).

8.2.4. ResizeRegionsAllocator

ResizeRegionsAllocator presenta la misma interfaz que el allocator anterior, por lo que pueden ser reemplazados fácilmente. A diferencia de *RegionsAllocator*, este soporta la extensión del tamaño de las regiones cuando no es suficiente para llevar a cabo la asignación de un objeto en alguna de ellas. También utiliza *RegionsPageResource* como administrador de páginas ya que se conservará la misma metodología para la reserva y liberación de las mismas.

Como es posible que una región no posea el tamaño suficiente se debe tener un mecanismo para extenderla mediante nuevos fragmentos de región. Se presentan dos casos:

- Si se trata de la última región es posible pedir más espacio, y si éste es contiguo se agrega al final.

- Si está limitada por otra región (no es la última en la pila), se debe crear una extensión de la región al final de la pila y apuntarla desde el fragmento de región original. Este proceso de extensión puede que ocurra sucesivas veces durante la vida de la región.

8.2.4.1. Estructura

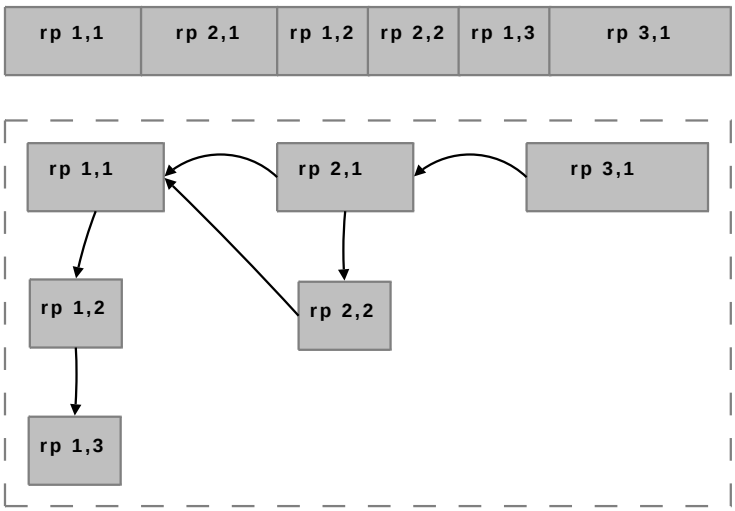

Figura 8.6: ResizeRegionsAllocator. Estructura.

Llamaremos a los fragmentos de región "parte de región" (*region part -rp -*). Cada región está conformada por una lista encadenada de una o más *partes* $[rp_{i,1}..rp_{i,j}..rp_{i,n}]$ siendo $rp_{i,j}$ la parte j de la región i con $1 \leq j \leq n$.

Dado que se utilizará también la idea de estructura de pila de regiones, para poder soportar las partes de regiones se debe definir una lista para cada una de ellas. Si se pudiera imaginar la estructura resultante en la memoria física, sería la de una lista encadenada de partes de regiones asignadas en lo posibles a rangos contiguos de memoria.

Características de las partes de región:

1. Dadas dos partes pertenecientes a la misma región $rp_{i,k}$ y $rp_{i,h}$ con $k < h$, la primera deberá crearse antes que la segunda, mientras que su posicionamiento en la pila no dependerá de eso. (Debido a la expansibilidad de las regiones siempre en el tope de la pila).

2. Dadas dos primeras partes de región $rp_{r,1}$ y $rp_{s,1}$ con $r < s$. $rp_{r,1}$ debe ser creada con anterioridad temporal y espacial (antes en la pila) que $rp_{s,1}$. Notar que sólo se cumple para la primera parte de región, las demás no contemplan un orden.

Como puede observarse, no es posible obtener un modelo de región contigua sin mecanismos de reagrupamiento y defragmentación (como en RegionsAllocator) y esto también influye en que los métodos de creación y eliminación de regiones no sean simétricos, dado que la asignación de objetos puede alterar la estructura de las regiones.

Por ejemplo una disposición posible de regiones en memoria podría ser la siguiente:

$[rp_{1,1}, rp_{2,1}, rp_{1,2}, rp_{2,2}, rp_{1,3}]$ donde la región $r1$ está formada por $[rp_{1,1}, rp_{1,2}, rp_{1,3}]$ y $r2$ por $[rp_{2,1}, rp_{2,2}]$.

Se puede observar que se cumplen las reglas 1 y 2. Si por ejemplo se requiere eliminar $r2$, que es la última región creada (no la última parte de región), se deberían eliminar $rp_{2,1}$ y $rp_{2,2}$ quedando dos extensiones de memoria inutilizadas en el espacio ocupado por ellas. Además no se podrían actualizar los punteros *cursor* o *pointer* dado que ninguna parte de $r2$ ocupa la posición más alta de memoria. Siguiendo con este mecanismo, si se quiere volver a crear $r2$, debería hacerse al final de la pila.

8.2.4.2. Implementación

Este allocator comparte casi los mismos datos internos que *RegionsAllocator*, incluyendo la misma la administración de páginas con sus punteros *cursor* y *limit*, y los *offsets* correspondientes para diagramar la memoria.

Si se retoma el ejemplo anterior, puede observarse que la disposición de las partes de regiones resultante tras haber eliminado una región, no se ajusta a los parámetros de un manejo de memoria eficiente, por lo que se debe pensar en un mecanismo de creación y eliminación de regiones diferente.

Este problema toma gran importancia sobre todo si en las posiciones de memoria más altas se hallan partes pertenecientes a regiones con índice pequeño (últimas en la pila de regiones), con lo cual no se liberaría la memoria hasta que hayan sido eliminadas todas las regiones anteriores. Y si utilizamos el algoritmo de creación de regiones de *RegionsAllocator* estaríamos desperdiciando espacio al crear regiones siempre a partir de *cursor*.

Existen varias formas de resolver este problema, desde soluciones que comprenden mover objetos para compactar el espacio, rearmar las regiones de forma que sus partes sean contiguas, o simplemente tener un control sobre los espacios libres disponiéndolos para futuras asignaciones (no asegura contigüidad total como la defragmentación).

Las dos primeras alternativas serían interesantes de aplicar para poder verificar los tiempos de respuesta a los algoritmos de mantenimiento, pero escapa al alcance de esta tesis. Se optó por implementar la lista de espacios libres. Si bien la alternativa de apuntar a espacios vacíos no es del todo óptima en reaprovechamiento del espacio, su mantenimiento es medianamente simple y cumple con el objetivo principal de no seguir consumiendo más recursos de memoria.

Se mantendrá dicha lista en el allocator y será definida como *freeList*. Cuando se realice un pedido para crear una nueva región o sea necesario extenderla (crear una nueva región es crear la parte $r_{i,1}$), primero se verificará la existencia de un espacio libre de tamaño suficiente, si no se halla, se deberá crear al tope de la pila, verificando el espacio disponible al igual que en allocator anterior. Esta lista hace que no se cumpla la restricción espacial del ítem 2.

Para poder operar sobre toda la estructura de regiones el allocator posee un puntero a la primer parte de la última región ($r_{last,1}$) denominada *lastRegion*. Cada parte conoce a la primera parte de su región predecesora y también a la siguiente parte de la región a la que pertenece, por lo tanto, desde *lastRegion* es posible acceder a cualquier parte dentro de la estructura.

Otra diferencia con RegionsAllocator es que en el header no se guarda el indicador de *slow-path*. El almacenar esta información en la parte de región carece de sentido ya que el orden en que se eliminan no depende del orden según su posicionamiento en memoria. Se utilizará una estructura

externa para almacenar esta información.

Estructura de una parte de región en memoria La estructura de una parte es muy parecida a la estructura de una región de *RegionsAllocator*. Está compuesta por un *header* y un área contigua de datos. La única diferencia con la región es que se mantiene un puntero a la próxima parte de región.

El header lo componen (ver figura 8.7):

- `PRE` (*Part Region End* Pointer): Dirección de memoria que indica el fin de la parte de región.
- `RID` (*Region ID*): número entero que identifica la región a la que pertenece.
- `OWN` (*Owner*): número entero que identifica al dueño de la región (puede ser nulo). Tiene el mismo significado que en *RegionsAllocator*.
- `STA` (*Statistics*): está compuesta por dos enteros. Llevan el conteo de objetos albergados y el tamaño actual de la parte de región.
- `DW` (Data Write Pointer): dirección de memoria que apunta a la primer dirección libre en el área de datos. Mismo significado en que *RegionsAllocator* pero aplicado a la parte de región.
- `PR` (*Previous Region Pointer*): puntero a la primer parte de región de la región anterior. Puede ser *zero* en el caso que la región en cuestión sea la primer región en memoria.
- `NPR` (*Next Part Region Pointer*): puntero a la próxima parte de región. Su valor es *zero* si es la última parte.

La otra parte que comprende una parte es el espacio de datos, que posee el mismo comportamiento que en el allocator anterior.

En la figura 8.7 se puede observar la estructura de una parte de región. La región anterior se la representó mediante un recuadro oscuro. Hay que tener en cuenta puede estar conformada por múltiples partes no contiguas. La siguiente parte de región también posee la misma estructura que la actual, se replica la información del header para tener conocimiento de la región en cada parte sin tener que recorrer la estructura (id de región, owner, y puntero a región anterior). El resto de los datos del header son propios de cada parte.

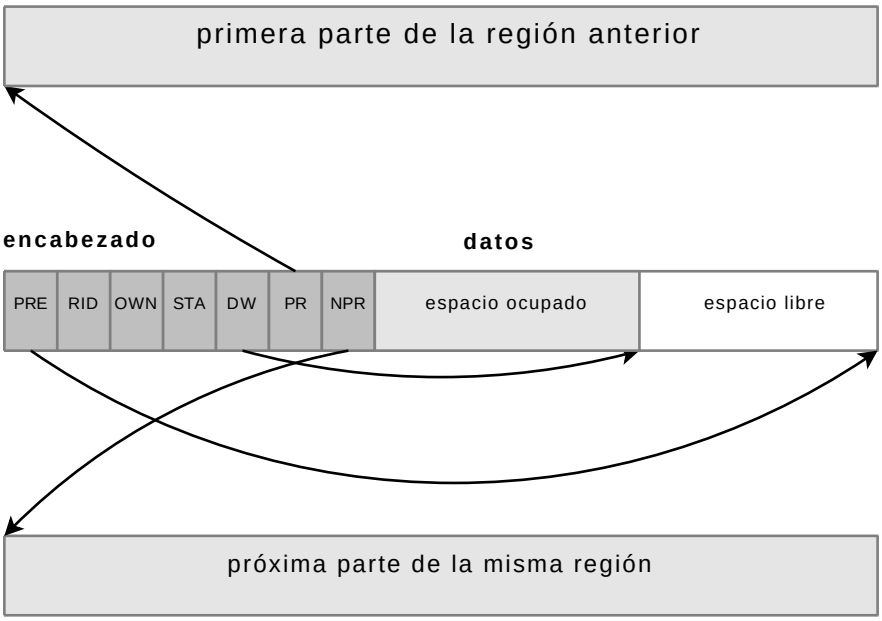

Figura 8.7: ResizeRegionsAllocator: parte de región

Algoritmos de creación y eliminación de regiones y asignación de objetos. El allocator contiene en memoria los siguientes objetos para poder realizar todas las operaciones sobre las regiones.

Constantes del allocator

- `Offset int RID_OFFSET`: desplazamiento del identificador de la región dentro del header.
- `Offset OWN_OFFSET`: desplazamiento del identificador owner dentro del header.
- `Offset PR_OFFSET`: desplazamiento del puntero a la región anterior dentro del header.
- `Offset RE_OFFSET`: desplazamiento del puntero fin de región dentro del header.
- `Offset DW_OFFSET`: desplazamiento del puntero de escritura de objetos dentro del header.
- `Offset STA_OFFSET`: desplazamiento del área de estadísticas dentro del header.
- `Offset NRP_OFFSET`: desplazamiento en el header al puntero a la próxima parte de región.
- `Offset REGION_HEADER_SIZE`: indicar el tamaño del header. Su valor es la suma de los desplazamientos anteriores.
- `int REGION_EXTENSION_SIZE`: tamaño en bytes que se utiliza al extender una región.
- `int REGION_EXTENSION_SIZE`: tamaño por defecto que tomarán las partes de región.

Información que mantiene el allocator

- `RegionsPageResource regionsPageResource`: instancia de *RegionsPageResource*. Se utilizará para pedir y liberar páginas de memoria.
- `freeList`: lista de direcciones de memorias y tamaños que representan lugares libres para creación de regiones en el espacio de memoria disponible. La estructura de un espacio libre es *<Address address, int size>* donde *address* identifica a la posición inicial del *gap* y *size* la extensión del mismo en bytes. Al inicio de la ejecución la lista está vacía, recién al eliminarse una región, si dejó *gaps* en el espacio disponible, dichos espacios son ingresados a la lista.

  Tiene los siguientes métodos:

  - `getSpace(int size)`: recibe un tamaño en bytes y devuelve una dirección de memoria que cumpla con el pedido (dicho espacio es removido de la lista. Si no es ocupado en su totalidad se genera un nuevo espacio con el tamaño restante). Si no hay un espacio disponible se devuelve *zero*.
  - `setSpace(Address address, Offset size)`: agrega a la lista de espacios disponibles el rango de memoria desde la dirección *address* hasta $address + size$. En caso de agregarse un espacio que limita con otro, se unen.
  - `freeSpace(Address from)`: elimina de la lista los espacios de memoria que no cumplen la condición $address + size \leq from$.
- `requestList`: lista de direcciones de memoria que registran pedidos de páginas a *RegionsPageResource* (*slow-path* requests)
- `Address cursor`: puntero a escritura de parte de región.
- `Address limit`: puntero centinela a la última dirección de memoria disponible en el allocator.
- `Address lastRegion`: puntero a la última región.
- `Address topRegionPart`: puntero a la parte de región que ocupa la posición de memoria más alta (no necesariamente pertenece a la última región).

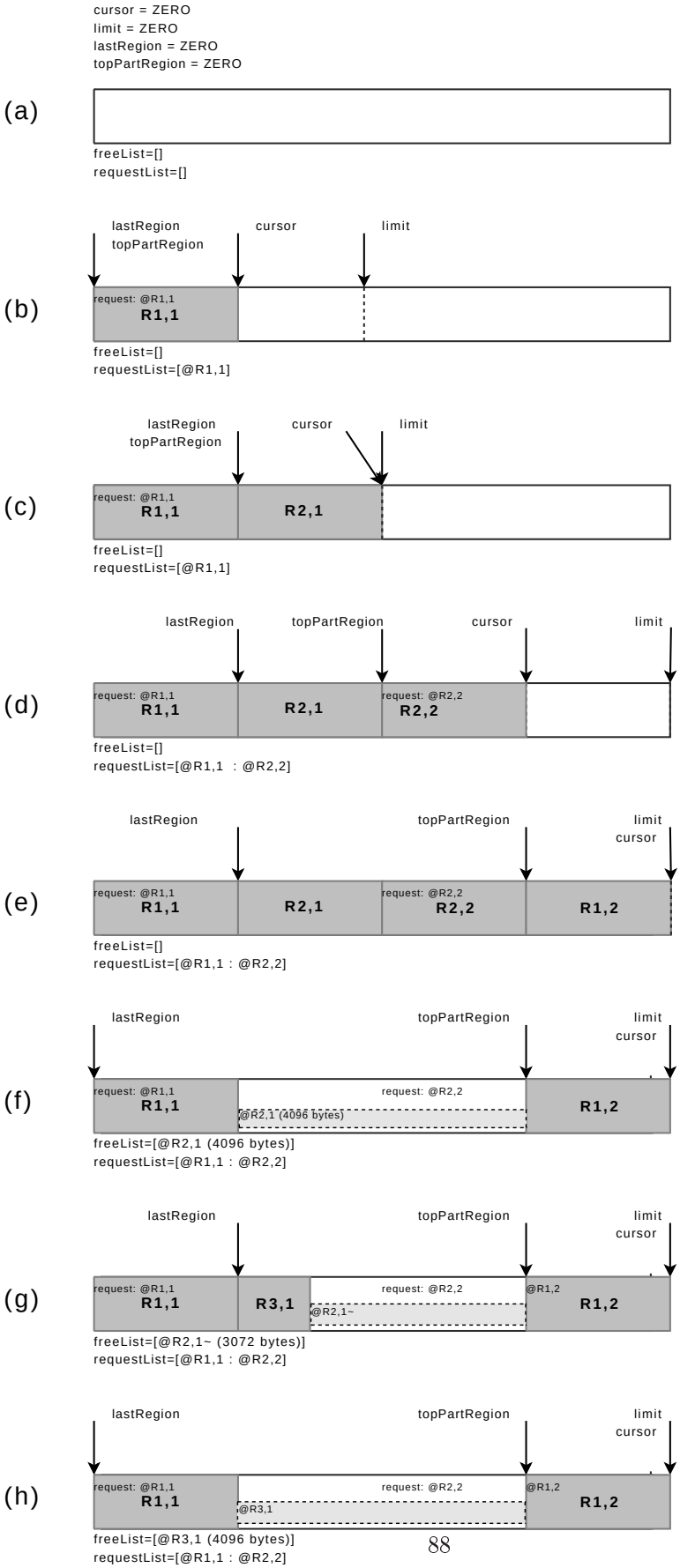

Figura 8.8: RegionsResizeAllocator: dinámica de creación y eliminación de regiones, y asignación de objetos.

Creación de una región Para crear una región se deberá crear la primera parte de la misma. Una vez obtenida una parte de región se deberá setear en el header el puntero a la región anterior y se inicializará en *zero* el puntero a la próxima parte de región. El crear la primer parte de región implica casi los mismos pasos que crear una región en *RegionsAllocator*. La diferencia más sustancial es la verificación de espacio disponible en *freeList,* y que los pedidos al administrador de páginas se registran en *requestList* en vez de en el header de la región. Adicionalmente se mantiene el puntero *topRegionPart* a la parte de región con la dirección más alta en memoria.

- Crear $rp_{regionId,1}$
  - Verificar espacio en *freeList,* ó crear en el espacio reservado del allocator, ó reservar espacio de *RegionsPageResource* (en este caso agregar dirección a *requestList*).
  - Inicializar header de la parte de región.
  - Si no es la primer región, apuntar a la región anterior.
- Apuntar *lastRegion* a $rp_{regionId,1}$

Algorithm 8.5 RegionsResizeAllocator: creación de una región.

```
public final int createRegion(int identifier, int owner, int size) {

      Address start = createRegionPart(identifier, owner, size)
      // seteo puntero a región anterior
      (start + PR_OFFSET).store(this.lastRegion)
      (start + NRP_OFFSET).store(Address.zero())
      this.lastRegion = start

}
private Address createPartRegion(identifier,owner,size){

      VM.assertions._assert(size > 0, "Negative region size")
      size += REGION_HEADER_OFFSET.toInt()
      Address end
      Address start = FreeList.getSpace(size)
      if(start==Address.zero()){
           start = this.cursor
           end = start.plus(size)
           boolean slowPath = !end.LE(this.limit)
           if (slowPath) {
                int pages = memoryRegionSize/PAGE_SIZE
                // Realizo pedido de páginas al RegionsPageResource
                Address temp = regionsPageResource.allocPages(pages)
                requestList.append(temp)
                // verifico que el pedido fue exitoso
                assert(!start.isZero(),"Out of memory")
                if(start.NE(limit)){
                   start = temp
                   end = start.plus(size)
                }
                // actualizo el limite
                this.limit = start + pages * PAGE_SIZE
           }
      }
      //si cursor es menor que el fin de la nueva parte, lo actualizo
      if(this.cursor.LT(end)){
           this.cursor = end
      }
      if(this.topRegionPart<start){
           // si la nueva parte está al tope de la memoria actualizo topRegionPart
           this.topRegionPart = start
      }
      start.store(RID_OFFSET,regionID) // regionID
      start.store(OWN_OFFSET,ownerID) // ownerID
      start.store(PRE_OFFSET,end) // partRegionEnd
      start.store(DW_OFFSET,start + REGION_HEADER_SIZE) // Data Writer
      start.store(STA_OFFSET,(0,0)) // Statistics

}
```

Orden algorítmico: $\mathcal{O}$(cantidad de entradas en *freeList*)

Eliminación de una región Para eliminar una región deberemos eliminar todas sus partes y apuntar como *lastRegion* a la región previa. En este allocator puede no cumplirse que la región a eliminar esté en las últimas posiciones de memoria, por lo que no es posible liberar memoria de *RegionsPageResource* en todos los casos. Al eliminar cada una de sus partes se deberá verificar si el fin de la región se corresponde con *cursor*. En este caso se podrá liberar memoria local del allocator para poder crear nuevas regiones en posiciones más bajas y analizar la posibilidad de liberar páginas de memoria verificando los pedidos de *slow-path* en *requestList* (liberación de pedidos que queden por encima de *cursor*). En el caso que la parte de región a eliminar sea "interna", se deberá registrar el espacio en *freeList*.

- Para cada $rp_{lastRegionId,i}$

    - Si es interna agragar el espacio a *freeList*. Sino actualizar *cursor*.

- Liberar páginas de *requestList* por encima de *cursor* y actualizar *limit*.
- Apuntar *lastRegion* a la primer parte de la región anterior.

Algorithm 8.6 RegionsResizeAllocator: eliminación de una región.

```
public final void removeTopMemoryRegion() {

      Address toDelete = this.lastRegion
      lastRegion = (lastRegion + PR_OFFSET).loadAddress()
      while(toDelete!=(Address.zero())){
           Address current = toDelete
           toDelete = (toDelete + NRP_OFFSET).loadAddress()
           if(this.topRegionPart==current){

                //obtengo la nueva parte de región que quedará en el tope de la memoria
                this.topRegionPart = getSecondTopPart()
                //actualizo cursor al la dirección de memoria final de la parte de región
                //que quedará en el tope de la memoria
                cursor = this.topRegionPart.loadAddress(RE_OFFSET)
                releaseAboveAddresses(this.topRegionPart)
           }
           else{
                deleteToOffset = current.loadAddress(RE_OFFSET) - current
                freeList.setSpace(toDelete, deleteToOffset)
                VM.memory.zeroFromTo(current, deleteToOffset)
           }
      }
}
private Address getSecondTopPart(){

      Address previous = Address.zero()
      Address regionFirstPart = lastRegion
      while(regionFirstPart!=Address.zero()){
           regionPart = regionFirstPart
           while(regionPart!=Address.zero()){
                if(regionPart!=this.topRegionPart && regionPart>previous){
                   //si existe una parte de región más alta que previous,
                   //y no es topRegionPart, reasigno previous con la nueva candidata
                   previous  = regionPart
                }
                //avanzo a siguiente parte de la región
                Address regionPart = (regionPart + NRP_OFFSET).loadAddress()
           }
           //paso a la región anterior
           regionFirstPart = (regionFirstPart + PR_OFFSET).loadAddress()
      }
      return previous

}
private void releaseAboveAddresses(Address current){

      previous = Address.zero()
      for(int i=0; i<requestList.length(); i++){
           Address address = requestList.get(i)
           if (address >= current){
                if (previous==Address.zero() || address < previous){
                   previous = address
                   regionsPageResource.release(address)
                   freeList.freeSpace(address)
                   limit = address
                   if (cursor > limit)
                      cursor=limit
                }
                requestList.remove(address)
           }
      }

}
```

$\mathcal{O}(getSecondTopPart)$: $\mathcal{O}$(regiones en la pila * partes de región).
$\mathcal{O}(releaseAboveAddresses)$: $\mathcal{O}$(direcciónes en requestList).
Orden algorítmico: $\mathcal{O}$(partes de región $*$ ($\mathcal{O}(getSecondTopPart) + \mathcal{O}(releaseAboveAddresses)$)).

Asignación de objetos en una región La asignación de los objetos (ver algoritmo 8.7) sucede en la última parte de la región especificada. El método *getMemoryRegionById* devuelve la posición inicial de la última parte de región con el *id* indicado. Cuando hay suficiente espacio en la región de datos de la parte de región, la asignación se ejecuta al igual que en *RegionsAllocator*. Se alinea la posición de asignación a partir del puntero de escritura de datos DW (de a cuerdo a *align* y *offset*), y finalmente se actualiza el valor de DW. Cuando el espacio no es el suficiente ($endData > endAddress$) se debe crear una nueva parte de región. La extensión de regiones tiene un tamaño mínimo especificado con la constante $REGION_EXTENSION_$SIZE. Si el objeto a alojar supera dicho tamaño, la región toma el tamaño del objeto. El método *createPartRegion* es similar al que se utiliza en la creación de regiones, la diferencia radica en que éste soporta la unión de las partes en el caso en que la extensión sea contigua con la región. De esta forma sólo se tendría que actualizar el tamaño de la región actual sin generar una nueva. Si la parte no es contigua se escribe el header duplicando la información de la región y se conectan las partes mediante el puntero a próxima parte de región *NRP*.

- Buscar $rp_{regionId,last}$ (última parte de la región $regionId$).
- Si el objeto cabe en la parte de región se lo asigna y actualiza el puntero DW.
- Si no hay suficiente espacio se intenta crear la parte de región $rp_{regionId,last+1}$
  - Verificar espacio en *freeList,* ó crear en el espacio reservado del allocator, ó reservar espacio de *RegionsPageResource* (en este caso agregar dirección a *requestList*).
  - Si $rp_{regionId,last}$ es contigua con $rp_{regionId,last+1}$ se hace un merge de la parte en $rp_{regionId,last}$. Sino se inicializa el header de la nueva parte de región y se las vincula.
  - Se vuelve a intentar el *alloc* sobre la rp resultante.

Algorithm 8.7 RegionsResizeAllocator: asignación de objetos en una región.

```
public final Address allocBump(int memoryRegionId, int bytes, int align,int offset) {

      Address startRegion = getMemoryRegionById(memoryRegionId)
      Address endAddress = (startRegion + MEMORY_REGION_END_OFFSET).loadAddress()
      Address writeCursor = (startRegion + MEMORY_REGION_BUMP_CURSOR_OFFSET).loadAddress()
      Address startData = alignAllocationNoFill(writeCursor, align, offset)
      Address endData = startData.plus(bytes)
      if (endData>endAddress) {
           if(createPartRegion(startRegion,
                bytes>REGION_EXTENSION_SIZE?bytes:REGION_EXTENSION_SIZE)
                   !=INVALID_REGION_ID){
                return allocBump(memoryRegionId,bytes, align, offset)
           }
           fillAlignmentGap(bCursor, startData)
           bCursor = endData
           (startRegion + MEMORY_REGION_BUMP_CURSOR_OFFSET).store(endData)
           (startRegion + MEMORY_REGION_STATISTICS_OFFSET).store(
                (startRegion + MEMORY_REGION_STATISTICS_OFFSET).loadInt() + 1)
           return startData
      }
      fillAlignmentGap(writeCursor, startData)
      // actualizo punteros internos de la región
      (startRegion + MEMORY_REGION_BUMP_CURSOR_OFFSET).store(endData)
      (startRegion + MEMORY_REGION_STATISTICS_OFFSET).store((startRegion +
                          MEMORY_REGION_STATISTICS_OFFSET).loadInt() + 1)
      return startData

}
private Address createPartRegion(previousPartAddress,size){

      VM.assertions._assert(size > 0, "Negative region size")
      size += REGION_HEADER_OFFSET.toInt()
      Address end
      Address start = FreeList.getSpace(size)
      if(start==Address.zero()){
           start = this.cursor
           end = start.plus(size)
           boolean slowPath = !end.LE(this.limit)
           if (slowPath) {
                int pages = Math.ceil(memoryRegionSize / PAGE_SIZE)
                // Realizo pedido de páginas al RegionsPageResource
                Address temp = regionsPageResource.allocPages(pages)
                requestList.append(temp)
                // verifico si el pedido fue exitoso
                assert(!start.isZero(),"Out of memory")
                if(start.NE(limit)){
                   start = temp
                   end = start.plus(size)
                }
                // actualizo el limite
                this.limit = start + pages * PAGE_SIZE
           }
      }
      //si cursor es menor que el fin de la nueva parte, lo actualizo
      if(this.cursor.LT(end)){
           this.cursor = end
      }
      if(start==previousPartAddress.loadAddress(RE_OFFSET)){
           //las partes son contiguas, se deben mergear.
           // solo cambio el limite de región y salgo.
           previousPartAddress.store(end, RE_OFFSET)
           return INVALID_REGION_ID
      }
      if(this.topRegionPart<start){
           // si la nueva parte está al tope de la memoria actualizo topRegionPart
           this.topRegionPart = start
      }
      (previousPartAddress +NP_OFFSET).store(start) // seteo puntero a la próxima región
      start.store(previousPartAddress.loadAddress(PR_OFFSET),PR_OFFSET) // replico región anterior
      start.store(RID_OFFSET,previousPartAddress.loadInt(RID_OFFSET)) // regionID
      start.store(OWN_OFFSET,previousPartAddress.loadInt(OWN_OFFSET)) // ownerID
      start.store(PRE_OFFSET,end) // partRegionEnd
      start.store(DW_OFFSET,start + REGION_HEADER_SIZE) // Data Writer
      start.store(STA_OFFSET,(0,0)) // Statistics
}
```


Orden algorítmico: $\mathcal{O}$(cantidad de regiones en la pila + cantidad de entradas en *freeList*)

8.2.4.3. Ejemplo

Supongamos el caso en que se quieran crear tres regiones y alojar objetos en ellas siguiendo el siguiente orden:

1. Crear *region1* con tamaño 2048 bytes.
2. Crear un objeto de 1000 bytes en *region1*.
3. Crear *region2* con tamaño 2048 bytes.
4. Crear en *region2* dos arrays de 2000 bytes.
5. Crear en *región1* un array de 2000 bytes.
6. Eliminar *region2*.
7. Crear *region3* con tamaño 1024 bytes.
8. Eliminar *region3*.
9. Eliminar *región1*.

El estado inicial del allocator - figura 8.8(a) - es el siguiente: *limit* y *cursor* en *zero*, ya que todavía no se han reservado páginas de memoria, el puntero *lastRegion* también en *zero*, y *freeList* vacía, ya que todavía no existe fragmentación del espacio.

1- Siguiendo el algoritmo de creación de regiones, al crear *region1* (figura (b)) se invoca a *createRegionPart*, que retornará la dirección de memoria inicial de la primer parte de región de *region1* (denominada $r_{1,1}$), una vez creada la parte de región se debe actualizar a la región previa (PR_OFFSET) mediante el puntero *lastRegion* (*zero*), y se setea en *zero* el puntero a la próxima parte de región. Finalmente se actualiza el valor de *lastRegion* apuntando a $r_{1,1}$.

El método para crear una parte de región es muy parecido al de crear una región en *RegionsAllocator*. Primero se debe calcular el tamaño real de la parte de región sumando el espacio que consumirá el header, luego se verifica en *freeList* la existencia de algún *gap* que cumpla el requerimiento de memoria. En el caso de $r_{1,1}$, como la lista está vacía se devolverá *zero* y se intentará crear en la dirección donde apunta *cursor*. Al igual que en RegionsAllocator, cuando sea necesario se deberá invocar a *RegionsPageResource* quien devolverá las páginas correspondientes de memoria para poder cumplir el pedido. Como $r_{1,1}$ es la primer parte de región y *limit* es *zero* se deberá llevar a cabo la asignación mediante *slow-path* finalmente actualizando el valor de *limit* y registrando la dirección obtenida en *requestList*. Luego se verificará la posición inicial de asignación (*start*) según la contigüidad del espacio obtenido y se deberá actualizar el puntero $lastRegion = start$. La información del *header* se almacenará al igual que en *RegionsAllocator*.

2 - En este paso se debe crear un objeto de 1000 bytes. Como la parte de región inicial tiene suficiente espacio disponible, la asignación se llevará a cabo en forma convencional: se obtiene *region1* y se aloca el objeto en la posición donde apunta *DW*, quedando 1048 bytes libres.

3 - En el ítem (c) puede verse el resultado de crear *region2*. Como *freeList* está vacía, se deberá posicionar en la dirección apuntada por *cursor*, pero esta vez no hará falta reservar nuevas

páginas de memoria ya que el espacio a ocupar es menor que *limit (fast-path allocation)*. El puntero *lastRegion* deberá apuntar a la dirección de memoria inicial de $r_{2,1}$.

4 - Como se deben crear dos objetos de 2000 bytes cada uno y el espacio de *region2* es de 2048, sólo se podrá asociar un objeto en la parte $r_{2,1}$ teniendo que extender la región con la parta $r_{2,2}$ para satisfacer la asignación del segundo array.

La creación del primer array sucede de manera convencional en $r_{2,1}$ mientras que al intentar asociar el segundo array no se cumple la condición $endAddress \leq endData$, por lo que se intenta extender la región con $r_{2,2}$. Al crear $r_{2,2}$ hará falta pedir más páginas de memoria a *RegionsPageResource* ya que no hay ningún espacio disponible en *freeList* y tampoco alcanza el espacio entre *cursor* y *limit*. La dirección de memoria obtenida se registrará en *requestList*.

Al obtener el espacio y verificar que no es contiguo con $r_{2,1}$ y no es posible extender $r_{2,1}$, se inicializa la nueva parte $r_{2,2}$ con la información correspondiente a la región y luego se guarda el puntero del header *NRP* de $r_{2,1}$ que deberá apuntar a $r_{2,2}$. Ver ítem (d) de la figura.

5 - Hasta el momento *region1* está conformado por $r_{1,1}$ y el espacio restante es de 1048 bytes, por lo que al crear un objeto de 2000 bytes hará falta extender la región. Al igual que en los casos anteriores la creación de la parte $r_{1,2}$ se realizará en cursor ya que freeList sigue estando vacía. Como la parte de región ocupa hasta limit (teniendo en cuenta que $REGION\_EXTENSION\_SIZE = 2048$) no hará falta reservar nuevas páginas de memoria, por lo que *requestList* queda inalterada.

6 - Para eliminar *region2* (*lastRegion*) se deberán eliminar las partes $r_{2,1}$ y $r_{2,2}$ simplemente borrando el espacio ocupado en memoria y registrando los *gaps* obtenidos en *freeList*, que serán unidos en uno solo (por ser contiguos). No se llevará a cabo modificaciones en los punteros *cursor* o *limit* ya que son espacios de memoria internos (no ocupan la posición de memoria más alta entre las demás partes). En el ítem (f) de la figura puede observarse el *gap* registrado en *freeList* como $@r_{2,1}$ cuyo tamaño es de $size(r_{2,1}) + size(r_{2,2}) = 4096$.

7 - La creación de region3 de 1024 bytes se diferencia de las anteriores. Al verificar espacios disponibles en freeList se hallará el liberado en el paso 6. El método `freeList.getSpace(1024)` devolverá y desregistrará el espacio $(@r_{2,1}, 4096)$ generando uno nuevo $(@r_{2,1} + Offset(1024), 4096 - 1024)$ que será llamado $@r_{2,1}$. Ver ítem (g).

8 - En el ítem (h) se muestra la configuración de memoria luego de eliminar region3. La eliminación de la parte $r_{3,1}$ se efectúa de la mima manera que como se eliminaron las partes de region2. Se borra la el espacio de memoria y se registra en freeList $@r_{3,1}$ uniendo el espacio libre con $@r_{2,1}$.

9 - Finalmente queda por eliminar region1 conformado por$r_{1,1}$ y $r_{1,2}$. El proceso de eliminación de $r_{1,1}$ es similar que en los casos previos, pero al eliminar $r_{1,2}$ se deberá actualizar *cursor* ya que apunta al final de la región y proceder a la liberación de los espacios libres mediante el método y devolver las páginas tomadas al administrador de páginas quedando finalmente la misma configuración de memoria que en el punto (a).

8.2.4.4. Optimizaciones

Uso de región activa sobre región inmortal Además de las optimizaciones utilizadas en *RegionsAllocator*, como este allocator es muy probable que sea utilizado sobre planes con un solo

espacio de memoria, la primer región será utilizada como espacio inmortal y tendrá una tasa alta de accesos. Al definirla también como región activa se baja el orden de acceso a $\mathcal{O}(1)$ en vez de $\mathcal{O}(n)$.

8.2.5. Gaps

La adquisición de extensiones de memoria no contiguas genera huecos *(gaps)*, que son porciones de memoria pedida que han quedado sin ocupar. En el caso de RegionsAllocator los *gaps* no deberían ser comunes o tener un tamaño "grande", dado que es esperable que la memoria administrada por el RegionsPageResource sea contigua en la mayoría de los casos.

8.2.5.1. Tipos de Gaps posibles en RegionsAllocator

- Fragmentación interna: es el espacio de datos de la región que nunca es ocupado. Se genera por la sobreestimación de su tamaño, y depende de la precisión de las herramientas automáticas de estimación de consumo.
- Generados por discontinuidad de chunks: son los generados por la adquisición de rangos de memoria discontiguos cuando una región no puede ser asignada por *fast-path* (la dirección de memoria retornada por RegionsPageResource no es contigua con *limit*).

8.2.5.2. Tipos de Gaps posibles en *RegionsResizeAllocator*

- Fragmentación interna: Ídem *RegionsAllocator* con el agregado de que cuando una región consume todo su espacio, se genera una nueva parte de región cuyo tamaño puede ser excesivamente grande. Una solución puede llegar a ser implementar un mecanismo adaptativo que mantenga estadísticas sobre la variación del tamaño de las regiones en runtime.
- Generados por discontiguidad de chunks: la generación de *gaps* de este tipo son más comunes que en RegionsAllocator debido al frecuente pedido de páginas al *RegionsPageResource* por no conocer de antemano el tamaño de la región.
- Generados por fragmentación de la memoria: la eliminación de partes de región genera huecos, que si bien pueden ser reasignados por medio de la freeList, es prácticamente imposible asegurar la contigüidad sin un mecanismo de compactación.

Capítulo 9

Estadísticas

El implementar los nuevos sistemas de memoria requiere poder evaluarlos y compararlos entre ellos y con los modelos preexistentes. Entre los datos relevantes a contemplar están el tiempo total de ejecución de los programas, el tiempo consumido por la administración de memoria, los datos de consumo de memoria entre los diferentes espacios, y dentro del espacio de regiones, el consumo en cada una de ellas.

El cómputo de las estadísticas está distribuido en los componentes implementados de manera que cada uno almacena la información relevante a su funcionamiento. Por ejemplo, en los allocators se tiene el control sobre las asignaciones de memoria teniendo la posibilidad de mantener el conteo de objetos y tamaño que ocupan para cada región. Para tener acceso a la información recopilada se provee una interfaz común en *MM_Interface* (dado que estos valores están almacenados en memoria).

9.1. Tiempos

Se añadieron registros de consumo de tiempos para medir los siguientes aspectos de un programa:

- *Execution-time*: tiempo total transcurrido desde el inicio hasta la finalización de la ejecución de la VM. Se toma como inicio la primera asignación en memoria.
- *Main-execution-time*: tiempo de vida del *thread* Main. Representa el tiempo transcurrido por la ejecución del programa excluyendo la inicialización y la finalización de la VM.
- *Memory-time*: es el tiempo dedicado a los procesos de administración de memoria a lo largo de toda la ejecución de la VM.
- *Main-memory-time*: tiempo dedicado a los procesos de administración de memoria en el thread Main.

El disponer de estas variables nos brinda la posibilidad de aislar el tiempo transcurrido por el administrador de memoria, y comparar las proporciones de tiempos entra las distintas mediciones.

Se mantienen las siguientes restricciones entre las variables de los tiempos calculados:

$executionTime > mainExecutionTime$: el tiempo de ejecución total es superior al de la ejecución del programa.

$memoryTime > mainMemoryTime$: el tiempo dedicado para administración de memoria durante toda la ejecución es superior al utilizado durante la ejecución del thread Main.

$executionTime > memoryTime$: el tiempo total transcurrido es mayor que el consumido en la administración de la memoria.

$mainExecutionTime > mainMemoryTime$: el tiempo consumido por el thread Main es superior al que consume su administración de memoria.

Para calcular estos valores se utilizaron cuatro *timers* independientes que provee la *MMTk* (ver `org.mmtk.utility.statistics.Timer`).

Mientras que los *timers* correspondientes a *execution-time* y a *main-execution-time* son inicializados una sola vez al comienzo de la ejecución y al comienzo del thread Main respectivamente, los *timers* dedicados a la administración de memoria sólo deben estar corriendo si se está llevando una operación de memoria. Éstas pueden ser: crear una región, eliminarla, crear objetos, o la ejecución del recolector de basura. Todas estas operaciones están implementadas en el plan de memoria y los métodos correspondientes están instrumentados para tomar los tiempos.

Para visualizar los tiempos tomados se agregó la opción de línea de comandos *timeStats*, que al finalizar la ejecución imprimirá por consola los resultados.

9.2. Contadores de páginas consumidas

Para llevar un control sobre la cantidad de memoria que consume una aplicación, la RVM dispone de la clase *Space,* que posee el método *reservedPages*. Este método retorna la cantidad de páginas reservadas en un espacio y el tamaño en bytes es calculado a partir de ese valor. Cada espacio debe implementar *reservedPages* en base al estado del administrador de páginas utilizado. De esta manera es posible tener información sobre el consumo de cada espacio en particular, pudiéndolos comparar. Es de mucha utilidad en los planes con dos sistemas de administración de memoria en donde se combinan un espacio destinado al uso de regiones y otro inmortal, y se puede ver la proporción de espacio utilizado por cada uno.

Si se quisiera obtener más información de consumo dentro de un espacio de memoria, la funcionalidad aportada por Jikes no es suficiente ya que no conoce la estructura interna de cada espacio. Para llevar un conteo más específico en el espacio destinado a regiones (*RegionsSpace*) es necesario definir las estructuras de datos que llevarán el registro de las estadísticas de conteo.

Si se especifica el argumento *regionsStats=true* al ejecutar la VM, cuando finaliza la ejecución se imprimirá por consola el estado de las páginas de memoria tomadas por cada espacio que interviene en la VM.

Ej: La siguiente salida indica los consumos en los espacios involucrados en la VM cuyo plan es Regiones - Mark&Sweep. Los espacios restantes son de uso interno de la VM (Jikes RVM los incluye por defecto).

```
used = 10.44 Mb = boot 0.00 Mb + immortal 0.12 Mb + meta 0.00 Mb + los 1.63 Mb + plos 0.05 Mb

      + sanity 0.00 Mb + non-moving 0.14 Mb + sm-code 0.22 Mb + lg-code 0.01 Mb
      + ms 7.87 Mb + regions 0.37 Mb

used = 2673 pgs = boot 0 pgs + immortal 32 pgs + meta 0 pgs + los 418 pgs + plos 13 pgs

      + sanity 0 pgs + non-moving 36 pgs + sm-code 58 pgs + lg-code 3 pgs
      + ms 2017 pgs + regions 96 pgs
```

9.3. Contadores por región/método

En la estructura del header de cada región, definida en los allocators, se reservó el espacio suficiente para almacenar dos enteros para uso estadístico. La dirección de memoria que los contiene se corresponden con el desplazamiento *STA - Statistics* - dentro del header y en ellos se mantendrá el conteo de objetos y el tamaño actual en bytes de la región. De esta manera es posible saber en algún momento particular de la ejecución el consumo de memoria de la región, o el tamaño de todas las regiones (calculado recorriendo la pila de regiones).

Para calcular estos valores, en cada asignación de memoria se incrementa el valor actual y es escrito en el header de la región.

Se podrá visualizar el estado de las regiones en memoria incorporando la anotación @PrintRegionInfo al método que se desee, y asignar el valor *true* a la propiedad statistics.

La información se mostrará por consola al momento de salir de cada método imprimiendo el estado de la pila de regiones.

Por ejemplo si tengo la cadena de llamadas de métodos *m0->m1->m2* y todos tienen regiones asociadas cuyos identificadores son 1, 2 y 3 respectivamente, si anoto a *m2* con *@PrintRegionInfo(statistics=true)*, al salir de *m2* se imprimirá la siguiente información por consola:

```
REGION STATISTICS:

      regions.TestRegions.m2(I)[Lregions/B; (3) -->

            Consumed: 52 bytes
            Total consumed from region: 52 bytes
            Number of objects: 2
            Total number of objects from region: 2

      regions.TestRegions.m1(I)V (2) -->

            Consumed: 124 bytes
            Total consumed from region: 176 bytes
            Number of objects: 7
            Total number of objects from region: 9

      regions.TestRegions.m0(I)[Lregions/B; (1) -->

            Consumed: 28 bytes // es lo que ocupa el header de la región
            Total consumed from region: 204 bytes
            Number of objects: 0
            Total number of objects from region: 9

      Spoiled memory: 5205812 bytes
```

La salida muestra la pila de regiones con el resumen de consumo para cada una, desde el tope hasta la base.

Datos por región:

- *Consumed:* indica el tamaño en bytes de la porción de memoria ocupada dentro de la región ($sizeOf(encabezadoRegion) + size(objetosEnRegion)$)
- *Total consumed from region*: es la suma del tamaño consumido desde la región hasta el tope de la pila de regiones.
- *Number of objects*: cantidad de objetos creados en la región.
- *Total number of objects from region*: es la cantidad de objetos creados en las regiones comprendidas entre la región en cuestión y el tope de la pila.

La diferencia entre los dos primeros datos y los restantes dos es que los primeros están medidos en bytes, mientras que en los otros sólo se lleva el conteo de los objetos.

Como dato adicional se muestra el valor *Spoiled Memory*, que indica el tamaño de memoria reservado que todavía no ha sido utilizado (*gaps*). Se calcula como la diferencia entre el espacio reservado por el allocator y la suma de las áreas de memoria ocupadas dentro de cada una de las regiones en la pila (se tiene en cuenta el header y el área de datos utilizada hasta el momento).

$$SpoiledMemory = reservedPages - totalConsumedFrom(firstRegion)$$

Si *SpoiledMemory* alcanza durante la ejecución valores pequeños en relación al tamaño neto consumido (espacio que ocupan los objetos y headers de regiones), es un indicador del "buen uso" de los recursos de memoria, si en cambio este valor siempre se mantiene elevado, indica que que hay regiones cuyo tamaño ha sido sobre-estimado, o en el caso de *ResizeRegionsAllocator*, que se han generado muchos *gaps* de memoria (fragmentación). *SpoiledMemory* es calculado en páginas de memoria por lo que la unidad mínima es 2^{15} bytes $= 32768$ *bytes* para arquitecturas x86.

9.4. Contadores generales

Para mantener información sobre el consumo de las regiones a lo largo de toda la ejecución, almacenar los datos en el header de la región no es suficiente, para ello es necesario crear una estructura en memoria aparte. En cada allocator de memoria (*RegionsAllocator* o *ResizeRegionsAllocator*) antes de realizar la primera asignación de memoria, se crea un área destinada a almacenar las estadísticas de uso. Al asignar la memoria para esta estructura de datos deberá solicitarse al administrador de páginas el espacio suficiente, al igual que cualquier asignación.

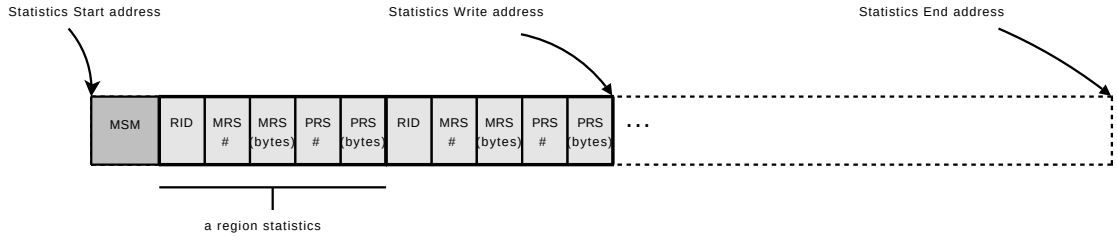

Figura 9.1: Estadísticas: estructura de datos.

En la figura 9.1 se presenta la estructura de la tabla. Los punteros *start address* y *end address* delimitan el espacio de memoria disponible y el puntero *write address* especifica el cursor de escritura para una nueva región. Al principio de la ejecución la tabla está vacía, sólo permanece especificado el registro MSM (Min Spoiled Memory) pero sin un valor. El puntero de escritura (write address) apunta al final de MSM. Recién cuando es eliminada una región por primera vez se guardan sus datos y se avanza el puntero de escritura. Cada región mantiene la siguiente información:

- `RId (Region Id)`: entero que identifica a la región.
- `Max_rsize (Max Region Size)`: es el tamaño máximo que puede tomar una región. `Max_rsize` debe actualizarse al eliminar cada región si el tamaño consumido es mayor que el valor actual. `Max_rsize` está compuesto por dos enteros, el primero lleva el conteo por cantidad de objetos en la región, mientras que el segundo mantiene el tamaño en bytes. Al finalizar la ejecución del programa este valor será la cota máxima de consumo real para una región dada. Será el tamaño con el que deberá especificarse el tamaño de región en las anotaciones Java.
- `Peak_rsize (Peak Region Size)`: lleva el valor máximo de los conteos de objetos y tamaños desde la región hasta el tope de la pila ($max(totalConsumedFrom)$). El proceso de actualización del valor es similar que en `Max_rsize`. `Peak_rsize` da una cota máxima de consumo de memoria a partir de la región en cuestión, y si se toma este valor desde la primer región (la de la base de la pila), se puede saber el consumo máximo de memoria de un programa en el espacio de regiones.

A diferencia de las estadísticas de consumo por método-región, estos valores registran los máximos obtenidos a lo largo de toda la ejecución.

Para visualizar la información correspondiente a las estadísticas de consumo en las regiones, también se debe utilizar la opción *regionsStats=true*. Los datos impresos por consola tendrán las siguiente estructura:

```
STATISTICS:

    Cantidad de objetos controlados por regiones: 12
    Region 2
    -----------
        Max. region size:
            number of allocations in region: 1
            consumed size in region: 32 bytes
        Peak region size (include child regions):
            number of allocations in region: 1
            consumed size in region: 32 bytes
    Region 1
    -----------
        Max. region size:
            number of allocations in region: 12
            consumed size in region: 400 bytes
        Peak region size (include child regions):
            number of allocations in region: 13
            consumed size in region: 432 bytes
```

Capítulo 10

Experimentación y Resultados

Se realizaron diferentes aplicaciones Java para comprobar el correcto funcionamiento de las máquinas virtuales y luego se ejecutó el conjunto de *benchmarks* JOlden [8] incorporando las anotaciones necesarias para la utilización de regiones con la ayuda de las herramienta provistas por [21]y [11]. Las pruebas sobre JOlden permitieron contrastar los resultados teóricos en el cálculo de consumo de memoria y verificar la precisión del método de medición. Finalmente se realizaron las pruebas de tiempos de ejecución y consumo total de memoria comparando las VM implementadas con la VM de Jikes original.

Las VMs evaluadas fueron: *Regiones-Inmortal*, *Regiones-Mark&Sweep* y *Regiones - Puro*, correspondientes a los planes de memoria explicados anteriormente en la sección 5.1.

10.1. Test de la máquina virtual

Los primeros tests consisten en pequeñas aplicaciones que se focalizan en determinados aspectos de la administración de memoria basada en regiones, y tienen como objetivo evaluar el correcto funcionamiento de la VM.

10.1.1. Formas de crear objetos

En este test se verifica la creación de objetos en el espacio de regiones. Para ello se creó una aplicación muy simple con las diferentes formas de asignar memoria.

Programa

```
@PrintRegionInfo(news = true,statistics=true)
@RegionDefinition(size = 172, allocations = { "main([Ljava/lang/String;)V:2,3,4,5,6" })
public static void main(String[] args) {

      int v1 = 1;  // no hay asignación
      int[] v2 = new int[2]; // se asigna 1 espacio para el array
      A v3 = new A(); // se asigna 1 espacio
      A[] v4 = new A[2]; // se asigna 1 espacio
      A[][] v5 = new A[2][3]; // se asignan 3 espacios. 1 de dimensión 2 y 2 de dimensión 3.
      String test= new String("hola mundo"); // se asigna 1 espacio
```

```
}
```

Resultados de conteo

region	max_rsize (#)	max_rsize (Bytes)	peak_rsize (#)	peak_rsize (Bytes)
main (1)	7	172	7	172

Cuadro 10.1: Contadores - test de creación de objetos de diferentes tipos.

En este programa se pueden ver las diferentes maneras de asignar memoria para los objetos. Notar que tanto en *max_rsize* como en *peak_rsize* los valores son los mismos. Esto se debe a que que es el único método del programa y por lo tanto se crea una sola región por única vez. Los valores representan el consumo real de la aplicación.

10.1.2. Selección de lugar de creación

En esta aplicación se asocian dos posibles regiones al objeto creado en m2. La primera vez (*main.2->**m1**.1->m2.1*) deberá ser alojado en la región del método *m1* y en la segunda oportunidad (***main**.3->m2.1*) en la de *main*.

$R_{cs_{m2,<1,0>}} = \{r_{main}, r_{m1}\}$

Programa

```
@RegionDefinition(size=68,allocations={"main([Ljava/lang/String;)V:1","m2()V:1"})
public static void main(String[] args){

      int[] i = new int[2]; //main.1 = cs_{main,<1,0>}
      m1(); //main.2
      m2(); //main.3

}
@RegionDefinition(size=48,allocations={"m2()V:1"})
public static void m1(){

      m2(); //m1.1

}
public static void m2(){

      int[] i = new int[2]; //m2.1 = cs_{m2,<1,0>}

}
```

Salida

```
creando region id 1 (main). Cantidad 1
creado en: 1
creando region id 2 (m1). Cantidad 2
creado en: 2
borrando region id 2 (m1). Cantidad 1
creado en: 1
borrando region id 1 (main). Cantidad 0
```

La región cuyo id es 1 corresponde a main, y la de id 2 a m1. En la salida por consola se muestra la creación de la región main (r_{main}) con la primer asignación de memoria en la región para la primer instrucción $cs_{main,<1,0>}$. Luego se crea la región de *m1* (r_{m1}) y se asigna $cs_{m2,<1,0>}$ a dicha región. La siguiente asignación de memoria en $cs_{m2,<1,0>}$ se realiza en r_{main} ya que r_{m1} ha sido eliminada de la pila.

Consumo en regiones

region	max_rsize (#)	max_rsize (Bytes)	peak_rsize (#)	peak_size (Bytes)
main (1)	2	68	2	96
m1 (2)	1	48	1	48

Cuadro 10.2: Contadores - test de selección de lugar de creación.

En este ejemplo se puede ver la memoria desperdiciada por el uso de regiones. Mientras que la suma de los *max_rsize* de ambas regiones da 68 *bytes* + 48 *bytes* = 116 *bytes* (que sería el tamaño de memoria que requiere la ejecución), el *peak_size* es 96 *bytes*, que representa el consumo real. El *peak_size* es menor que la suma del tamaño de las regiones porque cuando r_{m1} es eliminado, todavía no se llegaron a crear todos los objetos que contendrá la región r_{main}.

10.1.3. Recursión

El siguiente algoritmo presenta un caso básico de recursión sobre un único método. Este programa tiene como objetivo verificar el comportamiento de creación y eliminación de regiones con el mismo id. Se presenta una única región sobre m1 cuyo único elemento es el *StringBuilder* creado por la presencia del operador *"+"* en *m1(i-1)+1*. Esta región se deberá apilar 1000 veces y en cada una se albergará un objeto.

Programa

```
public static void main(String[] args) {

      String cadena = m1(1000);

}
@RegionDefinition(size = 44, allocations = "m1(I)Ljava/lang/String;:2")
private static String m1(int i) {

      if(i!=1)

           return m1(i-1)+i; //m1.2 = cs_{m1,<2,0>}

      return "1";

}
```

Salida

```
creando region id 1 (m1). Cantidad 1
creado en: 1
creando region id 1 (m1). Cantidad 2
creado en: 1

...
creando region id 1 (m1). Cantidad 1000
creado en: 1
borrando region id 1 (m1). Cantidad 999

...
borrando region id 1 (m1). Cantidad 1
borrando region id 1 (m1). Cantidad 0
```

Cada vez que se entra a m1 se crea la región r_{m1} y luego se crea el *StringBuilder* correspondiente al operador "+". Cuando se alcanza el caso base se procede a la eliminación de todas las regiones.

Consumo en regiones

region	max_rsize (#)	max_rsize (Bytes)	peak_rsize (#)	peak_size (Bytes)
m1 (1)	1	44	1000	44000

Cuadro 10.3: Contadores - test de recursión.

Cada región contiene sólo un objeto, sin embargo si contamos la cantidad de objetos desde la primer región deberan tenerse en cuenta el de todas las regiones apiladas 1 *objeto**1000 *regiones*=1000 *objetos*.

10.2. Validación con modelo teórico. JOlden

Los tests de JOlden servirán para analizar el consumo de las regiones y la correcta distribución de los objetos del programa dentro de las mismas. La información de regiones fue provista por el análisis realizado en [11]. Si bien la herramienta no calcula los consumos para los casos recursivos, en los ejemplos *BiSort* y *Health* se han calculado los consumos manualmente utilizando los conceptos teóricos del modelo. Dada la simplicidad del análisis de la mayoría de los programas evaluados, sólo se presentarán tres ejemplos (donde se utilizan varias regiones). Finalmente los valores obtenidos se contrastarán con los resultados estimados.

10.2.1. MST

Está compuesto por tres regiones, la principal es la vinculada con el método *main*, las restante son para el *parsing* de los argumentos del programa, y para el método *printValue*. Cada región se crea una sola vez y el tamaño máximo en la pila es de 2 regiones (sin contar la región inmortal para el caso de *Regiones - Puro*).

Comando `mst.MST -v n` (n es un valor entero positivo).

Salida (se omiten allocs)

```
creando region id 1 (mainOrig). Cantidad 1
creando region id 2 (parseCmdLine). Cantidad 2
borrando region id 2 (parseCmdLine). Cantidad 1
creando region id 3 (printValue). Cantidad 2
borrando region id 3 (printValue). Cantidad 1
borrando region id 1 (mainOrig). Cantidad 0
```

Consumo en regiones

region	max_rsize (#)	max_rsize estimado (#)	max_rsize (bytes)	peak (#)	peak estimado (#)	peak (bytes)
parseCmdLine (2)	1	2	40	1	2	40
mainOrig (1)	5052	$2 * v^2 + 2 * v + 7 = 5107$	83536	5101	$5107 + 49 = 5156$	84348
computeMST (3)	49	49	812	49	49	812

Cuadro 10.4: Contadores - MST (plan regiones) -v 50.

region	max_rsize (#)	max_rsize estimado (#)	max_rsize (bytes)	peak (#)	peak estimado (#)	peak (bytes)
parseCmdLine (2)	1	2	40	1	2	40
mainOrig (1)	11327	$2*v^2+2*v+7 = 11407$	187148	11401	11407+74 = 11481	188360
computeMST (3)	74	74	1212	74	74	1212

Cuadro 10.5: Contadores - MST (plan regiones) -v 75.

region	max_rsize (#)	max_rsize estimado (#)	max_rsize (bytes)	peak (#)	peak estimado (#)	peak (bytes)
parseCmdLine (2)	1	2	40	1	2	40
mainOrig (1)	20102	$2*v^2+3*v+7 = 20307$	332388	20201	20307+99 = 20406	334028
computeMST (3)	99	99	1640	99	99	1640

Cuadro 10.6: Contadores - MST (plan regiones) -v 100.

En las tablas se muestran los resultados tras haber corrido el programa con v = 50, 75 y 100. Las columnas “estimadas” son las calculadas por la herramienta externa, y proveen una cota superior

de consumo para que el programa pueda ser ejecutado en forma segura. Con estos datos, es posible definir el tamaño de cada región (con *max_rsize* estimado en bytes). Mientras más precisa sea la estimación, se hará un mejor uso de los recursos de memoria. A continuación se presenta un gráfico donde se muestra el error de cálculo en la región *main*.

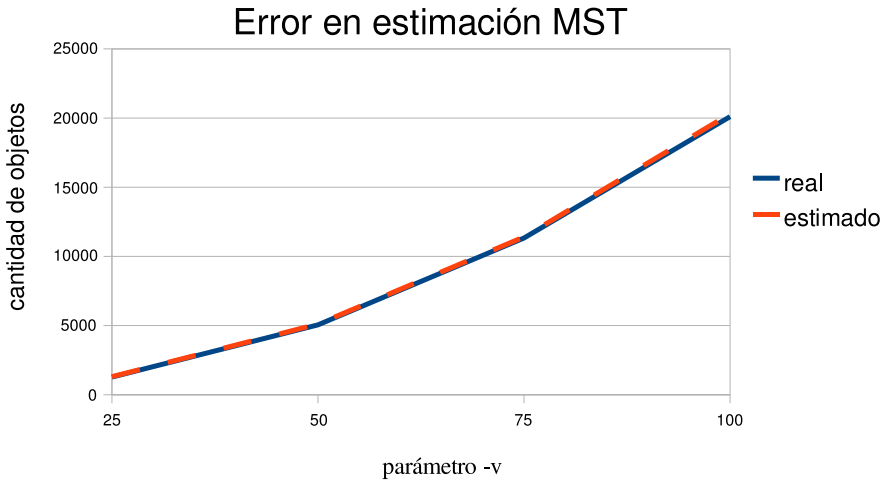

Figura 10.1: Error en estimación de la región main en MST.

10.2.2. BiSort

Este ejemplo trata de un caso recursivo donde la región principal es la correspondiente al método *main*, que alberga a la estructura del árbol binario. La región *printValue* es generada para imprimir por pantalla el valor del nodo. Este método es invocado $2 * s - 1$ veces desde el método recursivo *inOrder*, pero la presencia de *printValue* evita la generación de regiones recursivas, liberando la memoria luego de cada llamada.

Comando `bisort.BiSort -s` *size* `-p` (*size* es un valor entero positivo).

Salida (se omiten allocs)

```
creando region id 1 (mainOrig). Cantidad 1
creando region id 2 (parseCmdLine). Cantidad 2
borrando region id 2 (parseCmdLine). Cantidad 1
creando region id 3 (printValue). Cantidad 2
borrando region id 3 (printValue). Cantidad 1

...
creando region id 3 (printValue). Cantidad 2
borrando region id 3 (printValue). Cantidad 1
borrando region id 1 (mainOrig). Cantidad 0
```

Consumo en regiones

region	max_rsize (#)	max_rsize estimado (#)	max_rsize (bytes)	peak (#)	peak estimado (#)	peak (bytes)
parseCmdLine (2)	1	2	36	1	2	36
mainOrig (1)	512	$s+4 = 1004$	10260	513	$s+6 = 1006$	10300
printValue (3)	1	1	40	1	1	40

Cuadro 10.7: Contadores - BiSort (plan regiones - inmortal) -s 1000.

region	max_rsize (#)	max_rsize estimado (#)	max_rsize (bytes)	peak (#)	peak estimado (#)	peak (bytes)
parseCmdLine (2)	1	2	36	1	2	36
mainOrig (1)	1024	$s+4 = 204$	20500	1025	$s+6 = 2006$	20540
printValue (3)	1	1	40	1	1	40

Cuadro 10.8: Contadores - BiSort (plan regiones - inmortal) -s 2000.

region	max_rsize (#)	max_rsize estimado (#)	max_rsize (bytes)	peak (#)	peak estimado (#)	peak (bytes)
parseCmdLine (2)	1	2	36	1	2	36
mainOrig (1)	2048	$s+4 = 3004$	40980	2049	$s+6 = 3006$	41020
printValue (3)	1	1	40	1	1	40

Cuadro 10.9: Contadores - BiSort (plan regiones - inmortal) -s 3000.

Los resultados anteriores muestran que si bien el valor estimado por la herramienta es mayor al real, responde a un comportamiento lineal, mientras que en la ejecución del programa, la cantidad de objetos en la región *mainOrig* será la máxima potencia de 2 por debajo de s. ($2^{floor(\log_2(s))}$).

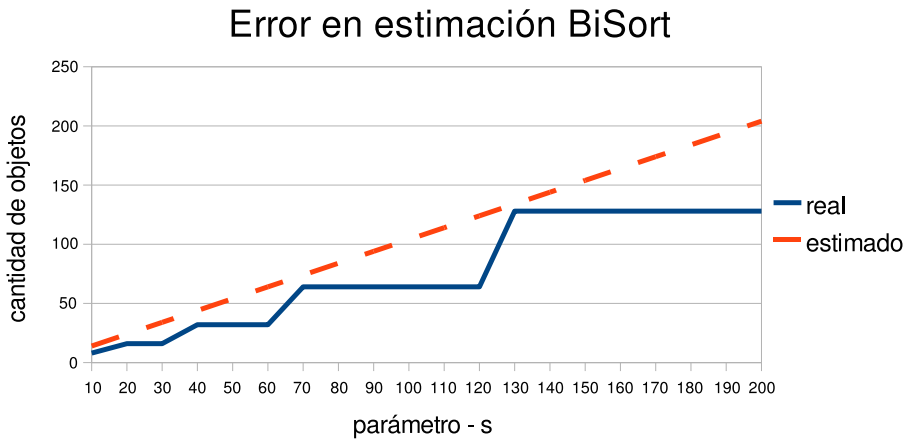

Figura 10.2: Error en estimación de la región *mainOrig* en MST.

10.2.3. Health

Este ejemplo provee un entorno de análisis interesante, ya que posee 7 regiones detectadas, 3 de las cuales albergan una cantidad variable de objetos (dependiendo de los parámetros del programa). Es un programa recursivo, donde también se presentan casos de iteraciones sobre una misma región.

Comando health.Health -l *size* -t *simulation-time* -s *seed* -p (*size, simulation-time y seed* son valores enteros. *Size* y *simulation-time* deben ser positivos).

Salida (se omiten allocs)

```
creando region id 1 (mainOrig). Cantidad 1
creando region id 3 (parseCmdLine). Cantidad 2
borrando region id 3 (parseCmdLine). Cantidad 1
ciclo (t veces)
      creando region id 4 (simulate2). Cantidad 2
      ciclo
           creando region id 5 (checkPatientsInside). Cantidad 3
           borrando region id 5 (checkPatientsInside). Cantidad 2
           creando region id 2 (checkPatientsAssess). Cantidad 3
           borrando region id 2 (checkPatientsAssess). Cantidad 2
           creando region id 6 (checkPatientsWaiting). Cantidad 3
           borrando region id 6 (checkPatientsWaiting). Cantidad 2
      fin ciclo
      borrando region id 4 (simulate2). Cantidad 1
fin ciclo
creando region id 7 (getResults2). Cantidad 2
borrando region id 7 (getResults2). Cantidad 1
borrando region id 1 (mainOrig). Cantidad 0
```

Consumo en regiones

region	max_rsize (#)	max_rsize estimado (#)	max_rsize (bytes)	peak (#)	peak (bytes)
Health.parseCmdLine (3)	3	3	64	3	64
Health.mainOrig (1)	9051	$((2*t+9)/3)*(4^l-1)+6=9895$	177068	9727	191976
Hospital.checkPatientsInside (5)	1	1	44	1	44
Village.simulate2 (4)	681	$(2/3)*(4^l-1)-1=681$	15016	682	15060
Hospital.checkPatientsAssess (2)	1	1	44	1	44
Hospital.checkPatientsWaiting (6)	1	1	44	1	44
Village.getResults2 (7)	682	$(2/3)*(4^l-1)=682$	15032	682	15032

*Cantidad de objetos controlados: 26776 objetos

Cuadro 10.10: Contadores - Health -l 5 -t 10.

region	max_rsize (#)	max_rsize estimado (#)	max_rsize (bytes)	peak (#)	peak (bytes)
Health.parseCmdLine (3)	3	3	64	3	64
Health.mainOrig (1)	36342	$((2*t+9)/3)*(4^l-1)+6=39591$	761080	39066	821072
Hospital.checkPatientsInside (5)	1	1	44	1	44
Village.simulate2 (4)	2729	$(2/3)*(4^l-1)-1=2729$	60072	2730	60116
Hospital.checkPatientsAssess (2)	1	1	44	1	44
Hospital.checkPatientsWaiting (6)	1	1	44	1	44
Village.getResults2 (7)	2730	$(2/3)*(4^l-1)=2730$	60088	2730	60088

*Cantidad de objetos controlados: 107315 objetos

Cuadro 10.11: Contadores - Health -l 6 -t 10.

region	max_rsize (#)	max_rsize estimado (#)	max_rsize (bytes)	peak (#)	peak (bytes)
Health.parseCmdLine (3)	3	3	64	3	64
Health.mainOrig (1)	147682	$((2*t+9)/3)*(4^l-1)+6=158375$	3100300	158598	3340516
Hospital.checkPatientsInside (5)	1	1	44	1	44
Village.simulate2 (4)	10921	$(2/3)*(4^l-1)-1=10921$	240296	10922	240340
Hospital.checkPatientsAssess (2)	1	1	44	1	44
Hospital.checkPatientsWaiting (6)	1	1	44	1	44
Village.getResults2 (7)	10922	$(2/3)*(4^l-1)=10922$	240312	10922	240312

*Cantidad de objetos controlados: 431647 objetos

Cuadro 10.12: Contadores - Health -l 7 -t 10.

Las tablas muestran que los cálculos de las regiones son bastante acertados. Sólo hay una sobre-estimación en la región *mainOrig*, que no es apreciable según la cantidad de objetos creados (ver figura 10.3).

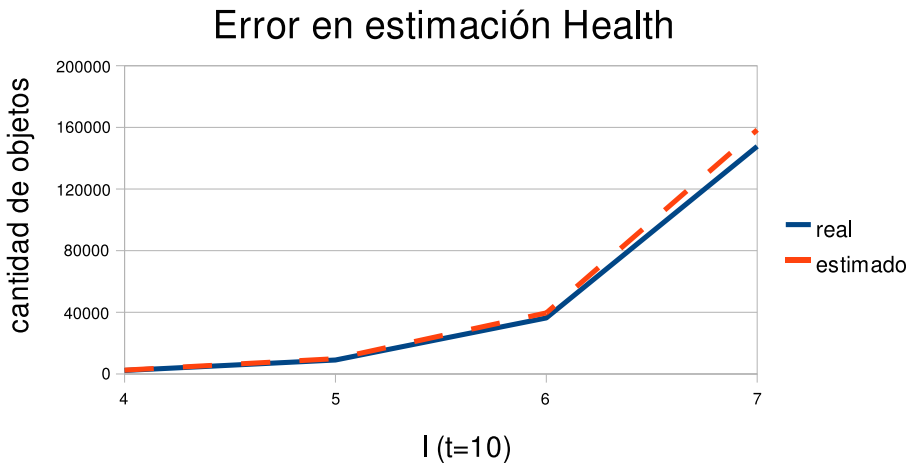

Figura 10.3: Error en estimación de la región *mainOrig* en Health.

10.2.4. Detalles sobre el análisis de escape utilizado

Dado que el análisis de escape efectuado no es sensible al contexto, los casos en donde podría haber un mayor error de cálculo, son en los métodos donde se presentan estructuras de decisión que incluyen la creación de muchos objetos. Los tests de JOlden no contienen casos críticos como el del siguiente ejemplo, por lo que la aproximación del método es bastante acertada.

Ej:

```
if(condition){
      new Object[1000];
}else{
      new Object[3000];
}
```

En este caso, el tamaño de la región deberá tener en cuenta a ambos objetos, aunque sólo contendrá a uno de ellos. En cambio, si el análisis estático detecta estas estructuras, es posible acotar el tamaño al mayor de ellos.

10.3. Utilización de los recursos tomados y performance

En esta sección se comparan diferentes aspectos de las VMs implementadas y se contrastan con la implementación original provista por Jikes RVM. Para ello se utilizaron los programas evaluados más significativos en cuanto a la utilización de regiones, que son los que la cantidad de regiones varían según los parámetros de entrada (*Health* y *BiSort*).

No se detallarán los tiempos obtenidos en todos los programas de JOlden debido a la no relevancia de los datos y a la complejidad del análisis de algunos ejemplos, en donde no es posible controlar todas las asignaciones, en particular las que provienen de objetos internos a la standard library, como es en el caso de *System.out.println*.

10.3.1. Memoria desperdiciada (spoiled memory)

La siguiente figura representa la memoria reservada no utilizada según la cantidad de objetos controlados en regiones. La misma es calculada como la suma del tamaño de los *gaps*.

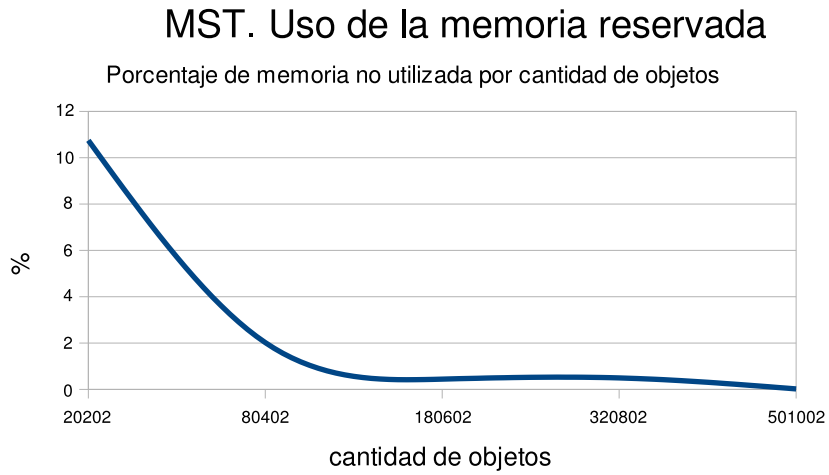

Figura 10.4: Porcentaje de memoria no utilizada por objetos controlados (RegionsAllocator).

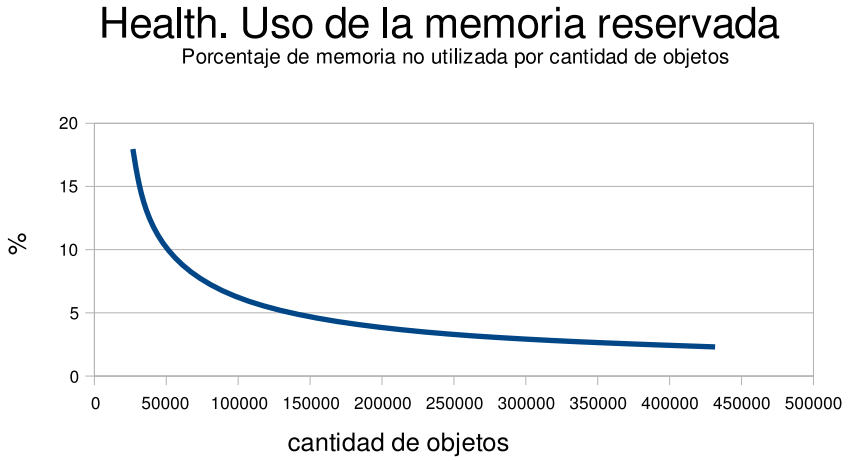

Figura 10.5: Porcentaje de memoria no utilizada por objetos controlados (ResizeRegionsAllocator).

Los gráficos muestran que cuando los objetos en regiones son relativamente pocos, la memoria pedida que está en desuso se acerca al 20 %, mientras que con una mayor cantidad de objetos, este porcentaje no sube del 2 %. Esto se debe a que la mínima unidad de memoria pedida por el allocator es de una página, y si el programa no administra muchos objetos en memoria, el tamaño de las regiones no alcanza a cubrir dicho espacio.

10.3.2. Comparación entre las máquinas virtuales implementadas

Para poder comparar todas las implementaciones realizadas, incluyendo la propia implementación de Jikes, se tomó el programa *Health*, ya que posee varios métodos (con regiones asociadas) invocados desde dos iteradores principales, lo que lo hace propicio para evaluar el consumo y la performance de la administración de la memoria.

10.3.2.1. Consumo de memoria por VM por cantidad de objetos creados.

En esta tabla se tomaron mediciones sobre el consumo de memoria de cada espacio de memoria perteneciente a las máquinas virtuales: *Mark&Sweep - Puro* (Jikes RVM), *Regiones - Mark&Sweep*, *Regiones - Inmortal* y *Regiones - Puro*.

Dado que todos los métodos iterados están asociados a regiones (no recursivas), la cantidad de memoria utilizada por la máquina virtual original debe ser mayor que la utilizada por las otras máquinas virtuales. En el cuadro comparativo se puede observar que el crecimiento de la memoria es lineal para todas las máquinas virtuales. La mayor diferencia se detecta en el caso de la máquina virtual *Mark&Sweep - Puro,* que crece de a 8MB en cada medición, mientras que las demás VMs con regiones lo hacen de a aproximadamente 3MB.

Si comparamos el consumo entre las máquinas virtuales que utilizan espacios con regiones, la VM *Regiones - Puro* consume 3MB adicionales, que son los correspondientes al overhead de la administración del espacio inmortal (fragmentado en múltiples partes discontinuas).

Parametros	# objetos	Mark &Sweep - Puro	Regiones – Mark &Sweep			Regiones – Inmortal			Regiones - Puro
t		**MSSpace**	Regions Space	MSSpace	**total**	Regions Space	Inmortal	**total**	**Regions Space**
300	720175	**20.05**	3.89	8.04	**11.93**	3.89	7.53	**11.42**	**14.79**
500	1198242	**28.53**	6.46	8.04	**14.5**	6.46	7.53	**13.99**	**17.57**
700	1675658	**36.98**	9.68	8.04	**17.72**	9.68	7.53	**17.21**	**20.35**

Cuadro 10.13: Comparación de Consumo - Health.

La siguiente figura representa la cantidad de memoria total para cada VM, evaluada para tres parámetros de entrada.

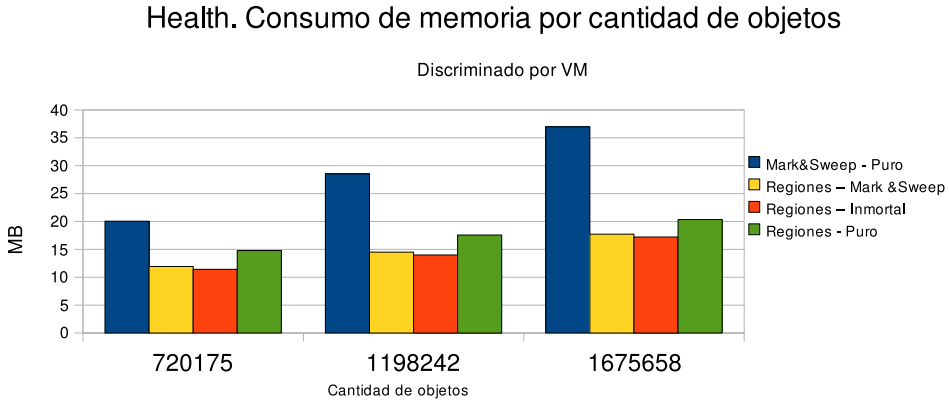

Figura 10.6: Health. Consumo de memoria por VM según cantidad de objetos.

10.3.2.2. Tiempos de ejecución por VM

Los siguientes resultados muestrar los tiempos de ejecución transcurridos al ejecutar el programa Health (-t 5 -l 300 -s 4). Los tiempos obtenidos están discriminados según si transcurren en el thread Main y si son consecuencia de la administración de memoria. Como comportamiento general se puede observar que el mayor consumo de tiempo es el transcurrido por la administración de memoria en el thread Main. Dicho valor está regido por el comportamiento de los allocators y el administrador de páginas.

En las VMs de regiones con tamaño predefinido (*Regiones - Mark&Sweep* y *Regiones - Inmortal*) la administración de memoria presenta mejor performance ya que las operaciones siempre se realizan en el tope de la pila y la región posible para cada lugar de creación es siempre una. Esto hace que el orden algorítmico sea $\mathcal{O}(1)$ en todas las operaciones, y como se utiliza un espacio contiguo la asignación es más rápida. *Regiones - Puro*, en cambio presenta una performance más baja debido a:

- La creación de objetos de forma alternada entre las regiones elegidas y la inmortal (Jikes internamente maneja algunos de sus procesos bajo el thread Main, y crea objetos en el heap). Esto genera mucha fragmentación de la región inmortal generando muchos gaps, que deben ser administrados por la freeList.
- El orden algorítmico al eliminar una región de memoria particionada. En este caso el mayor tiempo transcurre administrando la *freeList*, principalmente al eliminar la región inmortal (en un futuro se puede omitir el eliminado de esta región, pero actualmente se hace para obtener datos estadísticos -cuyos procesos están ligados a la eliminación de la región-).

Medición	Mark &Sweep - Puro	Regiones – Mark&Sweep	Regiones – Inmortal	Regiones - Puro
Ejecución	2640.55	2145.4	2139.46	10297.34
Adm. Memoria	988.56	824.24	837.76	8074.98
Ejecución Main	1353.29	810.97	821.93	9047.41
Adm. Memoria Main	556.09	372.67	360.72	7614.77

Cuadro 10.14: Health. Tiempos de ejecución por VM.

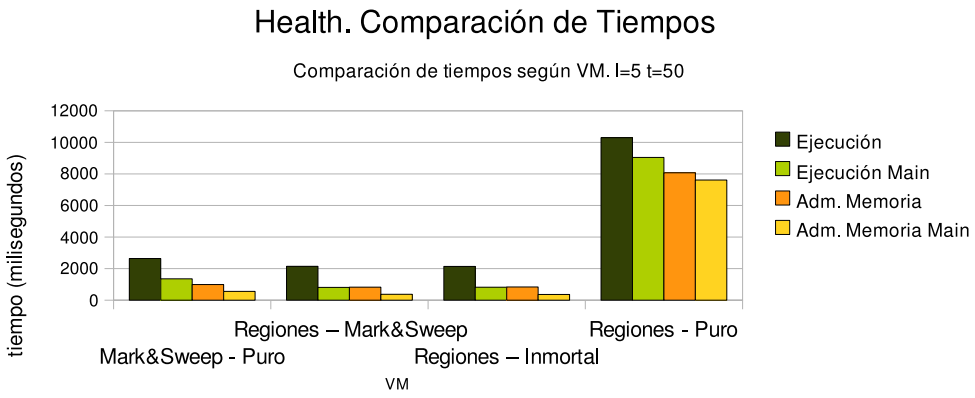

Figura 10.7: Health. Tiempos de ejecución por VM.

10.4. Detección de patrones algorítmicos

10.4.1. Iteración de regiones

Se tomó el programa *BiSort* como caso de estudio de uso de regiones. Una parte de este programa consiste en imprimir la estructura generada de la forma inOrder mediante el método *printValue*. En esta ocasión se modificó dicho método de modo que en él se cree un objeto "grande" (un array de *Object* de tamaño 1000) con el objetivo de demostrar el uso de regiones para este tipo de casos. Se aplicó una definición de regiones de memoria sobre este método para que la memoria ocupada por el arreglo se libere al final de cada llamada. Como el algoritmo es recursivo, pero la región fue extraída para el método *printValue* (en vez de en *inOrder*), se espera que la utilización de regiones mejore la performance tanto en espacio requerido como en el tiempo de ejecución.

```
void inOrder()   {

      if (this.left != null)

           left.inOrder();

      printValue();
      if (this.right != null)

           this.right.inOrder();

}
@RegionDefinition(size=4036,allocations={ "printValue()V:2"})
void printValue(){

      Value.contador ++;
      Object[] object =new Object[1000];

}
```

Se tomaron distintas muestras sobre el parámetro de programa, para cada VM implementada. Los datos de la siguiente tabla muestran el consumo en páginas de memoria, sin limitar la memoria. Mientras que las máquinas virtuales que utilizan regiones de memoria presentan valores de consumo similares y constantes (a pesar de cambiar la cantidad de objetos administrados), en *Mark&Sweep - Puro* la cantidad de páginas tomadas va incrementandose de a cuerdo a la figura 10.2. Ver siguiente figura 10.8.

Si se limita la cantidad de memoria a las 2700 páginas (10.8MB), como lo muestra la siguiente figura 10.9, En la VM *Mark&Sweep - Puro* el GC empezará a ejecutarse reiteradamente para liberar la memoria (para $s = 5000$ se ejecuta 40 veces). En las VMs que utilizan las regiones, los tiempos se mantienen casi constantes, en relación a la anterior.

parametro (s) #obj	Regiones - Puro	Regiones – Mark&Sweep	Regiones – Inmortal	Mark &Sweep - Puro
(s=1000) 89254	2554	2633	2502	3943
(s=2000) 178734	2562	2633	2502	5486
(s=3000) 357508	2562	2649	2518	8575
(s=4000) 357748	2562	2649	2518	8575
(s=5000) 715407	2578	2657	2526	14753

Cuadro 10.15: BiSort (modificado). Páginas de memoria reservadas por VM.

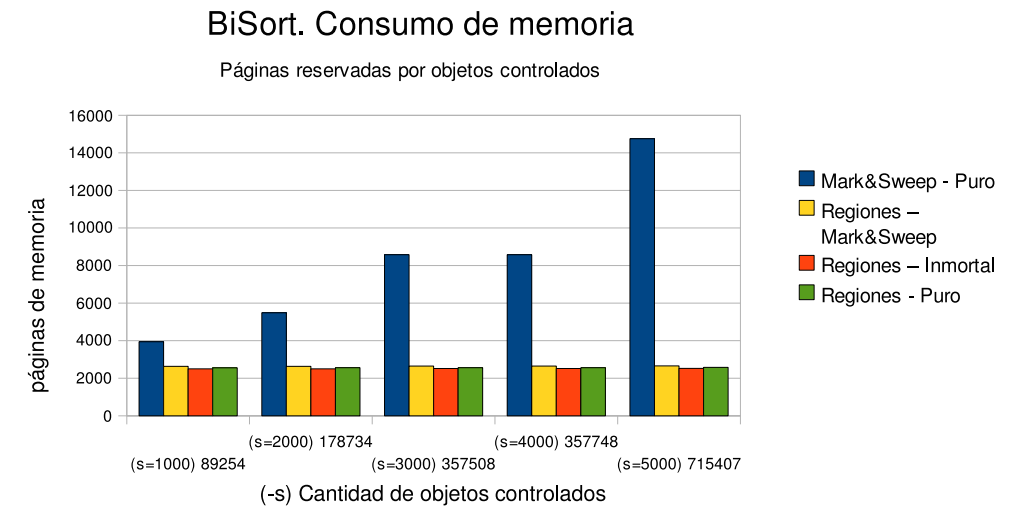

Figura 10.8: BiSort (modificado). Consumo de memoria por VM.

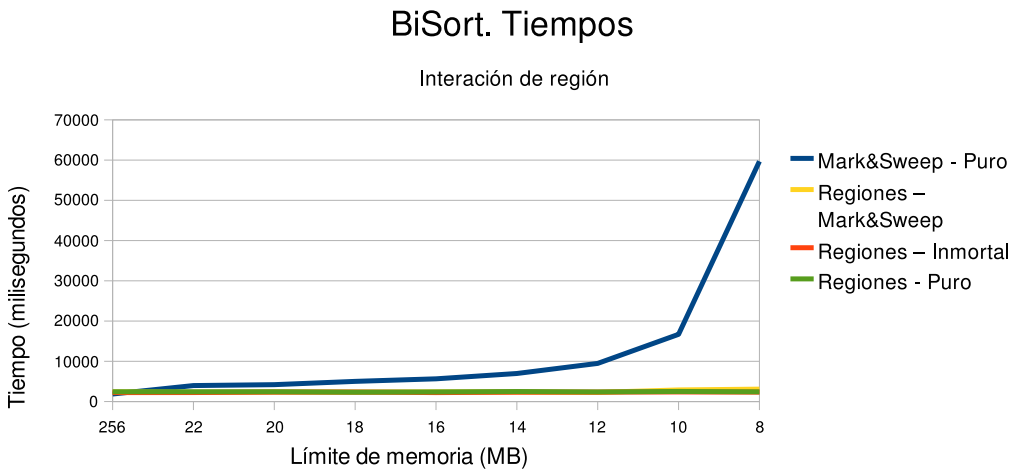

Figura 10.9: BiSort (modificado). Tiempos de ejecución por VM.

10.4.2. Región recursiva

Un caso donde la utilización de regiones de memoria no aporta beneficio alguno, y más bien genera utilización de recursos adicionales innecesarios, es en donde se presentan métodos/regiones recursivas, en las cuales algunos objetos albergados en ellas podrían liberarse antes de finalizar su ejecución. En un ambiente con GC, el programa presentado podrá ejecutarse con escasos recursos de memoria, aunque con malos tiempos de ejecución (se utilizará el GC), mientras que la utilización de regiones requerirá el espacio necesario para albergar la suma del tamaño de todas las regiones a apilar.

```
@RegionDefinition(size = 4040, allocations ={"m1(I)V:1" })
```

```
private static void m1(int i) {

      // el siguiente objeto no escapa,
      // podría ser liberado antes de invocar a m1
      currentObjects = new Object[1000];
      if(i!=0)

           m1(i-1);

}
```

La siguiente tabla muestra las páginas de memoria requeridas por cada VM, sin limitar la memoria. Si se limitara, las VMs de regiones darían el error *OutOfMemory*. Los resultados no muestran gran variación de consumo entra las diferentes VMs.

parametro (s=#obj.) bytes	Regiones - Puro	Regiones – Mark&Sweep	Regiones – Inmortal	Mark &Sweep - Puro
(s=1000) 1001	3418	3493	3374	3357
(s=2000) 2001	4418	4493	4374	4361
(s=3000) 3001	5410	5493	5374	5365
(s=4000) 4001	6402	6493	6374	6369
(s=5000) 5001	7394	7493	7374	7373

Cuadro 10.16: BiSort (modificado). Páginas de memoria reservadas por cantidad de objetos administrados.

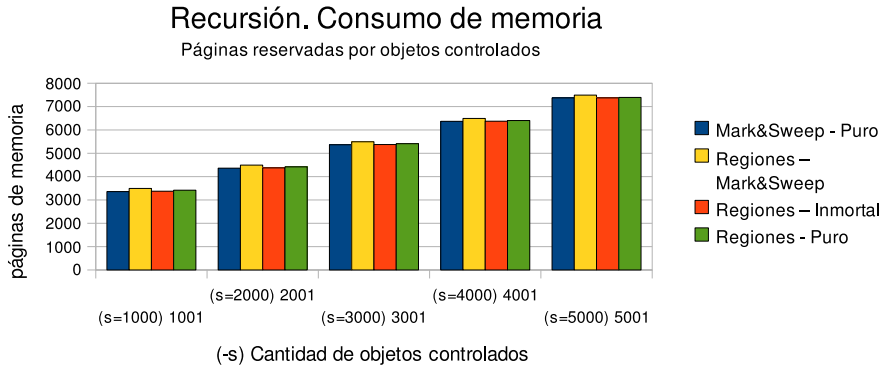

Figura 10.10: Recursión. Consumo de memoria por VM.

Al medir los tiempos de ejecución, si bien no se presentan diferencias significativas entre las diferentes máquinas, la administración por medio de *Mark&Sweep - Puro* es más eficiente que las que utilizan regiones. Este resultado es reflejo de la excesiva creación/eliminación de regiones en relación a la asignación de objetos en regiones, que es sólo uno por región. Entre las que utilizan regiones, la VM *Regiones - Inmortal* es la que posee el comportamiento más eficiente, y *Regiones - Puro*, el peor.

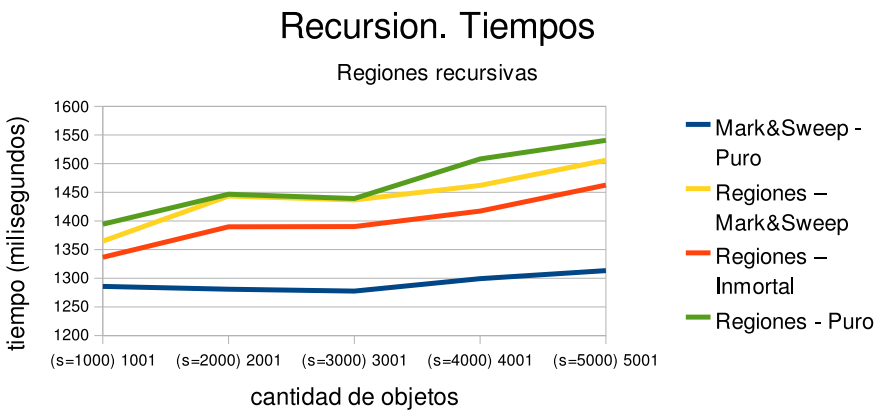

Figura 10.11: Recursión. Tiempos de ejecución por VM.

Capítulo 11

Conclusiones

11.1. Análisis de los resultados

Al evaluar las VMs en los diferentes aspectos presentados en el capítulo 10, además de proveer una visión sobre el estado actual de la implementación, los resultados obtenidos han servido para validar el modelo teórico empíricamente, y también para detectar casos en donde es conveniente la utilización de un sistema de memoria basada en este tipo de regiones.

11.1.1. Evaluación del modelo teórico

Como resultado de haber anotado los programas de JOlden (ver sección 10.2) puede observarse que las regiones detectadas por el analizador que provee la herramienta, son válidas. Esto quiere decir que sus configuraciones permiten realizar una ejecución del programa sin errores, respetando las restricciones de los ciclos de vida de los objetos, y que los valores calculados para el tamaño de las regiones nunca son inferiores a los reales, de hecho el error en el cálculo no es muy grande, por lo que el tamaño estimado es próximo al real. Basándose en esta premisa, si se pudieran detectar todas las regiones con sus tamaños de manera automática, se podría utilizar siempre el allocator *RegionsAllocator*, que realiza un manejo de memoria mejor que *ResizeRegionsAllocator* en términos de orden algorítmico y buen uso del espacio reservado, e implementar su automatización.

11.1.2. Consumo de memoria y tiempos

La administración de memoria basada en regiones intenta eliminar la utilización de los garbage collectors tradicionales sobre el heap, y para ello debe estructurar la memoria de manera que sea fácil detectar qué objetos eliminar sin necesidad de realizar un trazado sobre toda la memoria. A diferencia de los allocators que provee Jikes RVM, que tratan de optimizar la operación *alloc* haciendo esta tarea lo más eficiente posible (dejando al GC toda la responabilidad para la liberación de la memoria), al contemplar las nuevas operaciones sobre regiones y al tener en cuenta en qué región crear el objeto, se incorpora overhead. Esto provoca que en un sistema con suficiente cantidad

de memoria para correr toda la aplicación, la VM convencional sea más rápida y requiera menos espacio que las VMs implementadas en este trabajo.

	Segregated FreeListAllocator	RegionsAllocator	ResizeRegionsAllocator
crear región	-	$\mathcal{O}(1)$	$\mathcal{O}(l)$
eliminar región	-	$\mathcal{O}(1)$	$\mathcal{O}(p*(n*p+r))$
asignación de memoria	$\mathcal{O}(a)$	$\mathcal{O}(d*n)$	$\mathcal{O}(d*n+l)$
garbage collector	$\mathcal{O}(2*m)$. Se efectúan 2 pasadas.	-	-

a: cantidad de iteraciones para conseguir el espacio requerido en la *freeList* segregada.

m: cantidad de objetos del heap.

d: regiones posibles para un *cs* en runtime.

n: cantidad de regiones.

p: cantidad de partes de regiones.

l: cantidad de entradas en *freeList*.

r: cantidad de entradas en *requestList*.

Cuadro 11.1: Comparación de orden entre las operaciones de memoria.

En el cuadro 11.1 se puede apreciar que la VM que no administra regiones es la que provee mejor performance en tiempos y en espacio consumido (no se genera espacio adicional para estructuras de regiones). Aunque el orden algorítmico de las restantes VMs es mayor, el valor resultante del orden de complejidad mostrado es mucho menor que el orden m del GC (dado que en el proceso de GC están involucradas dos pasadas sobre todos los objetos del heap). En cambio, en regiones, el orden de complejidad está relacionado en la mayoría de los casos con la cantidad de regiones en la pila del allocator, que es a lo sumo la misma cantidad que métodos en la pila de llamadas.

Utilizando regiones, si las operaciones se realizan mayormente en el tope de la pila, y si para cada lugar de creación sólo hay unas pocas regiones posibles, el orden algorítmico también estará cercano o igual a $\mathcal{O}(1)$. Dicho comportamiento se puede apreciar en los resultados donde se evalúa el tiempo de las diferentes VMs. Para que la administración por regiones sea significativamente superior en performance a un espacio sin regiones, el programa deberá ejecutarse en un contexto de memoria limitada, y contener estructuras algorítmicas que ameriten su uso. De esta manera, se podrá evitar el uso del garbage collector, que es el proceso que más tarda y degrada la performance de la VM en este tipo de ambientes.

El beneficio de utilizar regiones de memoria es que si bien sus operaciones son menos performantes, están acotadas temporalmente y suceden en un instante determinado (conocido). Por otra parte en muchos casos su utilización brinda un mejor aprovechamiento del espacio en memoria (como en 10.4.1), evitando así su agotamiento y el consecuente uso del GC.

Se pueden distinguir tipos de programas o más bien patrones algorítmicos en donde

el uso de regiones provee mejoras importantes, y otros en donde éstas no ofrecen ningún beneficio y provocan un mal uso de la memoria.

Algunos de los casos en donde el uso de regiones es beneficioso, siempre hablando de un ambiente acotado en recursos de memoria, es en programas donde prevalece la invocación reiterada de métodos no recursivos, cuyos objetos (creados en él) son locales. En estos casos, al finalizar la invocación de los mismos, se liberará toda la memoria consumida durante su uso.

Un ejemplo claro es el de las iteraciones que invocan (directa o indirectamente) a métodos que disparan la creación de regiones en donde se genera una gran tasa de objetos locales. Por ejemplo en el caso de JOlden *BiSort* (subsección 10.4.1), el método *printValue* es creado reiteradas veces y al finalizar su ejecución el estado de la memoria no es alterado, en cambio, si no se utilizasen regiones, se generarían muchísimos objetos en el heap, entonces provocando las reiteradas ejecuciones del GC por falta de espacio.

En la misma subsección, si se observan las mediciones correspondientes a la VM *Mark&Sweep - Puro* puede observarse este comportamiento: el allocator siempre asigna memoria hasta que se agota y es liberada por el GC; y este proceso se repite reiteradas veces hasta que finaliza la ejecución del programa. Utilizando regiones, aunque la iteración de la región *printValue* hace las veces de GC, la diferencia radica en que el espacio es liberado en su totalidad sin tener que realizar una traza del heap para saber qué objetos eliminar individualmente.

Contrariamente, también existen casos en donde el uso de regiones no aporta ningún beneficio. Por ejemplo si se toma un algoritmo cuyo método/región es recursivo, por más que se cree una región por cada invocación, estas serán apiladas y el consumo de memoria será aún mayor que en un heap convencional. Esto se debe al espacio adicional requerido por el header de cada región, y porque en cada una de ellas pueden quedar objetos en desuso, que en una VM con GC, podría llegar a eliminarse. Adicionalmente, tampoco es posible obtener mejor performance que en un entorno con GC dado que nunca se iteran regiones. En el ejemplo 10.4.2 se presenta el caso mencionado.

Dado que la mayoría de los programas no presentan en su totalidad un comportamiento acorde con alguno de estos dos casos, es necesario analizarlos para poder detectar estos patrones (buenos o malos) y saber en dónde aplicar el uso de regiones. Una propuesta interesante podría ser realizar una herramienta que detecte las iteraciones de regiones, y simplemente aplicar regiones a dichos casos. Estos programas podrían ser corridos en el modelo híbrido *Regiones - Mark&Sweep*.

11.2. Aportes

11.2.1. Entorno de prueba

El desarrollo de este trabajo sirvió principalmente para tener una máquina virtual Java moderna con un sistema de memoria administrado por regiones, lo que permite evaluar cómo reaccionan ciertos programas (en tiempo y consumo) bajo una administración parcial o total de sus objetos bajo este esquema. Dado que la implementación responde al modelo propuesto en [7, 21], que utilizan regiones asociadas a métodos, la herramienta provee un entorno de prueba para corroborar el correcto funcionamiento de los modelos teóricos,

ya sea verificando la especificación de las regiones generadas, como la estimación de sus tamaños (en el caso de [7]).

El tener diferentes máquinas virtuales que permitan modelar esquemas de memoria híbridos, en particular la VM *Regiones - Mark&Sweep* permite en un principio definir manualmente regiones sobre algunos métodos del programa y evaluar su desempeño, sin necesidad de analizar el programa en su totalidad. Esta máquina virtual se comportará como una VM convencional si la mayor parte de los objetos van al heap, y si contrariamente los objetos están en su mayoría asociados a regiones, su funcionamiento será similar a la VM *Regiones - Inmortal*. La restante: VM (*Regiones - Puro*), si bien sus operaciones no están del todo optimizadas, posibilita la ejecución de programas donde las regiones no tienen un tamaño inicial definido, y de esta manera es posible calcularlos en runtime para luego fijarlos en las próximas ejecuciones, consiguiendo entonces una mejor performance.

Disponer de este conjunto de VMs con su entorno de desarrollo permite poder agilizar las pruebas sobre posibles soluciones teóricas (dentro del modelo implementado), y a su vez encarar futuros trabajos en base a los resultados obtenidos. **Claramente, esta implementación es sólo un paso inicial que permite validar el estado actual de las herramientas, para contemplar en un futuro la implementación de un modelo optimizado, donde no haga falta utilizar procesos de garbage collector garantizando la predictibilidad temporal y espacial.**

11.2.2. Integración con herramientas automáticas

La implementación provista propone dos formas de integración con herramientas que aporten información de regiones en forma automatizada. Debido a la implementación en capas, agregar un nuevo componente o cambiar el modo en que se compila el bytecode, permite definir diferentes formas de administrar regiones de memoria utilizando el mismo componente MMTk. Por ejemplo se podrían especificar nuevas instrucciones bytecode para el manejo explícito de regiones desligando su utilización únicamente a los métodos. En este caso el allocator *ResizeRegionsAllocator* puede ser adaptado fácilmente para ser utilizado no sólo como una pila de regiones (sólo hace falta implementar el método *removeRegion*(*regionId*) que es muy sencillo).

La segunda forma de integración se puede realizar sin programar componentes extras sobre la VM. La sintaxis de anotaciones permite anotar métodos tanto en el fuente Java como en el código bytecode, de modo que una forma simple de incorporar información adicional sería instrumentando el bytecode con las anotaciones de regiones. Este mecanismo tiene como ventaja que no es necesario desarrollar nuevos componentes sobre la VM. La desventaja es que debe utilizarse la sintaxis proporcionada y su uso sólo se limita a regiones de memoria asociadas a métodos.

11.3. Limitaciones y trabajo a futuro

11.3.1. Soporte multi-threading

Una de las principales limitantes que presenta la implementación propuesta es que el uso de las regiones de memoria deben restringirse al thread main. Esto tiene su origen en la administración directa de las regiones como una única pila. Una solución trivial, utilizando la VM Regiones expansibles, sería validar la región incorporando como owner al thread. De esta manera se mantendrían diversas pilas de memoria identificadas por su identificador de región (id asociado el método que asigna el compilador) y el identificador del thread. No es posible utilizar este modelo con el allocator *RegionsAllocator* debido a que las operaciones contemplan una sola pila contigua en memoria.

Uno de los temas más importantes a tratar a la hora de definir el soporte para multi-threading, es el manejo de los objetos compartidos. En el caso que se referencie al mismo objeto, este deberá vivir en la región del thread con mayor tiempo de vida. Otra solución sería generar una copia del objeto por thread, y administrar sus cambios por medio del pasaje de mensajes.

11.3.2. Control de objetos internos de Jikes RVM.

Otro de los problemas, que está relacionado con la arquitectura de Jikes RVM, esta vinculado con la meta circularidad de su implementación. Los procesos internos: como mantener el flujo del programa, imprimir por stdout, o la compilación de un método, están programados en Java, y en consecuencia crean objetos en el heap. Dada la gran cantidad de estos objetos (no pertenecientes al programa ejecutado), y por lo tanto no administrado por regiones, hace que sea muy dificultoso obtener mejoras en la performance, para entonces, limitar el uso de la VM sólo a un espacio de regiones (inhabilitando el heap con GC). Algunas soluciones consisten en analizar el código de Jikes mediante las herramientas automáticas utilizadas en esta tesis, y anotar el código como si formara parte del programa. Existe la posibilidad de hacerlo de manera segura dado que estos procesos no pueden ser invocados desde un programa de usuario (lo que podría llegar a modificar las regiones). De esta manera los procesos internos también harían un buen uso de la memoria.

11.3.3. Expresiones de tamaño paramétricas

La sintaxis provista no posibilita el poder especificar el tamaño de una región por medio de una expresión en base a los parámetros del método asociado, atributos de instancia, o estáticos. Esta limitación produce que el tamaño de las regiones tenga que ser definido previamente a la ejecución, lo que imposibilita además, la integración con herramientas automáticas que provean dicha parametrización. Por otra parte, no contemplar esta funcionalidad puede producir un exceso de espacio en las regiones, ya que estas son creadas siempre con el tamaño máximo que van a tomar durante toda la ejecución, aunque en algunos casos se requiera menos espacio.

Para poder implementarlo sobre Jikes RVM, habría que pensar en alguna forma de identificar los parámetros y objetos visibles al método (en runtime), para que antes de crear una región,

la expresión pueda ser evaluada teniendo en cuenta dichos objetos. En un principio, una solución podría ser sólo limitarse a los parámetros de entrada del método, con lo que la validación se llevaría a cabo en la clase *VM_Runtime*, accediendo al stack frame.

11.3.4. Definición de sintaxis sobre bloques o estructuras de código

Actualmente la sintaxis provista soporta regiones sobre métodos, sin embargo, en muchos casos es necesario asociar la región a unidades de cómputos más pequeñas, como podría ser un bloque de código, o una estructura dentro del mismo. Por ejemplo, sería útil especificar regiones dentro algunas iteraciones, o en condicionales. Un ejemplo claro de su utilización sería en el ejemplo 10.4.2 (regiones recursivas) donde se apilan regiones que albergan objetos que ya no son alcanzados. En este caso se podría proveer una región interna al método (en vez de asociada a el mismo), que sería eliminada antes de la llamada recursiva.

11.3.5. Cache de lugar de creación según pila de llamadas a métodos

Implementando la cache detallada en la subsección 8.1.1.1 se mejoraría la performance de todos los allocs que poseen diversos lugares de creación según la pila de llamadas a métodos. Sin ir muy lejos, los métodos de la clase *String* son utilizados desde muchos métodos en la mayoría de los programas. Para estos casos la performance de crear un objeto por primera vez sería de $\mathcal{O}(n^2)$ y para las futuras creaciones del mismo método y bajo la misma configuración de pila, $\mathcal{O}(1)$.

11.3.6. Optimización de *ResizeRegionsAllocator*

Actualmente la VM Regiones, que contempla regiones expansibles, no posee una buena performance en tiempos y aprovechamiento del espacio. Esto limita la utilización de herramientas automáticas que no contemplan regiones con tamaños predefinidos.

Algunas de las optimizaciones previstas para este allocator son:

- Administración de páginas con listas segregadas en vez de pila de regiones (bump pointer).
- Sistema adaptativo para estimar el tamaño de las regiones en tiempo de ejecución.
- Evaluar el uso de garbage collector de regiones (los cuales deberían ser eficientes ya que no es necesaria la traza de todo el heap).

Apéndice Manual de usuario

Referencia de Jikes : http://jikesrvm.org/Building+the+RVM

Instalación

La instalación del proyecto consta de las 3 máquinas virtuales correspondientes a los planes citados durante el trabajo, más la provista por Jikes RVM. Las mismas corren sobre sistemas operativos linux con procesadores AMD o Intel (ia32 o x86 de 32/64 bits).

Dependencias

- Java5 (JDK)
- Ant
- paquete build-essential
- paquete bison
- * paquetes multilib de gcc y g++

(sólo para arquitecturas AMD 64 bits)*

Variables de entorno

```
export JAVA_HOME=/usr/lib/jvm/java-1.5.0-sun
export PATH=$JAVA_HOME/bin:$PATH
```

Dentro de la carpeta INSTALL se encontrarán los archivos de instalación. Para instalar directamente todas las máquinas virtuales ejecutar el comando install.sh. Este descomprimirá en el lugar, los directorios correspondientes a las máquinas virtuales, y luego se llevará a cabo el proceso de build.

```
sh install-intel.sh (o install-amd para procesadores AMD).
```

Compilación Individual

Para compilar cada máquina virtual individualmente se debe ejecutar el siguiente comando dentro de la carpeta de la VM a compilar.

Para intel:

```
ant -Dhost.name=ia32-linux -Dconfig.name=regions
```

Para AMD32/64:

```
ant -Dhost.name=x86_64-linux -Dconfig.name=regions
```

Nota: si no se posee salida a internet descomprimir jikes-cache.tar.gz al mismo nivel que el directorio de la VM y añadir la opción:

```
-Dcomponents.cache.dir=../../..jikes-cache
```

Una vez terminado el proceso quedarán los correspondientes ejecutables junto a la biblioteca runtime en:

```
{VM_DIR}/dist/regions/regions_{host.name}/
```

ej:

```
/home/tesis/JikesRVM/dist/regions/regions_x86_64-linux/rvm
```

Para ejecutar la VM desde cualquier directorio, se puede crear un alias en el archivo ~/.bashrc

```
/home/tesis/JikesRVM/dist/regions/regions_x86_64-linux/rvmrt.jar
```

11.3.7. Compilación adicional para Eclipse

Referencia de Jikes: http://docs.codehaus.org/display/RVM/Editing+JikesRVM+in+an+IDE

Para poder levantar el proyecto en el eclipse se debe compilarlo añadiendo "eclipse-project" al comando.

```
ant -Dhost.name=ia32-linux -Dconfig.name=regions eclipse-project
```

Esto creará un proyecto Eclipse, que podrá ser incorporado fácilmente en la IDE.

Ejecución

Para utilizar la máquina virtual con soporte para la utilización de regiones, el programa debe tener en el classpath la biblioteca runtime provista por cada VM (reemplazar por la rt.jar):

```
/home/tesis/JikesOrginalMS/dist/regions/regions_x86_64-linux/rvmrt.jar
```

Básicamente estas bibliotecas proveen las anotaciones definidas en la clase `java.lang.regions`.

Como la versión de Jikes elegida todavía no es compatible para ser ejecutada dentro de Eclipse (la última versión si lo es), se debe hacerlo desde la línea de comandos.

```
rvm -classpath mi-aplicación/bin paquete.ClaseMain parameters
```

Dado que Jikes RVM toma por defecto la libraría rvmrt.jar, no hace falta incorporarla al classpath.

Opciones adicionales

Se incorporaron las siguientes opciones de línea de comando:

- `regionStats=true` (default en false): imprime las estadísticas en tiempo de ejecución tras salir de una región. Estos datos incluyen el estado de la pila de regiones con los max_rsize y peaks alcanzados hasta el momento.
- `timeStats=true` (default en false): imprime las estadísticas de tiempos transcurridos al finalizar la ejecución del programa
- verbose:gc=n (n=1, 2, 3, 4, 5). Es el nivel de verbose de la administración de memoria. Por ejemplo si verbose:gc=1 se imprimirán las corridas del GC, si se setea en 2 también se incluirán las operaciones de creación y eliminación de regiones, y así sucesivamente. Finalmente en 5 se imprime cada alloc que es controlado por alguna región.

```
rvm -verbose:gc=5 -regionStats=false -timeStats=true -classpath app/bin package.ClassMain
```

Índice de figuras

Índice de algoritmos

Índice de cuadros

Bibliografía

[1] David F. Bacon, Perry Cheng, David Grove, and Martin T. Vechev. Syncopation: generational real-time garbage collection in the metronome. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 183–192, New York, NY, USA, 2005. ACM Press.

[2] Stephen M. Blackburn, Perry Cheng, and Kathryn S. Mckinley. Oil and water? high performance garbage collection in java with mmtk. In *In ICSE 2004, 26th International Conference on Software Engineering*, pages 137–146, 2004.

[3] B. Blanchet. Escape analysis for object-oriented languages: application to Java. In *OOPSLA 99*, volume 34, pages 20–34, 1999.

[4] Bruno Blanchet. Escape analysis for javatm: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, November 2003.

[5] Greg Bollella and James Gosling. The real-time specification for java. *Computer*, 33(6):47–54, 2000.

[6] Chandrasekhar Boyapati, Alexandru Salcianu, Rasekhar Boyapati, Jr., Ru Salcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time java. In *In Programming Language Design and Implementation (PLDI*, pages 324–337. ACM Press, 2003.

[7] Victor Braberman, Federico Fernandez, Diego Garbervetsky, and Sergio Yovine. Parametric prediction of heap memory requirements. In *International Symposium on Memory Management*, sigplan, pages 141–150. ACM, ACM, jun 2008.

[8] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in java controller. In *PACT 2001*, pages 280–291, 2001.

[9] Sigmund Cherem and Radu Rugina. Region analysis and transformation for java programs. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 85–96, New York, NY, USA, 2004. ACM.

[10] J-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for Java. In *OOPSLA*, pages 1–19, 1999.

[11] Diego Garbervetsky. Sintesis de especificaciones parametricas de utilizacion de la memoria dinamica. 2007.

[12] Diego Garbervetsky, Chaker Nakhli, Sergio Yovine, and Hichem Zorgati. Program instrumentation and run-time analysis of scoped memory in Java. *RV 04, ETAPS 2004, ENTCS, Barcelona, Spain*, April 2004.

[13] David Gay and Alex Aiken. Language support for regions. In *In Programming Language Design and Implementation (PLDI*, pages 70–80, 2001.

[14] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction*, pages 82–93, London, UK, 2000. Springer-Verlag.

[15] IBM. Jikestm research virtual machine (rvm), 2009.

[16] Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996. With a chapter on Distributed Garbage Collection by Rafael Lins. Reprinted 1997 (twice), 1999, 2000.

[17] Kaffe.org. Java virtual machine, August 2008.

[18] Andreas Krall. Efficient JavaVM just-in-time compilation. In *PACT '98 proceedings*, 1998.

[19] Robert Lougher. Jamvm java virtual machine. 2003.

[20] Martin Maierhofer and M. Anton Ertl. Local stack allocation. In *CC*, pages 189–203, 1998.

[21] Guillaume Salagnac, Christophe Rippert, and Sergio Yovine. Semi-automatic region-based memory management for real-time java embedded systems. In *RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 73–80, Washington, DC, USA, 2007. IEEE Computer Society.

[22] Fridtjof Siebert. Hard real-time garbage-collection in the jamaica virtual machine. *rtcsa*, 00:96, 1999.

[23] Nikhil Swamy, Michael W. Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Safe manual memory management in cyclone. *Sci. Comput. Program.*, 62(2):122–144, 2006.

[24] Mads Tofte and Jean-Pierre Talpin. Region-based memory management, 1997.

[25] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. *ACM SIGPLAN Notices*, 34(10):187–206, 1999.