UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA
Departamento de Arquitectura de Computadoras y Automática

OPTIMIZACIÓN DE LA GESTIÓN DE MEMORIA DINÁMICA EN JAVA.

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

José Manuel Velasco Cabo

Bajo la dirección de los doctores

Katzalin Olcoz Herrero
Francisco Tirado Fernández

Madrid, 2010

ISBN: 978-84-693-7807-6

UNIVERSIDAD COMPLUTENSE DE MADRID

Facultad de Informática

Departamento de Arquitectura de Computadores y Automática

Optimización de la gestión de memoria dinámica en Java

TESIS DOCTORAL

José Manuel Velasco Cabo

MADRID, 2010

Optimización de la gestión de memoria dinámica en Java

Memoria presentada por
José Manuel Velasco Cabo
para la obtención del grado de Doctor
por la Universidad Complutense de Madrid,
realizada bajo la dirección de
Katzalin Olcoz Herrero,
David Atienza Alonso
y Francisco Tirado Fernández

16 de febrero de 2010

A toda mi familia

Agradecimientos

Quisiera dar las gracias de todo corazón a mis tres directores de tesis por la ayuda y el apoyo que me han demostrado durante estos años, confiando en mí en todo momento durante los momentos difíciles que he atravesado a lo largo de la elaboración de este trabajo. He aprendido mucho de los tres a nivel profesional y humano y han dejado en mí una honda impresión. Mí máximo agradecimiento y cariño a la profesora Olcoz, sin cuya guía, comprensión y ánimo me habría resultado imposible terminar esta tesis. Al profesor Atienza, por su generosidad y por las largas vigilias que me ha dedicado cuando teníamos un deadline próximo, mí afectuosa admiración. Y al profesor Tirado porque ha sido el impulsor y orientador de este trabajo en los momentos más cruciales.

También quiero expresar mí agradecimiento especial al doctor Gómez por las charlas mantenidas con él y por su orientación a la hora de realizar las estadísticas de accesos a la máquina virtual de Java, y al doctor Piñuel por su estrecha colaboración durante la modelización de una memoria scratchpad dentro del simulador Dynamic SimpleScalar.

Quiero agradecer a todos mis compañeros de despacho (presentes y pasados) el buen ambiente que siempre hemos tenido y que ha facilitado enormemente mí trabajo. Agradecimiento que me gustaría extender a todos los técnicos y, en general, a todos los compañeros de departamento por la atmósfera de cordialidad y compañerismo que he disfrutado.

Índice General

Capítulo 1

Introducción

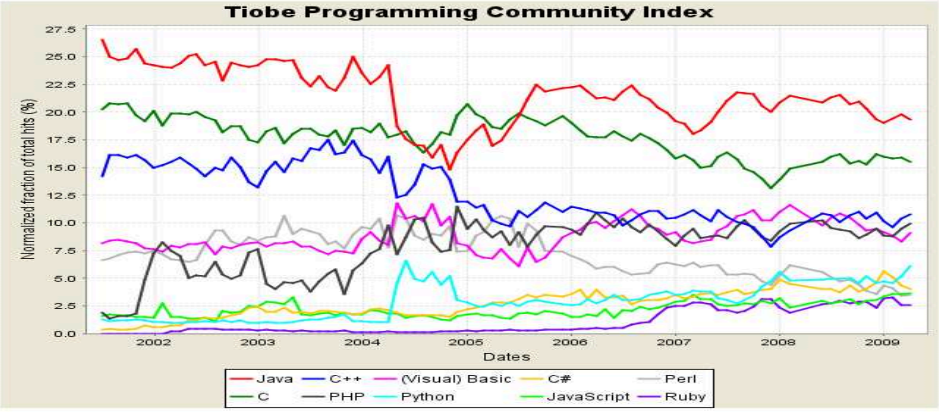
En este trabajo presentamos varias técnicas para mejorar el rendimiento y, al mismo tiempo, reducir del consumo de potencia del gestor automático de memoria dinámica de la máquina virtual de Java, habitualmente conocido como recolector de basura. Como punto de partida, en esta tesis se realiza un exhaustivo análisis de la influencia de la recolección de basura en el rendimiento de la máquina virtual de Java y en el consumo de potencia de ésta dentro de una jerarquía de memoria típica de los actuales sistemas empotrados, así como de la interacción de las distintas estrategias de recolección con el hardware subyacente. Como veremos en nuestros resultados experimentales, dentro de las distintas tareas encargadas a la máquina virtual de Java, la recolección de basura puede llegar a ser, en el contexto de los sistemas empotrados, el factor predominante en cuanto a consumo de energía y rendimiento se refiere. Este estudio culmina con la descripción de un completo espacio de diseño que nos proporciona las mejores configuraciones de la jerarquía de memoria para cada estrategia de recolección dentro de la dicotomía rendimiento-consumo de energía. Sobre esas configuraciones idóneas aplicaremos a continuación nuestras propuestas de optimización. Nuestras técnicas incluyen mejoras a nivel algorítmico por un lado, y por otro buscan aprovechar el comportamiento inherente y predecible de las distintas estrategias de recolección para propiciar una interacción fructífera entre la máquina virtual y la jerarquía de memoria, que produzca sustanciales reducciones en el consumo energético y en el tiempo empleado en la recolección de basura.

1.1 Motivación

La empresa Tiobe Software [sof] elabora mensualmente un índice de los lenguajes de programación más populares. Es un índice bastante complejo que tiene en cuenta visibilidad en internet, publicación de anuncios en prensa así como otra serie de variables. En la gráfica 1.1(a) se muestra la evolución histórica de ese índice. Podemos ver que desde que se lleva a cabo su realización, y con la única excepción de un breve periodo durante el año 2005, el lenguaje Java ha sido el lenguaje más popular para el desarrollo de todo tipo de aplicaciones. En la gráfica 1.1(b) se recogen los resultados porcentuales registrados en abril de 2009. En ellos se concluye que Java se ha convertido en el lenguaje de programación favorito por parte de los desarrolladores de software, cuatro puntos por delante del lenguaje C, que sería su único gran competidor, e incluye una comunidad de más de 6 millones y medio de programadores. Por todo el mundo existen más de 4.500 millones de dispositivos, pertenecientes a ámbitos de aplicación muy dispares, que utilizan la tecnología Java [sof]:

- Más de 800 millones de equipos de altas prestaciones (ordenadores personales y grandes servidores).
- 2.100 millones de teléfonos móviles y otros dispositivos portátiles, como cámaras web, sistemas de navegación para automóviles, terminales de lotería, dispositivos médicos, etc.
- 3.500 millones de tarjetas inteligentes.

Para introducir Java en mercados tan distintos como el de los servidores o el de los sistemas empotrados, la empresa Sun ha desarrollado varias versiones (”ediciones”) del lenguaje que podemos ver de una forma gráfica en la figura 1.2. En la segunda posición desde la izquierda de la figura tenemos la edición estandar de Java con todas sus funcionalidades típicas. A la izquierda de la figura tenemos la edición de Java para empresas, que es una ampliación de la edición estandar con librerías, módulos, entornos de desarrollo, etc (como, por ejemplo, NetBeans [6.7]), para el desarrollo de aplicaciones destinadas a servidores. A la derecha de la figura, podemos ver la edición Java Card [Incc], diseñada para ser usada en tarjetas inteligentes (como las tarjetas SIM de la telefonía móvil o las tarjetas monedero). La especificación de Java Card es un subconjunto muy pequeño del lenguaje Java, y aunque la máquina virtual de Java estandar puede ejecutar los



(a) Evolución histórica

Position Apr 2009	Position Apr 2008	Delta in Position	Programming Language	Ratings Apr 2009	Delta Apr 2008	Status
1	1	=	Java	19.341%	-1.55%	A
2	2	=	C	15.472%	-0.04%	A
3	3	=	C++	10.741%	-0.06%	A
4	4	=	PHP	9.888%	-0.32%	A
5	5	=	(Visual) Basic	9.097%	-0.69%	A
6	7	↑	Python	6.080%	+1.18%	A
7	8	↑	C#	4.059%	+0.00%	A
8	9	↑	JavaScript	3.678%	+0.75%	A
9	6	↓↓↓	Perl	3.462%	-2.09%	A
10	10	=	Ruby	2.569%	-0.07%	A
11	11	=	Delphi	2.272%	+0.25%	A
12	14	↑↑	PL/SQL	1.086%	+0.33%	A
13	12	↓	D	1.076%	-0.37%	A
14	13	↓	SAS	0.792%	-0.13%	A
15	15	=	Pascal	0.717%	+0.12%	A-
16	21	↑↑↑↑↑	Logo	0.707%	+0.37%	A
17	27	↑↑↑↑↑↑↑↑↑↑	ABAP	0.658%	+0.42%	B
18	26	↑↑↑↑↑↑↑↑	RPG (OS/400)	0.646%	+0.40%	B
19	19	=	Lua	0.491%	+0.12%	B
20	23	↑↑↑	MATLAB	0.482%	+0.22%	B

(b) Clasificación en abril de 2009

Figura 1.1: Índice de popularidad de los lenguajes de programación realizado por la empresa Tiobe Software.

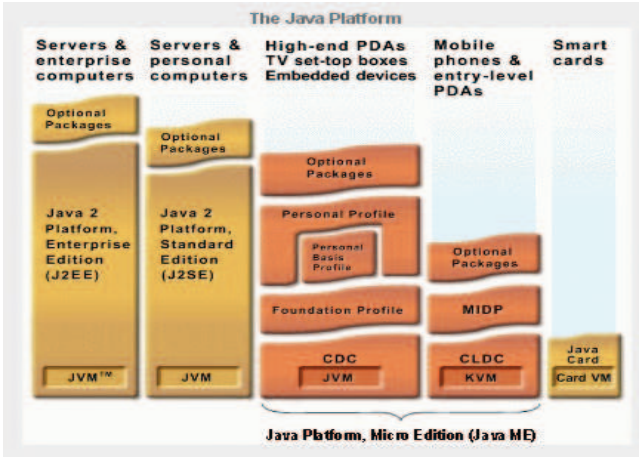


Figura 1.2: Ámbito de las diferentes versiones del lenguaje Java.

programas de Java Card, la máquina virtual de Java Card no dispone de un gran número de funcionalidades de la edición estandar. A medio camino entre estas dos versiones, se encuentra la edición Micro (Java ME) [Incd] pensada para sistemas empotrados. Java ME distingue entre sistemas empotrados con recursos muy limitados y sistemas con una cierta capacidad y, consecuentemente, ha desarrollado dos entornos separados cuyas características son:

- La especificación CLDC (*Connected Limited Device Configuration*) [Incb] enfocada a plataformas con una unidad de procesamiento de 16 bits y, al menos, una memoria de 160 KB. Evidentemente, la máquina virtual no dispone de la gran mayoría de funcionalidades de Java. Esta especificación tuvo su punto álgido con la llegada de los ”buscas” (*pagers*) y primeros teléfonos móviles.

- La especificación CDC (*Connected Device Configuration*) [Inca] que requiere, típicamente, de una unidad de procesamiento de 32 bits y una memoria RAM de 2MB o superior. La máquina virtual soporta todas las funcionalidades de la máquina virtual estandar salvo las relacionadas con la interfaz gráfica de ventanas (módulo dirigido al sistema operativo *Windows*), y claro está, usa ampliamente la recolección de basura. Esta especificación está pensada para dispositivos GPS, grabadores/ reproductores de DVD, teléfonos móviles con funcionalidades multimedia, etc. Como un ejemplo interesante por su éxito comercial, podemos poner el caso de la plataforma *SunSpot* [Ince] y la máquina virtual *Squawk* [(JV]. *SunSpot* es utilizado

en redes de sensores, tiene un *core* ARM920T [Ltd] (32-bits RISC) y una memoria de 1MB. *Squawk* es compatible con el entorno de desarrollo de NetBeans y puede ejecutarse en plataformas PowerPC, Intel o SPARC, bajo Windows, Linux o Solaris, aunque cuando se ejecuta sobre *SunSpot* no necesita de sistema operativo.

La versión de las librerías que utiliza la máquina virtual de Java ME en tiempo de ejecución (*Java RunTime Environment* 1.3) se está quedando anticuada frente a las disponibles para la versión estandar (JRE 1.6), y Sun no parece tener mucho interés en actualizarlas. La razón de este hecho puede estar en la opinión de Bob Vandette (ingeniero de Sun Microsystems) que afirma que, debido a las mejoras en prestaciones a nivel de hardware, próximamente la mayoría de sistemas empotrados utilizarán, sin restricciones, la versión estandar de Java. De hecho, actualmente, ya lo están haciendo gran cantidad de teléfonos móviles de última generación, multitud de PDAs o incluso controladores para surtidores de gasolina [Van]. Por esta razón hemos decidido basarnos en la versión estandar de Java para nuestro estudio de sistemas empotrados de altas prestaciones.

En el capítulo 2 se discute acerca de las diferentes razones del éxito de Java, pero básicamente podemos decir que uno de los principales motivos es la gestión automática de memoria dinámica. La gestión automática de memoria dinámica habitualmente se conoce como recolección de basura y su concepción se la debemos a John McCarthy [Paga], quien la desarrolló conjuntamente con el lenguaje LISP [LIS] a finales de la década de los cincuenta del siglo pasado. Sin embargo, el auge de la recolección de basura viene, más modernamente, de la mano de Java. De hecho, la recolección de basura se ha convertido en una de las señas de identidad de este lenguaje. No obstante, implementar la gestión de memoria dinámica para el lenguaje Java es una tarea mucho más compleja de lo que resultó hacerlo para el lenguaje LISP. Por ello, en los últimos años hemos presenciado una gran interés en la comunidad científica que ha producido diferentes algoritmos y optimizaciones enfocadas a reducir el impacto de la recolección de basura en el rendimiento final de la máquina virtual de Java de la edición estandar y en el contexto de plataformas de alto rendimiento con pocas restricciones de memoria. El investigador Richard Jones de la universidad de Kent [Pagb], ha recopilado minuciosamente todas las publicaciones sobre el tema en *the Garbage Collection Bibliography* [Jon] cuyo número de entradas, en junio de 2009, se acercaba a 1900. Sin embargo, como acabamos de ver, la mayoría de los dispositivos que utilizan Java hoy en día se incluyen dentro de algún tipo de sistema empotrado. Y la mayoría de estas plataformas, excluyendo las tarjetas, o bien

utilizan la versión Java ME CDC (con prácticamente todas las funcionalidades de Java) o simplemente la edición estandar de Java y, por tanto, en los dos casos van a hacer un uso significativo de la recolección de basura. De modo que, en la situación actual de expansión de Java en el mercado de los sistemas empotrados, nos encontramos con dos factores:

- En la revista Dr. Dobbs [BW], los investigadores John J. Barton (IBM's T.J. Watson Lab) y John Whaley (MIT) presentaron los resultados obtenidos con una herramienta desarrollada por ellos para obtener en tiempo de ejecución el perfil detallado de la actividad de la máquina virtual de Java. En la gráfica 1.3 podemos ver el resultado de sus análisis en el contexto de un procesador de propósito general. Como podemos observar, el recolector de basura (*Garbage Collector*, GC) es una de las tareas de mayor peso, siendo éste parecido a la propia ejecución de la aplicación Java. Y si la influencia de la recolección de basura es significativa cuando la máquina virtual no tiene restricciones de memoria especiales, en los sistemas con limitaciones de memoria severas la recolección de basura se convierte en un factor determinante en el rendimiento de las aplicaciones Java, como demostraremos en nuestros resultados experimentales (capítulo 4.

- Paralelamente, y dentro del contexto de los sistemas empotrados, el consumo de energía es una preocupación tan relevante como el rendimiento y la máquina virtual de Java ha de adaptarse a esta situación. Una vez más, como veremos en el capítulo 4, el consumo de energía debido a la recolección de basura es un factor decisivo dentro del consumo total producido cuando la máquina virtual de Java está ejecutando una aplicación.

Y es aquí, en la unión de estos dos factores, donde encontramos la motivación fundamental de esta tesis, cuyos principales objetivos podemos sintetizar como:

- Determinar experimentalmente la influencia de la recolección de basura cuando los programas Java son ejecutados por la máquina virtual en sistemas empotrados, tanto a nivel de rendimiento como de consumo energético.

- Realizar una comparación de las estrategias de recolección actuales dentro del contexto de memoria limitada propio de los sistemas empotrados, para buscar la política que mejor se adecúe a ellos.

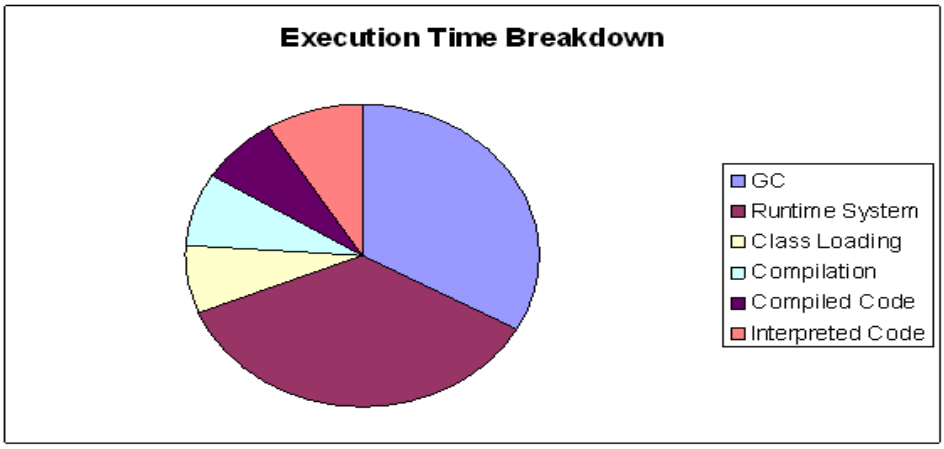


Figura 1.3: Porcentajes de tiempo asociados a las distintas tareas de la JVM durante la ejecución de una aplicación.

- Encontrar técnicas de optimización, tanto a nivel algorítmico como a nivel de la interacción entre la máquina virtual de Java y la plataforma hardware subyacente, que reduzcan el consumo de potencia y los tiempo de pausa asociados a la recolección de basura.

1.2 Estructura de la tesis

En la sección 1.1 hemos visto la motivación de este trabajo y los principales objetivos que perseguimos. Para describir el trabajo realizado en la consecución de estos objetivos, esta tesis se estructura de la siguiente forma:

- En el capítulo 2 se hace una recopilación de los conocimientos previos necesarios para seguir el resto de esta disertación. En primer lugar, se introducen la filosofía del lenguaje Java y el funcionamiento de su máquina virtual. A continuación se expone la mecánica básica de la programación orientada a objetos y el porqué ésta necesita asignar gran cantidad de memoria dinámicamente. Partiendo de este punto discutimos acerca de la complejidad inherente a la gestión automática de memoria dinámica y terminamos el capítulo presentado los distintos recolectores de basura utilizados a lo largo de este estudio y analizando su comportamiento.

- En el capítulo 3, en primer lugar, se detallan el entorno de simulación y el proceso completo de experimentacióin. A continuación, se presentan las distintas herramientas que hemos utilizado a lo largo de este trabajo: la máquina virtual de Java (JikesRVM), el simulador de la plataforma hardware (Dynamic SimpleScalar), los modelos de memoria (CACTI y Micron Power Calculator), así como el conjunto de benchmarks (SPEC).

- En el capítulo 4 se realiza una exploración experimental completa de la jerarquía de memoria, dentro de un rango típico de sistemas empotrados, para todos los recolectores de basura. Esto nos permitirá comprender la relación entre el recolector de basura y el sistema de memoria en la búsqueda de técnicas que nos permitan reducir el consumo energético y aumentar el rendimiento. Como primera consecuencia de este trabajo experimental, se obtiene la mejor estrategia de recolección para sistemas con memoria límitada. Además, en este capítulo también se presenta el espacio de diseño que nos proporcionan los datos de consumo energético y rendimiento obtenidos para cada recolector. En cada espacio de diseño se seleccionan los puntos que conforman las curvas de Pareto que nos servirán de punto de partida para, en los capítulos siguientes, desarrollar diferentes técnicas de optimización sobre ellas.

- En el capítulo 5 se proponen dos técnicas ideadas para reducir el consumo energético y mejorar el rendimiento. La primera está dirigida a reducir el consumo estático de los recolectores generacionales y utiliza el modo de bajo consumo de las actuales memorias SDRAM divididas en varios bancos. La segunda se enfoca a reducir el consumo dinámico y el tiempo total de recolección y utiliza una memoria *scratchpad* en el primer nivel de la jerarquía de memoria. En los dos casos, la máquina virtual es la encargada de interactuar dinámicamente con la plataforma subyacente utilizando información proporcionada por el recolector de basura.

- En el capítulo 6 se propone una optimización de los recolectores generacionales a nivel algorítmico. Nuestro recolector generacional adapta su comportamiento dinámicamente basándose en información recopilada en tiempo de ejecución. De este modo conseguimos reducir el número total de recolecciones, la cantidad de memoria copiada, el tiempo de recolección y el consumo de energía. Nuestro recolector generacional adaptativo puede emplearse con éxito para distintos tamaños

de memoria, pero está especialmente indicado para los sistemas empotrados. Al tratarse de una técnica software, puede usarse de forma paralela con las técnicas del capítulo 5, y al final de este capítulo se presentan los resultados experimentales obtenidos al aplicar conjuntamente las técnicas de los dos capítulos.

- Actualmente, los fabricantes de plataformas están abandonando la reducción de la frecuencia de reloj como objetivo principal, en favor de una búsqueda intensiva de nuevas arquitecturas multiprocesador construidas en un mismo circuito integrado. Ante esta situación el siguiente paso natural dentro de nuestra línea de investigación apunta a la gestión de memoria dinámica dentro de un sistema multiprocesador. En el capítulo 7, y dentro de las líneas actuales y futuras de trabajo, presentamos una técnica para la asignación eficiente de los objetos en los nodos de un sistema con memoria distribuida, de modo que se minimize el tráfico de datos, basada en la información suministrada por el recolector de basura. Además, hemos realizado un estudio experimental de distintos mecanismos de barrera como paso previo para la implementación de un recolector de basura global dentro de un entorno distribuido.
- En el apendice A se enumeran las abreviaturas utilizadas a lo largo de este trabajo en las leyendas de las gráficas.

Capítulo 2

Recolección de basura en Java

2.1 Introducción

En este capítulo presentamos en primer lugar, brevemente, las bondades del lenguaje Java y cómo éstas provocan un elevado grado de complejidad de la máquina virtual de Java (JVM). En segundo lugar se expone la mecánica básica de la programación orientada a objetos y el porqué esta necesita asignar gran cantidad de memoria dinámicamente. Partiendo de este punto, a continuación, discutimos acerca de la complejidad inherente a la gestión automática de memoria dinámica y terminamos el capítulo presentando los distintos recolectores de basura utilizados a lo largo de este estudio y analizando su comportamiento.

2.2 Java y la máquina virtual de Java

Como ya se indicó en la sección 1.1, durante la última década, Java ha sido el lenguaje de programación más popular. ¿A qué se debe el hecho de que la mayoría de programadores e ingenieros de sotware prefieran Java frente al resto de lenguajes?

Básicamente podemos decir que Java fué concebido para facilitar el desarrollo de aplicaciones, así como la comercialización del software. Para lograr este objetivo la filosofía de Java se enfoca en dos puntos:

- Liberar al programador de las tareas más complejas como la seguridad y la gestión de memoria.

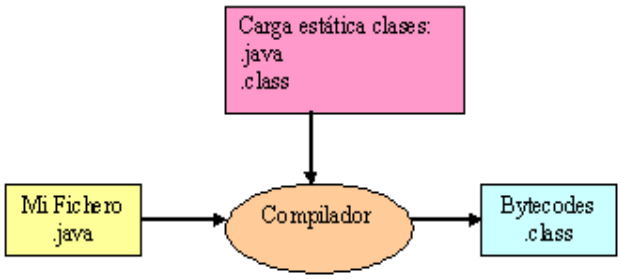
- Liberar a las empresas de la obligación de realizar multiples compilaciones para poder comercializar un software destinado a diferentes plataformas.

En síntesis, podemos decir que el lenguaje Java, conjuntamente con la máquina virtual de Java, simplifica el proceso de creación y comercialización de aplicaciones. Por supuesto, un éxito tan ambicioso conlleva un coste importante: la elevada complejidad de la máquina virtual de Java y de las múltiples tareas que ha de realizar en tiempo de ejecución. El rendimiento de la máquina virtual de Java ha sido, históricamente, la barrera que ha encontrado el lenguaje para su expansión. Más recientemente, con la llegada de los sistemas empotrados, al factor del rendimiento tenemos que añadir las limitaciones en cuanto a consumo de potencia. El objetivo de este trabajo es estudiar la interacción, tanto a nivel de rendimiento como de consumo energético, entre la máquina virtual (en especial el subsistema de gestión de memoria) y la plataforma hardware subyacente.

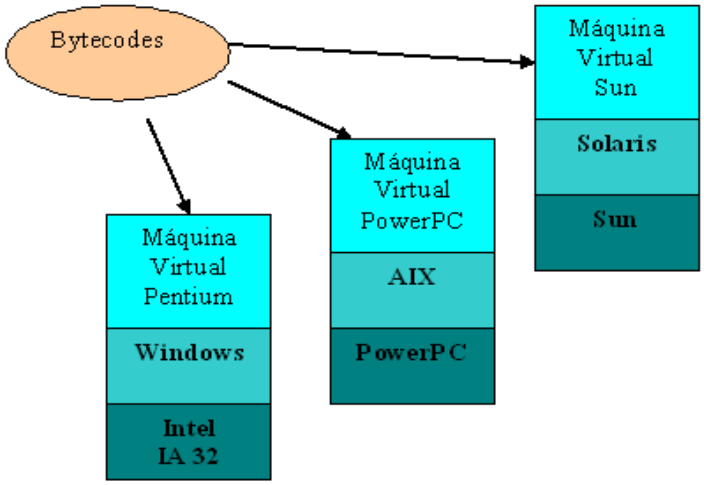
A continuación repasamos brevemente la historia del lenguaje y de las tareas encargadas a la JVM.

Java es un lenguaje de programación orientado a objetos desarrollado por la compañía Sun Microsystems [Inc03]. Su historia se remonta a la creación de una filial de Sun (FirstPerson) enfocada al desarrollo de aplicaciones para electrodomésticos tales como microondas, lavaplatos, televisiones... Esta filial desapareció tras un par de éxitos de laboratorio y ningún desarrollo comercial. Sin embargo, uno de los trabajadores de FirstPerson, James Gosling, desarrolló un lenguaje derivado de C++ que intentaba eliminar las deficiencias del mismo y al que llamó Oak. Cuando Sun abandonó el proyecto de FirstPerson rescató este lenguaje y, tras varias modificaciones (entre ellas la del nombre), decidió lanzarlo al mercado en verano de 1995.

Sun Microsystems es líder en servidores para Internet y al crear Java intentaba resolver simultáneamente todos los problemas que se le plantean a los desarrolladores de software por la proliferación de plataformas incompatibles, tanto a nivel de arquitecturas como a nivel de sistemas operativos o incluso de interfaces. Se añade a todo ello la dificultad actual de crear aplicaciones distribuidas en una red como Internet. Por ello Java nace con la promesa de permitir la ejecución de una aplicación sobre cualquier plataforma a partir de una única compilación. Esta primera compilación estática, figura 2.1(a), produce un fichero con *bytecodes* (instrucciones máquina dirigidas a la JVM) que después la máquina virtual tendrá que traducir y optimizar para la arquitectura final en el momento de la ejecución, figura 2.1(b).



(a) El compilador de Java, mediante una primera compilación estática, produce un fichero con *bytecodes*, dirigido a la arquitectura de la máquina virtual



(b) La máquina virtual se encarga, en tiempo de ejecución, de realizar la compilación de los *bytecodes* al conjunto de instrucciones máquina de la arquitectura final

Figura 2.1: Proceso de compilación y ejecución de las aplicaciones Java

Así, podemos resumir las caracteristicas principales del lenguaje Java como [Eck00]:

- Sencillez: los diseñadores de Java trataron de mantener las facilidades básicas del lenguaje en un mínimo y proporcionar un gran número de extras con las librerías de clases.
- Seguridad: se pretendió construir un lenguaje de programación que fuese seguro, esto es, que no pudiera acceder a los recursos del sistema de manera incontrolada. Por este motivo se eliminó la posibilidad de manipular la memoria mediante el uso de punteros y la capacidad de transformación de números en direcciones de memoria (tal y como se hace en C) evitando así todo acceso ilegal a la memoria. Esto se asegura porque el compilador de Java efectúa una verificación sistemática de conversiones.
- Portabilidad: el principal objetivo de los diseñadores de Java, dado el gran crecimiento de las redes en los últimos años, fue el de desarrollar un lenguaje cuyas aplicaciones una vez compiladas pudiesen ser inmediatamente ejecutables en cualquier máquina y sobre cualquier sistema operativo.
- Concurrencia (utilización de múltiples hilos): una de las características del lenguaje es que soporta la concurrencia a través de hilos (threads). En ocasiones puede interesar dividir una aplicación en varios flujos de control independientes, cada uno de los cuales lleva a cabo sus funciones de manera concurrente. Cuando los distintos flujos de control comparten un mismo espacio lógico de direcciones, se denominan hilos.
- Programación orientada a objetos (POO): abstracción del lenguaje de programación que permite al desarrollador expresar el programa en términos de la tarea a resolver y no en términos de la plataforma final donde se ejecutará la aplicación. En la sección 2.3 se profundiza en este punto.
- Robustez: uno de los problemas más comunes en los lenguajes de programación es la posibilidad de escribir programas que pueden bloquear el sistema. Algunas veces este bloqueo puede ser reproducido por el depurador ya que se trata de una falta de previsión del programador. Pero en otras ocasiones el bloqueo no es reproducible ya que es el resultado de un acceso no controlado a memoria. Si la ejecución de la aplicación se está llevando a cabo en exclusiva en el procesador, ese acceso a

memoria puede no producir ningún error. Sin embargo, si la ejecución es concurrente con otros programas, ese mismo acceso a memoria puede provocar interferencias en la ejecución de los otros programas y llevar a un bloqueo del sistema. Un ejemplo claro de lenguaje no robusto es C. Al escribir código en C o C++ el programador debe hacerse cargo de la gestión de memoria de una forma explícita, solicitando la asignación de bloques a punteros y liberándolos cuando ya no son necesarios. En Java, los punteros, la aritmética de punteros y las funciones de asignación y liberación de memoria (malloc( ) y free( )) no existen. En lugar de punteros se emplean referencias a objetos, los cuales son identificadores simbólicos. El gestor de memoria de la JVM es el encargado de liberar la memoria cuando ya no va a volver a ser utilizada. En la sección 2.4 se profundiza en este punto.

Para conseguir estos objetivos, la máquina virtual de Java consta de varios subsistemas que deben realizar una multitud de tareas al ejecutar una aplicación (figura 2.2) y que podemos resumir como:

- Compilación en tiempo de ejecución de los Bytecodes al repertorio de instrucciones de la arquitectura final.
- Gestión de los hilos de ejecución.
  - Sincronización
  - Conmutación entre hilos.
- Servicios en tiempo de ejecución.
  - Carga dinámica de clases. Esto comprende tanto las clases de la aplicación como las clases del *Java Runtime Environment* (JRE), entorno en tiempo de ejecución Java (conjunto de utilidades que, entre otras cosas, permite la ejecución de programas Java sobre todas las plataformas soportadas).
  - Control de tipos dinámico.
  - Interfaz entre la aplicación y la plataforma (sistema operativo y hardware).
    - Manejo de las excepciones.
    - Gestión de las interfaces.
    - Entrada/ salida.
    - Llamada a métodos nativos.

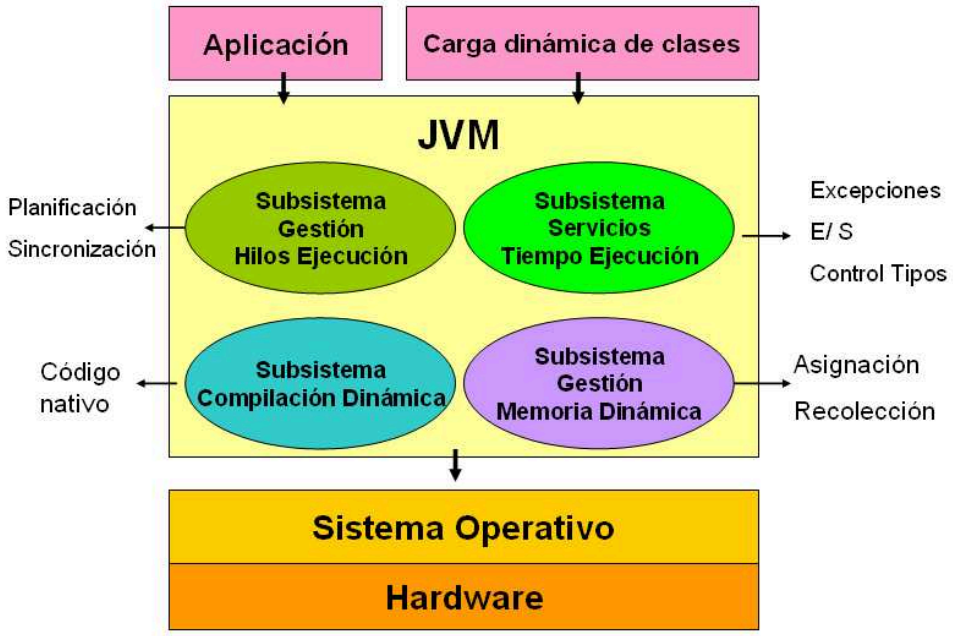


Figura 2.2: Tareas de la máquina virtual en tiempo de ejecución.

- Gestión automática de la memoria
  - Asignación de memoria dinámica.
  - Recolección de basura.

En el capítulo 4 presentaremos datos experimentales que demuestran que dentro del entorno de sistemas con limitaciones de memoria, el subsistema de gestión de memoria (y más concretamente el recolector de basura) es uno de los factores predominantes tanto a nivel de rendimiento como a nivel de consumo total de energía cuando la JVM está ejecutando una aplicación. Es por ello que en este estudio nos hemos centrado en la optimización del recolector de basura.

2.3 Programación orientada a objetos en Java

Para entender por qué es necesaria la gestión de memoria dinámica en Java vamos a explicar brevemente el funcionamiento de la programación orientada a objetos(POO), para más información se puede consultar el conocido libro de B. Eckel [Eck00]. Java es uno de los lenguajes orientados a objetos en su sentido más estricto. Java implementa la

tecnología básica de C++ con algunas mejoras y elimina algunas características para mantener el objetivo de la simplicidad del lenguaje. Java incorpora funcionalidades inexistentes en C++ como por ejemplo, la resolución dinámica de métodos. En C++ se suele trabajar con librerías dinámicas (DLLs) que obligan a recompilar la aplicación cuando se retocan las funciones que se encuentran en su interior. Este inconveniente es resuelto por Java mediante una interfaz específica llamada RTTI (RunTime Type Identification) que define la interacción entre objetos excluyendo variables de instancias o implementación de métodos. Las clases en Java tienen una representación en el *Java Runtime Environment* que permite a los programadores interrogar por el tipo de clase y enlazar dinámicamente la clase con el resultado de la búsqueda.

En Java todo es un objeto. Cada elemento del problema debe ser modelizado como un objeto. Un programa es un conjunto de objetos diciéndose entre sí qué deben hacer por medio de mensajes. Cada objeto tiene su propia memoria, que llena con otros objetos. Cada objeto puede contener otros objetos. De este modo se puede incrementar la complejidad del programa, pero detrás de dicha complejidad sigue habiendo simples objetos. Por tanto, podemos resumir básicamente los componentes de la POO de Java como:

- Clase: definición de las propiedades y comportamiento de un tipo de objeto concreto. La instanciación es la lectura de estas definiciones y la creación de un objeto a partir de ellas.
- Objeto: entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos).
- Método: algoritmo asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena tras la recepción de un ”mensaje”. Desde el punto de vista del comportamiento, es lo que el objeto puede hacer. Un método puede producir un cambio en las propiedades del objeto, o la generación de un ”evento” con un nuevo mensaje para otro objeto del sistema.

Java trabaja con objetos y con interfaces a esos objetos. Java soporta las tres características propias del paradigma de la orientación a objetos: encapsulamiento, herencia y polimorfismo.

- Encapsulamiento: significa reunir a todos los elementos que pueden considerarse

pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la cohesión de los componentes del sistema.

- Polimorfismo: esta característica permite extender un programa añadiendo diferentes tipos que sean capaces de manejar nuevas situaciones sin tener que rehacer el código desde el principio. De este modo los nuevos tipos tendrán funciones cuyos nombres coincidirán con nombres de funciones de los tipos antiguos, pero cuyos comportamientos pueden ser muy dispares. Así, en un programa Java, comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre. En tiempo de ejecución, la máquina virtual utilizará el comportamiento correspondiente al objeto que esté realizando la llamada.

- Herencia: las clases no están aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y el encapsulamiento permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir (y extender) su comportamiento sin que el programador tenga que volver a implementarlo. Un objeto, además, puede heredar de más de una clase (herencia múltiple).

2.4 Gestión de memoria dinámica

Los objetos se construyen a partir de una plantilla (la clase). Por ello, la creación de un objeto supone la creación de una *instancia* de la clase. Estas instancias, como en C++, necesitan ser construidas y destruidas en el espacio de memoria de forma dinámica. La correcta POO supone la creación y destrucción de numerosos objetos durante la ejecución de la aplicación. La asignación de memoria de forma dinámica se lleva a cabo de dos formas: En la pila (*Stack*) y en el montón (*Heap*). La pila y el montón permiten superar las principales limitaciones de la asignación estática de memoria que son:

- El tamaño de cada estructura de datos debe ser conocido en el momento de la compilación.

- Las estructuras de datos no se pueden generar dinámicamente.

- Una de las principales consecuencias de la limitación anterior es que no se puede implementar la recursividad. Distintas llamadas a un mismo procedimiento compartirían exactamente las mismas direcciones de memoria.

El problema de las llamadas recursivas, junto con las sustanciales mejoras en el lenguaje que conlleva, se puede resolver gracias a la pila. En la pila cada vez que un procedimiento es llamado se almacena un marco de referencia (*frame*). De este modo, diferentes llamadas a un mismo procedimiento no comparten las mismas direcciones para sus variables locales. Sin embargo, la rígida disciplina *last-in*, *first-out* de la pila sigue imponiendo numerosas limitaciones al lenguaje. Gracias a la asignación en el *Heap*, el diseñador de lenguajes obtiene su máxima libertad. Los datos son creados dinámicamente y su tamaño puede cambiar en el transcurso de cada ejecución libremente (desaparecen los errores debidos a los arrays de tamaño fijo). Las listas y árboles pueden crecer y cortarse en cualquier sentido. Un procedimiento puede devolver objetos de tamaño variable o incluso otro procedimiento. Las estructuras de datos pueden sobrevivir al procedimiento que los creó siempre que sean referenciados desde otro objeto.

2.4.1 Creación de objetos en Java

La forma típica de crear un objeto en Java es:

NombreClase nombreObjeto = new nombreClase ().

Al hacerlo, la máquina virtual realiza dos tareas:

- Reserva una porción de memoria en el Heap en función de la clase a la que pertenece el objeto.
- Crea una referencia, un puntero, en la pila dentro del marco de referencia que origina el nuevo objeto.

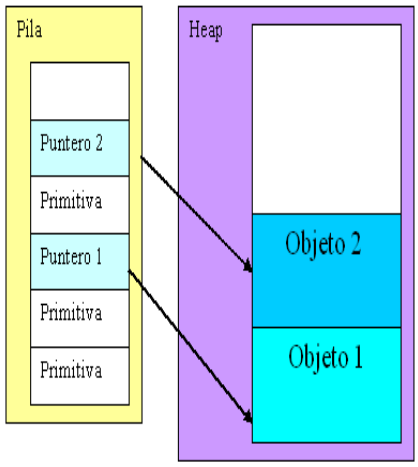
Como ya se ha mencionado, en Java todo son objetos. La única excepción son los tipos ”primitivos”, como enteros, caracteres, cadenas, etc, que el programador elige si van a ser objetos y por tanto se almacena su valor en el heap, o van a ser variables en el sentido de los lenguajes procedimentales, en cuyo caso se almacena su valor directamente en la pila.

Los valores que la máquina virtual de Java puede manipular directamente se conocen como el conjunto raíz (*root set*). Este comprende los registros del procesador, las variables globales (que son inmortales) y las variables almacenadas en la pila (incluyendo variables

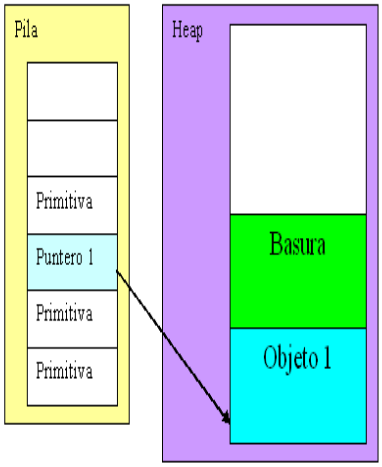
```
{ Clase1 obj1 = new Clase1 ();
   /* sólo obj1 es alcanzable */
  {
      Clase1 obj2 = new Clase1 ();
      /* obj1 y obj2 son alcanzables */
  }
   /* sólo obj1 es alcanzable */
   /* La referencia al Obj2 desapareció de la pila */

  Clase2 obj3= new Clase2 ();
   /* obj1 y obj3 son alcanzables */
}
```

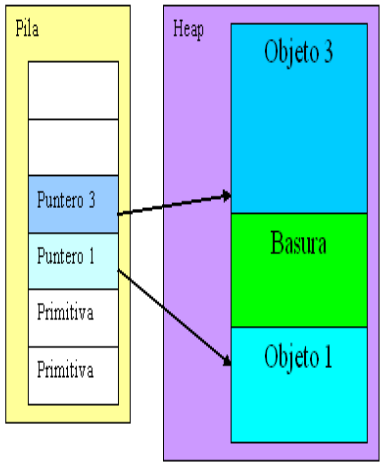
(a) Código



(b) Creación de dos objetos



(c) Muerte del segundo objeto



(d) Creación de un nuevo objeto

Figura 2.3: Creación de objetos y producción de basura

locales y temporales). Calificamos a un objeto como vivo si existe una cadena de referencias que tenga su origen en el conjunto raíz y que concluya en el objeto. En la figura 2.3(a) podemos ver un trozo de código Java típico en el que se crean tres objetos. Asumiendo que la asignación de memoria es contigua, en la figura 2.3(b) se muestra como sería la situación tras la creación de los dos primeros objetos. Cuando no hay referencias a un objeto desde el *root set* o siguiendo una cadena de referencias desde éste, la información del objeto en el Heap queda inalcanzable de forma permanente. Entonces decimos que el objeto está muerto o que es basura, figura 2.3(c). Y en la figura 2.3(d) la situación final, tras la creación del tercer objeto. El espacio reservado en la memoria para un objeto que se convierte en basura puede ser reutilizado para la creación de nuevos objetos, evitando así que la ejecución de un programa se quede sin memoria disponible cuando el sistema tiene memoria que no está utilizando. Sin embargo, el primer requisito indispensable para el ”reciclaje” de la basura es el conocimiento de su existencia. Lenguajes como C++, han dejado la responsabilidad de este conocimiento en el programador. En el lenguaje Java se optó por liberar al programador de esta tediosa responsabilidad y su máquina virtual es la encargada de recolectar esta basura. En la sección 2.4.2 se presentan las principales estrategias para llevar a cabo el reciclaje de basura y se profundiza en los recolectores que implementa la máquina virtual de Java utilizada en esta tesis, JikesRVM.

2.4.2 Recolección de basura

En la sección 2.3 hemos visto que la programación orientada a objetos conlleva la creación de un gran número de objetos. La mayoría de estos objetos no van a ser utilizados durante todo el tiempo que dure la ejecución del programa. A partir del momento en que un objeto no va a ser utilizado, el espacio que ocupa en el heap está siendo desaprovechado, es basura. Si no queremos que la máquina virtual se quede sin memoria para asignar a los nuevos objetos, tenemos que reciclar de algún modo esta basura en memoria disponible para el asignador. Esa es la tarea del recolector de basura [Jon00]. Pero, ¿cómo puede saber el recolector de basura si un objeto va a ser utilizado antes de que concluya la ejecución de la aplicación? La respuesta es que sólo el programador lo puede saber. El recolector de basura ha de adoptar una actitud conservadora. A ciencia cierta, la máquina virtual sólo puede estar segura de que un objeto está muerto si no hay ninguna referencia que apunte a ese objeto desde un objeto que esté vivo. O puede adoptar una estrategia

más sencilla y considerar que un objeto no ha muerto mientras haya referencias a ese objeto sin importar el origen. Estas dos ideas producen las dos únicas estrategias de recolección de basura: recolección por traza y recolección por cuenta de referencias.

En la recolección por cuenta de referencias la JVM mantiene un contador (normalmente en la cabecera del objeto) con el número de referencias que apuntan a ese objeto. Si el contador llega a cero, el recolector sabe sin lugar a dudas que el objeto no va a ser usado de nuevo y que su espacio puede ser reciclado. Hay varios problemas asociados con la cuenta de referencias:

- El almacenamiento extra: ¿Cuánto espacio empleamos para almacenar el contador? ¿Cuántas referencias puede llegar a tener un objeto? Normalmente los recolectores de cuenta de referencias utilizan una palabra para guardar el contador. En muchos casos, esto puede ser excesivo, pero en otros casos puede ser necesario. Otro enfoque utilizado es emplear sólo dos bits y si las referencias son más de tres el objeto nunca podrá ser reciclado.

- El elevado coste de la actualización de los contadores: Para la actualización de los contadores necesitamos un cierto número de instrucciones que pueden sobrecargar a la máquina virtual si la aplicación crea muchas referencias con un tiempo de vida corto. Además, si el contador de un objeto llega a cero, hay que actualizar los contadores de todos los objetos a los que el objeto recién muerto referencie. En el caso de una lista simplemente enlazada, esto puede dar lugar a una cadena de actualizaciones que necesite de un número impredecible de instrucciones.

- Las referencias cíclicas: El recolector por cuenta de referencias no es capaz de reciclar las estructuras de datos cíclicas o simplemente con referencias cruzadas. Como esta situación es muy común, podemos decir que este es el principal problema de esta estrategia. Para prevenir este defecto, los recolectores comerciales implementan conjuntamente un recolector de traza que periódicamente se encarga de estos objetos.

Este conjunto de problemas hacen que la cuenta de referencias no sea una solución universal. Por ello, en este trabajo nos hemos centrado en la recolección por traza, sección 2.5.

Durante la actividad normal de la máquina virtual, cuando no está ejecutando el recolector de basura, el grafo de relaciones entre objetos se modifica (”muta”), por ello, y

siguiendo la terminología de Dijkstra [W.D75], esta actividad se conoce como ”mutador” (*mutator*). Así, la realización de las distintas tareas de la máquina virtual (salvo la recolección) y la ejecución de la aplicación Java son parte del mutador. De este modo, y dentro del entorno de la gestión automática de memoria dinámica, la máquina virtual puede actuar únicamente como recolector o como mutador.

2.5 Recolectores de traza

Los recolectores de traza se basan en encontrar todos los objetos que están vivos en un determinado momento y asumir posteriormente que los demás objetos están muertos. Para ello, el recolector recorre todo el grafo de relaciones entre objetos, marcando los objetos visitados. En este grafo, los nodos son los objetos Java y las aristas son los punteros. Al finalizar el recorrido, los objetos que no han sido visitados y marcados son considerados basura. El recolector de basura comienza buscando referencias dentro del *root set* (sección 2.3), las referencias halladas se guardan en una cola para ser posteriormente visitadas en busca de nuevos punteros. En la máquina virtual de Java para investigación, JikesRVM, el recorrido del grafo de relaciones entre objetos se lleva a cabo ”primero en anchura”, pero en el punto 2.5.2 se discute cuando puede ser más interesante el recorrido ”primero en profundidad”.

En la figura 2.4 se muestra una clasificación de los distintos tipos de recolectores de traza utilizados en esta tesis. El primer factor de clasificación es el recorrido del grafo de relaciones entre objetos. Si el recorrido se produce a lo largo de toda la memoria tenemos los recolectores clásicos. Si el recorrido está limitado a ciertas regiones de la memoria hablamos de los recolectores generacionales. El segundo factor de clasificación es el modo en que los objetos vivos son tratados durante las recolecciones globales. Si los objetos no son movidos estamos ante el recolector de marcado y barrido. El movimiento de los objetos a través del *heap* es la seña de identidad de la política de copia. Finalmente en la gráfica se muestra el nombre con que se conoce la versión de estos recolectores en la máquina virtual de Java creada por IBM para la investigación, JikesRVM (capítulo 3, sección 3.3), que es la herramienta principal utilizada en esta tesis. En las secciones siguientes se presentan estos recolectores de traza y algunas de las características principales de su implementación en JikesRVM .

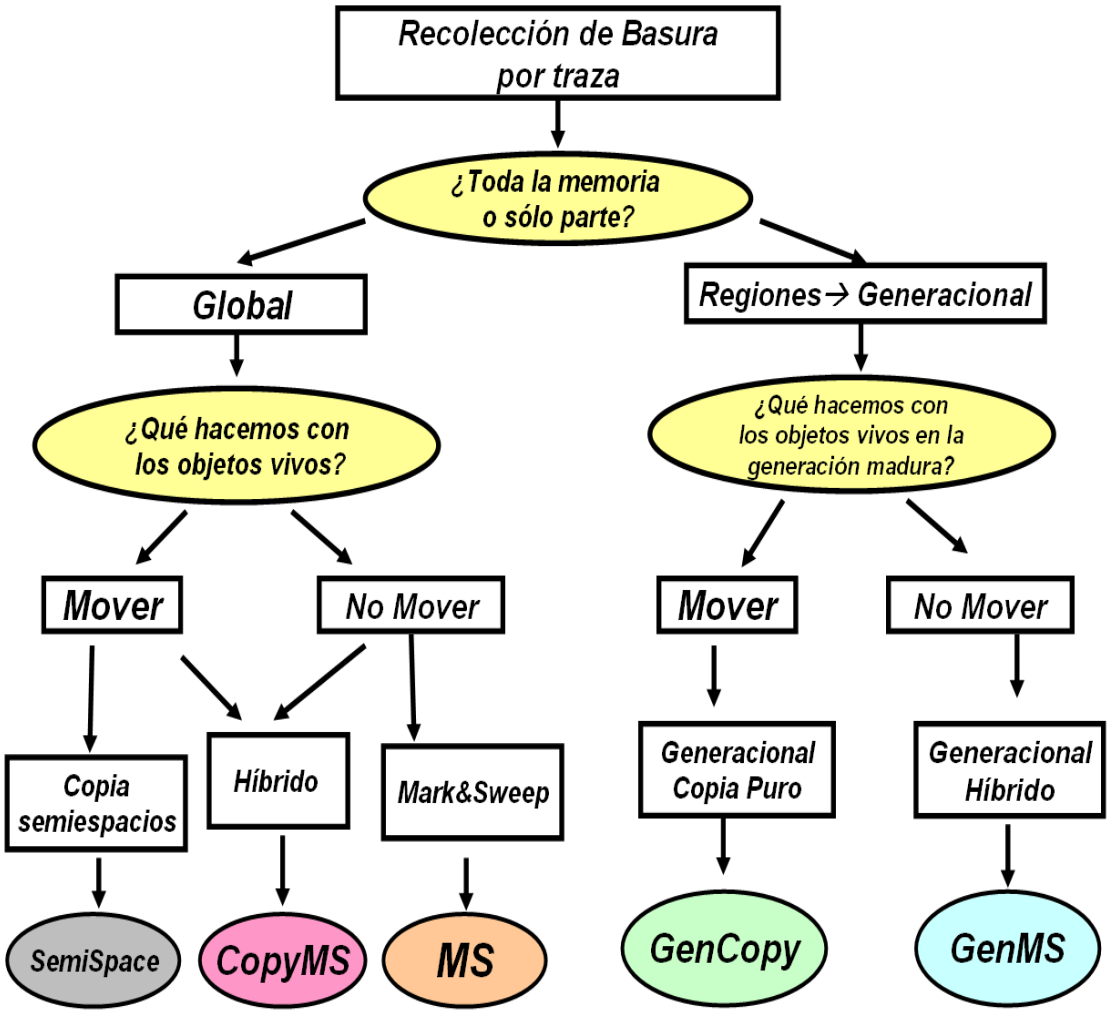


Figura 2.4: Clasificación de los recolectores de traza.

2.5.1 Recolector de marcado y barrido

El recolector *Mark&Sweep* (MS) durante la fase de recorrido del grafo de relaciones en busca de objetos vivos simplemente va marcando los objetos que visita. Así pues el primer requisito de este recolector es que los objetos tengan una cabecera (*header*) con un bit que indicará si el objeto ha sido visitado o no. Una implementación alternativa sería tener una estructura de datos con las referencias a los objetos y un bitmap global. La fase final de barrido consiste en recorrer todo el heap buscando los objetos que no han sido marcados y enlazándolos en una lista de objetos libres para su posterior uso por el asignador.

La utilización de una única lista de objetos libres produce, tras unas cuantas recolecciones, graves problemas de fragmentación interna y externa del heap. De modo que el asignador podría verse incapaz de asignar nuevos objetos aunque la memoria libre total fuese muy superior a la necesaria. La fragmentación interna (perdida de memoria por asignar bloques de memoria más grandes que los objetos que las ocupan) se puede reducir teniendo varias listas de distintos tamaños (*segregated free lists*). No obstante, el problema de la fragmentación externa es inherente a la estratagia de marcado y barrido. Tras la asignación de objetos nuevos y el reciclaje de basura, la memoria libre puede estar dividida en bloques muy pequeños y rodeada de bloques en uso. Las peticiones de memoria superiores a estos pequeños bloques no pueden ser atendidas aunque haya suficiente memoria libre en conjunto para ello. Para solucionar este problema se puede implementar algún tipo de algoritmo de compactación, que reduzca la fragmentación moviendo los objetos (recolector *Mark&Compact* [Jon00]). La fase de compactación es muy costosa ya que los objetos han de ser movidos por el heap, puede que hasta en varias ocasiones, y por ello el recolector debe reservar esta opción para cuando la fragmentación esté degradando claramente el rendimiento de la máquina virtual.

También inherente a este recolector es la baja localidad de referencia: tras unas cuantas recolecciones, objetos de muy distinta edad y con poca relación entre sí estarán situados contiguamente.

La fase de barrido de todo el heap reconstruyendo las listas libres produce un coste proporcional al número de objetos muertos. Coste importante, que quizás no se vea compensado si la ejecución finaliza antes de una nueva recolección. Una posible solución, implementada en JikesRVM, es el *lazy sweep*. Esta estrategia saca la fase de barrido de la recolección y la sitúa durante la actividad del mutador. Cuando el asignador necesita memoria para un objeto de un determinado tamaño recorre la lista de bloques de ese

tamaño (o tamaños superiores) hasta que encuentra un objeto no marcado, que será reciclado para su uso por el objeto nuevo.

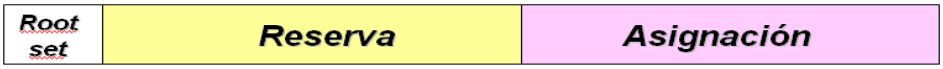
Mark&Sweep es un recolector clásico implementado en varias máquinas virtuales de Java, como Kaffe [Kaf05], JamVM [Sou04] o Kissme [Sou05], o en las máquinas virtuales de otros lenguajes como Lisp [LIS], Scheme [Sch] o Ruby [Rub]. También se usa como complemento al recolector por cuenta de referencias (para así evitar las perdidas debidas a las listas circulares [Jon00]) como en las máquina virtuales de Perl [Per] o Python [Pyt].

2.5.2 Recolector de copia

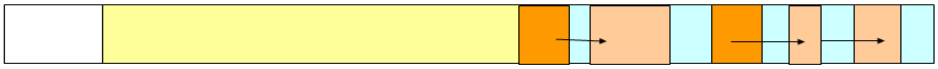
El recolector de copia también es conocido como recolector de semi-espacios (*semi-Spaces*) ya que inevitablemente ha de dividir el espacio disponible en dos mitades. Este recolector trabaja de la siguiente forma (figura 2.5):

- Divide el espacio disponible de memoria en dos semiespacios (Asignación o *fromSpace* y Reserva o *toSpace*)(figura 2.5(a)).
- La asignación de memoria se lleva a cabo en uno de los semi-espacios (*fromSpace*), dejando el otro semi-espacio como reserva para copiar en él los objetos estén vivos cuando se produzca la recolección (figura 2.5(b)).
- Partiendo, como en todos los recolectores de traza, del *root set* se recorren recursivamente los bloques de memoria referenciados (figura 2.5(c)).
- La diferencia con el *Mark&Sweep* es que ahora los bloques visitados son copiados en el otro semi-espacio (*toSpace*) en orden y de forma contigua.
- Las referencias entre los bloques se actualizan.
- Finalmente se actualizan las referencias de las variables del programa al nuevo semi-espacio.
- *Flip* (cambio de roles de los semi-espacios): las nuevas asignaciones se realizan en el semi-espacio que antes se utilizó de reserva. El primero se considera completamente libre, y sobre él se llevará a cabo la siguiente copia.

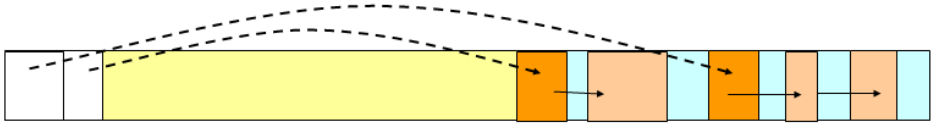
La asignación en el recolector de copia es muy sencilla. A medida que el mutador crea nuevos objetos, el asignador simplemente los situa contiguamente en el *fromSpace* sin



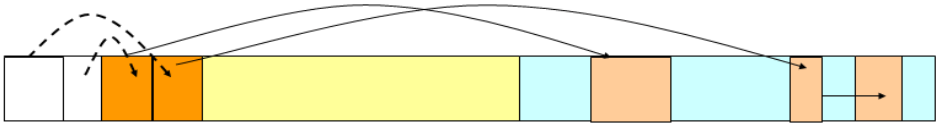
(a) El espacio disponible se divide en dos mitades



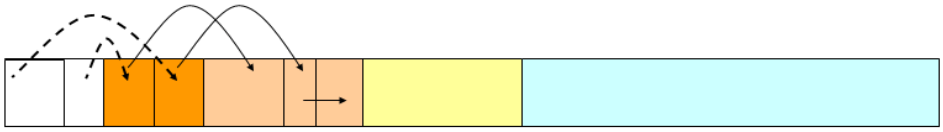
(b) Se produce la asignación hasta agotar una de las mitades



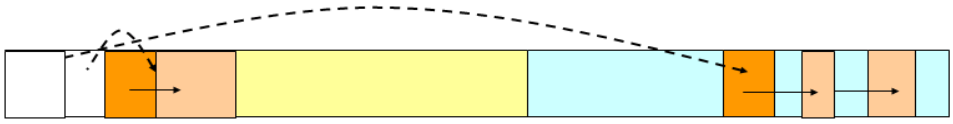
(c) Partiendo del ”root set” se recorre el grafo



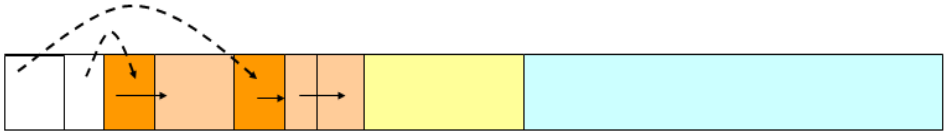
(d) Fase intermedia de la copia de objetos primero en anchura



(e) Fase final de la copia de objetos primero en anchura



(f) Fase intermedia de la copia de objetos primero en profundidad



(g) Fase final de la copia de objetos primero en profundidad

Figura 2.5: Recorrido del grafo y copia de objetos

considerar su tamaño. La única operación necesaria es sumar al puntero que referencia la siguiente dirección de memoria el tamaño del nuevo objeto (*bump pointer*). Puesto que posteriormente los objetos supervivientes a una recolección van a ser copiados al semiespacio de reserva no es necesario el uso de listas de distintos tamaños y no tiene sentido hablar de fragmentación tanto interna como externa (excepto la producida por tener un semi-espacio de reserva en el que no se produce asignación). Además, al copiar conjuntamente los objetos relacionados y de edad similar se mejora la localidad de referencia (ver figuras 2.5(d) y 2.5(e)). Este efecto se puede intensificar haciendo el recorrido del grafo de relaciones ”primero en profundidad” (ver figuras 2.5(f) y 2.5(g)). De este modo los bloques a los que referencia un objeto son copiados inmediatamente después de éste.

En la estrategia de copia no es necesario tener un bit en la cabecera en los objetos para indicar si están vivos o no (como en el Mark&Sweep), sin embargo, para evitar que un mismo objeto sea copiado varias veces es necesario que en la cabecera se indique si un objeto ha sido copiado ya (*forwarded*). Cuando un objeto ha sido copiado, en su ubicación anterior se marca el bit de copia y junto a él, se sobreescribe la nueva dirección del objeto. Así, cuando otros punteros llevan al recolector a visitar de nuevo un objeto se puede realizar la actualización de referencias.

Las principales desventajas del recolector de copia son:

- La copia de datos junto con la actualización de referencias requiere de una serie de accesos a memoria y, por tanto, tiempo extra.
- Al igual que el recolector Mark&Sweep necesita de un bit en la cabecera con el consiguiente gasto de espacio.
- La memoria disponible se reduce a la mitad. Esto es un serio inconveniente en sistemas de memoria limitada o para programas con *footprint* grande. El *footprint* es el número máximo de objetos vivos al mismo tiempo registrados en la ejecución de una aplicación.
- Los datos de vida prolongada son copiados en repetidas ocasiones.

Puesto que este recolector no necesita de la fase de barrido su coste depende únicamente del número de objetos vivos cuando se produce la recolección y no del tamaño del heap. Este recolector funciona bien cuando el tiempo medio de vida de los objetos es

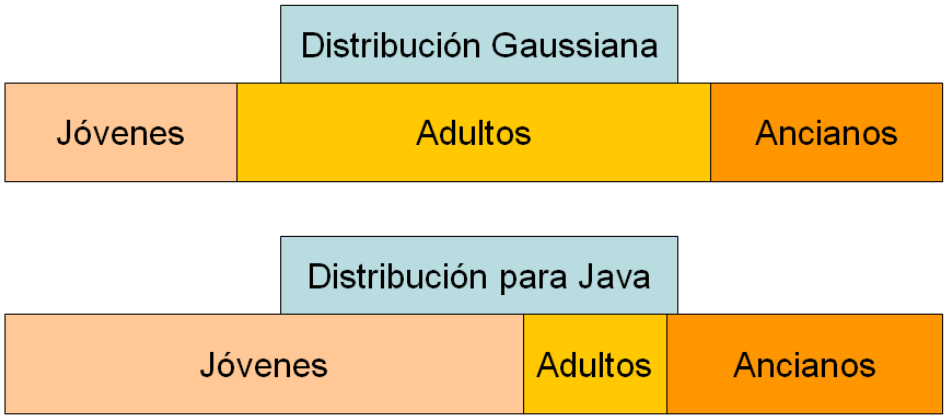


Figura 2.6: Distribución de los tiempos de vida de los objetos. Hipótesis generacional débil.

muy corto o cuando la memoria disponible es muy grande y, por tanto, las recolecciones son infrecuentes.

2.5.3 Recolector híbrido

En la figura 2.6 podemos ver como sería una distribución Gaussiana de los objetos Java según su tiempo de vida. Un porcentaje pequeño de objetos morirían con un tiempo de vida muy corto o muy grande, teniendo la mayoría de objetos un tiempo de vida medio en referencia al tiempo total de ejecución de la aplicación Java. Sin embargo, la distribución empírica, en promedio, de los tiempos de vida de los objetos se corresponde con la segunda distribución mostrada en la gráfica. En ella podemos ver que la mayoría de objetos muere muy joven, siendo ”bebés”. De los objetos que llegan a un tiempo de vida ”adulto”, la gran mayoría llegarán a un tiempo de vida muy alto y sólo una pequeña parte tendrán una vida de duración media. Esta distribución se conoce como ”hipótesis generacional débil” expuesta por D. Ungar [Ung82]. A partir de esta distribución parece claro que:

- Sería interesante que el recolector pudiese distinguir entre los objetos según su tiempo de vida y se comportase de forma distinta acorde a éste, es decir, tuviese un comportamiento híbrido.
- Puesto que el coste del recolector de copia depende del número de objetos que están vivos durante la recolección, parece lógico aplicar esta estrategía en los objetos

recientemente creados, pues esperamos una alta mortalidad.

- Como suponemos que la mayoría de los objetos que sobreviven una recolección van a tener una vida larga y por tanto la mortalidad en ellos va a ser baja, lo ideal es aplicar aquí una estrategía de marcado y barrido, cuyo coste es proporcional al número de objetos muertos durante la recolección. Así evitamos el coste asociado a la copia en objetos que esperamos que sobrevivan numerosas recolecciones.

Basados en estas conclusiones surgen el recolector híbrido, que en Jikes recibe el nombre de *CopyMS* y los distintos recolectores generacionales que se discuten en la sección 2.5.4. El recolector híbrido utiliza la sencilla asignación continua (*bump pointer*) para situar a los objetos recién creados. Cuando se produce la recolección, los objetos supervivientes son copiados pero ahora se les asigna memoria en función de su tamaño utilizando un conjunto de listas (*segregated free lists*). De este modo, en la siguiente recolección en la región de estos objetos (”maduros”) se aplica una estrategia de marcado y barrido. Para ello, los objetos recién nacidos necesitan llevar la cabecera de copia (con el bit de *forwarding* y la dirección de copia) y al pasar a maduros cambiar ésta por la cabecera con el bit de marcado.

Teóricamente, el algoritmo óptimo sería recolectar primero la zona de maduros reconstruyendo las listas libres. De este modo, al copiar los objetos supervivientes a la recolección ya se podría utilizar el espacio que dejan los maduros muertos. Sin embargo, siempre que queramos llevar a cabo la recolección en una parte del heap exclusivamente y no en el heap completo debemos conocer las referencias que parten del resto del heap hacia la región que queremos recolectar. Guardar estas referencias requiere de un espacio extra y de unas barreras de escritura que vigilen las modificaciones del grafo de relaciones entre objetos. *CopyMS* evita este coste asignando a los objetos maduros nuevos bloques de las listas libres y marcándolos como vivos. Tras la fase de recorrido del grafo los objetos en el espacio de maduros son escaneados para reconstruir las listas de bloques libres. De modo que se incurre en una penalización de espacio por fragmentación externa.

2.5.4 Recolectores generacionales

Todos los recolectores hasta ahora discutidos realizan un recorrido completo del grafo de relaciones a través del heap. Muchos de los nodos de ese grafo pertenecen a la región de objetos que sabemos que van a ser inmortales (objetos globales de la máquina virtual),

quasi-inmortales (objetos globales de la aplicación Java) o que van a tener un tiempo de vida prolongado (sobreviven a varias recolecciones).

Los recolectores generacionales se enfrentan a este problema recolectando sólo una parte del heap. Están inspirados en la hipótesis generacional débil y por tanto distribuyen los objetos basándose en su edad. Desde la creación de un objeto (nacimiento) hasta la siguiente recolección, el objeto está en la región del heap llamada guardería (*nursery*). Los objetos que sobreviven a la recolección son movidos (”promocionados”) a una región diferente (*mature space*) destinada a objetos con una esperanza de vida prolongada. La frecuencia de recolección en esta zona de objetos maduros es inferior a la frecuencia de recolección en la guardería, de este modo estamos dando más tiempo a los objetos para finalizar su tiempo de vida.

La recolección que se lleva a cabo sobre la guardería exclusivamente se conoce como *minor collection*. Y la recolección sobre la generación madura como *major collection*. En JikesRVM, cuando se recolecta el espacio de maduros se recolectan también las regiones para objetos inmortales y objetos grandes, se trata por tanto de una recolección global (*full heap collection*). Estas recolecciones globales se disparan cuando, tras una *minor collection*, la memoria disponible para la generación *nursery* es inferior a un determinado valor umbral (en JikesRVM por defecto es 512 KB).

En teoría, se pueden diseñar recolectores generacionales con múltiples generaciones. Pero, como ya se comentó en la sección 2.5.3, siempre que la recolección se lleve a cabo en una parte del heap y no en el heap en su totalidad, debemos conocer todos los punteros del resto del heap que referencien la región que queremos recolectar. Para conocer estas referencias se necesitan unas barreras de escritura que controlen la evolución del grafo de relaciones entre objetos y un espacio extra donde se registren los punteros (*meta-data*) que referencian al *nursery*. Además, muchas de estas referencias guardadas (de generaciones maduras a generaciones más jóvenes) con el paso del tiempo pertenecen a objetos que han muerto, por tanto ocupan un espacio inútil y hacen perder el tiempo al recolector obligándole a copiar objetos muertos (”nepotismo”). Por ello, el coste añadido para implementar más de 2 o 3 generaciones hace inviables en la práctica recolectores de este tipo. En JikesRVM los recolectores generacionales por defecto tienen dos generaciones.

En la literatura existen distintas soluciones para la implementación de las barreras de escritura [EHE+92], así como para almacenar el hecho de que se ha producido una referencia de una generación madura a una generacíon joven. Se puede guardar el

objeto que tiene el puntero (requiere de un bit en la cabecera), la página donde se produjo la referencia (necesita una estructura auxiliar) o como en JikesRVM [BM02], el slot concreto donde se guarda la dirección de memoria del objeto *nursery*, incluyéndolo en una estructura de datos tipo ”cola” (*Remembered set*).

JikesRVM separa en dos métodos la realización de las barreras de escritura. El *fast path* donde se produce la comprobación de si la referencia es intergeneracional, y el *slow path* donde se registra en una cola (*Remembered set*) la dirección del slot del objeto que contiene el puntero al nursery. Siguiendo la idea de Stefanovic [SMEM99], el nursery está situado en regiones de memoria superiores a la generación madura y el resto de regiones (inmortales, y *LOS*), estando todos los subheaps alineados (2^k), de este modo el *fast path* se reduce a una operación *mask-bit-and-shift*.

En teoría, se puede gestionar cada generación con cualquier política de recolección, lo cual daría lugar a numerosos tipos de recolectores generacionales híbridos. En la práctica, en las generaciones jóvenes, puesto que los objetos han de ser movidos a las generaciones más maduras, es lógico usar recolectores de copia. Por eso se dejan las estrategias de recolección que no mueven los datos para las generaciones maduras. De este modo, en JikesRVM tenemos dos posibles configuraciones: *GenMS* y *GenCopy*.

- *GenCopy* es el recolector generacional de copia puro. Sus dos generaciones se gestionan con el recolector de copia y utilizan, por tanto, asignación contigua (*Bump Pointer*). Ambas generaciones necesitan dividir el espacio disponible en dos mitades para así reservar una de ellas para la posible copia de objetos supervivientes.

- *GenMS* es el recolector generacional híbrido. La generación *nursery* utiliza recolección de copia, pero la generación *mature* es gestionada mediante la política de marcado y barrido. Como la generación de maduros no necesita reservar espacio para la copia, este recolector dispone de más memoria para la generación *nursery* y, por ello, se alcanza con menor facilidad el umbral que dispara la recolección global (ver la tabla 2.1 donde se muestra el número de recolecciones globales en promedio para todos los benchmarks con un heap de 16 MB).

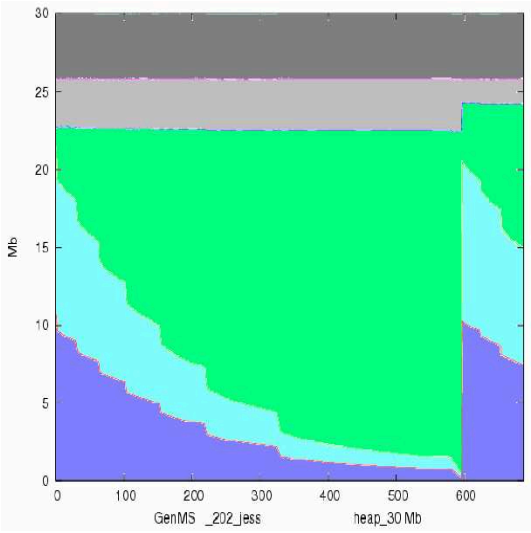
Recolector	Nursery	Global	Totales
Mark&Sweep	0	31	31
SemiSpace	0	106	106
CopyMS	0	59	59
GenMS	121	3	124
GenCopy	134	8	142

Tabla 2.1: Número de recolecciones en promedio para todos los benchmarks con un heap de 16MB.

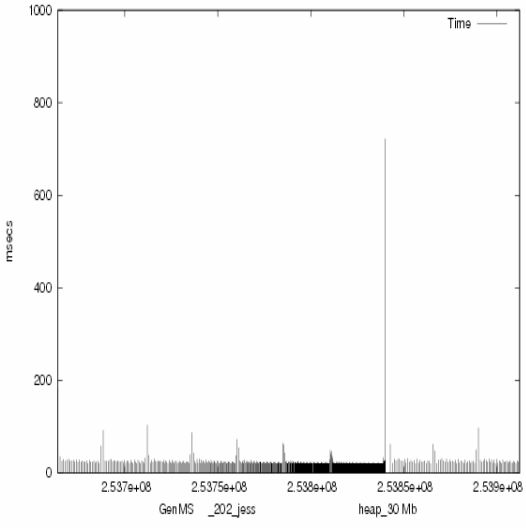
2.6 Análisis de los recolectores de traza

Para analizar el comportamiento de cada algoritmo de recolección, tradicionalmente se definen una serie de métricas:

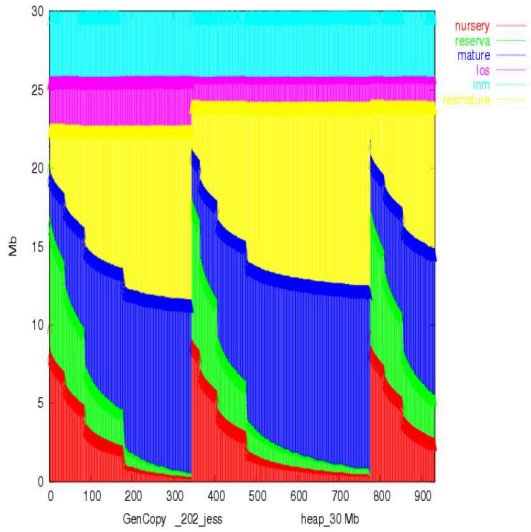
- Tiempo máximo de pausa: El mayor tiempo gastado en una recolección de basura.
- Número total de recolecciones.
- Tiempo total de recolección: es la suma de los tiempos de cada una de las recolecciones a lo largo de la ejecución de la aplicación.
- Tiempo total de ejecución: esta métrica engloba el tiempo total de recolección, la influencia del asignador, el efecto de las barreras de escritura, etc.
- MMU: durante el tiempo que dura el recorrido del grafo de relaciones la solución más segura es detener por completo la actividad del mutador. Para las aplicaciones que necesitan dar una respuesta en tiempo real esto es un incoveniente serio. Por ello se definió la métrica MMU (*Minimum Mutator Utilization*) como un modo de estudiar el tiempo mínimo en que la máquina virtual puede garantizar un progreso de la actividad del mutador.
- Heap mínimo: tamaño de memoria mínimo que la pareja asignador/ recolector necesita para poder ejecutar una aplicación sin quedarse sin memoria. Es una consecuencia de la pérdida de memoria por distintas causas (fragmentación interna, externa, etc) junto con la huella (*footprint*) de la aplicación.



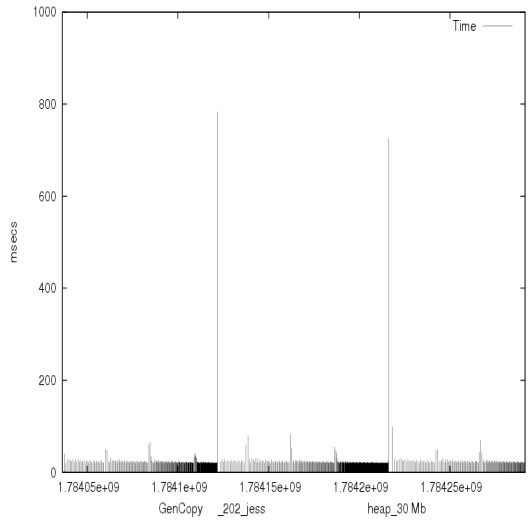
(a) Heap antes de la recolección para GenMS



(b) Pausas para recolección a través del tiempo de ejecución para GenMS

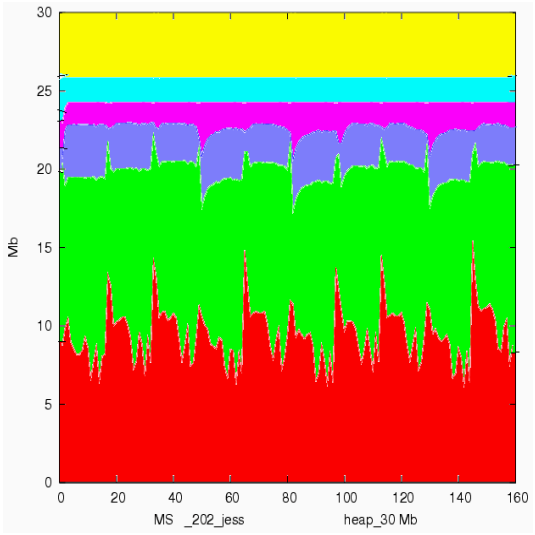


(c) Heap antes de la recolección para GenCopy

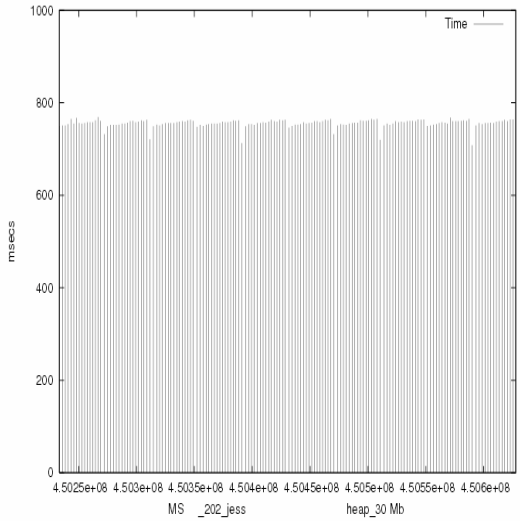


(d) Pausas para recolección a través del tiempo de ejecución para GenCopy

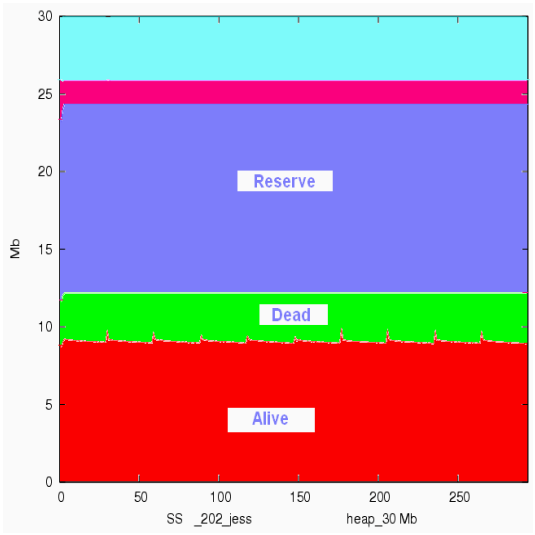
Figura 2.7: Comportamiento de los recolectores generacionales, para el benchmark _202_jess.



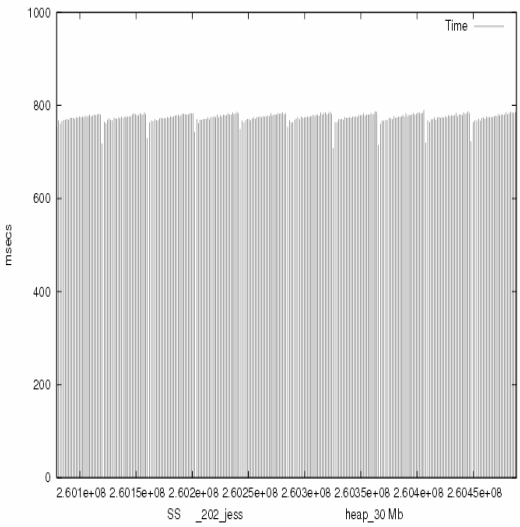
(a) Heap antes de la recolección para MarkSweep



(b) Pausas para la recolección a través del tiempo de ejecución para Mark&Sweep



(c) Heap antes de la recolección para el recolector de copia



(d) Pausas para la recolección a través del tiempo de ejecución para el recolector de copia

Figura 2.8: Comportamiento del recolector *Mark&Sweep* y del recolector de copia, para el benchmark _202_jess.

Para ilustrar estas métricas se muestran los resultados de cada recolector cuando la máquina virtual de Java está ejecutando el benchmark _202_jess. Este benchmark es uno de los más representativos del conjunto de aplicaciones que componen SPECjvm98 (sección 3.6).

En la tabla 2.1 se muestra el número de recolecciones en promedio para todos los benchmarks del SPEC con un *heap* de 16MB. Comprobamos que dentro de los recolectores no generacionales, el recolector de copia es el que más recolecciones necesita debido a la pérdida de memoria que supone el espacio de reserva. Como se aprecia, el recolector híbrido necesita la mitad de recolecciones, y el recolector de marcado y barrido la tercera parte. Los recolectores generacionales tienen un mayor número de recolecciones, pero de ellas sólo un 5% son recolecciones globales, comparables a las recolecciones de los otros recolectores. De modo que un 95% de las recolecciones son ”menores” y por tanto de un coste muy inferior.

En la figura 2.7(a) y figura 2.7(c) podemos ver la distribución de la memoria entre las distintas regiones del heap antes de cada recolección para los dos recolectores generacionales (*GenMS* y *GenCopy*) y la aplicación _202_jess. En la figuras 2.8(a) y 2.8(c), tenemos las correspondientes gráficas para los recolectores clásicos de marcado y barrido y copia. En el eje horizontal se sitúan las distintas recolecciones a través del tiempo de ejecución, mientras que el eje vertical muestra el espacio ocupado por cada región antes de la recolección. La distribución es como sigue:

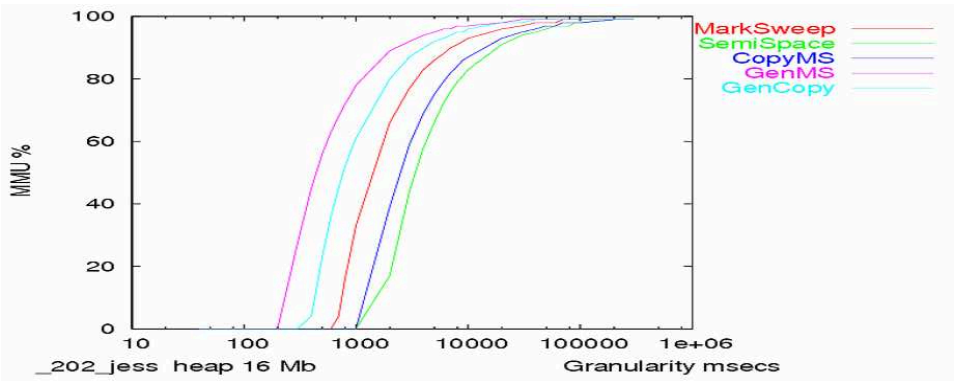
- Para GenMS: el espacio para datos inmortales se muestra en color gris oscuro, el espacio para objetos grandes (LOS) en gris claro, los objetos maduros en verde, la reserva del *nursery* en azul claro y el espacio de asignación *nursery* está en azul oscuro.

- Para GenCopy: el espacio para datos inmortales se muestra en color azul claro, el espacio para objetos grandes (LOS) en fucsia, los objetos maduros en azul oscuro, la reserva del *nursery* en verde y el espacio de asignación *nursery* está en rojo.

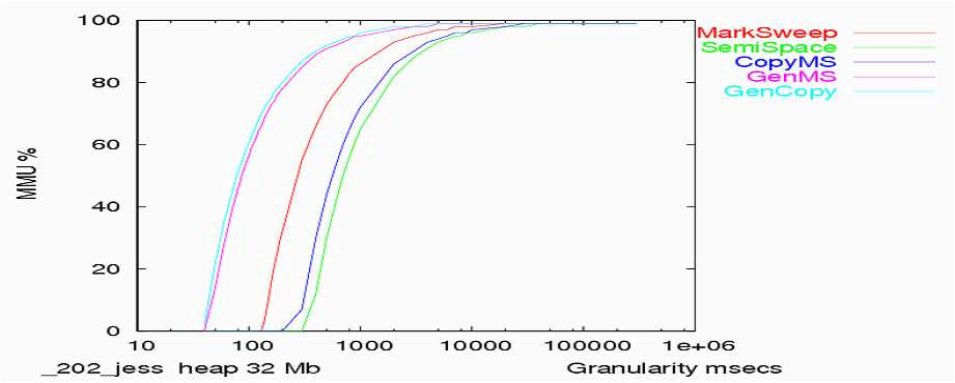
- Para el recolector de marcado y barrido: el espacio para datos inmortales se muestra en color amarillo, el espacio LOS en azul claro, fragmentación interna en fúcsia, fragmentación externa en morado, objetos muertos en verde claro y los objetos supervivientes se muestran en rojo.

- Para el recolector de copia: inmortales en azul claro, el espacio LOS en fúcsia, el espacio de reserva en morado, objetos muertos en verde claro y los objetos supervivientes en rojo.

En el caso de los recolectores generacionales, se puede apreciar como el espacio disponible para la generación *nursery* va disminuyendo tras cada recolección, de igual modo el espacio de maduros va aumentando, hasta que se produce una recolección global. Tras la recolección global, la cantidad de memoria destinada a maduros se reduce según el número de objetos que no sobreviven y el asignador dispone de ese espacio para los objetos recién creados. En la figura 2.7(b) y figura 2.7(d) podemos apreciar este comportamiento en términos de tiempos de pausa. En el eje de abcisas tenemos la distribución de las recolecciones a lo largo del tiempo de ejecución. En el eje de ordenadas la duración de cada una de las recolecciones. En esta figura se aprecia la diferencia en los tiempos de pausa entre las recolecciones que se limitan al espacio *nursery* (cuyos tiempos más altos a lo sumo alcanzan unos 70 milisegundos) y las recolecciones globales (cuyos tiempos ascienden a más de 700 milisegundos). También se puede apreciar como a medida que la generación *nursery* dispone de menos memoria el tiempo entre recolecciones se hace menor y la frecuencia de recolección, en términos de tiempo de ejecución, es mayor. En el capítulo 6 se propone una técnica para reducir el impacto de esta situación. Tras la recolección global, la frecuencia de recolección disminuye. También se puede apreciar, para *GenMS*, comparando con figura 2.7(a), como la pausa de las recolecciones menores depende de la cantidad de memoria copiada, mientras que la pausa de la recolección global es similar a cualquiera de la recolecciones del recolector de marcado y barrido (figura 2.8(b)). En el caso del recolector generacional de copia puro, se aprecian diferencias significativas (15%) en el tiempo de pausa de las dos recolecciones globales (dependiendo de la cantidad de memoria copiada en el espacio de maduros). En las gráficas 2.8(a) y 2.8(b) podemos ver que para el recolector de marcado y barrido, a pesar de que el número de supervivientes difiere mucho para cada recolección, los tiempos de pausa son muy similares para todas ellas. En el caso del recolector de copia, podemos ver que los objetos asignados apenas llegan a 12MB (figura 2.8(c)), mientras que el recolector de marcado y barrido asigna de media 20MB sin contar con las pérdidas por fragmentación. Esto produce que la cantidad de objetos supervivientes sea similar para todas las recolecciones (a diferencia de Mark&Sweep) y por tanto, los tiempos de pausa no difieran mucho entre si(figura 2.8(d)).



(a) El heap es 16MB



(b) El heap es 32MB

Figura 2.9: Utilización mínima del mutador (MMU)

También podemos ver que el espacio LOS inicialmente ocupa 4 MB, para este benchmark, y que, en el caso de los recolectores clásicos con recolecciones globales, en las primeras recolecciones este espacio se ve reducido a la mitad y permanece inalterado hasta el final de la ejecución. Sin embargo, los recolectores generacionales no pueden aprovechar este hecho hasta que no se produce una recolección global, y por ello están desaprovechando 2MB de espacio con objetos muertos durante gran parte de la ejecución (en el caso de GenMS durante casi toda la ejecución).

En la gráfica 2.9(a) se muestra la utilización mínima de mutador para los distintos recolectores con el benchmark jess y un *heap* de 16MB. La gráfica 2.9(b) es similar, pero ahora con un *heap* de 32MB. En el eje de ordenadas se situan los distintos intervalos de tiempo en milisegundos, y en el eje de abcisas, el porcentaje MMU. Así, para un

determinado intervalo de tiempo, esta gráfica nos indica, el porcentaje de utilización del mutador que cada recolector puede garantizar. Es decir, es la utilización mínima registrada a lo largo de toda la ejecución para cada intervalo. Cuanto más rápidamente suba esta gráfica hacia una utilización del 100% y cuanto más cerca del origen empiece, mejores condiciones tiene el recolector para garantizar una respuesta en tiempo real. En el caso de este benchmark, se aprecia que los mejores resultados son los registrados por los recolectores generacionales. Con un tamaño de heap de 16MB, el primero situado es el recolector GenMS seguido por GenCopy. Con tamaño de heap de 32MB, estos recolectores intercambian sus posiciones, aunque con líneas muy parejas. En los dos casos el recolector de marcado y barrido se sitúa en tercer lugar. Esto es debido a que la fase de barrido está implementada conjuntamente con la fase de asignación (*Lazy Sweep*), de modo que en la recolección sólo estamos teniendo en cuenta el recorrido del grafo de relaciones marcando los objetos vivos. En el caso de este benchmark (con un porcentaje de mortalidad muy elevado), esta métrica se ve beneficiada por ello. En último lugar, para los dos tamaños de heap, está el recolector de copia puro. Debido a que este recolector reduce el espacio disponible a la mitad, su comportamiento no es competitivo con los demás recolectores a no ser que disponga de un tamaño de heap superior en cuatro veces el tamaño mínimo necesario para la ejecución. Si comparamos las dos gráficas, podemos ver como al aumentar la memoria disponible, todos los recolectores acercan sus líneas al origen, es decir, todos mejoran sensiblemente su comportamiento respecto a esta métrica. La razón es obvia: a mayor tamaño de *heap*, mayor distancia entre recolecciones (en términos de cantidad de memoria asignada). Cuanto mayor es la distancia entre recolecciones, mayor es el tiempo disponible para que los objetos mueran. Y una mayor mortalidad implica recolecciones menos costosas.

2.7 Sinopsis

En este capítulo se han expuesto brevemente las bases de la programación orientada a objetos y se ha discutido por qué este estilo de programación necesita asignar y eliminar gran cantidad de memoria dinámicamente. La gestión manual de esta memoria dinámica es una de las tareas más complejas para el desarrollador de aplicaciones. En el capítulo hemos visto que una de las razones de la enorme popularidad de Java es la implementación de un gestor automático de la memoria dinámica que ha liberado a los

programadores de esta laboriosa tarea. El capítulo termina presentando los principales algoritmos utilizados en la recolección de basura. Como resultado de aplicar y mezclar estos algoritmos, surgen los distintos recolectores empleados en este estudio. Este capítulo termina presentando estos recolectores, algunas de las características principales de su implementación en la máquina virtual JikesRVM y analizando su comportamiento.

Capítulo 3

Entorno de simulación

3.1 Introducción

En este capítulo vamos a presentar el entorno de simulación y las distintas herramientas que hemos utilizado a lo largo de este trabajo. En la sección 3.2 se da una visión en conjunto del proceso de simulación y de los distintos elementos necesarios para ello. La principal herramienta utilizada en esta tesis es la máquina virtual de Java JikesRVM, por ello, en la sección 3.3 se realiza un estudio de ella en detalle, tratanto todos los aspectos relevantes para nuestro trabajo como son la gestión de memoria, la distribución del *heap* o la estructura de los objetos entre otros. En la sección 3.4 se trata el simulador utilizado para estudiar la interacción entre la máquina virtual y la plataforma subyacente, *Dynamic SimpleScalar*. En la sección 3.5 se comenta el simulador CACTI que, junto con el modelo de memoria SDRAM de la empresa Micron, nos proporciona los datos finales de consumo energético. El capítulo termina (sección 3.6) presentando el conjunto de aplicaciones utilizadas en nuestro estudio.

3.2 Metodología de simulación

En la figura 3.1 podemos ver el entorno completo de simulación que hemos utilizado en nuestros experimentos. Los distintos pasos necesarios para la obtención final de los resultados son:

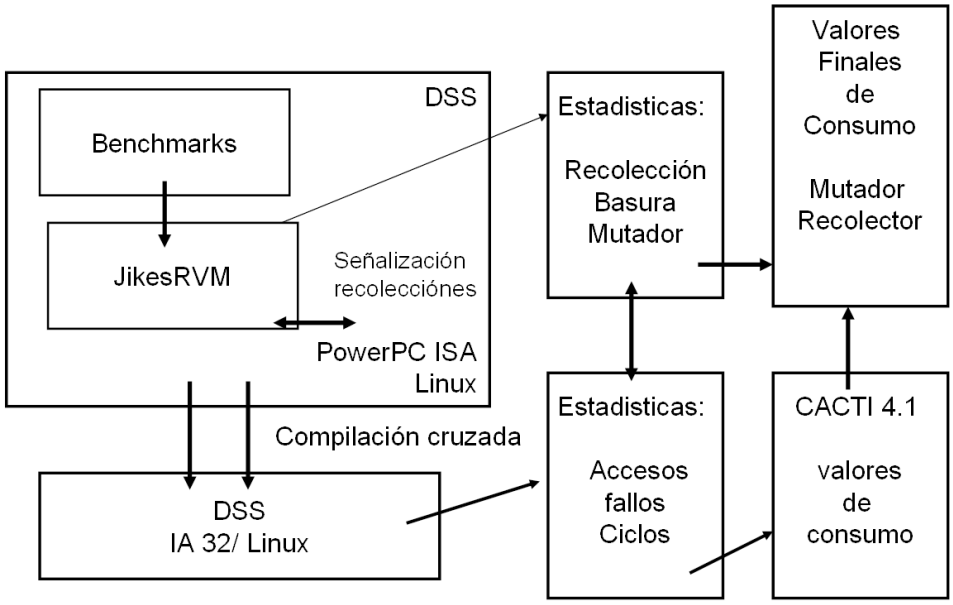


Figura 3.1: Esquema de todo el entorno de simulación

- La máquina virtual de Java, JikesRVM (sección 3.3), ejecuta una aplicación perteneciente al conjunto de benchmarks SPECjvm98 (sección 3.6). La plataforma objetivo es PowerPC/ Linux. JikesRVM nos proporciona los datos relativos al momento de inicio y fin de la recolección, lo cual nos permite diferenciar todos los resultados finales en las dos fases de la máquina virtual (mutador y recolector).

- La ejecución de JikesRVM es gestionada por el simulador Dynamic SimpleScalar (DSS sección 3.4). DSS se ejecuta sobre una plataforma IA 32/Linux. DSS nos proporciona los datos relativos a la utilización del hardware: número de ciclos, accesos y fallos a los distintos niveles de la jerarquía de memoria, fallos de página en memoria principal, etc.

- La información proporcionada por el DSS es utilizada por CACTI (sección 3.5), junto con el modelo de memoria SDRAM de la empresa Micron, para calcular el consumo de energía dinámico y estático.

Dynamic SimpleScalar es una variante del conocido simulador SimpleScalar [Aus04] que permite la ejecución de código Java. DSS se ejecuta sobre la plataforma IA 32/ Linux, sin embargo, hasta la fecha sólo simula la arquitectura PowerPC (el sistema operativo puede ser Linux o AIX). En nuestro laboratorio no teníamos acceso a esta arquitectura,

de modo que nos hemos visto obligados a realizar una compilación cruzada [Keg04] de la máquina virtual. Es decir, JikesRVM está compilado sobre hardware IA 32 y sistema operativo Linux, pero su ejecución sólo es posible sobre una plataforma PowerPC/ Linux. Aunque JikesRVM está fundamentalmente programada en Java (y por tanto debería ser independiente del hardware objetivo), esta compilación cruzada es necesaria, ya que tanto la inicialización de la máquina virtual (sección 3.3.2) como la manipulación a bajo nivel de la memoria se lleva a cabo gracias a métodos escritos en lenguaje C (sección 3.3.2).

3.3 Jikes RVM

3.3.1 Introducción a Jikes RVM

JikesRVM (Jikes Research Virtual Machine) [IBMc], versión en código abierto del proyecto originalmente llamado Jalapeño (IBM T.J. Watson Research Centre), es una máquina virtual para Java pensada para su uso en investigación y con un rendimiento competitivo respecto a otras máquina virtuales comerciales. JikesRVM soporta todas las funcionalidades de la edición standard de Java con algunas excepciones como la interfaz gráfica de ventanas, por lo cual podríamos incluirla dentro de la edición para sistemas empotrados (Java ME). Para su uso por parte de la comunidad científica, JikesRVM está diseñado modularmente, de modo que se facilita la inclusión de nuevas estrategias en el módulo correspondiente de las distintas tareas de la máquina virtual de Java. En el momento de creación de una imagen concreta de JikesRVM, el investigador puede escoger entre los distintos recolectores y optimizadores que vienen por defecto, o añadir sus propios módulos. La construcción modular se ve favorecida por el hecho de que JikesRVM está escrito principalmente en lenguaje Java. Para su inicialización, la máquina virtual necesita de un subprograma escrito en C (sección 3.3.2) y de otra máquina virtual de Java. Nosotros hemos usado la JVM de Kaffe [Kaf05]. A partir del momento en que JikesRVM está plenamente operativa, todo el código Java (incluidas sus propias clases), es ejecutado por ella. De modo que JikesRVM es una máquina virtual meta-circular. Esto implica dos factores: el código de JikesRVM está mayoritariamente escrito en Java y, en tiempo de ejecución, la máquina virtual no sólo carga y ejecuta las clases de las aplicaciones, sino también sus propias clases Java. Lo interesante de este enfoque es que otras máquinas virtuales comerciales lo han adoptado también, como es el caso de

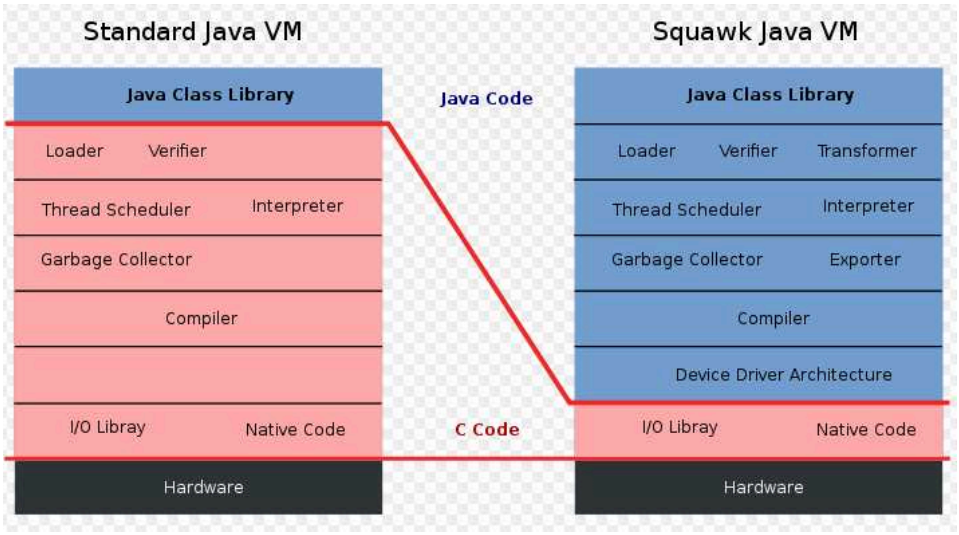


Figura 3.2: Comparativa entre la máquina virtual estandar y *Squawk*.

la máquina virtual *Squawk* (Java ME) [(JV], que ya se comentó en la sección 1.1. En la figura 3.2 podemos ver una comparación entre la máquina virtual estandar y *Squawk*. En color rosa se muestran los módulos escrito en lenguaje C, mientras que en color azul se muestran los módulo escritos en lenguaje Java. Como se aprecia en la figura, la mayoría de los módulos de *Squawk* están escritos en Java. El éxito comercial de Squawk nos hace pensar en una tendencia futura siguiendo esta línea que inició JikesRVM dentro del desarrollo de máquinas virtuales.

JikesRVM puede utiliar como compilador a Jikes (compilador de Java disponible en código abierto [IBMb] también de IBM) o a Javac (compilador de lenguaje Java de Sun).

JikesRVM, como el resto de máquinas virtuales de Java, necesita acudir, en tiempo de ejecución, a un conjunto de librerías (*Classpath*). Nosotros hemos utilizado las librerías en código abierto del proyecto GNU [url].

Para poder realizar la asignación y liberación de memoria de forma directa (lo cual está rigurosamente prohibido en la especificación del lenguaje Java) fue necesario extender Java con primitivas para el manejo de memoria. Esto se realiza en la clase VM.Magic() a la cual sólo tiene acceso la máquina virtual, nunca la aplicación. VM.Magic() se comunica (mediante la interfaz con código nativo, JNI) con métodos escritos en lenguaje C que se encargan de la manipulación directa de la memoria.

Jikes RVM dispone de tres compiladores que pueden actuar conjuntamente en la misma imagen de una máquina virtual:

- Baseline Compiler: tiene el rol de intérprete. Realiza la traducción directa de los bytecodes de Java en instrucciones máquina simplemente emulando la pila de la máquina virtual de Java.
- Quick Compiler: lleva a cabo la asignación de registros y algunas optimizaciones simples.
- Optimizing Compiler: puede realizar numerosas optimizaciones a diferentes niveles.

En los experimentos llevados a cabo conjuntamente con *Dynamic SimpleScalar*, hemos tenido que limitarnos al compilador base pues es el único soportado por éste.

3.3.2 Inicialización de la máquina virtual

El investigador debe decidir que módulos va a incluir en la RVM y crear una imagen de ella (*boot image*). Esta imagen es creada por un programa Java llamado *boot image writer*. Este programa puede ser ejecutado por cualquier JVM (*source JVM*), y como resultado proporciona una máquina virtual de Java (*target JVM*). Básicamente, el *boot image writer* hace la función de compilador y enlazador: convierte las instrucciones java (bytecodes) a código máquina y reescribe las direcciones físicas de memoria para convertir los componentes de la imagen en un programa ejecutable. El código de este subprograma se encarga de:

- Establecer el mapa de memoria virtual utilizado por la máquina virtual.
- Realizar el mapeado de los ficheros de la imagen de la RVM. En especial de aquellos, escritos en C, responsables de la comunicación con el sistema operativo.
- Preparar los registros y las tablas de contenidos Java (JTOC) (sección 3.3.3) para cada procesador virtual (hilo del procesador).

La inicialización de la RVM comienza en la instancia VM.boot(), donde se realizan una serie de operaciones: cargar las distintas clases e inicializar los distintos subsistemas (memoria, compilador, cargador de clases, planificador de hilos). El planificador se encarga de crear los hilos del procesador (pThreads) para dar soporte a los procesadores virtuales necesarios. A continuación se inicializa el subsistema encargado de la interfaz con el código nativo (Java Native Interfaz, JNI). Las clases que necesitan esta interfaz son compiladas e inicializadas (por ejemplo java.io.FileDescriptor).

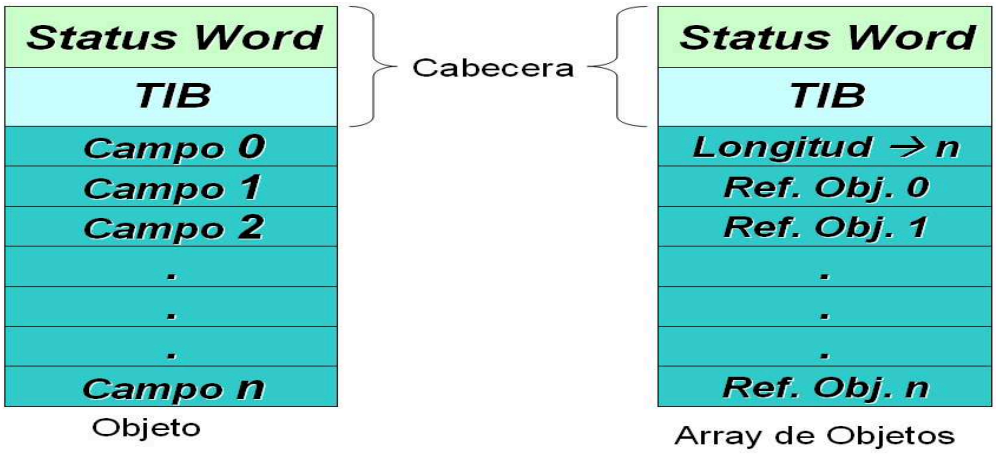


Figura 3.3: Estructura de los objetos en Java

Finalmente, la rutina de inicialización de la máquina virtual carga la clase principal de la aplicación a ejecutar. El nombre de esta clase se suministra como parámetro a la llamada a la RVM. Se crea y planifica un hilo principal (Java main thread) donde se situa el método principal de la aplicación. La ejecución termina cuando este hilo principal de la aplicación ha terminado y siempre que todos los hilos de la máquina virtual (que no son ”demonios”) hayan terminado o están ociosos (por ejemplo, los hilos del recolector están ociosos mientras no están ejecutandose o no sean planificados).

3.3.3 Modelo de objeto Java

El modelo de objeto fija el modo en que los objetos se almacenan en memoria y la forma de acceso a las operaciones más frecuentes del lenguaje. En JikesRVM el modelo de objeto viene definido en la clase *ObjectModel*.

En Java las variables almacenadas en la pila pueden ser primitivas (*int, long, double, . . .*) o punteros a objetos que residen en el heap. El objeto puede ser un *array* de objetos, o un objeto escalar (*scalar object*).

La estructura de un objeto Java tal como se implementa en Jikes se muestra en la figura 3.3. Como se aprecia, la estructura es muy similar para los objetos y los *array* de objetos. A nivel lógico, en ambos, encontramos dos zonas diferenciadas: la cabecera y los campos.

La cabecera del objeto está formada por dos posiciones de memoria:

- La palabra de estado (*status word*), la cual se divide en tres campos distintos. Puesto que la función de estos campos varía con los distintos módulos de RVM, su longitud se fija en el momento de la creación de la imagen de la máquina virtual a partir de unas constantes.
  - Cada objeto Java tiene asociado un estado de ”cerrojo” (*lock state*). Este campo puede ser una representación directa del estado o un puntero a un objeto de la clase *lock*.
  - El código Hash del objeto: los objetos Java tienen un identificador Hash único durante todo su tiempo de vida. Este identificador puede ser leído por el método de JikesRVM *Object.hashCode*, o, si este no estuviese disponible, por *System.identityHashCode*. Por defecto, el identificador Hash del objeto es su dirección en memoria, y este campo no tiene uso. Pero si la máquina virtual emplea una política de copia para la recolección, necesita utilizar este campo para guardar el identificador único del objeto a lo largo de las posibles copias a través del heap.
  - El campo utilizado por el recolector. Dependiendo del recolector implementado en la máquina virtual, este campo puede consistir en el bit de marcado (*Mark&SWeep*), bit de copia y *forwarding* (capítulo 2, sección 2.5.2), contador de referencias, ...

- El puntero al TIB (*Type Information Block*). El TIB es una estructura donde se almacena toda la información que define a un tipo. Un tipo puede ser una clase, un array o una variable primitiva. En memoria, el TIB es un array de objetos. En el caso de las clases, el primer elemento es un puntero a otra estructura donde se guarda la información de las clases que hereda o interfaces que implementa y por tanto la información de todos los campos, métodos disponibles y su situación en el array del tipo. El resto de los elementos son referencias a las direcciones donde se guardan compilados los métodos de la clase. De este modo, los pasos que la máquina virtual sigue para ejecutar un método son (figura 3.4):
  - Buscar en la cabecera del objeto la referencia al TIB.
  - En el TIB se guarda la referencia a la información de la clase.
  - En la información de la clase se guarda la localización de los campos en el

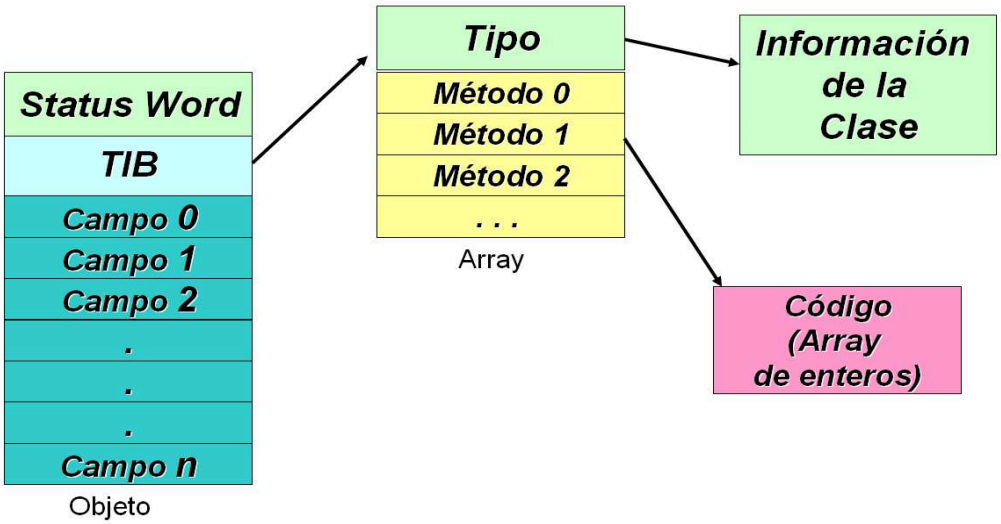


Figura 3.4: Ejecución de un método a partir de la referencia al TIB

objeto y la situación de los punteros a los métodos en el TIB (desplazamiento correspondiente en el array de referencias).

- En el array del tipo en el TIB están los punteros a las direcciones donde se guardan, en forma de array de enteros (PowerPC) o array de bytes (Intel), los métodos de la clase compilados.

Para facilitar el acceso a los elementos de un array, la referencia al array apunta al elemento cero. El resto de elementos están situados en orden ascendente. De este modo, y asumiendo que la referencia al array está en un registro, el acceso a un elemento se realiza en una única instrucción de direccionamiento más índice. El número de elementos del array se sitúa justo en la dirección previa al elemento cero.

En los objetos los elementos del array son atributos del objeto, valores primitivos o bien referencias a otros objetos. Si se trata de un array de objetos, cada elemento es un puntero a la dirección donde se encuentra el objeto.

Normalmente los campos de los objetos se sitúan en el mismo orden en que son declarados por el programador. Sin embargo, JikesRVM hace algunas excepciones a esta regla para mantener una alineación correcta. Los datos *long* y *double* se sitúan primero ya que constan de 8 bytes. Igual sucede con la mayoría de las referencias (mientras consten de 8 bytes). En los demás casos JikesRVM junta datos y los rellena con bytes vacíos para mantener la alineación de 8 bytes. Por ejemplo, un dato *entero* y un dato *byte* se agrupan

del siguiente modo: primero los cuatro bytes del dato *entero*, a continuación tres bytes vacíos, y por último el byte del dato *byte*. La información relativa a la situación de estos agujeros se encuentra en la información de la clase y se transmite a las clases herederas.

En Java para poder ejecutar un método definido en una clase, primero hay que crear (instanciar) un objeto de esa clase, con la excepción de los métodos estáticos. El programador puede ejecutar estos métodos sin necesidad de disponer de un objeto (una disyuntiva de la POO). Puesto que no hay un objeto, el acceso a estos métodos no se realiza a través del TIB, sino de otra estructura llamada JTOC.

EL JTOC (tabla de contenidos Jalapeño) se implementa en la clase VM_Statics. En él se guardan referencias a todas las estructuras de datos globales, y a cualquier variable definida como constante. Para poder realizar comprobaciones de tipos en tiempo de ejecución de forma rápida, el JTOC tiene referencias a todas las entradas de cada clase en el TIB. Estas referencias incluyen tanto a las clases de la aplicación como a las clases de la RVM. La referencia al inicio del JTOC se almacena en un registro (*JTOC Register*). Paralelamente al JTOC, se crea una estructura llamada *JTOC Descriptor*. Esta estructura es un array de bytes que guarda en la posición i-ésima el tipo de información almacenada en elemento i-ésimo del JTOC.

3.3.4 Distribución del Heap

El heap es el lugar de la memoria donde se guardan los objetos. En JikesRVM, el heap consta de tres regiones (figura 3.5):

- Región para datos inmortales. En esta región se almacenan los datos definidos como globales, estáticos, las constantes, . . . Es decir, datos que permanecen vivos durante toda la ejecución de la aplicación.

- Región para objetos grandes, LOS (*Large Object Space*). En el LOS se guardan los objetos cuyo tamaño es superior a un cierto umbral. Este umbral ha sufrido modificaciones con las distintas versiones de JikesRVM. En la versión utilizada por nosotros el umbral es 16KB. Estos objetos nunca son copiados. Su asignación se lleva a cabo mediante una lista enlazada de bloques de tamaño 16KB.

- Región para los objetos de tamaño inferior al umbral. Esta región, dependiendo de la política de recolección, puede estar a su vez subdividida en regiones o en

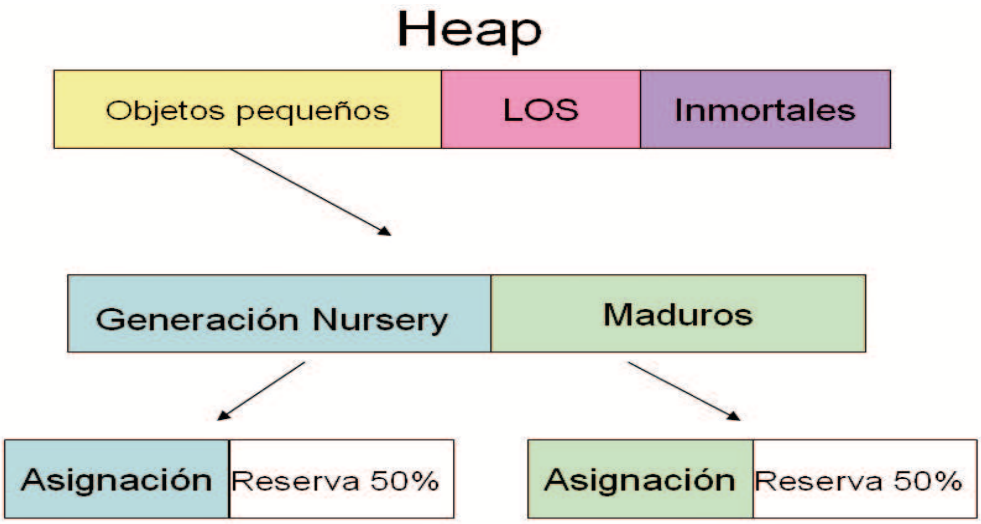


Figura 3.5: Distribución del Heap en JikesRVM para GenCopy

generaciones. En esta región se guardan tambien los datos llamados *meta-data*. Estos datos son creados por la máquina virtual y no por la aplicación. En el *meta-data* JikesRVM guarda datos necesarios para el control de las listas libres (política de marcado y barrido) o el resultado de las barreras de escritura (estrategia generacional).

En la figura 3.5 se ve un ejemplo de distribución cuando la región para objetos de tamaño menor que el umbral está gestionada por el recolector generacional de copia puro, GenCopy. Este recolector divide el espacio para los objetos pequeños en dos generaciones. A su vez, cada generación está gobernada por la política de copia y por tanto se divide en dos mitades, una para la asignación de objetos y otra que se reserva para la copia de los objetos supervivientes a la recolección.

3.3.5 Gestión de memoria

La Gestión de memoria en JikesRVM se basa en :

- Los hilos de la aplicación Java y de JikesRVM se multiplexan sobre los hilos del sistema operativo, siendo muy inferior en número los últimos respecto de los primeros. Los hilos del sistema operativo son considerados por JikesRVM como procesadores virtuales (VP). Un gestor central de almacenamiento es el

encargado de la distribución de memoria para cada VP. Para evitar la necesidad de sincronización entre los distintos hilos del sistema operativo cada vez que se produce una asignación, los procesadores virtuales reciben grandes bloques de memoria (*chunks*) para su uso privado. Cuando un procesador virtual necesita un nuevo *chunk* y el gestor central de almacenamiento no dispone de ninguno, se dispara una recolección de basura.

- Cuando la máquina virtual está funcionando como intérprete puede iniciar una recolección de basura en cualquier momento, sin embargo cuando está en funcionamiento el ”optimizador” la ejecución sólo puede detenerse en puntos concretos llamados *safe-points*. Estos puntos permiten al compilador realizar optimizaciones complejas con seguridad ya que la información relativa a la localización de punteros está almacenada en unas estructuras llamadas *GCMap* (mapas describiendo el contenido de la pila). Los *safe-points* se producen en las llamadas a métodos y retornos. JikesRVM permite al usuario escoger entre distintos niveles de optimización o dejar que sea la máquina virtual quien dinámicamente se mueva entre ellos durante la ejecución. Sin embargo, Dynamic SimpleScalar sólo soporta el nivel más bajo de optimización.

- La pila puede contener una mezcla de los datos generados por cada compilador. Para ello, estos están agrupados dentro de marcos de referencia en la pila. Cada compilador genera objetos llamados *MapIterator* que son usados para moverse a través de estos marcos de referencia de la pila. El *MapIterator* presenta una interfaz común, independiente del compilador, que es usada por los diferentes recolectores cuando acuden a la pila para buscar las variables raíz del programa.

- La utilización conjunta de los *safe-points*, los *MapIterator* y los distintos descriptores de clase permite al recolector conocer la localización de todos los punteros y sus correspondientes tipos durante la ejecución (*Type Accuracy GC*).

3.3.6 Soporte para la recolección en paralelo

Por defecto, todos los recolectores heredan de la clase *Stop-the-world-Collection*.

Cuando una petición de memoria no puede ser satisfecha se dispara la llamada para una recolección de basura a todos los procesadores virtuales, figura 3.6. El hilo de cada

procesador, donde se está ejecutando el mutador, al llegar a un *safe point* transfiere la ejecución al planificador el cual:

- Suspende la actividad del hilo del mutador.
- Planifica el hilo del recolector.

La coordinación entre los hilos de la recolección de basura se logra mediante barreras (*barrier synchcronization*). De modo que al iniciarse la recolección todos los procesadores virtuales han cesado su actividad como mutador (*Stop-the-world-Collection*) y se dedican por completo a la recolección de basura.

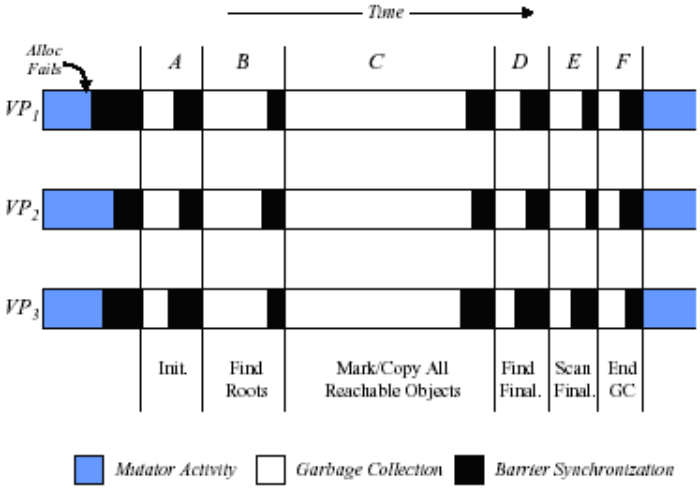


Figura 3.6: Stop-the-World-Collection. Todos los hilos se paran en *safe-points* para iniciar la recolección.

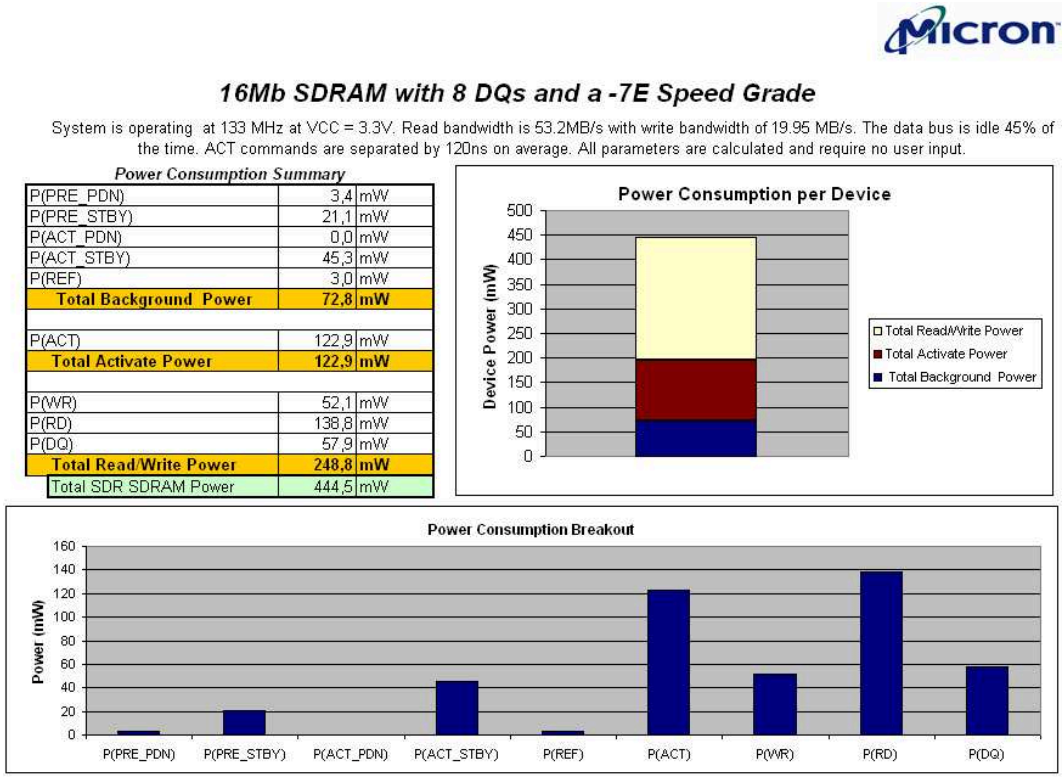


Figura 3.7: Resumen del cálculo de potencia consumida por una memoria SDRAM realizada por el *Micron System-Power Calculator*

3.4 Dynamic SimpleScalar

Dynamic SimpleScalar (DSS) [oMAtUoT04] es una evolución de SimpleScalar [Aus04], la conocidísima herramienta de simulación e investigación en arquitectura de computadores creada por Todd Austin. DSS es un proyecto conjunto de ”the University of Massachusetts at Amherst” y ”the University of Texas at Austin”. DSS implementa soporte para la generación dinámica de código ejecutable (tal como sucede en los compiladores *Just-In-Time* utilizados en la JVM), carga dinámica (típica de Java), sincronización entre hilos, mecanismos de señalización y gestión y recuperación de excepciones. Todo ello permite la simulación completa de la interacción entre las aplicaciones Java y la máquina virtual de Java, como es el caso de los benchmarks SPECjvm98 siendo ejecutados por JikesRVM. Además, DSS mejora el modelo de memoria de SimpleScalar, e implementa algunas nuevas características como el *checkpointing*, necesario para la compilación dinámica.

DSS permite simular la arquitectura PowerPC con sistema operativo Linux o AIX. Aunque puede ser ejecutado sobre varias plataformas, la arquitectura objetivo de simulación en esta tesis es PowerPC conjuntamente con sistema operativo Linux. Para realizar las simulaciones se utilizó como plataforma anfitrión la Intel IA32 con sistema operativo Linux.

DSS proporciona unos dispositivos virtuales para poder comunicarse con el programa que está siendo ejecutado. De este modo la máquina virtual de Java puede ”avisar”, mediante un flag, al DSS cuando está a punto de iniciarse o terminarse una recolección de basura, de modo que el DSS, en esos momentos precisos, imprima las estadísticas correspondientes. Esto nos permite computar los distintos factores (número de ciclos, accesos y fallos) diferenciando entre mutador y recolector, para su posterior procesamiento por parte de CACTI.

Hemos simulado una jerarquía de memoria, típica de sistemas empotrados actuales, mediante el DSS:

Primer nivel de memoria: cache de instrucciones y cache de datos separadas. La latencia es de 1 ciclo. El tamaño de línea es de 32 bytes. En los distintos experimentos hemos cambiado el número de entradas de 8 a 64 y la asociatividad entre 1,2 y 4 vías.

Segundo nivel de memoria: una cache unificada de datos e instrucciones. La latencia es de 6 ciclos. Hemos experimentado con tamaño de línea 32 y 128 bytes. La asociatividad es 4. Y el número de entradas 256.

Memoria principal de 16 MB. La latencia promedio es del orden de 36 ciclos (puede variar dependiendo de si se producen fallos de página).

3.5 CACTI

CACTI [AKB02] es una herramienta de los laboratorios de HP que integra un conjunto de modelos de los aspectos más significativos de un sistema de memoria basados en las mismas presunciones y siendo, por tanto, coherentes entre sí. Estos modelos son:

- Tiempo de acceso a memoria.
- Tiempo de ciclo.
- Área.
- Corrientes de fuga (*leakage*).
- Potencia dinámica.

El uso de CACTI está enfocado a los investigadores en arquitectura de computadores cuyo ámbito de estudio es la dinámica inherente a la organización en los sistemas de memoria.

CACTI 4.1 [Lab] incluye un modelo de potencia para la pérdida de energía por corrientes de fuga y actualiza diversos parámetros de los dispositivos para reflejar los últimos avances en la tecnología de semiconductores.

El modelo para la memoria principal es el proporcionado por la empresa Micron technologies, Inc [tIa] para memorias SDRAM, mediante el *Micron System-Power Calculator* [tIb]. En la figura 3.7 se resumen sus principales características.

3.6 SPECjvm98

SPECjvm98 [SPE99] es un producto de la Standard Performance Evaluation Corp. (SPEC [SPE00a]). Esta organización ha sacado recientemente una actualización, SPECjvm08, pero la versión actual de Dynamic SimpleScalar no puede simular su ejecución.

SPECjvm98 es un conjunto de aplicaciones escogidas para medir aspectos de rendimiento de la máquina virtual de Java (compilación Just-In-Time, etc) y su interacción

con el hardware subyacente (operaciones enteras y de punto flotante, memorias cache, etc).

Previamente a la ejecución se puede definir el tamaño de los datos (s1, s10 y s100).

En los experimentos realizados utilizando sólo Jikes RVM se utilizó el tamaño ”s100” para la entrada de datos, cada benchmark fue ejecutado un número diferente de veces de forma continua para obtener un promedio de ejecución de 10 minutos y así alcanzar un régimen estacionario. Se utilizó la opción standard ”autorun”, que realiza un numero prefijado de ejecuciones sin descartar el contenido de memoria entre ellas. Este proceso se repitió 10 veces para obtener un promedio fiable.

En los experimentos realizados utilizando Dynamic SimpleScalar y debido a la complejidad de las simulaciones, se utilizó el tamaño ”s10” y diez ejecuciones de cada benchmark con la opción ”autorun”.

Descripción de los benchmarks:

- _201_compress: Es un compresor basado en el método Lempel-Ziv (LZW). Este benchmark recorre una tabla de datos buscando subcadenas comunes. Una vez halladas son reemplazadas a partir de un diccionario de código de tamaño variable. Este proceso es determinístico, de modo que la fase de descompresión no necesita de la tabla de entrada ya que ésta puede ser reconstruída inequívocamente[Wel84]. Es la versión Java de 129.compress de CPU95, aunque difiere de éste en que _201_compress comprime datos reales sacados de un fichero y no datos generados de forma arbitraria como en 129.compress. Asigna 334 MB de datos, en su mayoría en el espacio para objetos grandes (LOS), siendo además estos de larga duración. Éste comportamiento no es típico de la programación orientada a objetos.

- _202_jess: Jess es la versión Java de CLIPS, sistema experto de la NASA [FH]. Este programa resuelve problemas de coincidencia de patrones utilizando el algoritmo para el procesamiento de reglas, RETE [For82]. Por ello, la ejecución de Jess recorre una lista de datos aplicando un conjunto de cláusulas condicionales. La ejecución de CLIPS se encarga de resolver un conjunto de puzzles. Jess asigna 748 MB de datos que en su mayoría tienen un periodo de vida muy corto, siendo el promedio de supervivientes durante la recolección de basura de 2MB.

- _209_db: Este benchmark [Hei] realiza una serie de operaciones típicas de bases de datos, tales como: añadir, borrar, encontrar u ordenar datos. La base de datos ocupa

un 1MB y consiste en una tabla con nombres, direcciones y números de teléfono. Este fichero se carga de forma residente en memoria y sobre él se aplica el conjunto de operaciones predeterminadas por el fichero scr6, de 19KB. _209_db asigna 224MB de datos con un promedio de vida variado.

- _213_javac: Este programa es el compilador para Java [Hei] que viene incluido en la aplicación comercial JDK 1.0.2., de modo que el código fuente no está disponible. Durante la ejecución de _213_javac se producen un montón de recolecciones de basura exigidas por la aplicación. En nuestros experimentos, y a no ser que se especifique lo contrario, hemos desabilitado esta posibilidad para que todas las recolecciones se produzcan cuando la máquina virtual se quede sin memoria. _213_javac asigna objetos por un tamaño de 518MB, con tiempos de vida variados.

- _222_mpegaudio: Este benchmark descomprime ficheros de audio que siguen las especificaciones ISO MPEG Layer-3 [Tea]. _222_mpegaudio descomprime un fichero de 4 MB de datos de audio. Esta aplicación es un ejemplo de programación en Java que no sigue el modelo de la programación orientada a objetos, por ello la cantidad de memoria dinámica asignada es muy pequeña.

- _228_jack: Jack [McM01] es un analizador sintáctico (parser) basado en el Purdue Compiler Construction Tool Set (PCCTS) [PCC]. El fichero a analizar es el propio Jack, de modo que durante una ejecución Jack se analiza a si mismo varias veces. Este benchmark asigna 481 MB de datos en su mayoría de corta vida.

- _205_raytrace y _227_mtrt: _205_raytrace es una aplicación para la síntesis de imágenes tridimensionales basada en el trazado de rayos. En el algoritmo de trazado de rayos se determinan las superficies visibles en la escena que se quiere sintetizar trazando rayos desde el observador (cámara) hasta la escena a través del plano de la imagen. Se calculan las intersecciones del rayo con los diferentes objetos de la escena y aquella intersección que esté más cerca del observador determina cuál es el objeto visible. El algoritmo de trazado de rayos extiende la idea de trazar los rayos para determinar las superficies visibles con un proceso de sombreado (cálculo de la intensidad del pixel), que tiene en cuenta efectos globales de iluminación como pueden ser reflexiones, refracciones o sombras arrojadas. _227_mtrt es una versión de _205_raytrace que utiliza dos threads. Ambos benchmarks realizan el renderizado

de la imagen de un dinosaurio desde un fichero de 340 KB de tamaño. En total se asignan 355 MB de memoria dinámica.

Aunque _201_compress y _222_mpegaudio no suelen incluirse en los estudios de recolección de basura, ya que no asignan una cantidad relevante de memoria dinámica, nosotros los hemos incluido en este estudio por dos factores: como representación de programas reales de uso frecuente en sistemas empotrados de alto rendimiento y para así estudiar el efecto del asignador de memoria en estas aplicaciones.

Capítulo 4

Exploración del sistema de memoria

4.1 Introducción

En este capítulo mostramos los resultados experimentales obtenidos utilizando el entorno de simulación explicado en la sección 3.2. Nuestro objetivo es caracterizar el comportamiento de la máquina virtual implementada con los distintos recolectores al ejecutar los diferentes benchmarks, y registrar las principales métricas desde el punto de vista de la gestión de memoria: número de accesos, fallos, ciclos y, finalmente, consumo energético. Esto nos permitirá comprender la relación entre el recolector de basura y el sistema de memoria en la búsqueda de técnicas que nos permitan reducir el consumo energético y aumentar el rendimiento.

El análisis comparativo de los distintos algoritmos de recolección ya había sido presentado a nivel de rendimiento en el trabajo de Blackburn et al. [BCM04], pero el añadido de datos de consumo y el enfoque a nivel de sistemas empotrados es novedoso en el área. Dentro de la literatura podemos encontrar trabajos relacionados como el de Eeckout et al. [EGB03], en el que se hace una comparación de las implicaciones a nivel hardware de varias máquina virtuales de Java (incluyendo JikesRVM). En dicho estudio el recolector de basura no es el mismo para cada máquina virtual, por lo que los resultados no son consistentes desde el punto de vista de la gestión de memoria y tampoco los resultados distinguen entre las fases de mutador y recolector. En un trabajo similar ,

Sweeney et al. [SHC+04] concluyen que el recolector de basura incrementa el número de fallos tanto para la cache de datos como para la cache de instrucciones. Aunque igualmente en su estudio no se hace una comparación entre los distintos algoritmos de recolección ni se obtienen datos de energía. En un estudio más reciente [HJ06] se explora el comportamiento de la JVM a nivel de rendimiento y consumo de potencia para proponer cambios a nivel de la micro arquitectura. Sin embargo, este estudio se centra en plataformas de altas prestaciones como servidores con memorias muy grandes, de modo que sus resultados complementan nuestro estudio, que se centra en los sistemas empotrados. También podemos citar el trabajo de Shuf et al. [SSGS01] en el que se realiza una caracterización de los benchmarks incluidos en el conjunto SPECjvm98. En su estudio se muestra información detallada sobre los fallos en la cache de datos utilizando la máquina virtual de Java de Sun. Nuestro trabajo amplía el suyo presentando un análisis de la influencia de los distintos algoritmos tanto en la cache de datos como en la cache de instrucciones dentro de una jerarquía de memoria típica de sistemas empotrados.

En estos experimentos, la máquina virtual ejecuta cada benchmark 1 vez y el tamaño de los datos de entrada es el tamaño medio (s10). Con estas restricciones, se consigue que la simulación a cargo del simulador *Dynamic SimpleScalar* se sitúe en valores razonables de miles de billones de instrucciones. Así la simulación completa de cada benchmark junto con una determinada configuración de memoria y el recolector correspondiente requiere un lapso de tiempo que oscila entre dos y cinco dias en un procesador AMD Opteron 270.

Para este estudio hemos utilizado una jerarquía de memoria (típica de sistemas empotrados actuales) que consta de tres niveles:

- Conjuntamente con el procesador tenemos dos niveles de memoria cache:
  - El primer nivel cache lo constituyen dos memorias SRAM, una para instrucciones y otra para datos. Hemos experimentado con los tamaños 8K, 16K, 32K y 64K. El tamaño de bloque es de 32 bytes. La asociatividad varía entre 1, 2 y 4 vías. Inicialmente experimentamos con distintas políticas de reemplazo: *Least Recently Used (LRU)*, *First-In First-Out (FIFO)* y política aleatoria (random). Sin embargo, como los resultados fueron muy similares y con una ligera ventaja para la política LRU, en este estudio y a no ser que se especifique lo contrario, la política empleada es LRU.
  - El segundo nivel con una memoria SRAM unificada para instrucciones y datos.

Recolector	menor	Global	Totales
Mark&Sweep	0	31	31
SemiSpace	0	106	106
CopyMS	0	59	59
GenMS	121	3	124
GenCopy	134	8	142

Tabla 4.1: Número de recolecciones en promedio para todos los benchmarks con un heap de 16MB.

En los experimentos principales el tamaño de esta memoria es de 256KB, asociatividad 4 vías y tamaño de bloque de 128 bytes. La política de reemplazo es LRU.

- Exteriormente, una memoria principal SDRAM. El tamaño de esta memoria ha sido 16 MB para la mayoría de los experimentos, aunque también hemos experimentado con 32MB.

Los datos obtenidos permiten a CACTI el cálculo del consumo de energía dinámico junto con el debido a las corrientes de fuga para la generación tecnológica de $.09\mu$m.

Configuraciones similares están siendo usadas hoy en día por dispositivos comerciales de gran difusión. Como ejemplo podemos citar a:

- La familia imx31 de Freescale [iAP] con un segundo nivel de cache unificada de 128KB, y un primer nivel que, dependiendo del modelo, varía entre 4KB y 64KB. Este dispositivo empotrado dispone de varios millones de unidades vendidas en los mercados automovilístico y médico.

- El ARM1176JZ(F)-S [Proa], con tecnología *jazelle* (tecnología de ARM para el uso eficiente de Java en sistemas empotrados), con un primer nivel de cache separado para instrucciones y datos de 16KB, y un segundo nivel unificado.

Inicialmente, los experimentos comprendían cinco recolectores: marcado y barrido(Mark&Sweep), copia puro (SemiSpace), híbrido (CopyMS), generacional puro (GenCopy) y generacional híbrido (GenMS) (en la tabla 4.1 se puede consultar el número de recolecciones necesarias para cada recolector con un heap de 16MB). Sin embargo, como se aprecia en las figuras 4.1 y 4.2, para este tipo de entorno, con fuertes limitaciones de

memoria, y en promedio para todas las aplicaciones, el recolector clásico de copia puro provoca una sobrecarga en los resultados inadmisible si lo comparamos con los resultados de los otros cuatro algoritmos. El consumo de energía de la máquina virtual implementada con este recolector durante la fase de recolección es algo más que el doble del consumo cuando está actuando como mutador. El número de ciclos durante la recolección es más de tres veces el número de ciclos registrados con el mutador. Por ello, y siempre teniendo en mente las restricciones de memoria a las que está enfocado este estudio, hemos optado por excluir este recolector en el resto de los resultados mostrados en este capítulo.

Como justificación de la importancia de este estudio, podemos ver la figura 4.3 en la que se muestra el peso porcentual de cada una de las fases (mutador y recolector), sobre el total del número de ciclos y de consumo de energía. Los resultados están promediados para todos los benchmarks y se ha excluído el recolector de copia puro. El tamaño del primer nivel de cache es de 32K, pero se han registrado resultados similares para los otros tamaños. Como se aprecia en esa figura, el peso de la fase de recolección, en cuanto a consumo de energía se refiere, se sitúa en valores que oscilan entre un 20% y un 50% del consumo total registrado durante la ejecución de las aplicaciones a cargo de la máquina virtual. La situación, en cuanto al número de ciclos, es aún más crítica, pues el peso de la fase de recolección nunca se sitúa por debajo del 40%, pudiendo llegar a representar el 70% del número total de ciclos necesarios para la ejecución. A la vista de estos valores se comprende la necesidad de un estudio en profundidad de la interacción entre la máquina virtual de Java y la jerarquía de memoria, y en especial del comportamiento del recolector de basura frente al comportamiento del mutador.

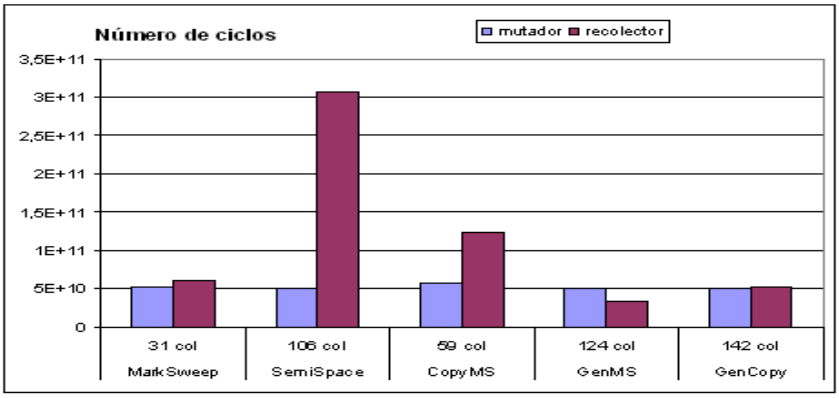


Figura 4.1: Número de ciclos para los cinco recolectores. El tamaño de L1 es 32K.

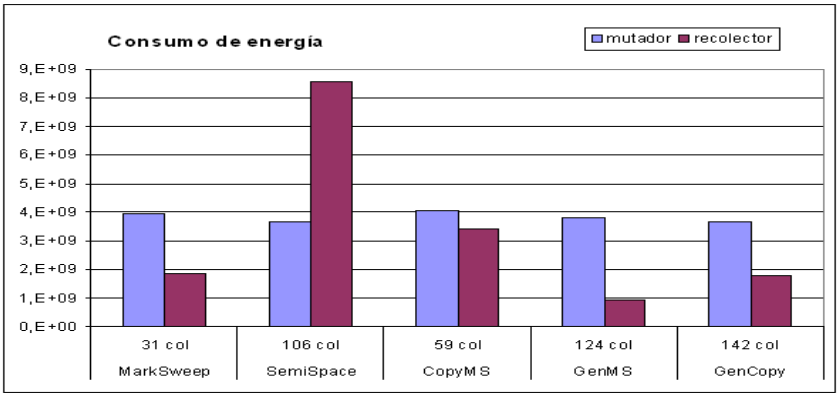


Figura 4.2: Consumo de energía (nanojulios) para los cinco recolectores. El tamaño de L1 es 32K.

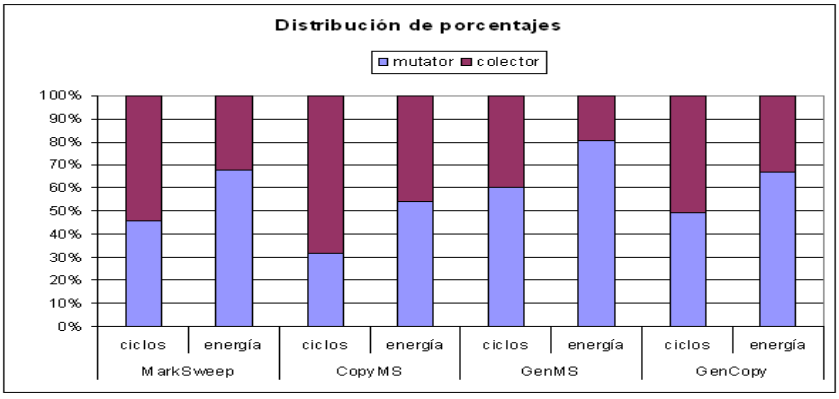


Figura 4.3: Porcentajes entre mutador y recolector para número de ciclos y consumo de energía. El tamaño de L1 es 32K.

4.2 Comparación entre algoritmos de recolección

En la introducción de este capítulo (sección 4.1) se ha mostrado el significativo peso que supone la ejecución del recolector de basura frente al mutador (ejecución de la aplicación y del resto de tareas de la máquina virtual). En esta sección vamos a mostrar que la elección del algoritmo de recolección es un factor clave, tanto en el rendimiento como en el consumo energético total, cuando la máquina virtual de java ejecuta una aplicación.

En las gráficas 4.4, 4.5, 4.6 y 4.7 se muestra una comparación de las distintas asociatividades (directa, dos y cuatro vías) para cada uno de los tamaños del primer nivel de cache estudiados. En cada una de las gráficas se pueden ver los resultados totales promediados para todos los benchmarks, del número de accesos (Global Access), fallos (Global Misses), ciclos y consumo de energía (nanojulios). Los resultados están agrupados en bloques correspondientes cada uno a un recolector. Encima de cada bloque se muestra el nombre del recolector y el número total de recolecciones (el número total de recolecciones se puede consultar en la tabla 4.1). Para comprender la leyenda consultar el apéndice A. El término Cache A-BxC hace referencia a:

- A es el tamaño en KB del primer nivel de cache.
- B y C son respectivamente el número de conjuntos y el número de vías del segundo nivel de cache.

A la vista de estas gráficas podemos sacar las siguientes conclusiones:

El número total de accesos a los distintos niveles de la jerarquía de memoria durante la fase de mutador es bastante similar para los distintos recolectores. Las diferencias se encuentran en torno a un 10%-15%, siendo el número menor el registrado por el recolector de marcado y barrido y teniendo los demás valores similares. El número de accesos durante la fase de recolección oscila entre un 30% y un 50% del total de accesos, y viene claramente marcado por el algoritmo de recolección. La distribución de accesos entre colector y mutador es similar para instrucciones y datos. En el caso del recolector de marcado y barrido la fase de mutador supone algo más del 50% de los accesos. En el caso del recolector híbrido (CopyMS), la fase de mutador no alcanza el 50%. Sin embargo, para los recolectores generacionales los porcentajes llegan al 60% (GenCopy) y 70% (GenMS). Claramente se aprecia el menor coste, en términos de accesos, de las recolecciones centradas en la generación *Nursery* frente a las recolecciones globales, que aunque son mucho mayores en número tienen peso porcentual netamente inferior.

El número de accesos durante la recolección es el dato clave que va a marcar las diferencias en cuanto a fallos, número de ciclos y consumo de energía.

El recolector con un menor número de accesos, y por tanto con mejores valores de ciclos y energía, es el recolector generacional híbrido (GenMS), seguido de cerca por el generacional de copia puro.

El número de fallos totales está directamente relacionado con el número total de accesos y por ello, depende principalmente del algoritmo de recolección. El aumento del tamaño del primer nivel de cache (así como el aumento de la asociatividad) reduce significativamente el número de fallos tanto en instrucciones como en datos, para las fases de mutador y recolector. Tradicionalmente los investigadores de la comunidad java se han preocupado principalmente por los fallos en la cache de datos durante la fase de recolección de basura. Si el recolector es de traza, es previsible que al recorrer el grafo de relaciones se produzcan muchos más fallos dentro de la jerarquía de memoria que cuando la máquina virtual está en la fase de mutador. En nuestros experimentos, el aumento de la asociatividad afecta de forma desigual a la cache de instrucciones y datos, dependiendo de la fase estudiada. En la tabla 4.2 se muestran las tasas de fallos de los distintos niveles de la jerarquía de memoria y su evolución con el aumento de la asociatividad en el primer nivel de cache (recordamos que el tamaño de línea para el primer nivel es de 32 bytes, y para el segundo de 128 bytes). La primera conclusión que podemos sacar es que las tasa de fallos de datos son menores que las tasas de instrucciones. Por otro lado, se aprecia que las tasas de fallos del primer nivel de cache de instrucciones durante la recolección son mayores que durante la fase de mutador en 2-4 puntos. Este efecto se reduce con el aumento de la asociatividad. No hay diferencias significativas en las tasas de fallos durante la recolección dependiendo del algoritmo del recolector. En el primer nivel de cache de datos, encontramos tasas de fallos más próximas entre las dos fases disminuyendo las distancias con el aumento de la asociatividad. De hecho, cuando la asociatividad es 4 vías, las tasas de fallos de datos de los cuatro recolectores son inferiores a las tasas registradas durante las correspondientes fases de mutador. Estos datos nos llevan a la siguiente conclusión: contrariamente a lo esperado, el responsable principal del consumo durante la fase de recolección, no es la cache de datos sino la cache de instrucciones. En el capítulo 5 presentamos una técnica para abordar esta situación.

En el segundo nivel de la cache nos encontramos con una situación distinta. Las tasas de fallos (tabla 4.2) durante la fase de recolector son muy inferiores a las tasas durante la

TipoGC	L1-vías	mil1	mdl1	ml2	cil1	cdl1	cl2
M&S	1	10.98	5.97	1.45	13.47	7.49	0.15
M&S	2	8.36	2.64	2.28	9.57	2.90	0.25
M&S	4	4.50	1.07	4.46	5.76	0.78	0.45
CopyMS	1	10.53	7.31	0.98	14.10	5.86	0.19
CopyMS	2	7.14	2.20	1.93	10.35	2.19	0.30
CopyMS	4	3.94	0.86	3.74	6.46	0.50	0.54
GenMS	1	10.42	6.99	1.03	13.57	6.37	0.22
GenMS	2	7.56	2.19	1.89	10.40	2.63	0.34
GenMS	4	4.19	0.90	3.58	7.40	0.76	0.49
GenCopy	1	10.19	6.14	1.20	14.11	6.50	0.38
GenCopy	2	7.52	2.24	2.04	9.93	2.64	0.63
GenCopy	4	4.18	0.91	3.84	6.86	0.77	0.91

Tabla 4.2: Tasas de fallos en promedio para todos los benchmarks. Tamaño del primer nivel de cache es 32K. El heap es 16MB. Para comprender la leyenda consultar el apéndice A

fase de mutador. Esta situación va a provocar que la memoria principal esté más inactiva durante la fase de recolector que durante la fase de mutador y , por tanto, las corrientes de pérdida van a ser también mayores. En el capítulo 5 profundizaremos en este punto y presentaremos una técnica enfocada a reducir el problema. En la tabla se aprecia que con el incremento de la asociatividad del primer nivel de cache, aumenta la tasa de fallos en el segundo nivel durante las dos fases, pero siempre manteniéndose las distancias entre fases. Pero hay que tener en cuenta que el número de accesos al segundo nivel es mucho menor por lo que un aumento en la tasa no significa necesariamente un aumento en el número de fallos en términos absolutos. A nivel de algoritmos, durante la recolección podemos ver que las tasas de los recolectores generacionales son las mayores, y el recolector generacional de copia puro es el que obtiene la tasa más elevada. Durante la fase de mutador tenemos la situación contraria, el recolector generacional de copia puro es el que obtiene las tasas más bajas de los cuatro recolectores. Este problema es muy distinto con tamaños de línea menores del segundo nivel de la cache. Así, para las configuraciones mostradas en este capítulo, hay un importante decremento lineal en el número de accesos al segundo nivel de cache con el aumento del tamaño del primer nivel

Tipo Recolector	mutador %	recolección %
Mark&Sweep	0.04	13.01
CopyMS	0.15	11.70
GenMS	0.65	9.43
GenCopy	0.16	4.31

Tabla 4.3: Porcentaje del número de accesos a memoria principal que producen un fallo de página, en promedio para todos los benchmarks con un heap de 16MB y asociatividad directa en el primer nivel de cache.

de cache. Pero este efecto está másmarcado con el aumento de la asociatividad en L1. Con un tamaño de 32K en L1, tenemos una reducción en los fallos del 65%, al pasar de asociatividad directa a asociatividad de 4 vías para el conjunto de recolectores. Para el resto de tamaños del primer nivel de cache tenemos resultados similares. Para el tamaño 8K, tenemos una reducción del 45%, para 16K la reducción es del 50% y para el tamaño máximo estudiado (64K), la reducción llega al 70%.

El número de ciclos, como los demás factores, está gobernado principalmente por el algoritmo de recolección, y siempre, como era previsible, disminuye con el aumento del tamaño del primer nivel de cache y de la asociatividad.

El aumento del coste energético que conlleva el aumento de la asociatividad en el primer nivel de cache contraresta el efecto de la disminución en el número de accesos que produce. De este modo, el aumento de la asociatividad conlleva un aumento del consumo de energía.

En la tabla 4.2 se puede ver que el número de fallos en el segundo nivel no alcanza el 2% y por tanto, los cambios en tamaño y asociatividad en el primer nivel de la jerarquía de memoria son el principal causante de las variaciones del consumo dinámico.

En la tabla 4.3 se muestra el porcentaje del número de accesos a memoria principal que producen un fallo de página, en promedio para todos los benchmarks con un heap de 16MB y asociatividad directa en el primer nivel de cache (valores similares se obtienen para las otras dos asociatividades), distinguiendo entre las fases de mutador y recolector. Como se aprecia en la tabla, los porcentajes durante la recolección son muy superiores (los valores están entre uno y dos ordenes de magnitud por encima) a los registrados cuando la máquina virtual está comportándose como mutador. Este alto número de fallos de página es debido al recorrido del grafo de relaciones inherente a la recolección basada en traza.

Podemos ver que los recolectores generacionales tienen los valores más bajos de la tabla, gracias a la sustitución de las recolecciones globales por las recolecciones menores, que no necesitan del recorrido completo del grafo de relaciones del programa. Dentro de los recolectores generacionales, el recolector de copia puro produce más accesos que el generacional híbrido, lo cual se traduce en una tasa menor de fallos de página (ver tabla 4.1).

4.3 Espacio de diseño

Este conflicto entre el aumento del consumo de energía y la reducción del número de ciclos nos lleva a la necesidad de explorar el espacio de diseño para cada tipo de recolector. En la figura 4.8 se muestra la evolución del consumo de energía en julios (eje de ordenadas) frente al tiempo de ejecución en segundos (eje de abcisas) para doce configuraciones de la jerarquía de memoria. Estas doce configuraciones provienen de los cuatro tamaños del primer nivel de cache (L1) y las tres asociatividades estudiadas (1,2 y 4 vías). La política de reemplazo es siempre LRU. La curva de Pareto óptima está compuesta por los puntos 7, 5, 8, 6 y 9. El punto 12 (tamaño de cache 64K, asociatividad 4 vías), aunque teóricamente podría incluirse en esta curva óptima, ha sido desechado ya que conlleva un aumento en el consumo de un 40% respecto del punto 9, aportando sólo una mejora en el tiempo de un 5%. El punto 6 queda completamente fuera de la curva en el caso del recolector de marcado y barrido (y ligeramente desplazado en las otras gráficas), pero lo hemos incluido porque forma parte de la curva en el caso de los demás recolectores, en especial del recolector generacional híbrido que es el que más nos interesa (por ser el de mejores resultados a nivel global). Este espacio de diseño nos indica que la solución óptima desde el punto de vista de la energía es el punto 7, que se corresponde con un tamaño de L1 de 32K y asociatividad directa. En un segundo lugar tenemos el tamaño de 16K (punto 5) y el tamaño 32K (punto 8), ambos con asociatividad 2 vías. Los puntos con mejores soluciones a nivel de rendimiento se corresponden con las asociatividades de 4 vías y tamaños 16K y 32K (puntos 6 y 9).

Hay que hacer notar que ninguna de las configuraciones para el primer nivel de cache con tamaño 64K está incluida dentro de la curva óptima ya que todas ellas resultan muy caras, porque lo que podemos concluir que no son realmente eficicentes en el contexto de la tecnología que nosotros hemos usado. Por ello, en el capítulo 5 propondremos

como alternativa al aumento de tamaño del primer nivel de cache, la inclusión de una memoria scratchpad con código de la máquina virtual de Java. Ésta y otras propuestas que mostramos en el capítulo 5, nos permitirán mejorar los resultados de consumo y rendimiento en todos los puntos del espacio de diseño, incluidos los puntos de la curva de Pareto.

Si diferenciamos el espacio de diseño entre la fase de mutador y la fase de recolector nos encontramos con diferencias en el caso del recolector de marcado y barrido. La curva de Pareto óptima para este recolector durante la fase de mutador sí comprende el punto 6. Sin embargo, es durante la recolección, donde el punto 6 se desplaza claramente fuera de la curva, mientras que el punto 11 (tamaño 64K, asociatividad 2 vías) se erige en la mejor opción a nivel de rendimiento.

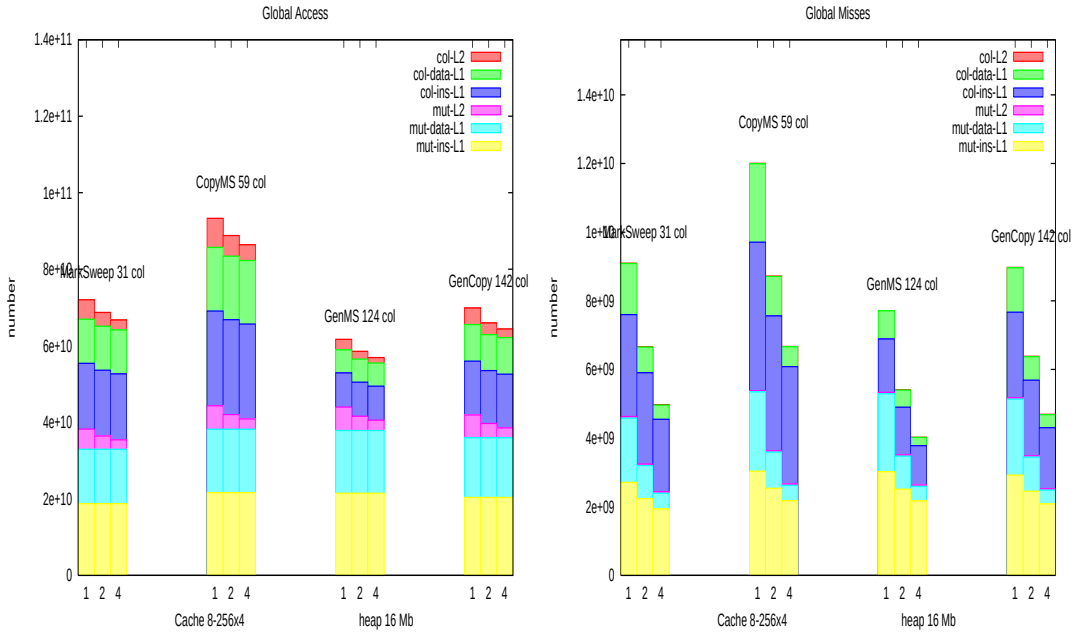
4.4 Resultados para cada benchmark

En la sección 4.2 se ha concluido que, en promedio para todos los benchmarks, la elección del algoritmo de recolección es un factor clave, tanto a nivel de rendimiento como de consumo de energía total, dentro del proceso conjunto que supone la ejecución de un aplicación por parte de la máquina virtual de java. El objetivo de esta sección es comprobar si todas las aplicaciones estudiadas se comportan de igual modo frente a la elección del algoritmo de recolección y las distintas configuraciones de la jerarquía de memoria de nuestros experimentos.

Hemos agrupado los 8 benchmarks estudiados en 3 escenarios basándonos en su comportamiento desde el punto de vista de la gestión de memoria.

En el primer escenario encontramos los benchmarks jess, jack, javac, raytrace y mtrt (figuras 4.9, 4.10, 4.11, 4.12 y 4.13). Estas cinco aplicaciones son las utilizadas habitualmente por la comunidad de investigadores en el área de gestión automática de memoria. Todas ellas asignan dinámicamente una gran cantidad de datos y son ejemplos claros de programación orientada a objetos. En su comportamiento podemos encontrar una serie de elementos comunes:

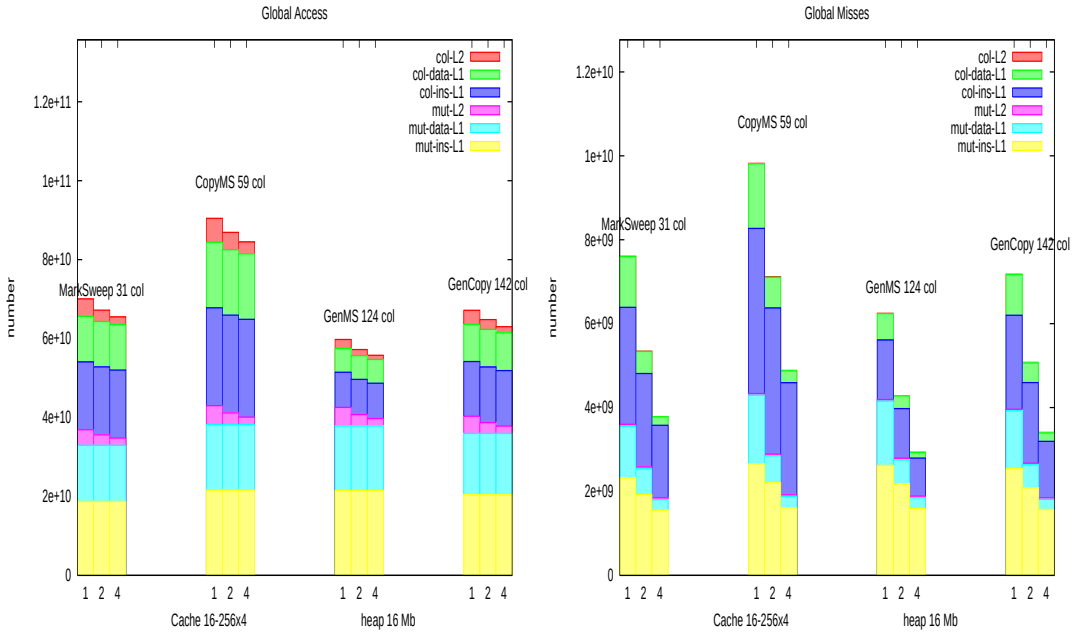
- Los accesos, en conjunto para toda la jerarquía de memoria, durante la fase de mutador son muy parecidos para la máquina virtual con los dos recolectores generacionales y el recolector híbrido. El recolector con mayor número de accesos durante la fase de mutador es el recolector clásico de marcado y barrido. Esto



(a) Accesos (b) Fallos

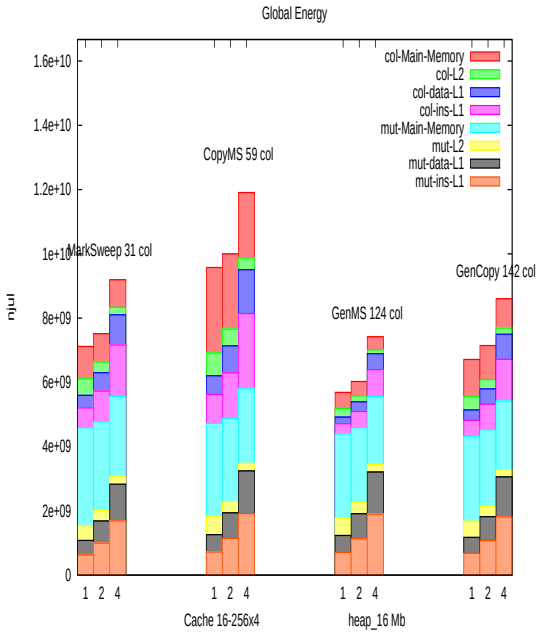
(c) Energia (d) Ciclos

Figura 4.4: Comparación entre las diferentes asociatividades. Tamaño del primer nivel de cache es 8K. Promedio de todos los benchmarks.

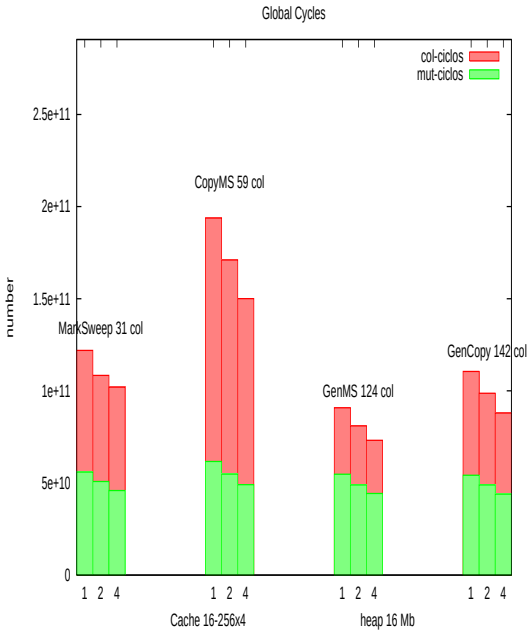


(a) Accesos

(b) Fallos

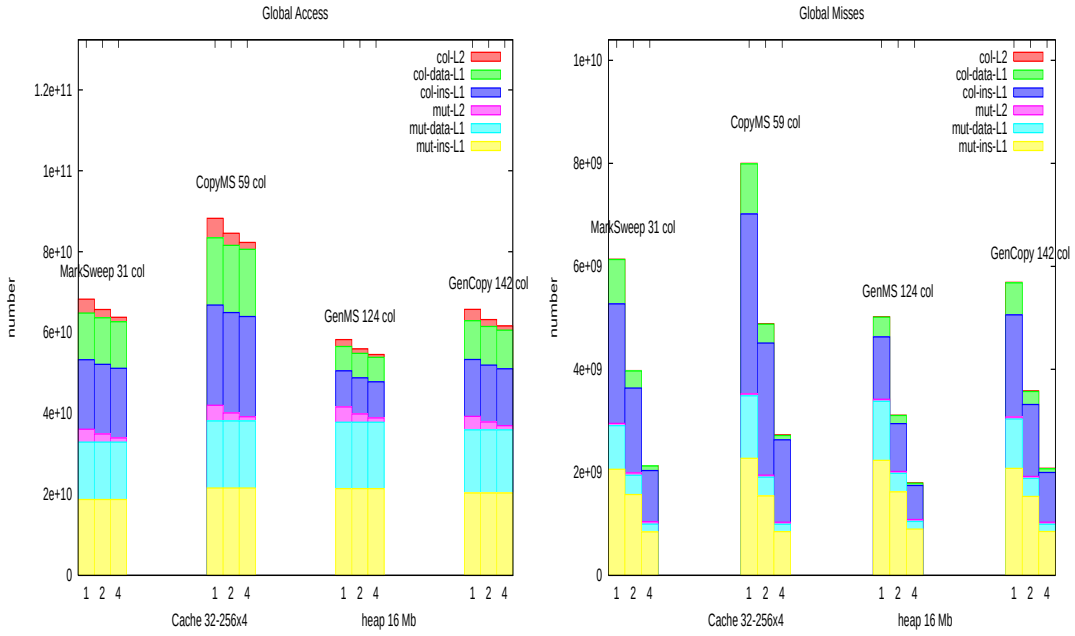


(c) Energia



(d) Ciclos

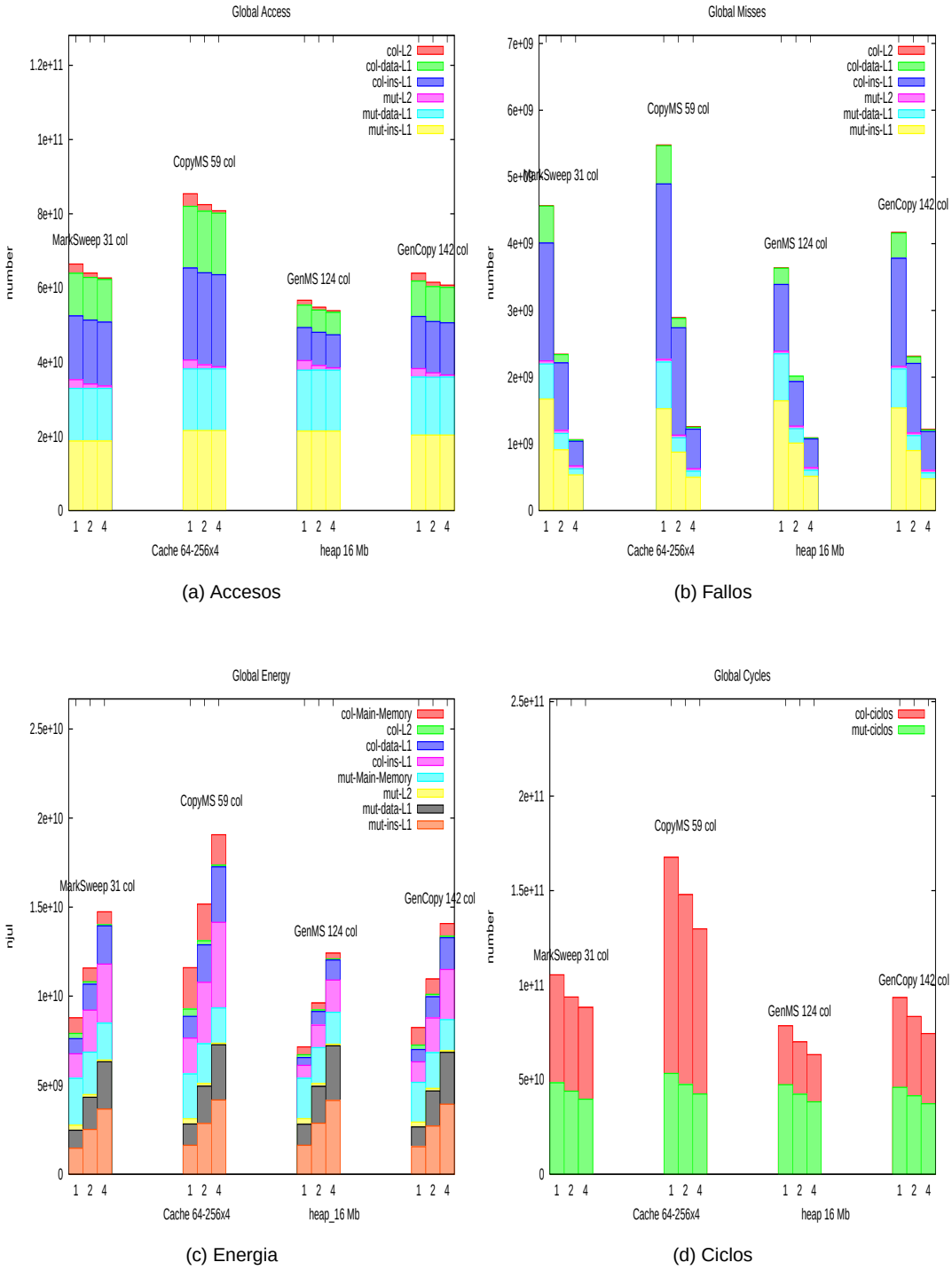
Figura 4.5: Comparación entre las diferentes asociatividades. Tamaño del primer nivel de cache es 16K. Promedio de todos los benchmarks.



(a) Accesos (b) Fallos

(c) Energia (d) Ciclos

Figura 4.6: Comparación entre las diferentes asociatividades. Tamaño del primer nivel de cache es 32K. Promedio de todos los benchmarks.



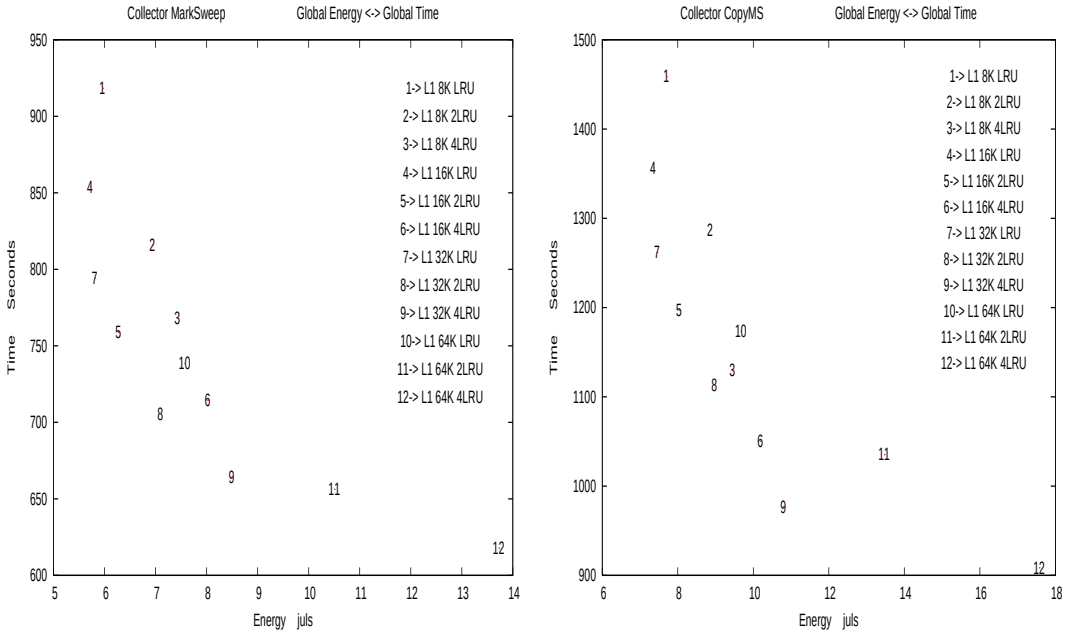
(a) Accesos

(b) Fallos

(c) Energia

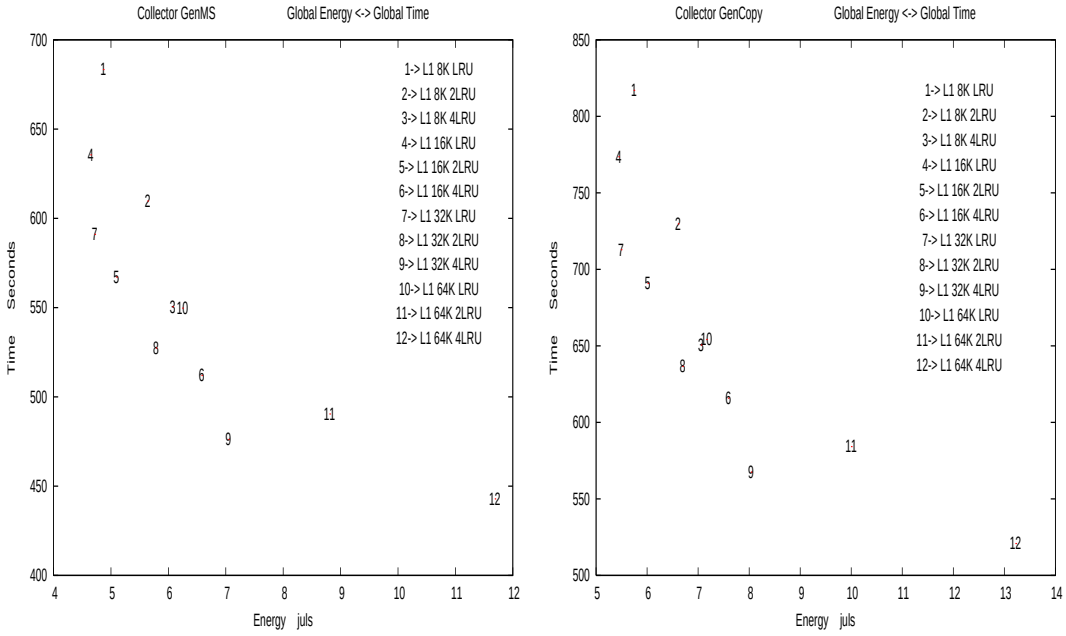
(d) Ciclos

Figura 4.7: Comparación entre las diferentes asociatividades. Tamaño del primer nivel de cache es 64K. Promedio de todos los benchmarks.



(a) MarkSweep

(b) CopyMS



(c) GenMS

(d) GenCopy

Figura 4.8: Energia versus tiempo. Comparación entre las diferentes configuraciones para un mismo recolector. Promedio de todos los benchmarks.

es debido a que en JikesRVM el barrido se produce en la fase de mutador conjuntamente con la asignación de un objeto recién creado. El gran número de recolecciones de todos los benchmarks de este escenario produce este mayor número de accesos durante la fase de mutador para el recolector de marcado y barrido. Es interesante señalar que la situación es muy distinta para los otros dos escenarios.

En el caso del segundo escenario (aplicaciones con poca producción de memoria dinámica y pocas recolecciones), el recolector que tiene un menor número de accesos durante la fase de mutador es el recolector de marcado y barrido. En el caso del tercer escenario, nos encontramos con valores muy similares entre los cinco recolectores.

- El recolector con el menor número de accesos es en todos los casos el recolector generacional híbrido seguido por el recolector generacional de copia puro. Para el benchmark jack tenemos el ejemplo con la mayor diferencia de los algoritmos generacionales frente a los no generacionales. Además, para esta aplicación se ha registrado muy poca diferencia entre el generacional GenMS y el generacional GenCopy aunque este último tiene un 30% más de recolecciones que el primero. Esta situación se debe al comportamiento de los objetos de jack. En esta aplicación encontramos en promedio el ratio más alto de supervivientes en las recolecciones nursery de modo que la generación madura se llena con una relativa rapidez. En contraste, hay muy pocos supervivientes cuando se produce una recolección global. De modo que las recolecciones que se producen sobre todo el heap tienen un coste muy bajo y parecido para los dos tipos de recolector generacional.

- En el extremo contrario al anterior se encuentra el benchmark mtrt. En esta aplicación encontramos que los valores del número de accesos durante la recolección se situan muy próximos entre los recolectores generacionales y el recolector clásico de marcado y barrido. Al mismo tiempo durante la fase de mutador, el recolector de marcado y barrido alcanza la mayor diferencia en el número de accesos respecto del resto de recolectores. Este doble efecto se explica por los tiempos de vida extremadamente cortos de la mayoría de los datos de la aplicación conjuntamente con tiempos de vida muy largos para un porcentaje pequeño de objetos. Este benchmark asigna un gran número de objetos, de modo que la memoria disponible disminuye

rápidamente, lo cual provoca un gran número de recolecciones. Sin embargo, en la mayoría de estas recolecciones el porcentaje de objetos supervivientes es nulo o prácticamente nulo. Esto produce que el coste de la fase de marcado del recolector de marcado y barrido sea muy pequeño, ya que el recorrido del grafo de relaciones es mínimo. La otra cara de la moneda para este recolector es que la mayor parte del trabajo se produce en la fase de barrido para rehacer el conjunto de listas de bloques de distintos tamaños (ejecutado en el mutador). El ratio de supervivientes tan bajo registrado en este recolector provoca que la generación de objetos maduros crezca muy poco. Este hecho genera el único caso registrado en este escenario en el que el recolector de copia puro necesita menos recolecciones que el recolector generacional híbrido. A pesar de ello, y debido al coste asociado a la copia de los objetos que sobreviven a las recolecciones globales, el recolector generacional híbrido obtiene los valores más bajos como en el resto de aplicaciones de este escenario.

En el segundo escenario tenemos a los benchmarks db y mpegaudio (figuras 4.14 y 4.15). Estas aplicaciones asignan dinámicamente muy poca memoria lo cual provoca un número muy pequeño de recolecciones, por lo cual no son incluidos, normalmente, en los estudios de recolección de basura. Como nuestro objetivo es el estudio global del comportamiento de la máquina virtual de java tanto durante la fase de recolector como de la fase de mutador, hemos creído interesante incluirlos. El comportamiento de los dos benchmarks es similar, siendo mpegaudio el caso más extremo de los dos. En estas aplicaciones la influencia de la elección del algoritmo de recolección se debe, fundamentalmente, al algoritmo de asignación. El número de accesos más bajo es el registrado por el recolector generacional de copia puro. Ello se debe a la mejor localidad que proporciona la asignación contigua y a la sencillez del asignador *bump pointer* (ver sección 2.5.2) utilizado tanto en la asignación en la generación nursery como en la generación madura. La asignación contigua también proporciona a GenCopy el número más bajo de fallos en la cache de datos registrados (figura 4.15(b)). Es interesante señalar aquí, que en el caso de mpegaudio, el recolector clásico de copia puro obtiene resultados muy parecidos al recolector generacional, contrariamente a la situación encontrada en el resto de aplicaciones.

En el tercer escenario tenemos el caso especial del benchmark compress. Esta aplicación tampoco se incluye normalmente en los estudios de gestión de memoria dinámica. Sin embargo, el comportamiento de compress es muy distinto de los programas

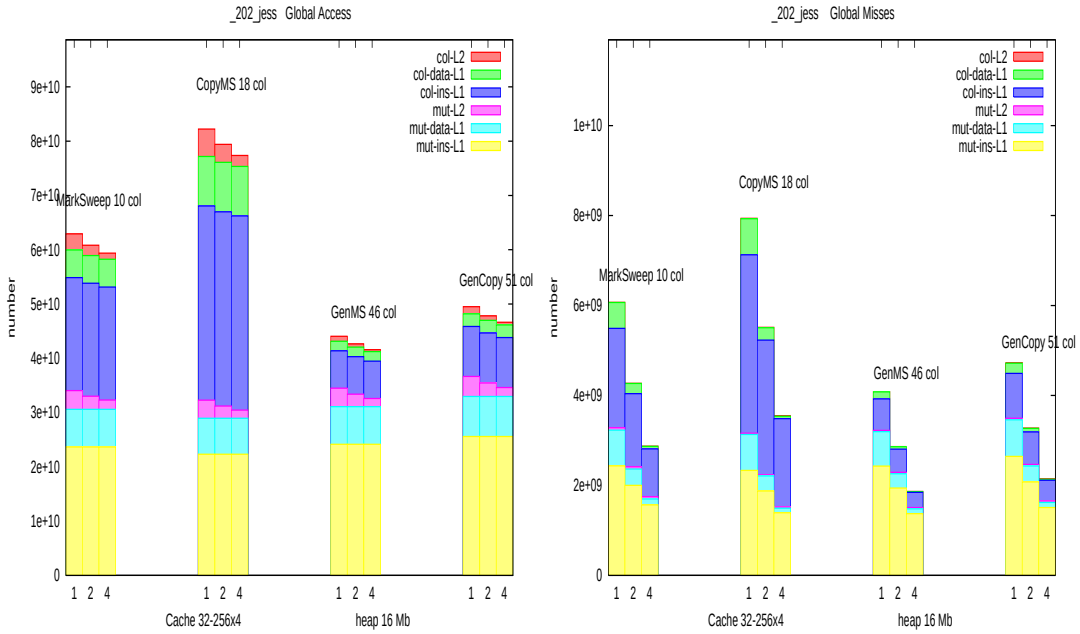
incluidos en el segundo escenario. La distribución del heap en esta aplicación difiere de la distribución en el resto de aplicaciones. En este caso encontramos una mayoría de objetos grandes (con un tamaño superior a 16K) que además son inmortales en la mayoría de los casos. De modo que la región para objetos grandes (LOS) ocupa la mayor parte del heap. En el caso de los recolectores generacionales este hecho provoca un espacio muy reducido para las generaciones nursery y madura y, consiguientemente, un gran número de recolecciones globales. Además, tenemos un gran número de referencias de objetos de la región LOS a la generación nursery que han de ser registradas por las barreras de escritura, generando gran cantidad de meta-data que aún reduce más el espacio para la generación nursery. Todo ello provoca que el recolector híbrido registre un número de recolecciones menor que los recolectores generacionales y obtenga los mejores valores en el número de accesos, ciclos y consumo de energía aunque con muy poca diferencia respecto a GenMS.

4.5 Conclusiones

En este capítulo hemos presentado los resultados experimentales obtenidos tras la simulación completa de la máquina virtual de java ejecutando los distintos benchmarks de la corporación SPEC sobre distintas configuraciones de la jerarquía de memoria, todas ellas típicas de sistemas empotrados.

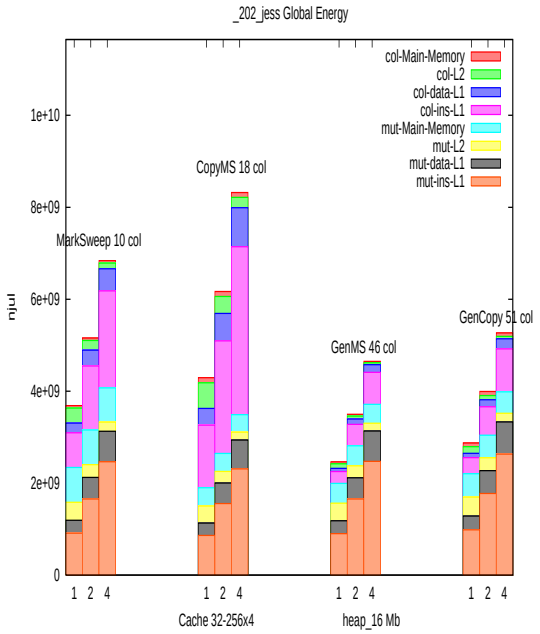
En promedio, el peso de la fase de recolección para el consumo de energía (dinámico y estático) oscila entre un un 20% y un 50% del total registrado. La influencia de la fase de recolección en el número de ciclos que la máquina virtual necesita para ejecutar las aplicaciones se sitúa entre un 40% y un 70%. Estos datos justifican la importancia de este estudio y la necesidad de profundizar en la interacción entre el recolector de basura, el asignador y la memoria.

En nuestros resultados experimentales hemos comprobado que la elección del algoritmo de recolección puede significar una diferencia de hasta 20 puntos porcentuales en el número total de accesos a los distintos niveles de la jerarquía de memoria durante la ejecución completa. Y como el número de accesos es el rasgo principal que marca el comportamiento de la máquina virtual en cuanto a fallos, número de ciclos y consumo de energía se refiere, podemos concluir que el algoritmo de recolección es una elección clave a la hora de implementar la máquina virtual de Java que va a determinar el rendimiento y

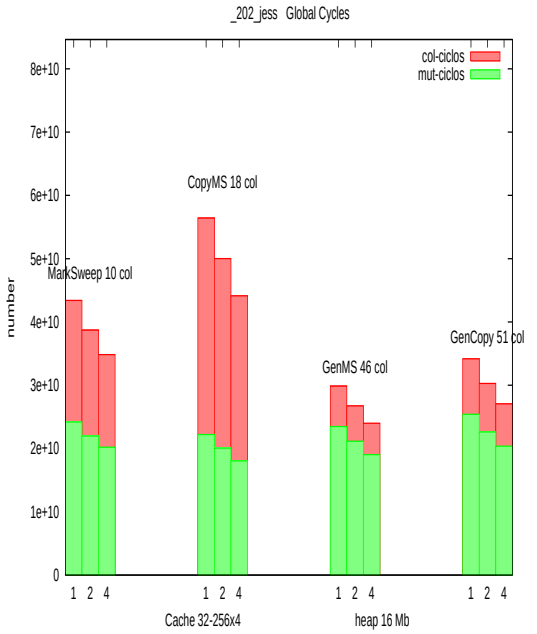


(a) Accesos

(b) Fallos

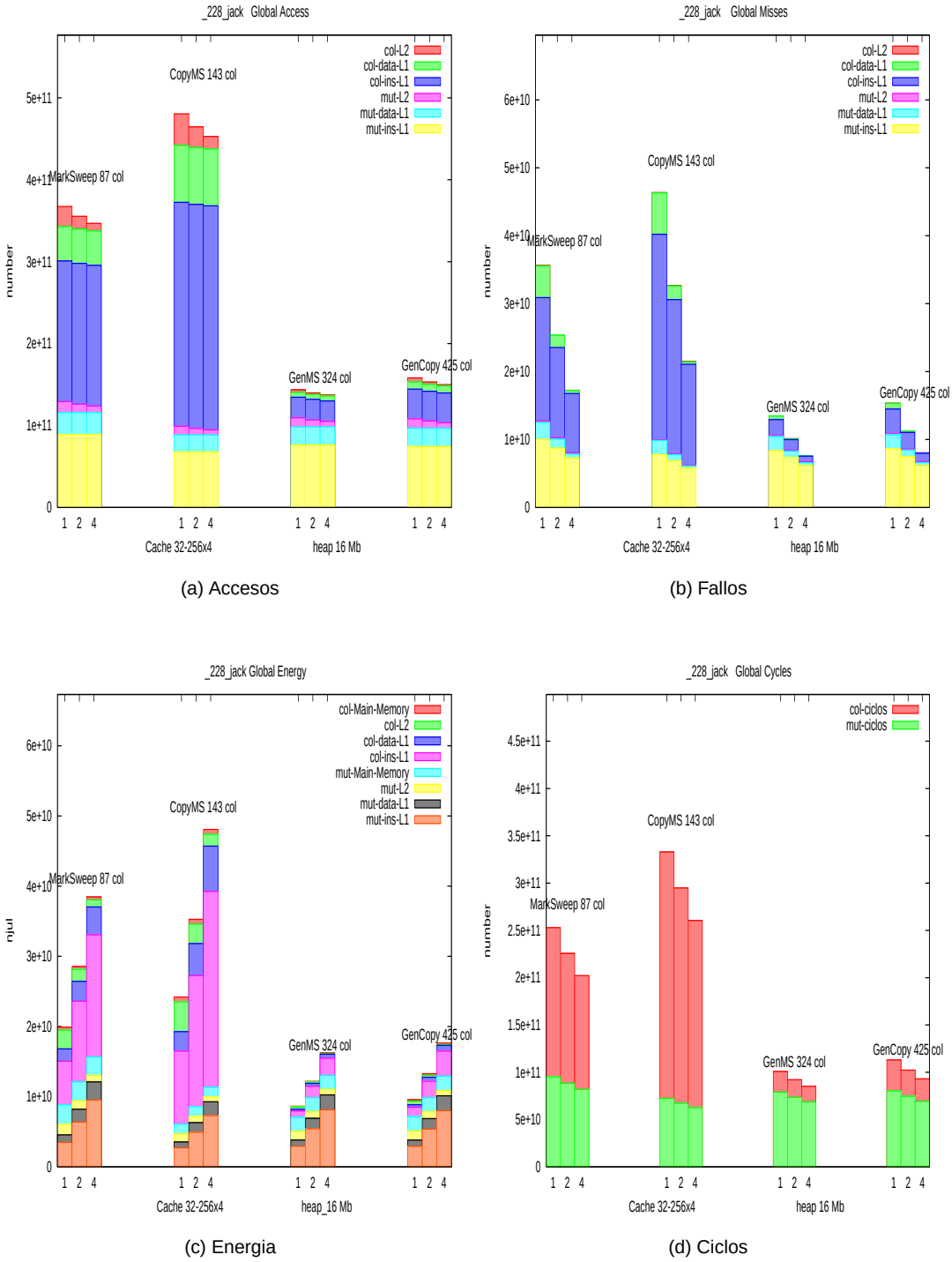


(c) Energia



(d) Ciclos

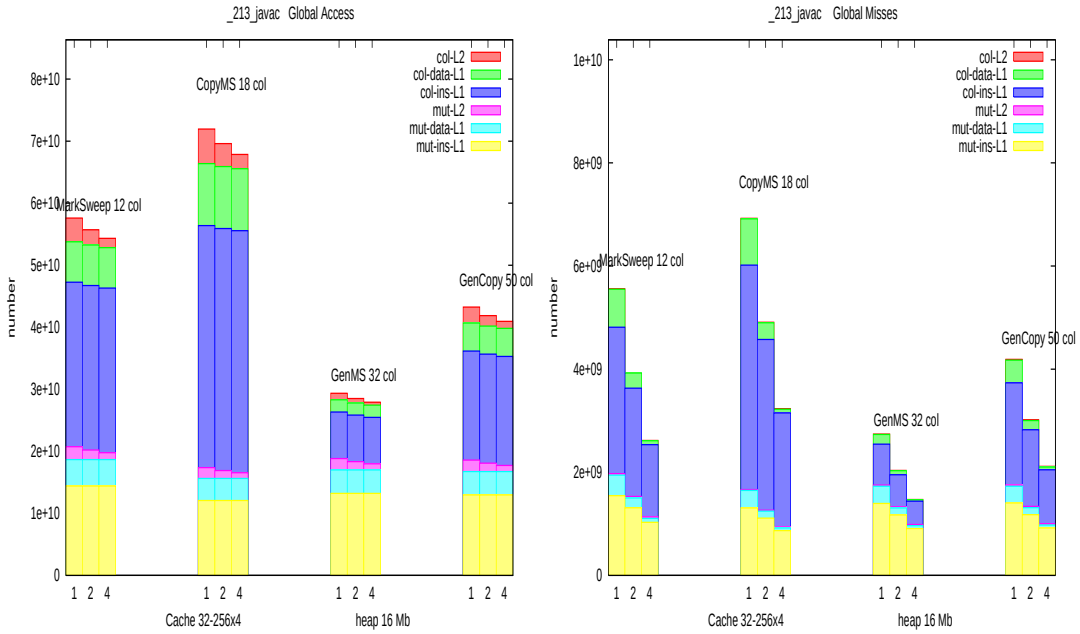
Figura 4.9: Comparación entre las diferentes asociatividades. Tamaño del primer nivel de cache es 32K. jess.



(a) Accesos (b) Fallos

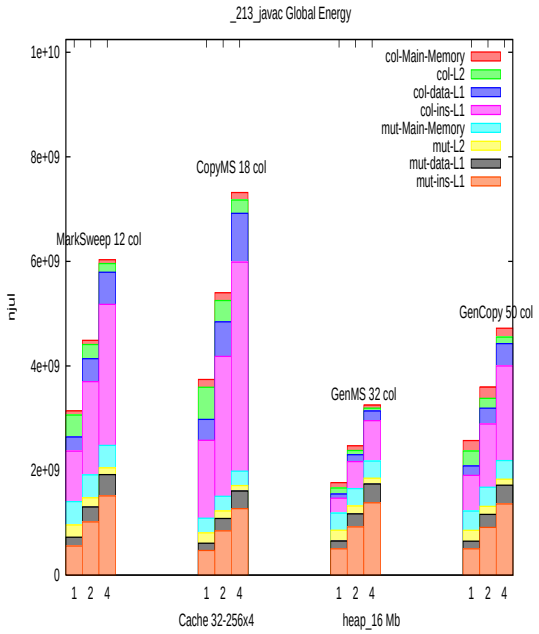
(c) Energia (d) Ciclos

Figura 4.10: Comparación entre las diferentes asociatividades. Tamaño del primer nivel de cache es 32K. jack.

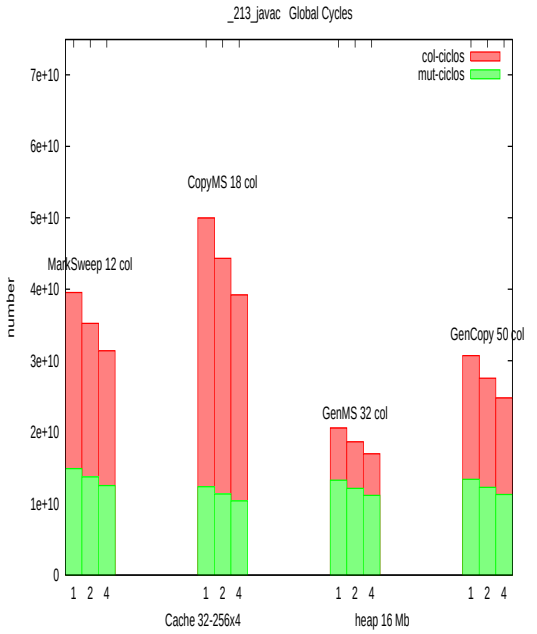


(a) Accesos

(b) Fallos

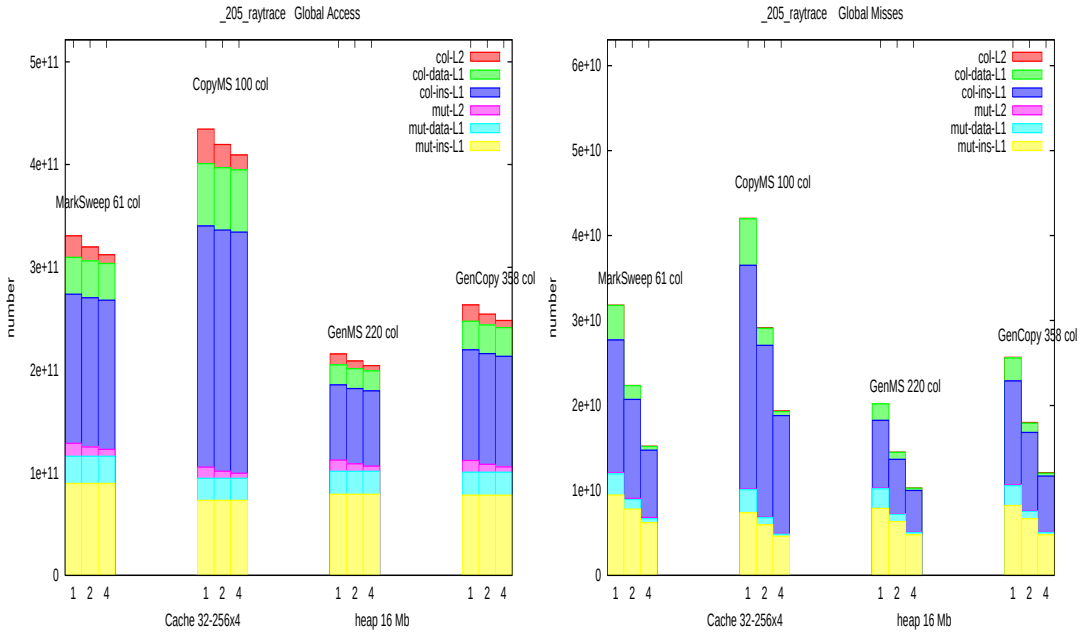


(c) Energia



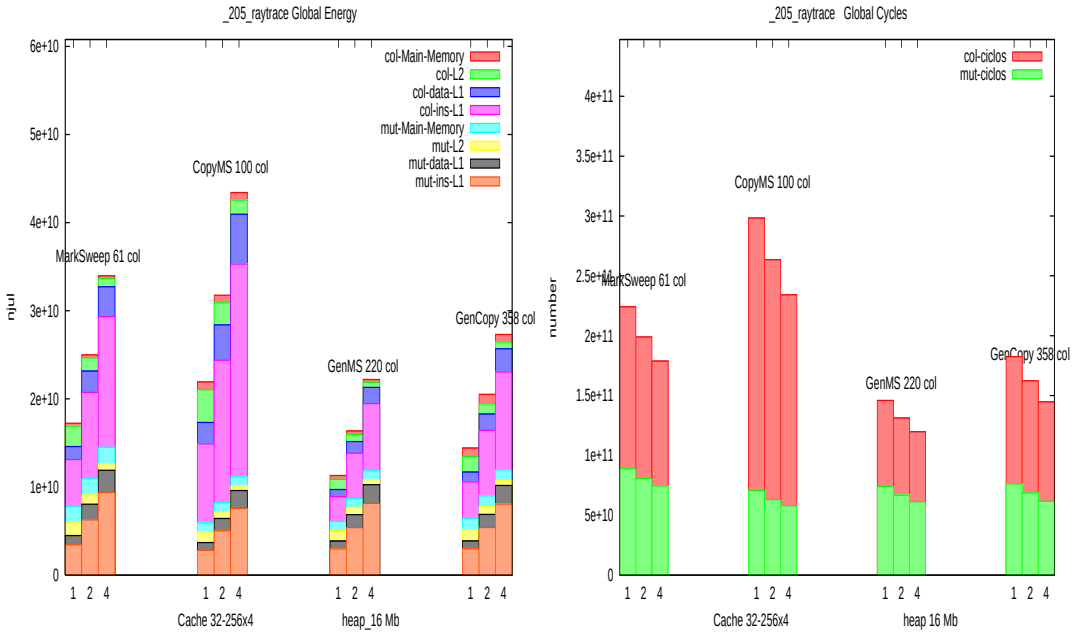
(d) Ciclos

Figura 4.11: Comparación entre las diferentes asociatividades. Tamaño del primer nivel de cache es 32K. javac.



(a) Accesos

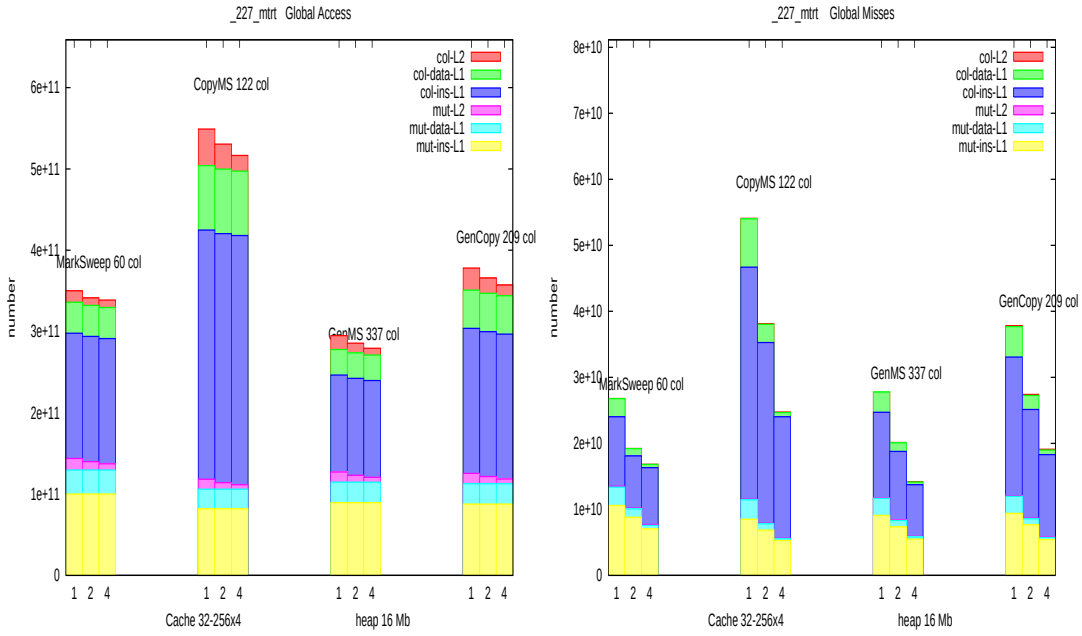
(b) Fallos



(c) Energia

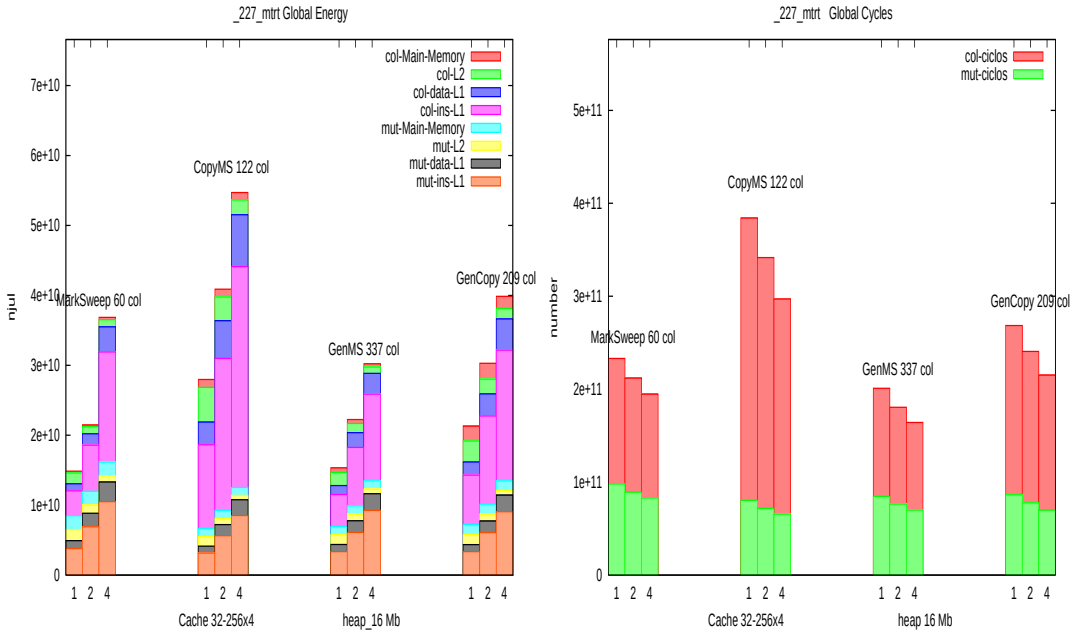
(d) Ciclos

Figura 4.12: Comparación entre las diferentes asociatividades. Tamaño del primer nivel de cache es 32K. raytrace.



(a) Accesos

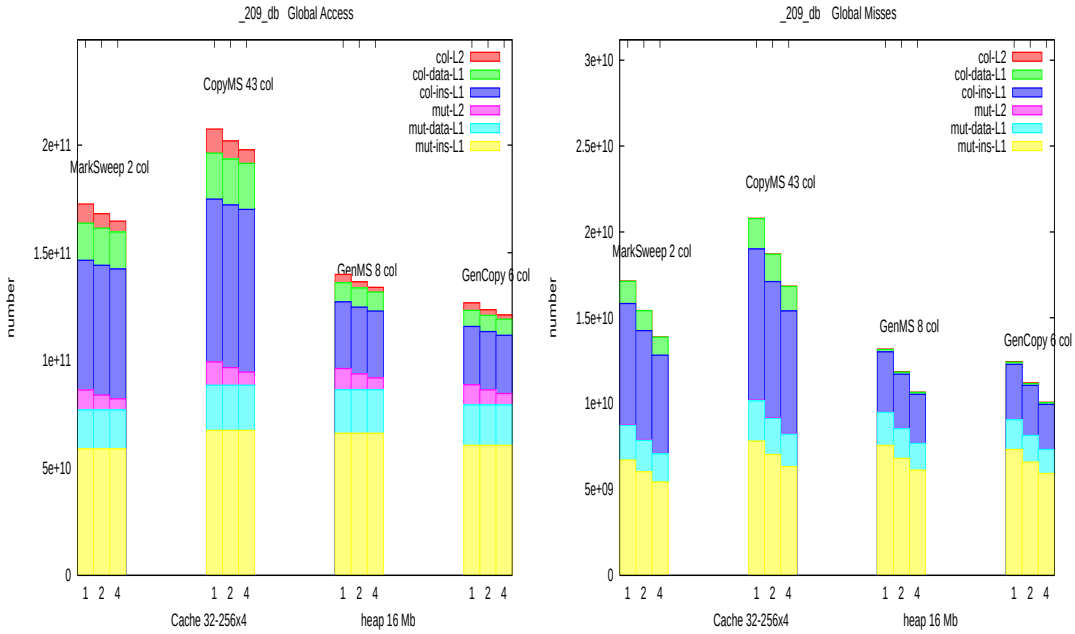
(b) Fallos



(c) Energia

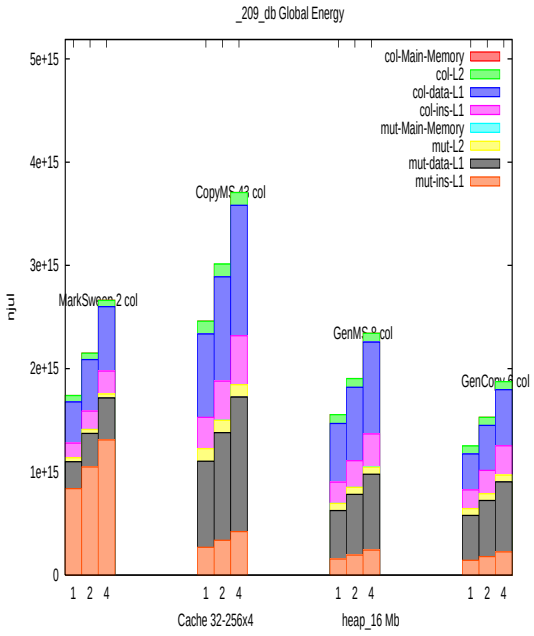
(d) Ciclos

Figura 4.13: Comparación entre las diferentes asociatividades. Tamaño del primer nivel de cache es 32K. mtrt.

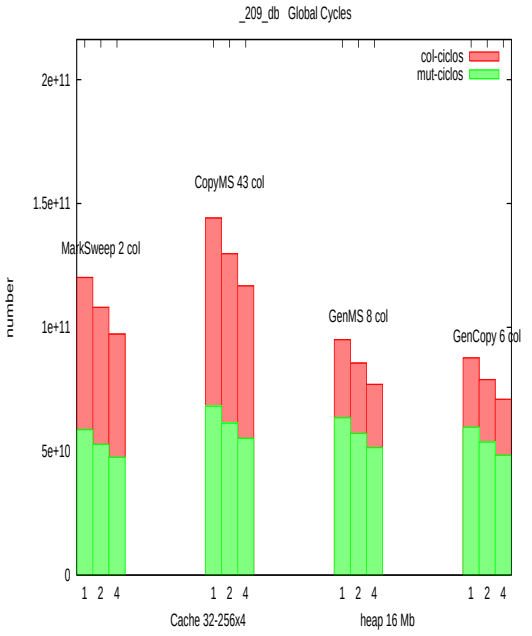


(a) Accesos

(b) Fallos

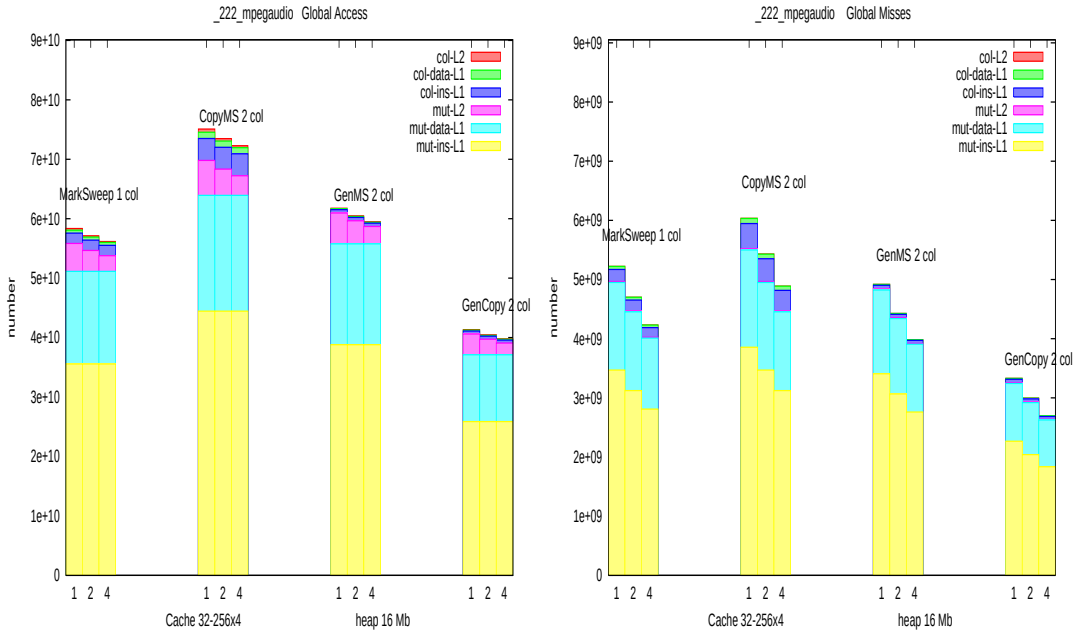


(c) Energia



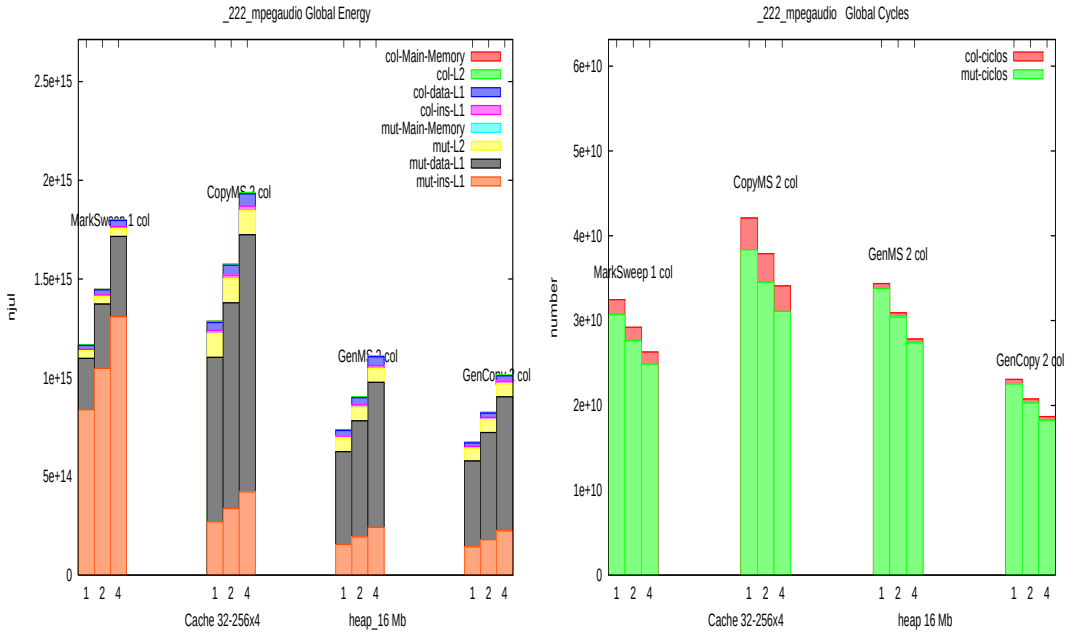
(d) Ciclos

Figura 4.14: Comparación entre las diferentes asociatividades. Tamaño del primer nivel de cache es 32K. db.



(a) Accesos

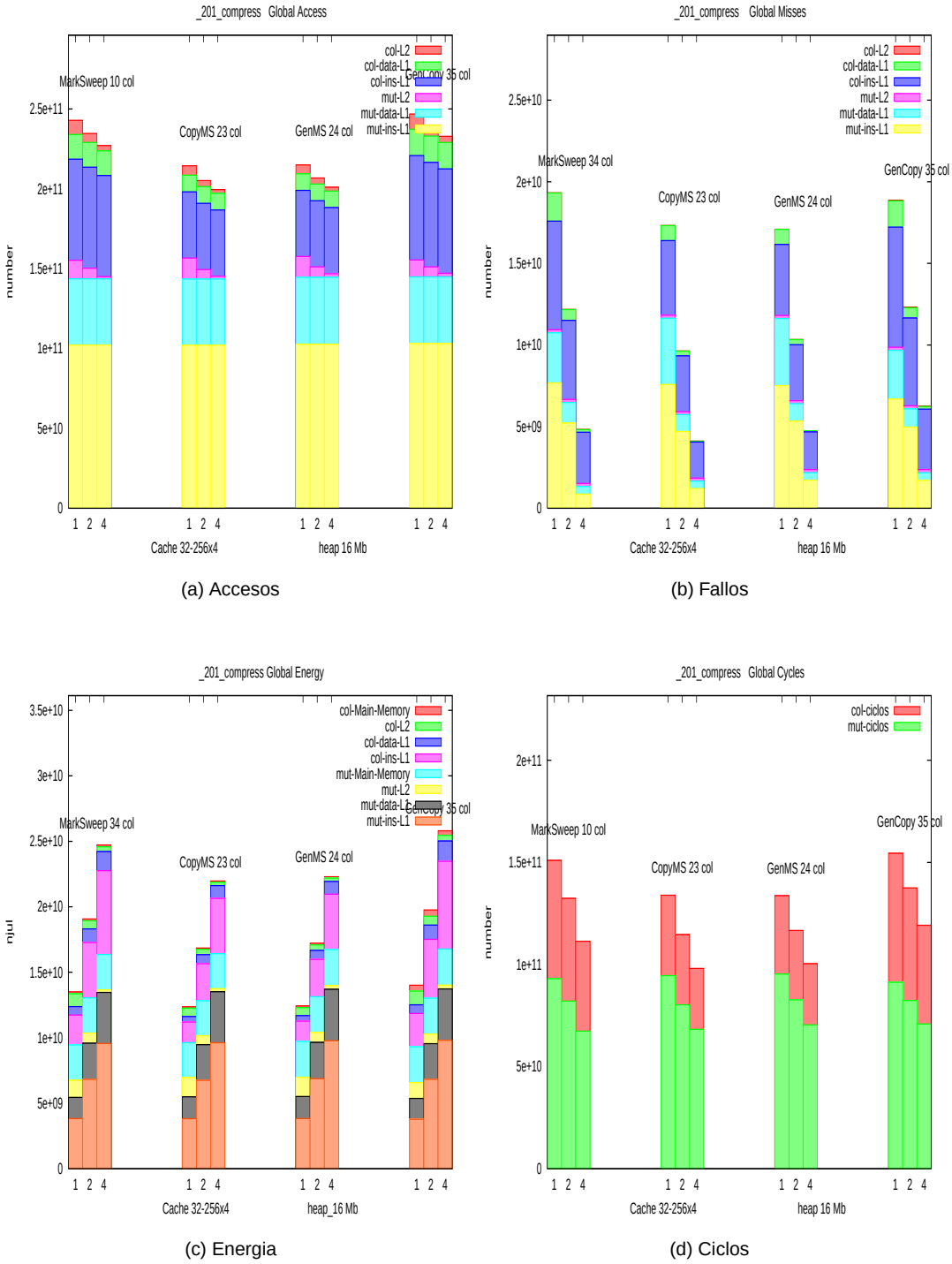
(b) Fallos



(c) Energia

(d) Ciclos

Figura 4.15: Comparación entre las diferentes asociatividades. Tamaño del primer nivel de cache es 32K. mpeg.



(a) Accesos

(b) Fallos

(c) Energia

(d) Ciclos

Figura 4.16: Comparación entre las diferentes asociatividades. Tamaño del primer nivel de cache es 32K. compress.

el consumo final del sistema empotrado.

El recolector con un menor número de accesos, y por tanto con mejores valores de ciclos y energía, es el recolector generacional híbrido (GenMS), seguido de cerca por el recolector generacional de copia puro.

En este capítulo, también, hemos presentado el espacio de diseño del consumo de energía frente al tiempo de ejecución, que nos ha proporcionado las mejores configuraciones (tamaño y asociatividad) de la jerarquía de memoria para cada uno de los recolectores estudiados, y en las que no se incluyen ninguna de las configuraciones de primer nivel de cache con tamaño de 64K.

Finalmente, hemos realizado un estudio individualizado de cada benchmark, que nos ha llevado a definir tres posibles escenarios basándonos en el comportamiento de las aplicaciones. En nuestras simulaciones hemos visto que cada escenario tiene unas peculiaridades específicas lo cual hace, con un tamaño de *heap* de 16MB, que sea un recolector distinto para cada uno de los tres casos el que obtenga los mejores resultados. No obstante, no parece aconsejable la utilización de una máquina virtual que pudiese optar por distintos recolectores según el tipo de aplicación que fuese a ejecutar. En realidad GenMS o bien es la mejor opción o bien consigue unos resultados muy próximos al mejor recolector. Por ello podemos concluir que el recolector generacional híbrido es la mejor opción ante cualquier escenario.

Recordemos que estos datos son siempre para tamaños de memoria principal muy reducidos. Para tamaños mayores, la política de copia mejora sus resultados y GenCopy iguala al recolector generacional híbrido. Por ello en los capítulos siguientes vamos a buscar estrategias para mejorar ambos recolectores.

Capítulo 5

Oportunidades de optimización

5.1 Introducción

En la información recopilada en el capítulo 4 encontramos varios aspectos del comportamiento del recolector que nos brindan distintas oportunidades de optimización. En este capítulo aprovecharemos nuestro conocimiento del funcionamiento de los distintos algoritmos de recolección y los avances en las tecnologías de fabricación de memorias para proponer dos técnicas ideadas para reducir el consumo energético y mejorar el rendimiento:

- Una técnica dirigida a reducir el consumo estático de los recolectores generacionales (sección 5.2) que utiliza el modo de bajo consumo de las actuales memorias SDRAM divididas en varios bancos.

- Una técnica enfocada a reducir el consumo dinámico (sección 5.3 y el tiempo total de recolección y que utiliza una memoria *scratchpad* [IC04] con el código de los métodos del recolector que son más accedidos durante la ejecución. En este trabajo también se presentan los resultados de ampliar esta técnica al código de otras tareas de la máquina virtual asociadas a la fase de mutador.

Dentro del trabajo previo relevante, podemos citar la propuesta inspiradora de Chen et al. [CSK+02] que también aprovecha el modo de bajo consumo de una memoria multibanco. Su propuesta está pensada para el recolector de marcado y barrido y busca reducir el consumo estático ajustando la frecuencia de recolección y ”apagando” los bancos de memoria que no contienen datos vivos. Nuestra propuesta difiere de ésta

en que está enfocada a los recolectores generacionales y no necesita de información recopilada en tiempo de ejecución sobre localización de datos muertos. En cambio nuestra técnica utiliza el comportamiento particular de la pareja asignador/ recolector de la estrategia generacional para desactivar una serie de bancos en las fases de mutador y recolector de la JVM .

Por otro lado, en el trabajo de Nguyen et al. [NDB07] encontramos una técnica para asignar objetos dentro de una memoria *scratchpad*. Su trabajo no explora los efectos en el consumo de potencia ni el efecto para diversos algoritmos de recolección. Puesto que nuestra estrategia utiliza la memoria *scratchpad* sólo durante la fase de recolección, vemos factible la utilización conjunta de las dos técnicas.

En el trabajo de Kim et al. [KTV+04] se presentan dos técnicas para la asignación de objetos ideadas para reducir el consumo de energía de la máquina virtual de Java. Una estrategia está pensada para reducir los fallos en la cache de datos. La otra utiliza un enfoque similar a la propuesta de Nguyen, aprovechando una memoria local en un estilo similar a la memoria *scratchpad*. Por tanto, las dos estrategias pueden ser usadas paralelamente a nuestras técnicas.

5.2 Reducción del consumo estático

Tradicionalmente el consumo de potencia en circuitos CMOS estaba dominado por el consumo dinámico, siendo el consumo estático (debido a las corrientes de fuga) ignorado. Sin embargo, la reducción paulatina del tamaño de los transistores conjuntamente con la disminución de la tensión de alimentación provocan que el consumo debido a las corrientes de fuga tenga cada vez un mayor peso en los sistemas empotrados. De hecho, se prevee que el consumo causado por las corrientes subumbrales sea un factor predominante en el futuro [Rab03]. Reconocidas autoridades en este campo, como Handel Jones (CEO of International Business Strategies Inc.) estiman que para la tecnología de 90nm, el peso del consumo debido al leakage, es de un 50%, pudiendo llegar a alcanzar un 80% en 65nm. Robert Hoogenstryd (director de marketing de Synopsys) piensa que la potencia disipada por las corrientes de fuga puede llegar a ser cinco veces superior a la encontrada en las tecnologías precedentes, lo cual, de no ser corregido, provocará inevitablemente el mal funcionamiento de los circuitos integrados [Ind]. También podemos citar el analizador de consumo de la empresa Altera, el PowerPlay Early Power Stimator [ALT], el cual estima un

consumo estático que oscila entre un 30% y un 45% para las FPGAs Stratix II y Stratix II GX, con memorias TriMatrix, y valores similares aunque menores para el ASIC Hardcopy II.

Nosotros vamos a aprovechar nuestro conocimiento acerca de la estrategia de recolección de basura generacional para ahorrar consumo estático durante las fases de mutador y recolector.

En la sección 5.2.1 se presentan los datos experimentales de consumo estático y se justifica la importancia de su reducción. En la sección 5.2.2 presentamos el funcionamiento de nuestra estrategia y en la sección 5.2.3 se muestran los resultados experimentales obtenidos.

5.2.1 El consumo debido a las corrientes de fuga

En la tabla 5.1 se muestra el porcentaje que representa el consumo debido a las corrientes de fuga (*leakage*) respecto del consumo total en la memoria principal para los cuatro tamaños de nivel uno de cache estudiados, distribuidos para las fases de mutador y recolector y para los distintos algoritmos de recolección. Cuando el leakage aporta un porcentaje alto en el consumo final, podemos concluir que la memoria no está siendo utilizada frecuentemente. Las primeras conclusiones que podemos sacar al estudiar estas tablas son:

- Durante la fase de mutador el porcentaje de consumo debido a las corrientes de fuga está situado en torno al 20%. Los porcentajes no varían significativamente para los distintos algoritmos, tamaños del primer nivel de cache y las distintas asociatividades, siendo el porcentaje menor el del recolector de marcado y barrido. Esto es debido, sin duda, al *lazy sweep* implementado en JikesRVM que ya vimos que era causante de un 10% más de accesos durante el mutador con respecto a los otros recolectores. El porcentaje mayor corresponde al recolector híbrido CopyMS, estando por debajo los recolectores generacionales por el coste extra que suponen las barreras de escritura.

- Los porcentajes durante la recolección varían significativamente dependiendo de la política de recolección. Los recolectores enfocados a la estrategia de copia tienen los porcentajes más bajos (en torno al 50%), causados por el mayor número de accesos necesarios para mover los objetos a través del heap. En el caso del recolector de marcado y barrido, el porcentaje se sitúa por encima del 60%. No obstante, el dato

Recolector	L1 vías	Leakage %							
		8KB		16KB		32KB		64KB	
		Mut	Rec	Mut	Rec	Mut	Rec	Mut	Rec
MS	1	18.28	64.51	18.28	64.51	18.26	65.01	18.27	62.90
	2	18.27	64.39	18.27	64.39	18.26	64.98	18.25	62.97
	4	18.25	64.30	18.25	64.30	18.22	65.27	18.28	63.85
CopyMS	1	21.28	49.41	21.28	49.41	21.12	49.40	21.12	49.32
	2	21.27	49.37	21.27	49.37	21.12	49.31	21.07	49.37
	4	21.25	49.30	21.25	49.30	21.07	49.28	21.07	49.03
GenMS	1	20.80	70.80	20.80	70.80	20.79	70.81	20.79	70.55
	2	20.79	70.70	20.79	70.70	20.77	70.76	20.73	70.55
	4	20.76	70.98	20.76	70.98	20.75	70.84	20.85	71.93
GenCopy	1	20.40	47.70	20.40	47.70	20.41	48.30	20.3	47.73
	2	20.38	47.36	20.38	47.36	20.43	46.76	20.49	47.47
	4	20.36	46.79	20.36	46.79	20.48	47.95	20.70	50.22

Tabla 5.1: Porcentaje del consumo total en memoria principal debido a las corrientes de fuga durante las fases de mutador (Mut) y recolector (Rec). El tamaño de L1 es 8K, 16K, 32K y 64K.

más relevante es que los porcentajes durante la recolección son significativamente más altos que durante el mutador. Para todos los casos el porcentaje de consumo se sitúa por encima del 47%, llegando a superar el 70% para el recolector generacional GenMS.

Está claro que la memoria principal durante el tiempo que dura la recolección está mucho más inactiva que durante la fase en la que la máquina virtual actúa como mutador, lo cual nos proporciona una oportunidad de optimización.

5.2.2 Apagado de bancos de memoria

Vamos a provechar nuestro conocimiento sobre el comportamiento de los recolectores generacionales para desarrollar una técnica que ahorre consumo estático durante las dos fases en la ejecución de una aplicación (mutador y recolector) por parte de la máquina virtual de Java.

Nuestra técnica de reducción del consumo estático está ideada para los recolectores generacionales y más específicamente para el recolector generacional de copia puro. Pensemos en el comportamiento de la máquina virtual durante una recolección menor, enfocada en la generación *nursery*. A partir del *root set* se recorre el grafo de relaciones entre objetos. Si el recolector encuentra un puntero que referencia a una dirección que no está dentro de los límites de la generación *nursery* la ignora. Al finalizar el grafo de relaciones que tiene su origen en el *root set* empezará de nuevo el proceso, pero ahora a partir del *remembered set* (cola donde se han guardado las referencias a la generación *nursery* desde las otras generaciones y regiones). Durante esta fase, el recolector necesita acceder a las direcciones de las generaciones maduras para leer las direcciones del *nursery* y posteriormente, tras la copia, actualizar las nuevas referencias. En la tabla 5.2 se muestra el porcentaje de tiempo que esta última fase representa respecto del tiempo total de recolección para los diferentes benchmarks. Podemos ver que en promedio esta fase representa sólo un 3-4% del total. De modo que, con la excepción de la fase de recorrido del *remembered set*, que es sólo el 3-4% del tiempo final de la recolección, durante una recolección menor tenemos la seguridad de que los objetos maduros, los objetos inmortales y objetos grandes no van a ser accedidos. Y puesto que conocemos perfectamente el momento en que se inicia una recolección y el momento en que comienza la fase de recorrido del *remembered set*, nuestra propuesta es aprovechar la capacidad que tienen las actuales memorias SDRAM, que están divididas en bancos, para colocar uno o varios de estos bancos en modo ”durmiente” (drowsy) y evitar por completo las corrientes de fuga. Al ”apagar” los bancos donde están situadas las regiones que no son el *nursery* podremos evitar la mayor parte del leakage de la memoria principal durante la recolección. Esta estrategia conlleva una penalización: el número de ciclos necesarios para encender de nuevo los bancos (de 500 a 1000 ciclos) cuando se va a iniciar la fase de recorrido del *remembered set*. No obstante, esta penalización no es significativa comparada con el tiempo de recolección. Pensemos que mil ciclos por unas cien recolecciones nos da una carga añadida a la recolección de cien mil ciclos, cuando los ciclos de recolección superan los diez mil millones.

Nuestra segunda idea se enfoca en la fase de mutador. Recordemos que la política de copia necesita siempre reservar espacio para copiar los objetos que van a sobrevivir a la recolección. Así pues, mientras la máquina virtual está ejecutándose como mutador, tenemos la absoluta certeza de que el espacio de reserva no va a ser accedido y por tanto

	Porcentaje %	
Benchmark	GenMS	GenCopy
compress	0.4	0.3
db	2	1.5
mpeg	1	1
jess	3	2.5
jack	5.4	5.3
javac	6.4	6.5
raytrace	5.55	5.9
mtrt	5.9	5.3
promedio	3.7	3.5

Tabla 5.2: Porcentaje de tiempo destinado al procesamiento del *remembered set* durante la recolección. Tamaño de datos s10. Heap 16 MB.

podemos dejar los bancos que dicho espacio ocupe en modo ”drowsy” sin riesgo. Esto es aplicable al espacio de reserva de la generación *nursery* y de la generación madura. En el caso de este último, además, podemos tener apagados sus bancos durante las recolecciones menores.

Al iniciarse la recolección tenemos que encender los bancos de reserva que van a ser utilizados (reserva del *nursery* en las recolecciones menores y reserva de maduros en las recolecciones globales). esto implicaría una penalización como en el caso del encendido de los bancos destinados a las generaciones maduras que hemos señalado antes. Sin embargo, en este caso estamos ante una situación distinta. Si recordamos la gráfica 3.6 mostrada en el capítulo 3), podemos observar que desde el momento en que la máquina virtual sabe que necesita ejecutar una recolección de basura hasta el momento real en que ésta empieza se suceden tres periodos:

- Un primer periodo de sincronización hasta que todos los hilos de ejecución del mutador finalizan.
- Un segundo periodo de sincronización hasta que todos los hilos del recolector se preparan.
- El periodo necesario para que el recolector recopile y organize la información del *root*

set.

Nuestra idea es que en el momento en que un hilo de ejecución haga la llamada al recolector, la máquina virtual proceda al encendido de los bancos de reserva. Mientras se producen los tres periodos mencionados los bancos tienen el tiempo necesario para recobrar su modo operativo, de modo que la recolección se inicia sin penalización.

En la figura 5.1 se muestra un ejemplo de la distribución de las distintas generaciones del heap para un recolector GenCopy y como se ubicarían en una SDRAM con 8 bancos. En la figura 5.1(a) se muestra la situación justo antes del inicio de la primera recolección de basura. Los datos inmortales ocupan algo menos de 4 MB y se sitúan en el primer y segundo banco de la memoria ya antes de que la máquina virtual comience a ejecutar la aplicación java. Recordemos que usamos un recolector generacional con tamaño de *nursery* flexible, así que toda la memoria disponible (10 MB) se asigna a la generación *nursery*. Como en la generación *nursery* tenemos una política de copia, tenemos que dividir la memoria libre en dos espacios iguales, en este caso 5 MB. El espacio de reserva ocupa los bancos tres, cuatro y parte del cinco, que se comparte con el espacio de asignación. El banco tres lo dejamos en reserva especial por si hay que asignar datos grandes, es decir para el espacio LOS. De modo que durante la ejecución del mutador hasta la llegada de la primera recolección podemos tener en modo *drowsy* los bancos cuatro y cinco. Durante la ejecución del mutador, la máquina virtual va asignando espacio a los objetos en el *nursery* a la vez que registra los punteros que llegan al *nursery* desde direcciones exteriores. El registro de estas referencias se guarda como meta-data en el espacio del *nursery*. Al producirse la recolección activamos los bancos cuatro y cinco y puesto que es una recolección *nursery*, podemos ”apagar” la memoria que no es *nursery*, ya que el recolector no va acceder a ella hasta el momento que procese los datos guardados en el meta-data. En este caso únicamente podemos apagar los bancos donde están los datos inmortales y el espacio LOS, bancos uno, dos y tres.

En la figura 5.1(b) se muestra una situación similar pero ahora a mitad de la ejecución, justo antes del inicio de una recolección cualquiera. En este caso tenemos:

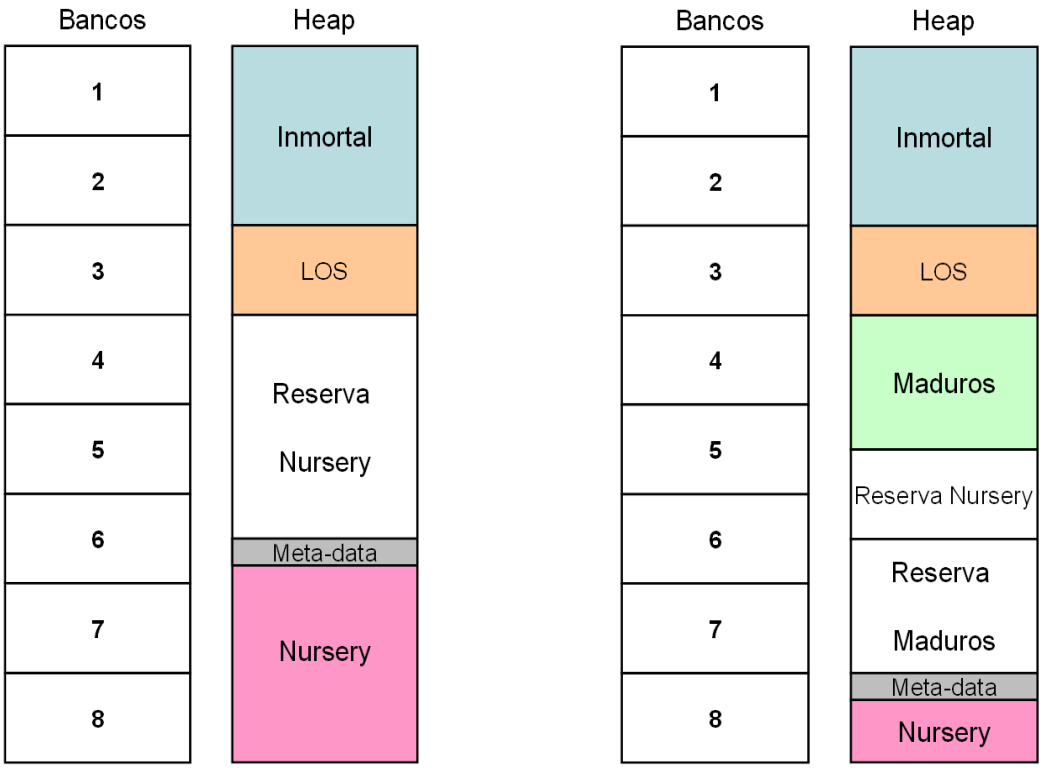
- El primer y segundo banco ocupado por los datos inmortales.
- El tercer banco parcialmente ocupado por el espacio reservado para objetos grandes.
- Los objetos maduros, que ya han sobrevivido la recolección en la generación *nursery*, ocupan el cuarto banco completamente y parcialmente el quinto banco.

- La parte libre del quinto banco, junto con los bancos sexto y séptimo están reservados para copiar los objetos que sobrevivan a las recolecciones menores y mayores. Puesto que los objetos que sobreviven a la recolección *nursery* se copian contiguamente a los objetos maduros, hemos indicado que el banco quinto y sexto parcialmente son la reserva *nursery*, mientras que la otra parte parte del sexto y el séptimo banco forman la reserva para los objetos maduros.

- Por último, en el octavo banco, están los datos recientemente creados, la generación *nursery*, junto con el resultado de la acción de las barreras de escritura (meta-data).

Podemos comprobar que en este caso, durante la ejecución del mutador podemos tener en modo ”drowsy” dos bancos (los número seis y siete). Al iniciarse la recolección menor tenemos que encender el sexto banco (para copiar los objetos supervivientes), pero podemos mantener apagado el banco siete. Además, al ser una recolección menor sabemos que no vamos a acceder a ninguna generación que no sea la *nursery*, y podemos apagar los cuatro primeros bancos. De modo que tenemos en modo ”drowsy” cinco bancos en total. Cuando se ha recorrido todo el grafo de relaciones entre objetos y ya sólo queda procesar el ”*remembered set*” tenemos que encender los cuatro primeros bancos, correspondientes a los espacios para objetos inmortales, grandes y maduros, para que las actualizaciones de las referencias se puedan llevar a cabo. Durante el procesamiento de estos datos, que experimentalente se corresponde con un porcentaje 3-4% de tiempo de recolección, podemos mantener apagado el séptimo banco, destinado a la reserva de maduros.

Esta propuesta no es aplicable directamente al recolector GenMS. En el caso de GenMS, el espacio de reserva, necesario para copiar los objetos supervivientes a la recolección menor, no es un espacio claramente delimitado (a partir de la primera recolección global) como en el caso de GenCopy. Puesto que la generación madura utiliza la política de marcado y barrido, el espacio de reserva está compuesto de un conjunto de listas de distintos tamaños en los que objetos maduros y bloques libres están intercalados. Como trabajo futuro, y posible mejora para GenMS, se podría utilizar una política de compactación en la generación madura [Jon00]. De este modo la estrategia de apagado de bancos se podría implementar completamente en este recolector. En los resultados experimentales actuales, hemos aplicado nuestra estrategia hasta la primera recolección global.



(a) Recolección primera

(b) Recolección i-ésima

Figura 5.1: Ejemplo típico de la distribución de las generaciones de GenCopy en una SDRAM con 8 bancos.

5.2.3 Resultados experimentales

En nuestro primer conjunto de simulaciones hemos utilizado una memoria principal SDRAM de 16 MB y 8 bancos [tIb]. La configuración de los dos niveles de memoria cache se corresponde con la mejor opción encontrada desde el punto de vista del consumo en el capítulo 4, pero los resultados son mejores para los otros casos. En la tabla 5.3 se muestran los resultados experimentales tras ejecutar todos los benchmarks y calcular el promedio para cada recolector generacional. En primer lugar se muestra la distribución del consumo de energía de la máquina virtual en memoria principal entre las fases de mutador y recolector. En segundo lugar, la tabla muestra el tanto por ciento del consumo debido a las corrientes subumbrales distinguiendo nuevamente entre las fases de mutador y recolector y la quinta columna nos indica el tanto por ciento del consumo total en memoria principal que es debido al leakage. En la tabla 5.5 se muestra el porcentaje de reducción en el consumo por corrientes de fuga al aplicar la técnica propuesta en el mutador y en el recolector para los distintos recolectores y benchmarks. Finalmente se muestra el porcentaje de reducción acumulado sobre el leakage total y sobre el total de consumo en memoria principal. En la tablas 5.4 y 5.6 se muestran los resultados para una memoria SDRAM de 32 MB y 16 bancos.

A la vista de estos resultados podemos sacar las siguientes conclusiones:

- EL consumo en memoria principal en los recolectores generacionales se debe en su mayoría a la fase de mutador. Esto es debido al limitado número de recolecciones de estos recolectores. Esta tendencía se acentúa al aumentar el tamaño de la memoria principal. A mayor tamaño de la memoria principal, menor número de recolecciones de basura necesarias y, en general, menor peso de la fase de recolector frente a la de mutador.

- La fase de recolección, para el recolector generacional híbrido consume en torno al 5% (6.7% para 16MB y 4.5% para 32MB) del total de consumo en memoria. Sin embargo, para el recolector generacional de copia, el porcentaje se sitúa en torno al 20% (22.55% para 16MB y 17.3% para 32MB). El mayor peso que tiene la fase de recolección en el consumo para GenCopy se debe a que este recolector, además de tener un 10% más de recolecciones menores (en promedio), tiene un mayor número de recolecciones globales (en promedio 8.125) que GenMS (en promedio 3.375). Recordemos también que durante la recolección global, GenCopy utiliza la política de

copia mientras que GenMS realiza sólo el recorrido del grafo de relaciones marcando los objetos vivos.

- El porcentaje de consumo debido a las corrientes de fuga es, como ya se ha mencionado, mucho mayor para la fase de recolector que para la de mutador. Los distintos recolectores muestran porcentajes de leakage muy dispares debido a los dos factores mencionados en el punto anterior. Con un Heap de 16MB, GenMS tiene un 70% del consumo de recolección debido al leakage, mientras que GenCopy no llega al 50%. Con un heap de 32 MB, el número de recolecciones globales baja para ambos recolectores pero más significativamente para GenCopy. Este hecho se traduce en una memoria principal más inactiva durante la recolección y por tanto el porcentaje de pérdidas por corrientes de fuga aumenta. Podemos comprobar además que los porcentajes de los dos recolectores se acercan.

- Podemos apreciar que el peso del consumo debido a las corrientes de fuga en la fase de mutador aumenta diez puntos porcentuales al pasar de tamaño de Heap de 16MB (20%) a tamaño de 32MB (30%). Esto es debido a que el aumento en el tamaño de memoria principal no produce una reducción equivalente en el tiempo de ejecución.

- Dado que el porcentaje de consumo debido a las corrientes de fuga (tanto para la fase de mutador como para la fase de recolector) aumenta al aumentar el tamaño de memoria principal, el efecto acumulado es que pasamos de un peso del leakage del 25% a el 35% del consumo total. Esto significa que reducir la influerncia de las corrientes de fuga será cada vez más importante a medida que memorias de tamaño mayor se incorporen dentro de los dispositivos empotrados.

- La aplicación de la estrategia durante las recolecciones *nursery* (apagando de los bancos donde se sitúan el resto de las generaciones y regiones) obtiene un porcentaje de reducción en torno al 75% (69% para 16MB y 80% para 32MB) de las corrientes de fuga para GenCopy. Puesto que la técnica no se puede aplicar integramente en el recolector GenMS, los resultados obtenidos son significativamente menores (22% para 16MB y 28% para 32MB).

	Distribución %		Leakage %		
	mutador	recolector	mutador	recolector	Acumulado
GenMS	93.3	6.7	20.7	70.1	24
GenCopy	77.4	22.55	20.4	49.5	26.9

Tabla 5.3: Consumo en memoria principal. Distribución entre las fases de mutador y recolector y porcentaje del consumo debido a las corrientes de fuga. SDRAM de 16MB y 8 bancos.

	Distribución %		Leakage %		
	mutador	recolector	mutador	recolector	Acumulado
GenMS	95.4	4.55	33.8	76.1	35.7
GenCopy	82.7	17.3	31.3	60.5	36.55

Tabla 5.4: Consumo en memoria principal. Distribución entre las fases de mutador y recolector y porcentaje del consumo debido a las corrientes de fuga. Memoria SDRAM de 32MB y 16 bancos.

	Reducción Leakage %			Reducción %
	mutador	recolección	Acumulada	Memoria principal
GenMS	8	22.5	10.8	2.55
GenCopy	23	69.2	42.1	11.3

Tabla 5.5: Porcentaje de reducción en el consumo por corrientes de fuga tras aplicar nuestra propuesta. SDRAM de 16MB y 8 bancos.

	Reducción Leakage %			Reducción %
	mutador	recolección	Acumulada	Memoria principal
GenMS	10.5	28.2	12.2	4.3
GenCopy	38.4	80.9	52.3	19

Tabla 5.6: Porcentaje de reducción en el consumo por corrientes de fuga tras aplicar nuestra propuesta. Memoria SDRAM de 32MB y 16 bancos.

- Al aplicar la estrategia propuesta en la fase del mutador, es decir, al apagar los bancos destinados como espacio de reserva para copiar los objetos supervivientes a la recolección, se consiguen reducciones menores en porcentaje (23% para 16MB y 38% para 32MB). Pero pensemos que el peso de la fase de mutador es mayor que el de la fase de recolección (sobretodo para GenMS), de modo, que la reducción global obtenida durante esta fase es finalmente más significativa. Además, esta situación se agudiza al aumentar el tamaño del heap. De modo que, previsiblemente, la reducción en el gasto por corrientes de pérdida al utilizar esta técnica durante el mutador mejorará cuanto mayor sea el tamaño de la memoria principal, al contrario que la fase de recolector que pierde relevancia con el aumento del tamaño de la memoria.

- Finalmente se puede observar que esta propuesta consigue, para el caso de GenCopy, una reducción de más del 10% del consumo total en memoria principal (16MB) sin pérdida de rendimiento. Esta reducción es aún más significativa al aumentar el tamaño de memoria principal, ya que con 32MB, la reducción se sitúa en torno al 20%, lo que nos indica que esta técnica producirá mejores resultados en el futuro. Para GenMS, la reducción es mucho menor, no llegando al 3%. En este sentido, estimamos que sería muy interesante la utilización un algoritmo compactador para la generación madura, esto nos permitiría utilizar la estrategia completa con el recolector generacional híbrido. Por ello, este punto queda como trabajo futuro.

Este último punto se puede apreciar mejor en las gráficas 5.2 y 5.3 en las que se muestran el porcentaje de reducción total de las corrientes de fuga y la contribución de las fases de mutador y recolector para los dos tamaños de memoria principal y el recolector generacional de copia puro. En color azul se representa la fase de mutador y en rojo la de recolector. Podemos apreciar como la contribución de la fase de mutador aumenta en todos los benchmarks al pasar de 16MB a 32MB, mientras que la contribución de la fase de recolector disminuye.

Podemos ver que en el caso de la aplicación compress, cuando la memoria principal sólo tiene 16MB, casi todo el heap se llena con objetos grandes (LOS) dejando muy poco espacio para la generación *nursery*. De este modo, el espacio de reserva de esta generación nunca llega a ocupar un banco en exclusiva y por tanto nuestra técnica no se puede aplicar durante la fase de mutador. Este problema desaparece cuando el tamaño de

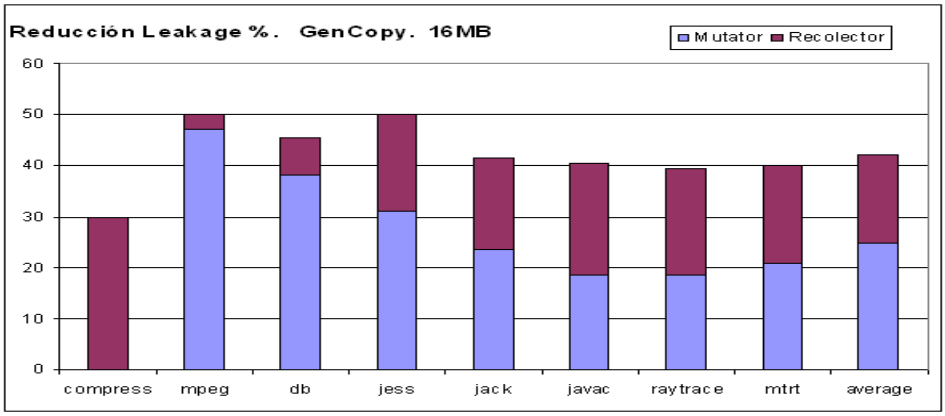


Figura 5.2: Reducción de las corrientes de fuga en una memoria principal de 16MB y 8 bancos.

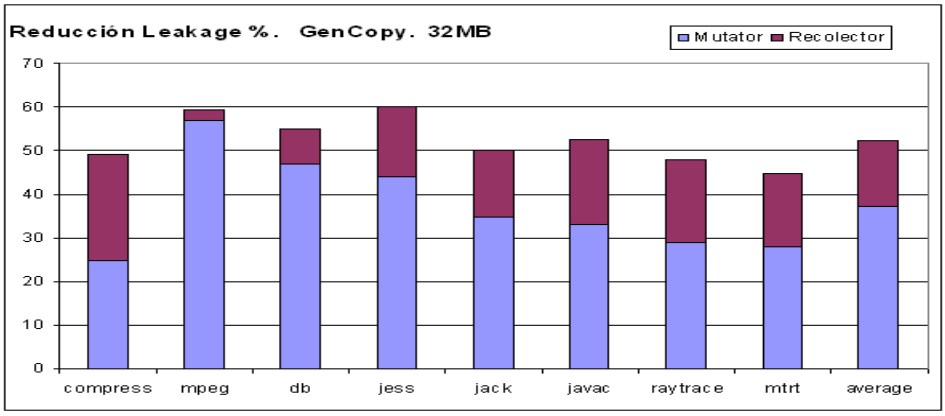


Figura 5.3: Reducción de las corrientes de fuga en una memoria principal de 32MB y 16 bancos.

la memoria principal es 32MB. En el caso de los demás benchmarks, cuanto mayor es el número de objetos asignados y mayores son sus tiempos de vida, mayor peso tiene la fase recolección. Por ello, podemos ver que las aplicaciones mpeg y db por su bajo número de objetos asignados, y en menor medida jess (por el corto tiempo de vida de sus objetos), son las aplicaciones con mayor peso porcentual de reducción en la fase de mutador.

5.3 Utilización de una memoria *Scratchpad*

5.3.1 Memorias Scratchpad

La memoria scratchpad es una memoria SRAM de alta velocidad usada normalmente conjuntamente con las memorias cache de primer nivel. La información almacenada en la memoria scratchpad se corresponde con un rango específico de direcciones de memoria principal (figura 5.4) que no puede ser accedida por las memorias cache, por lo cual, en estas memorias no podemos hablar de aciertos (*hits*) o fallos (*misses*), ya que el acceso dentro del rango de direcciones correspondiente siempre resulta en un acierto. De este modo, y puesto que no es necesario incluir el hardware asociado a la comprobación de direcciones, reescrituras, llamadas a la cache de segundo nivel, etc, la memoria scratchpad es más sencilla, ocupa menos área y consume menos que una memoria cache convencional. Su principal utilidad es el almacenamiento de direcciones de memoria de uso muy frecuente. La elección de estas direcciones se lleva a cabo mediante software.

Actualmente el ámbito de aplicación de las memorias scratchpads está restringido a los sistemas empotrados, los procesadores de propósito específico y las consolas de video juegos. Como ejemplos podemos poner:

La familia de procesadores imx31 de Freescale [iAP], enfocada a toda clase de sistemas empotrados, en especial los de aplicación en la industria médica y del automóvil, usa una memoria scratchpad junto con la jerarquía de memorias cache. La CPU de la Playstation 2 (Emotion Engine [Sot00]) usa una memoria scratchpad de 16KB conjuntamente con la memoria cache. En la R3000, CPU de la consola predecesora (la Playstation 1), se había sustituido el primer nivel de cache por una memoria scratchpad donde se guardaban los datos de la pila del procesador. El microprocesador Cell [IBMd], resultado de la alianza ”STI” (entre Sony Computer Entertainment, Toshiba, e IBM), así como el procesador para cálculos físicos, el PhysX de AEGIA (NVIDIA [Hom]) utilizan una memoria scratchpad dividida en bancos. La GPU 8800 (NVDIA), de la serie GeForce 8 [Was], dentro de la arquitectura CUDA [NVI], utiliza una scratchpad de 16KB por thread para el procesamiento gráfico.

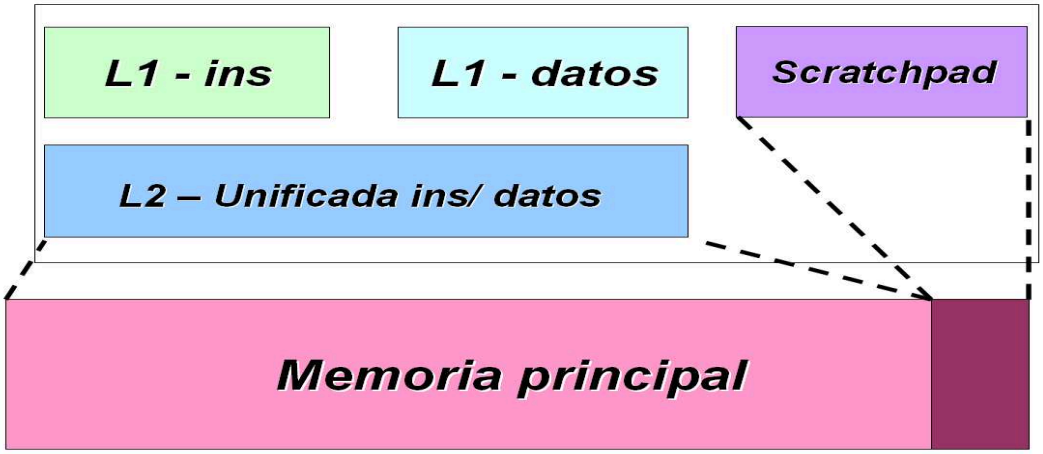


Figura 5.4: Rango de direcciones asignado a la memoria scratchpad.

5.3.2 Motivación

En el capítulo 2 se mostró el comportamiento de la máquina virtual de java y así vimos las distintas tareas que ha de afrontar durante la ejecución de una aplicación. Algunas de estas tareas están directamente influenciadas por el comportamiento de la aplicación y se realizan de forma breve y aleatoriamente intercaladas con la ejecución de ésta. Sin embargo, en el caso de la recolección de basura nos encontramos con la siguiente situación:

- Antes de iniciarse la recolección, la jerarquía de memoria está llena de las instrucciones y datos debidos a la ejecución del programa java.
- La máquina virtual sabe perfectamente cuando empieza y acaba la recolección.
- Todos los hilos de la máquina virtual se dedican a la recolección durante periodos de tiempo del orden de centenas de miles de ciclos.
- Durante la recolección, los métodos ejecutados dependen de la política seguida de asignación y reciclaje. El comportamiento de la aplicación sólo infuirá a la hora del reciclaje en las estrategias de copia. Dependiendo del número de objetos copiados la máquina tendrá que ejecutar más o menos los métodos encargados de ello. Los métodos encargados de recorrer el grafo de referencias van a ser una constante para todas las aplicaciones.
- A medida que la máquina virtual utiliza estos métodos y sus datos la jerarquía de

memoria tendrá que desechar los datos que tenía guardados en los primeros niveles y por tanto se perderá por completo el estado en que se encontraban las memorias cache. Problema que se repite al terminar la recolección, ya que hay que recuperar el estado previo y desechar por completo las instrucciones y datos utilizados durante la búsqueda y reciclaje de basura.

A partir de los puntos anteriores parece lógico deducir que la recolección de basura perturba el comportamiento normal de la jerarquía de memoria, pero al mismo tiempo su comportamiento es predecible y poco dependiente de la aplicación concreta que en cada momento esté ejecutando la máquina virtual. Por todo ello es un candidato ideal para aplicar alguna técnica de *prefetching* de datos y/ o instrucciones. Nosotros hemos utilizado una memoria scratchpad controlada por la máquina virtual sobre la que hemos probado diferentes ideas en la búsqueda del contenido idóneo:

- Cada vez que la máquina virtual tiene que acceder a los diferentes campos de un objeto primero ha de conocer como se estructura la clase a la que pertenece ese objeto (ver subsección 3.3.3). En Jikes esa información se guarda de forma global en una estructura llamada JTOC (*Jikes table of Contents*). Puesto que la máquina virtual ha de recurrir constantemente al JTOC, nuestra siguiente idea fue guardar esta estructura de forma permanente en una memoria *scratchpad* al iniciarse la ejecución de la aplicación. Con esta idea buscabamos liberar a la cache de datos de estos accesos inevitables para evitar interferencias con otros datos de comportamiento más impredecible. Los resultados experimentales dieron una reducción en el número de accesos en la cache de datos de un 3-5%, junto con una reducción en el número de fallos de un 6-8%. Sin embargo, los accesos a la *scratchpad* no eran suficientes para justificar su inclusión desde el punto de vista del consumo final. El gasto inevitable por leakage en la *scratchpad*, tanto en la fase de mutador como en la de recolector, contrarresta la mejora conseguida en la cache de datos.

- En proyectos previos que buscaron crear una implementación física de la máquina virtual de Java, se probó la utilización de memorias cache sin etiquetas (proyecto *Mushroom* de Wolczko et al. [Proc], Universidad de Manchester) para guardar la generación *nursery*. En nuestros experimentos en esa dirección utilizando una memoria *scratchpad* sólo encontramos desventajas. Si la generación *nursery* es de tamaño fijo y pequeño (implementada en una *scratchpad*), aumenta en 6-8 veces

el número de recolecciones y lo que es peor, el número de datos copiados al sobrevivir las recolecciones menores, resultando en un decremento grande tanto en rendimiento como en el consumo final.

- Finalmente, experimentamos con la idea de incluir el código más accedido durante la recolección en la memoria scratchpad para así liberar a la cache de instrucciones de una carga de trabajo altamente previsible, dejando espacio para las instrucciones de comportamiento más aleatorio.

5.4 Memoria *scratchpad* con el código más accedido

En las gráficas de accesos y fallos (capítulo 4) pudimos ver que durante la recolección la cache de instrucciones tiene más relevancia que la cache de datos. Las instrucciones producen un 10-15% más de accesos que los datos. Esta información junto con la experiencia obtenida tras las ideas anteriores nos llevó a la siguiente propuesta: utilizar una memoria *scratchpad* controlada por la máquina virtual de Java para guardar los métodos más utilizados durante cada una de las fases que atraviesa la ejecución de una aplicación. La máquina virtual al iniciarse cada una de las fases realizaría la carga de una imagen específica en la *scratchpad*. De este modo se liberaría a la cache de primer nivel de las instrucciones que se repiten de forma cíclica y predecible.

En las siguientes subsecciones estudiamos los beneficios de tener una memoria *scratchpad* incluída en la jerarquía de memoria y controlada por la máquina virtual de Java. En la subsección 5.4.1 se muestran los resultados experimentales al utilizar una memoria *scratchpad* para guardar los métodos más utilizados durante la recolección, en la subsección 5.4.2 se amplia esta estrategia a la fase de mutador y en la subsección 5.4.3 se presenta una estrategia adaptativa para que el recolector escoja dinámicamente entre tres posibles conjuntos de métodos para cargar en la *scratchpad*.

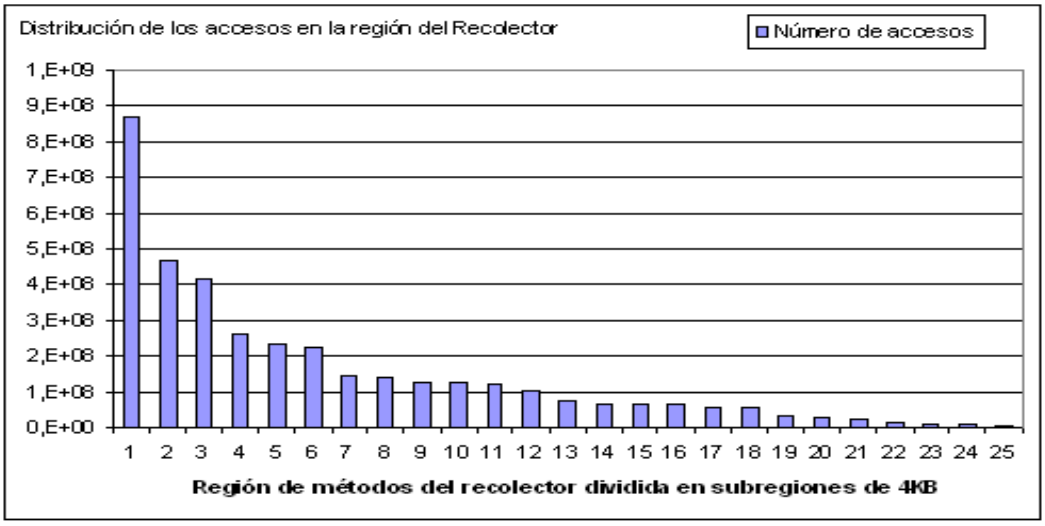


Figura 5.5: Distribución de los accesos al dividir la región del Recolector GenMS en subregiones de 4KB .

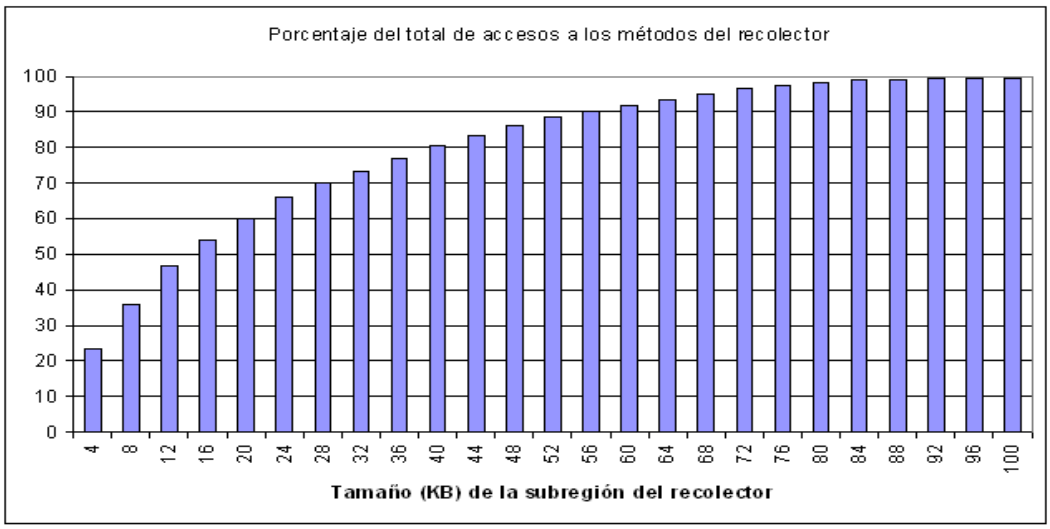
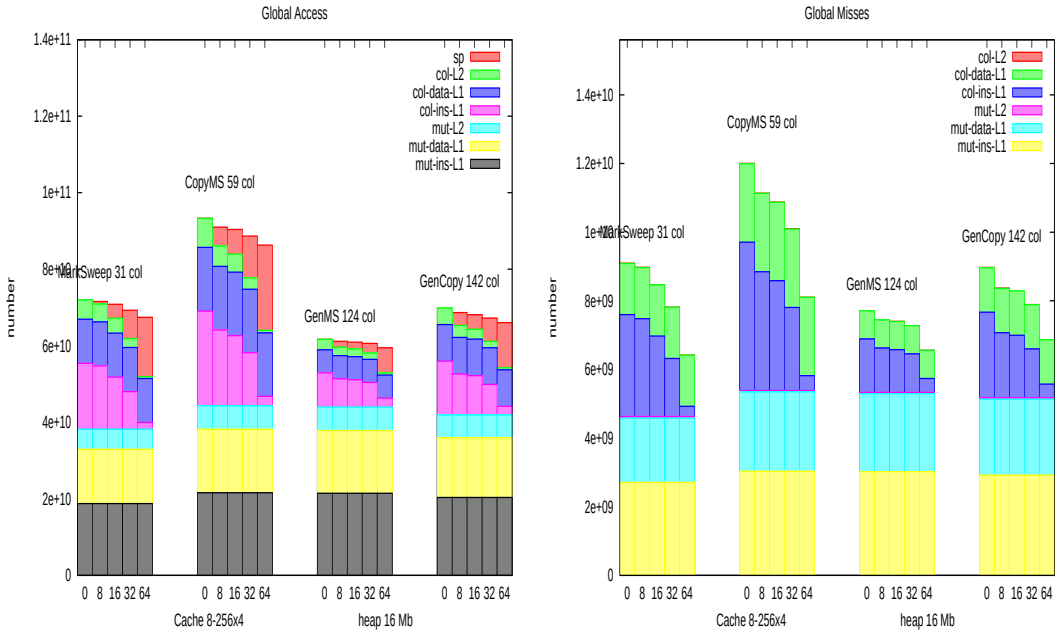


Figura 5.6: Porcentaje del total de accesos a las métodos del recolector GenMS.

5.4.1 Memoria *scratchpad* durante la fase de recolector

En esta subsección estudiamos los beneficios de tener una memoria *scratchpad* controlada por la máquina virtual de Java incluida en la jerarquía de memoria y donde se guardan los métodos más utilizados durante la fase de recolección. Los métodos fueron escogidos tras realizar un exhaustivo profiling del comportamiento del recolector y promediando los accesos a cada método para la totalidad de los benchmarks estudiados. Durante la fase del mutador la memoria *scratchpad* está en modo ”drowsy”. La máquina virtual es la encargada de activar y desactivar la *scratchpad* al inicio y al final de la recolección. Las características que hacen interesante la fase de recolección para implementar esta estrategia son:

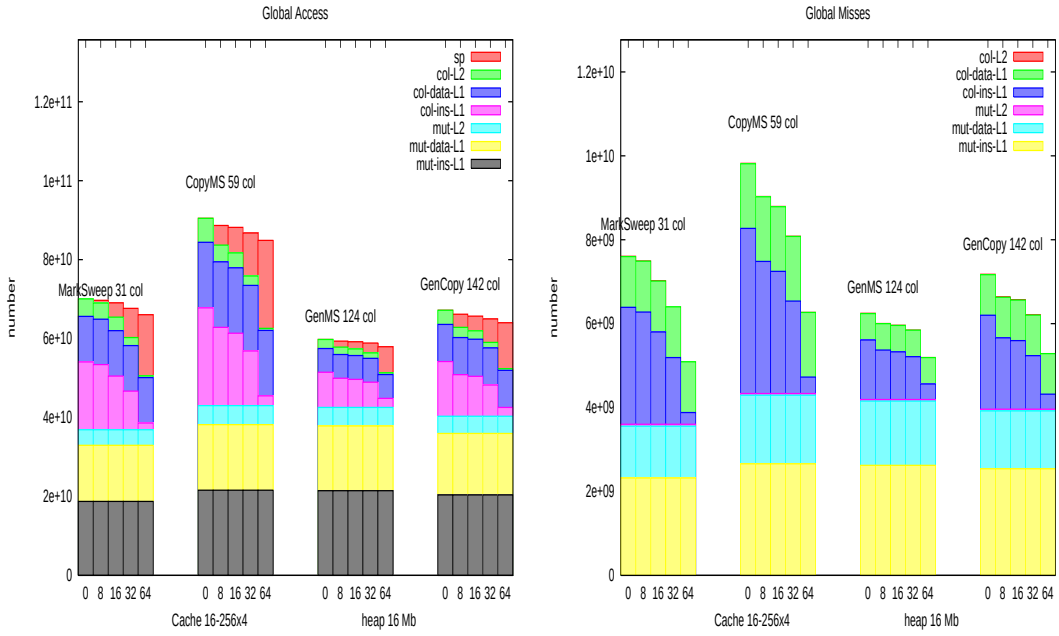
- La máquina virtual conoce perfectamente el momento de inicio y finalización de la recolección.
- El tamaño de la región donde están compilados los métodos del recolector es superior a los 300KB. Sin embargo, no todos los métodos reciben el mismo número de accesos, de hecho se cumple la ley que dice: ”un 20% del código es responsable del 80% de las instrucciones”. En la figura 5.5, se muestra: en el eje ordenadas la región de los métodos del recolector GenMS dividida en subregiones de 4KB. Recordemos que el recolector generacional híbrido es el más complejo de los cinco recolectores, ya que utiliza los métodos propios de las estrategias de copia, marcado y barrido y generacional. En el eje de abcisas el número de accesos a cada subregión. Sólo se muestran las 25 subregiones más accedidas, ya que el resto prácticamente no registra accesos. De este modo, y como se resume en la figura 5.6, podemos ver que a 8KB de la región del recolector le corresponde el 35.8% de los accesos, a 16KB el 53.8%, a 32KB el 73.55% y a 64KB un poco más del 93% de los accesos.



(a) Accesos (b) Fallos

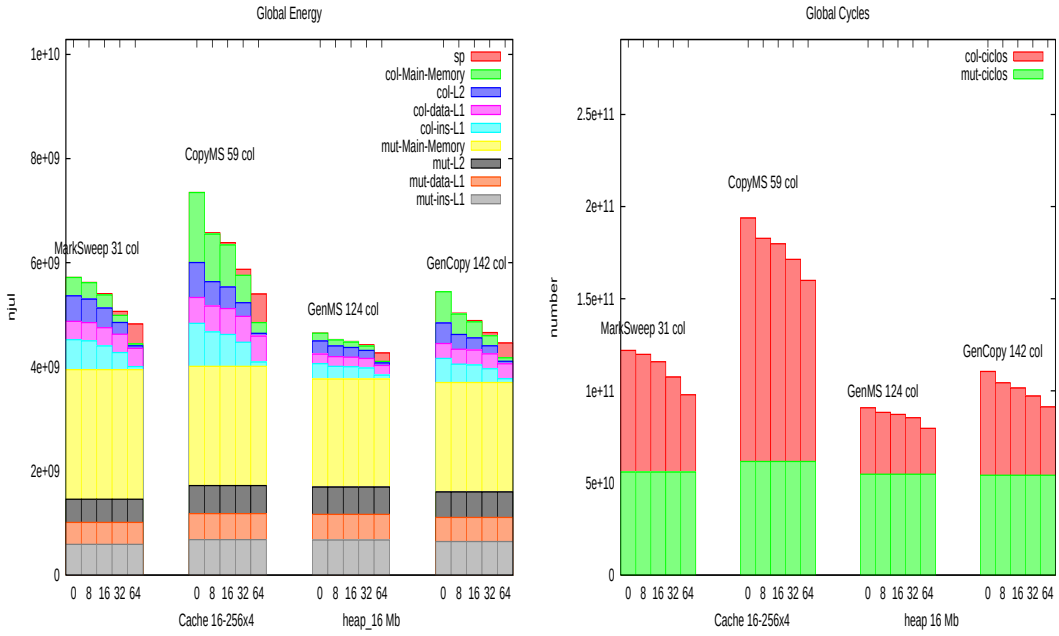
(c) Energia (d) Ciclos

Figura 5.7: Comparación entre los diferentes tamaños de scracthpad. Tamaño del primer nivel de cache es 8K. Promedio de todos los benchmarks.



(a) Accesos

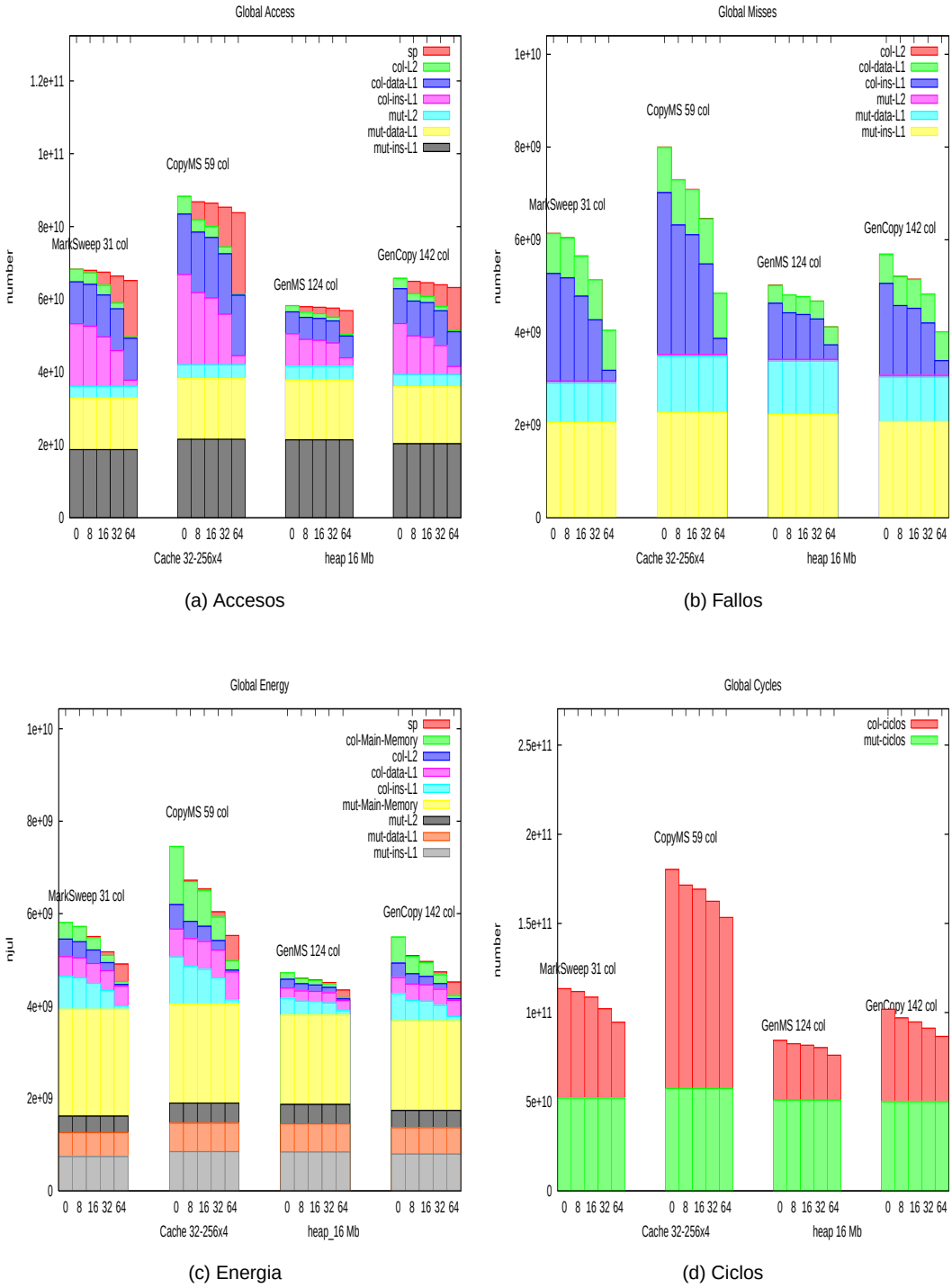
(b) Fallos



(c) Energia

(d) Ciclos

Figura 5.8: Comparación entre los diferentes tamaños de scracthpad. Tamaño del primer nivel de cache es 16K. Promedio de todos los benchmarks.



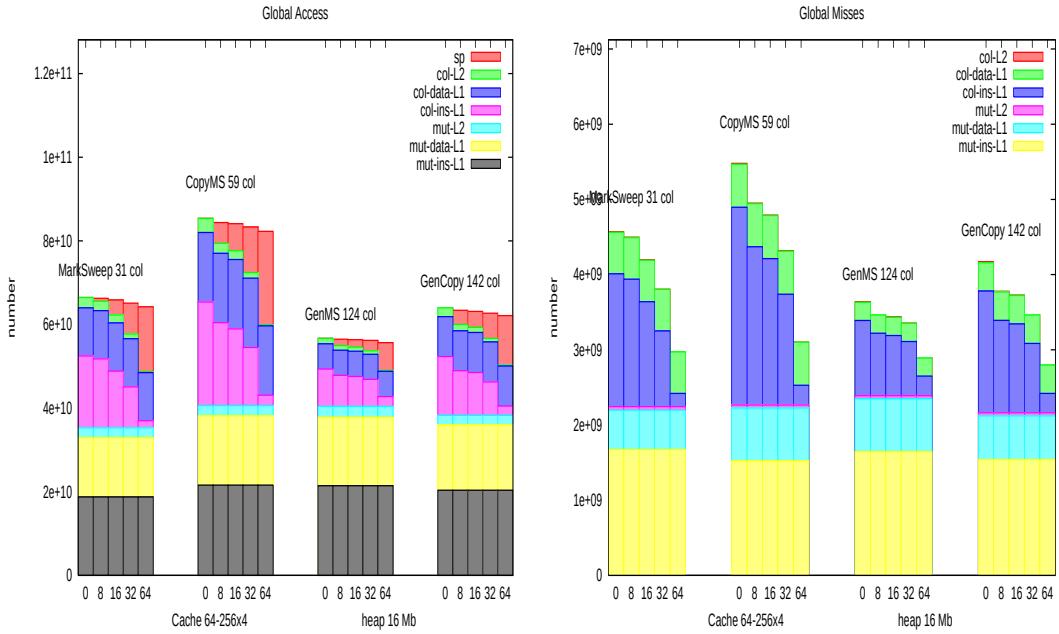
(a) Accesos

(b) Fallos

(c) Energia

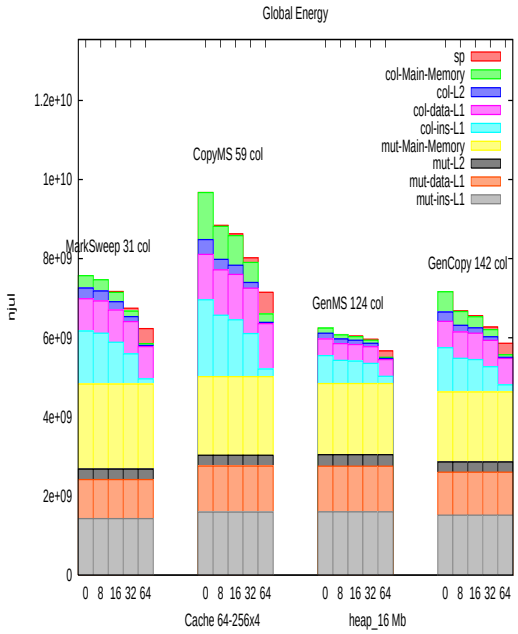
(d) Ciclos

Figura 5.9: Comparación entre los diferentes tamaños de scracthpad. Tamaño del primer nivel de cache es 32K. Promedio de todos los benchmarks.

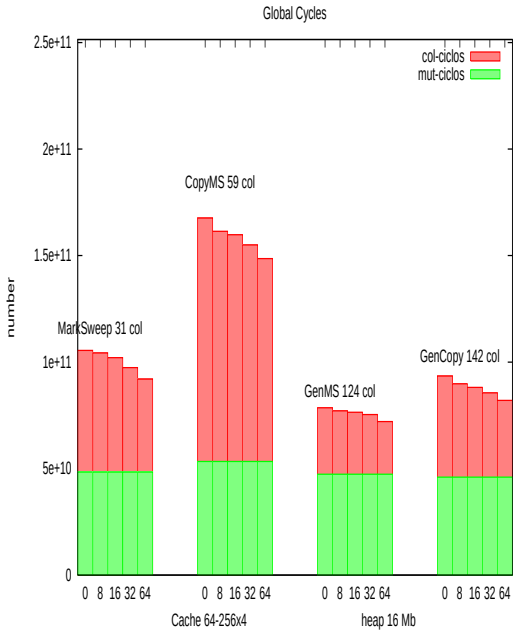


(a) Accesos

(b) Fallos



(c) Energia



(d) Ciclos

Figura 5.10: Comparación entre los diferentes tamaños de scracthpad. Tamaño del primer nivel de cache es 64K. Promedio de todos los benchmarks.

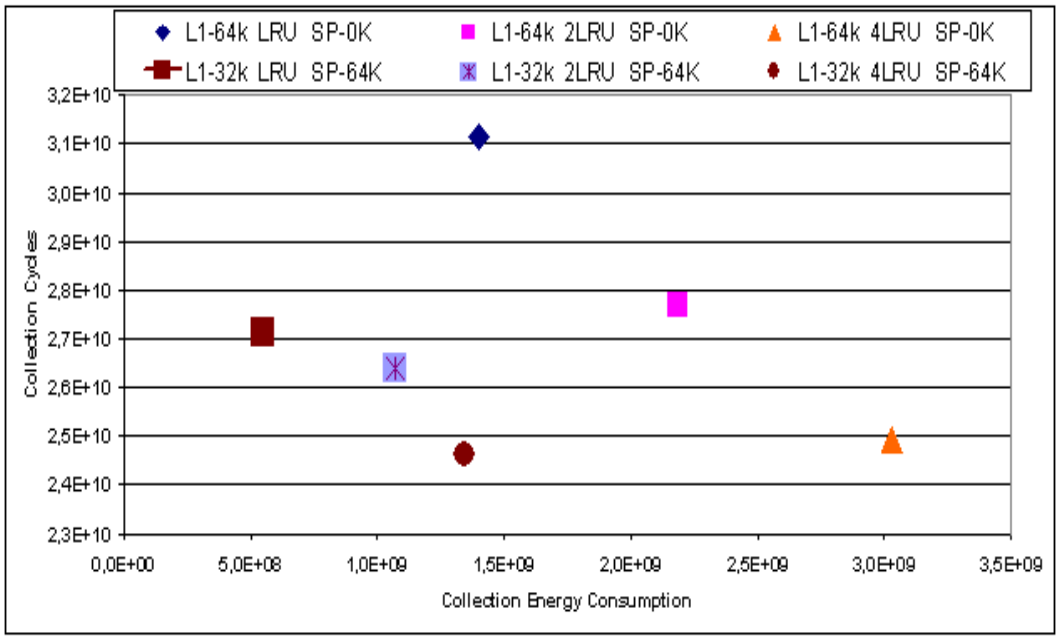


Figura 5.11: Consumo de energía (nanjulios) del recolector frente a número de ciclos del recolector para las configuraciones con y sin *scratchpad* en el primer nivel de cache

- El comportamiento del recolector es bastante independiente del comportamiento de la aplicación java. De modo que la selección de métodos escogida es una solución universal. La unica excepción a la afirmación anterior entre los benchmarks estudiados es compress. Las diferencias entre aplicaciones se centran en el número de objetos copiados y en la cantidad de objetos grandes que se asignan. Por ello podemos entender que el comportamiento de compress se separa mucho del comportamento del resto de aplicaciones y se ve beneficiado de una selección de métodos distinta a la del resto.

En la figura 5.7 se muestra el resultado de incluir una *scratchpad* con el código más accedido de los métodos de los diferentes recolectores en la jerarquía de memoria con un tamaño de primer nivel de cache de 8K y asociatividad directa. Los tamaños estudiados de la *scratchpad* son 0K, 8K, 16K, 32K y 64K. Los resultados están, como en el capítulo 4, distribuidos en los diferentes niveles de la jerarquía de memoria y diferenciando las fases de mutador y recolector. El resto de instrucciones del recolector que no tienen cabida en la scratchpad, así como otras instrucciones del classpath de Java utilizadas durante la recolección son ubicadas en la memoria cache de instrucciones. En las figuras 5.8, 5.9

y 5.10 se muestran las mismas gráficas cuando el tamaño del primer nivel de cache es 16K, 32K y 64K respectivamente.

Al estudiar estas gráficas sacamos dos conclusiones:

- Esta técnica consigue reducir el número de accesos al primer nivel de la cache de instrucciones de forma directa y de forma indirecta al segundo nivel de cache. La cache de segundo nivel, que recordemos que es una memoria unificada de datos e instrucciones, ve reducidos sus accesos debidos al código previsible y puede dedicar sus recursos al resto del código y a los datos.

- Durante la recolección llegamos a alcanzar una reducción tanto en consumo (33%) como en número de ciclos (10%). Este efecto se satura con una *scratchpad* de 64K, ya que en ese tamaño ya conseguimos capturar la mayoría de instrucciones que van a ser utilizadas durante la recolección.

Hemos visto que la inclusión de una memoria *scratchpad* en la jerarquía de memoria, manteniendo constante el tamaño del resto de memorias cache, provoca una reducción tanto en el consumo como en el número de ciclos. Pero nuestro objetivo es comprobar qué es más efectivo: utilizar el espacio para aumentar el tamaño de las memorias cache de primer nivel o utilizar ese espacio para incluir una memoria *scratchpad*. En la gráfica 5.11 se muestran los valores de energía frente al número de ciclos para dos configuraciones del primer nivel de la jerarquía de memoria. La primera configuración tiene en el primer nivel dos memorias cache de 64K para instrucciones y datos, es decir, en total 128K. La segunda configuración tiene dos memorias cache de 32K para instrucciones y datos, más una memoria *scratchpad* de 64K, en total, de nuevo, 128K. La política de reemplazo es LRU y la asociatividad varía entre directa, 2 y 4 vías. En la gráfica se aprecia que las configuraciones con *scratchpad* en los tres casos obtienen mejores resultados tanto en consumo de energía como en número de ciclos empleados. También podemos observar que al aumentar la asociatividad disminuye el beneficio en el número de ciclos, pero aumenta la reducción en el consumo. De una configuración sin memoria *scratchpad* y asociatividad directa, podemos pasar a tener 4 vías y una memoria *scratchpad*, consiguiendo una reducción en el número de ciclos del 21%, junto con una reducción de 4% del consumo energético. O podemos ver que con una configuración con asociatividad 4, obtenemos una reducción del 55% del consumo al incluir una memoria *scratchpad*, sin perjudicar el número de ciclos (de hecho se produce una reducción del

1%). Para los demás algoritmos de recolección que, recordemos, sufren de una influencia mayor del recolector tanto en rendimiento como en consumo, las ganancias obtenidas son mayores.

5.4.2 Memoria *scratchpad* durante la fase de mutador

Tras ver los buenos resultados de incluir una memoria *scratchpad* en la jerarquía de memoria para guardar los métodos más utilizados durante la fase de recolección, el paso lógico siguiente es ampliar esta estrategia a la fase de mutador de la máquina virtual de java. La fase de mutador incluye la ejecución de la aplicación java, así como las distintas tareas de las que es responsable la máquina virtual: compilación, carga de clases, gestión de hilos, . . . (ver sección 2.2). De este modo la memoria *scratchpad* estaría siendo utilizada durante todo el proceso de ejecución de la aplicación y obtendríamos un mayor rendimiento de ella. En este trabajo nos hemos limitado al estudio de los métodos propios de la máquina virtual de java. La inclusión en la *scratchpad* de los métodos más utilizados de cada aplicación java queda pendiente como trabajo futuro. Esta idea, aunque parece un enfoque prometedor, requiere de un estudio off-line específico para cada programa y no es, por tanto, una solución universal como ocurre con la inclusión de métodos de la máquina virtual.

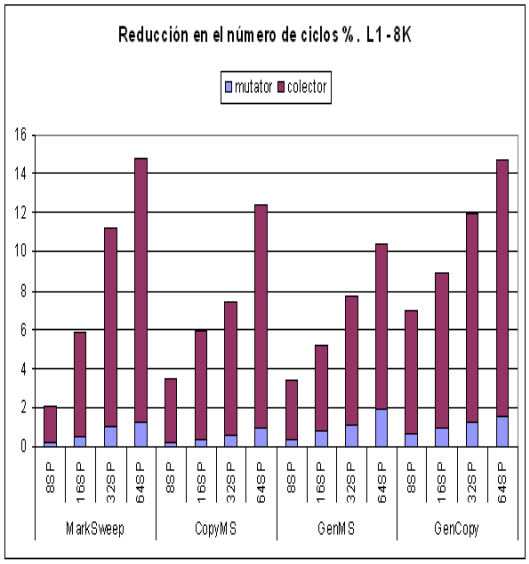
El principal inconveniente de la ampliación de nuestra estrategia a la fase de mutador es encontrar tareas en la máquina virtual que posean las mismas características idóneas que tiene la recolección de basura y que mecionamos en la sección 5.4.1. Las tareas que hemos seleccionado como candidatas han sido el compilador, la carga de clases y el asignador de memoria libre. La compilación dinámica es la tarea del mutador que mayor número de ciclos ocupa seguida por la carga de clases. Así, en nuestros experimentos, durante la fase del mutador la *scratchpad* tiene cargados los métodos del asignador de memoria libre. Cuando la máquina virtual va iniciar dinámicamente la compilación o la carga de una clase, realiza una llamada al hardware para que se cargen en la *scratchpad* los métodos más accedidos correspondientes a esa tarea. Al finalizar su ejecución, en la *scratchpad* vuelven a cargarse los métodos del asignador de memoria. Inevitablemente con cada carga de instrucciones en la *scratchpad* hay una penalización debido a los ciclos necesarios para ello.

En la figura 5.12 se muestra el porcentaje de reducción total en el número de ciclos conseguido distinguiendo entre mutador y recolector para los cuatro recolectores. Con

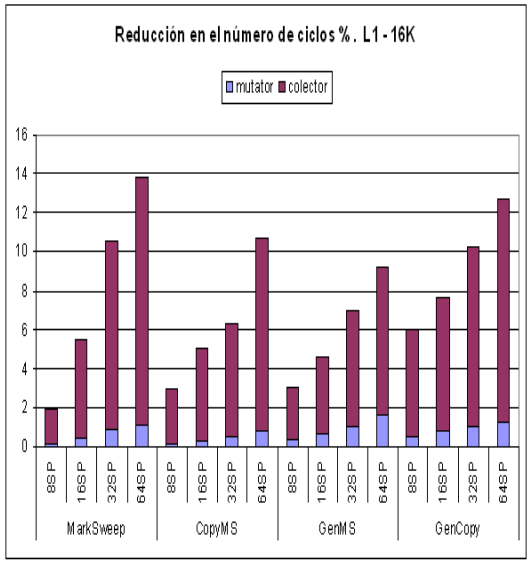
la utilización de la *scratchpad* se consigue reducir en un 2-3% el número de ciclos de la fase de mutador que se corresponde con un peso de 5-15% de la reducción total registrada. Estos porcentajes son claramente inferiores a los conseguidos durante la fase de recolección. La causa la podemos encontrar en que estas tareas no se ejecutan en un número pequeño de ocasiones y durante un número de ciclos seguidos suficientemente grande como ocurre con la recolección. Estas tareas, al contrario, son utilizadas un número grande de veces y durante periodos cortos.

Desde el punto de vista del consumo energético, ver figura 5.13, la situación empeora. El número de accesos a la *scratchpad* durante la fase de mutador no logra compensar el gasto debido a las corrientes de fuga, inevitable al tener encendida la scracthpad durante esta fase. Esta situación provoca un mayor consumo de energía, durante la fase de mutador al tener incluida una memoria *scratchpad* en la jerarquía de memoria, en el caso de los recolectores no generacionales y para los tamaños menores del primer nivel de cache. Como dato esperanzador podemos ver que en el caso de los recolectores generacionales sí se obtienen reducciones de consumo, reducciones que se ven incrementadas al incrementar el tamaño del primer nivel de cache.

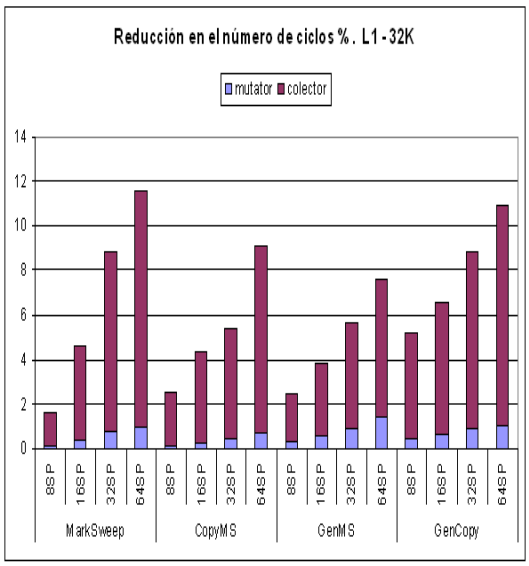
Así, podemos concluir que dentro de la fase de mutador, ninguna de las diferentes tareas de la máquina virtual que hemos estudiado tiene un peso lo suficientemente significativo que justificar el uso de una memoria *scratchpad* para almacenar el código de sus métodos más accedidos. Como trabajo futuro, seguiremos investigando para encontrar otros enfoques que nos permitan sacar partido a la memoria scratchpad durante la fase de mutador.



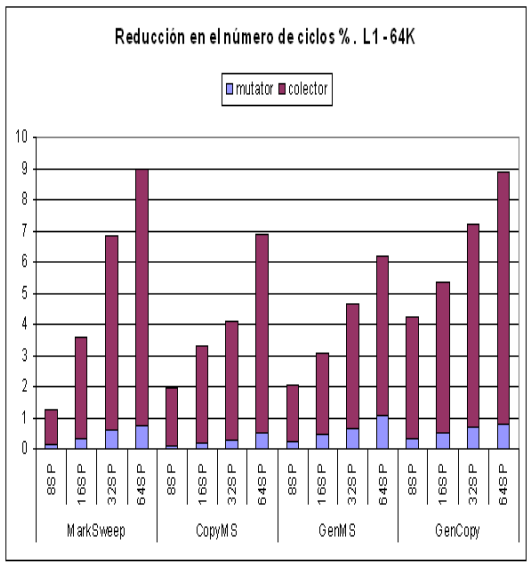
(a) Tamaño del primer nivel de cache es 8K



(b) Tamaño del primer nivel de cache es 16K

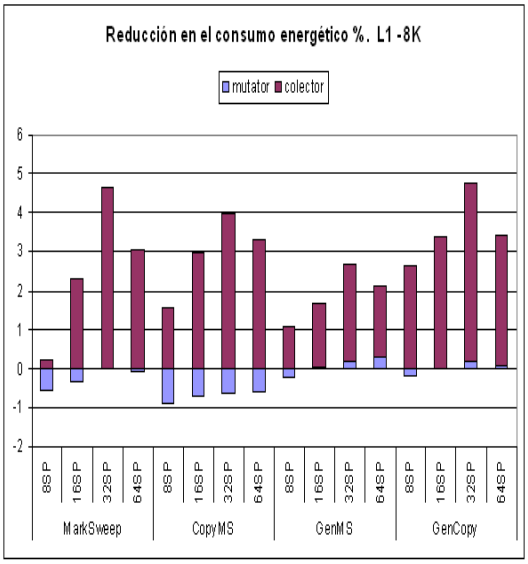


(c) Tamaño del primer nivel de cache es 32K

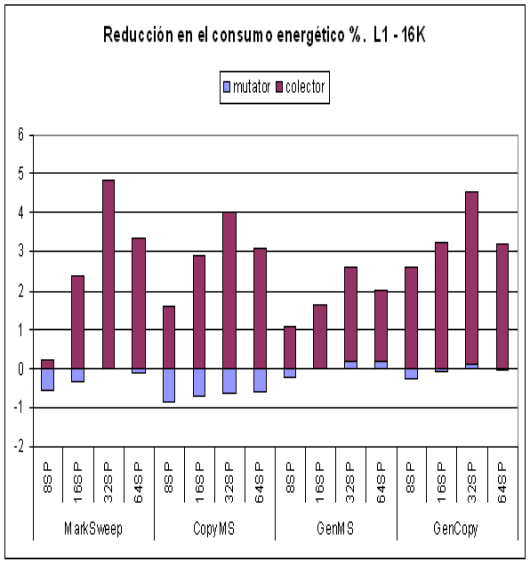


(d) Tamaño del primer nivel de cache es 64K

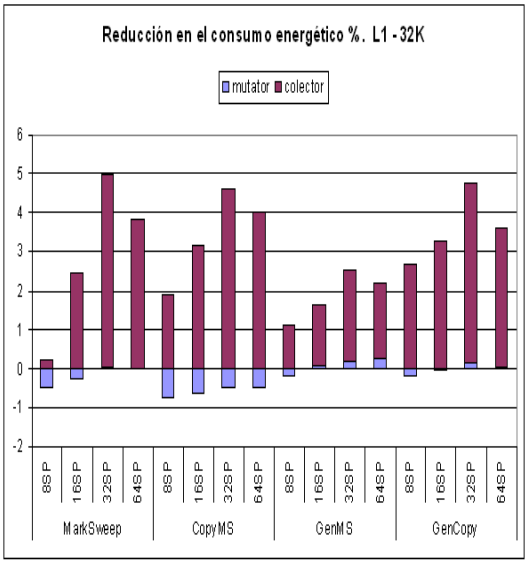
Figura 5.12: Porcentaje de reducción en el número de ciclos. Comparación entre los diferentes tamaños de scracthpad. Promedio de todos los benchmarks.



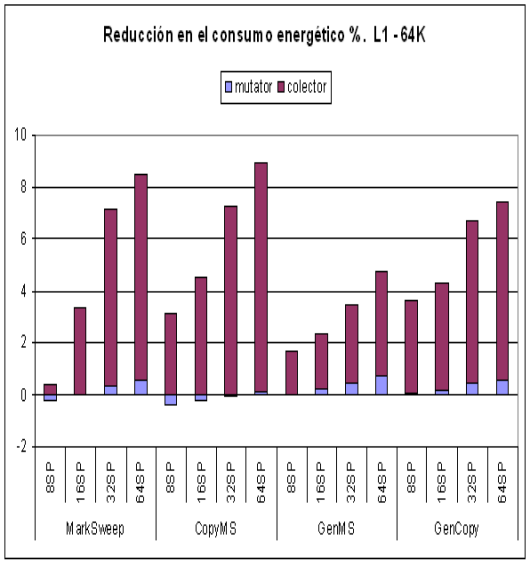
(a) Tamaño del primer nivel de cache es 8K



(b) Tamaño del primer nivel de cache es 16K



(c) Tamaño del primer nivel de cache es 32K



(d) Tamaño del primer nivel de cache es 64K

Figura 5.13: Porcentaje de reducción en el consumo de energía. Comparación entre los diferentes tamaños de scracthpad. Promedio de todos los benchmarks.

5.4.3 Selección dinámica de la imagen de la *scratchpad*

En la sección 5.4.1 se analizó experimentalmente el efecto de incluir, junto al primer nivel de memorias cache, una memoria *scratchpad* con los métodos más utilizados durante la fase de recolección. La elección de estos métodos se llevó a cabo después de estudiar el número de accesos a cada método durante la ejecución de todos los benchmarks y tras promediar los resultados para cada recolector.

En el caso del recolector de marcado y barrido obtuvimos una selección de métodos altamente representativa y con buenos resultados para el conjunto de aplicaciones. Pensemos que para este recolector las diferencias en el número de accesos a cada método dependen del peso de la fase de barrido respecto a la fase de marcado. A mayor número de objetos muertos, mayor peso de la fase de barrido. Pero, puesto que la fase de barrido en JikesRVM, se realiza durante el mutador y junto a la asignación, para este recolector no encontramos diferencias importantes en los métodos más accedidos entre los distintos benchmarks y podemos considerar nuestra selección como idónea.

Sin embargo, para los recolectores que utilizan política de copia, los métodos más accedidos pueden variar significativamente de un benchmark a otro dependiendo de la cantidad de memoria que sobrevive a la recolección. Además en el caso de los recolectores generacionales los métodos difieren radicalmente entre recolecciones menores y mayores. Esto quiere decir que la selección de métodos basada en el promedio de accesos puede alejarse de la solución óptima para algunos benchmarks.

Para afrontar esta situación podemos elegir entre dos estrategias:

- Realizar un estudio previo de cada aplicación y realizar la selección de métodos de forma individual. En este caso, la selección de métodos es óptima. Como contrapartida tenemos que pensar que no es una solución universal y conlleva el coste del estudio previo.

- La selección de métodos que se incluyen en la memoria *scratchpad* se realiza de forma dinámica a partir del comportamiento que la aplicación muestra durante la recolección. El origen de las instrucciones que ejecuta el recolector se reparte entre los métodos de copia y los métodos utilizados para el recorrido del grafo de relaciones en busca de objetos vivos. En última instancia, las diferencias entre aplicaciones están marcadas por el número de objetos que son copiados durante la recolección. Por ello, si hacemos que la máquina virtual registre la cantidad de memoria que

es copiada en cada recolección, el recolector dispondrá de la información clave que le permitirá escoger de forma dinámica la mejor selección de métodos para cada caso concreto. Esta estrategia se podría aplicar al recolector de copia, el recolector híbrido CopyMS y los recolectores generacionales, pero puesto que en el capítulo 4 (sección 4.5) concluimos que, en el caso de sistemas con limitaciones de memoria, los mejores resultados eran obtenidos por los recolectores generacionales, hemos limitado este trabajo a su estudio. En la figura 5.14 se muestra el pseudo-código del algoritmo que hemos utilizado en nuestros experimentos para realizar la selección dinámica de la imagen con los métodos a cargar en la *scratchpad* para los recolectores generacionales. Los pasos principales de este mecanismo adaptativo son:

- Durante la preparación de la recolección:
  - Si estamos ante una recolección global, en la *scratchpad* se carga la imagen que contiene los métodos específicos para recolecciones globales.
  - Si la recolección se limita a la generación *nursery*, se carga en la *scratchpad* una de las imagenes específicas para las recolecciones *nursery*. Inicialmente, esta imagen es la que contiene una mayoria de métodos para el recorrido del grafo de relaciones (”imageMark”).
  - Se registra el tamaño del espacio de maduros y el tamaño del espacio de asignación de la generación *nursery*.
- Tras finalizar la recolección:
  - Se registra el tamaño de la generación madura. A partir de ese dato y de los datos registrados durante la preparación de la recolección se calcula la cantidad de memoria que ha sido copiada tras sobrevivir a la recolección *nursery*. Finalmente se obtiene el ratio que supone la memoria copiada relativa al total de memoria que fue asignada en la generación *nursery*.
  - Cada N recolecciones se calcula el ratio promedio de esas últimas N recolecciones.
  - Si el ratio promedio es menor que una cierta cantidad umbral, la imagen que se cargará durante las próximas N recolecciones será la imagen con un mayor número de métodos destinados a realizar el recorrido del grafo de relaciones (”imageMark”). En caso contrario, durante las próximas N

recolecciones se cargará la imagen que contiene una mayoria de métodos de copia (”imageCopy”)

Cada una de estas imágenes se han obtenido promediando los resultados al igual que se hizo en la estrategia original, pero ahora distinguiendo entre recolecciones con un número alto y un número bajo de objetos copiados. De igual modo se ha procedido con la imagen para las recolecciones globales.

En el siguiente esquema se resumen las tres estrategias estudiadas:

- Average. Es la estrategia promedio antes estudiada (ver sección 5.4.1). En primer lugar se realiza un profiling minucioso del número de accesos a los distintos métodos durante la recolección para cada uno de los benchmarks. Los métodos incluidos en la memoria *scratchpad* son escogidos tras promediar los resultados totales.

- Adaptive. Esta estrategia escoge dinámicamente los métodos a incluir en la memoria *scratchpad* entre tres imágenes distintas de métodos. El criterio utilizado es el ratio de la memoria copiada entre la memoria asignada en la generación *nursery*. Esta estrategia también carga una imagen específica en caso de que la recolección sea global.

- Off-line. En esta estrategia se cargan en la memoria *scratchpad* los métodos más accedidos dutrante el profiling de cada aplicación en concreto, tanto en la recolección *nursery* como en la recolección global. Esta estrategia nos proporciona el beneficio máximo que podría obtenerse con la inclusión de una memoria *scratchpad* en la jerarquía de memoria y que fuese utilizada para guardar métodos del recolector de basura.

Hemos realizado experimentos para comparar los beneficios anteriormente estudiados de la estrategia promedio con el nuevo enfoque adaptativo. Los resultados experimentales se presentan conjuntamente con la estrategia Off-line como punto de referencia que nos muestra el margen de mejora de nuestras estrategias. En la figura 5.15 se muestra el porcentaje de reducción en el número total de ciclos necesarios para la recolección de basura y en la figura 5.16 el porcentaje de reducción en el consumo energético también durante la recolección. En ambos casos se muestran los porcentajes para las tres estrategias, los dos recolectores generacionales y cada uno de los tamaños de la memoria *scratchpad*. El tamaño del primer nivel de cache es 32K con asociatividad directa pero

```
globalPrepare
-------------------------------------------------------------------------
If (globalCollection==true){
        load(imageMature)
}
else{
load(scratchpadImage)
recordAllocationNurserySizeBeforeCollection
recordMatureSizeBeforeCollection
}
-------------------------------------------------------------------------

globalRelease
-------------------------------------------------------------------------
recordMatureSizeAfterCollection
copiedMemory= MatureSizeAfterCollection - MatureSizeBeforeCollection
 lastRatio = copiedMemory / allocationNurserySizeBeforeCollection
sumRatio += lastRatio
profileCounter ++

if  (profileCounter== N) {
                newAverageRatio = sumRatio/ N
                sumRatio =0
                profileCounter = 0

        if (newAverageRatio<=thresholdRatio){
                scratchpadImage=imageMark
        }
        Else{
                scratchpadImage=imageCopy
        }
}
-------------------------------------------------------------------------
```

Figura 5.14: Algoritmo para la selección dinámica de los métodos que se incluyen en la memoria *scratchpad* para los recolectores generacionales

para los otros tamaños los resultados son similares. La política de reemplazo es ”LRU”. A la vista de estas gráficas podemos sacar las siguientes conclusiones:

- Para los dos recolectores estudiados y los distintos tamaños de memoria *scratchpad* y tanto a nivel del número de ciclos como del consumo de energía, la estrategia adaptativa obtiene mejores porcentajes de reducción que la estrategia promedio. A su vez y como era previsible, la estrategia off-line supera los resultados de la estrategia adaptativa y nos proporciona el posible margen de mejora.

- De forma global, el comportamiento de las tres estrategias es análogo para las gráficas de número de ciclos y de consumo de energía.

- En el caso del recolector GenCopy para el tamaño de *scratchpad* más extremo (64K), no hay mucha posibilidad de mejorar los resultados de la estrategia promedio. Podemos observar que la estrategia off-line produce una reducción superior solo en 2-4 puntos, tanto para el número de ciclos como para el consumo. Para los tamaños menores (16K y 32K), la posible mejora es de 6-8 puntos. En todos los casos, la estrategia adaptativa obtiene unos resultados sólo ligeramente inferiores a los obtenidos por la estrategia off-line. De hecho, con la estrategia adaptativa se alcanza en torno a un 95% de la posible mejora obtenida con la selección off-line. Como única excepción, podemos citar el caso del consumo de energía y con tamaño de memoria *scratchpad* de 8K, en el que la cifra es de un 90%.

- La situación es distinta para el recolector GenMS. El comportamiento de la estrategia adaptativa es dispar: los tamaños más pequeños de memoria scratchpad superan los resultados de la estrategia promedio, pero tienen aún un margen de mejora apreciable. En el caso de los tamaños grandes, la estrategia adaptativa obtiene valores muy próximos a la estrategia off-line. Para el tamaño de 16K, la estrategia adaptativa supera a la estrategia promedio, pero obtiene unos resultados inferiores a la estrategia off-line en 7 puntos para el número de ciclos y en 12 puntos para el consumo de energía. Sin embargo, en el caso de tamaño 32K, la estrategia adaptativa obtiene unos resultados que están respecto a la estrategia off-line, tan solo a 2 puntos en el caso de número de ciclos y a 4 puntos en el caso del consumo energético. De hecho, si comparamos los resultados de la estrategia adaptativa y tamaño de *scratchpad* 32K frente a los resultados obtenidos por la estrategia promedio y tamaño de *scratchpad* 64K, podemos ver que son muy similares. La

estrategia adaptativa está ligeramente por debajo en cuanto a la reducción en el número de ciclos, pero obtine una reducción ligeramente superior en el consumo de energía.

- Finalmente, podemos concluir que la técnica adaptativa consigue una mejora en torno al 90-95% de la máxima ganancia posible para el recolector GenCopy y del 85-90% para el recolector GenMS, en ambos casos con tamaños de scratchpad de 32K y 64K.

En la gráfica 5.17 se muestran los valores (recolector) de energía frente al número de ciclos para tres configuraciones del primer nivel de la jerarquía de memoria. La primera configuración tiene en el primer nivel dos memorias cache de 64K para instrucciones y datos, es decir, en total 128K. La segunda configuración tiene dos memorias cache de 32K para instrucciones y datos, más una memoria *scratchpad* de 32K, que hacen en total 96K. Es decir, esta configuración supone un ahorro en el tamaño total de la memoria del primer nivel de cache. La tercera configuración tiene dos memorias cache de 32K para instrucciones y datos, más una memoria *scratchpad* de 64K, en total son 128K, como en la primera configuración. El recolector es GenMS y la selección de los métodos que se incluyen en la *scratchpad* se lleva a cabo mediante la estrategia adaptativa. La política de reemplazo es LRU y la asociatividad varía entre directa, 2 y 4 vías.

Los puntos de las configuraciones sin *scratchpad* están dibujados con el color rosa. Los puntos de las configuraciones con memoria *scratchpad* de tamaño 32K se han dibujado con color naranja. Y cuando el tamaño de la *scratchpad* es de 64K se ha utilizado el color verde. Para dibujar los puntos con asociatividad directa en el primer nivel de memoria cache se ha utilizado como símbolo el rombo. Los puntos con asociatividad 2 vías se han dibujado utilizando un cuadrado. Y los puntos con asociatividad de 4 vías se muestran con un triángulo.

En la gráfica se aprecia que los puntos de la segunda configuración obtienen siempre mejores resultados tanto en consumo de energía como en número de ciclos empleados respecto a la primera configuración. También podemos observar que al aumentar la asociatividad disminuye el beneficio en el número de ciclos, pero aumenta la reducción en el consumo. De una configuración sin memoria *scratchpad* y asociatividad directa, podemos pasar a tener 4 vías y una memoria *scratchpad* de 32K, consiguiendo una reducción en el número de ciclos del 20%, junto con una reducción del 5% del consumo energético. O, podemos ver, que de una configuración con asociatividad 4, obtenemos una

reducción del 55% del consumo al incluir una memoria *scratchpad* de 32K, sin perjudicar el número de ciclos. Recordemos que la segunda configuración, además, supone un ahorro en el tamaño total de la memoria del primer nivel de cache.

Si comparamos los puntos de la tercera configuración respecto a los puntos de la segunda configuración, podemos observar que siempre hay una mejora tanto en número de ciclos como de consumo de energía. Sin embargo, la reducción en el número de ciclos es considerablemente mayor (10-15%) que la obtenida en cuanto el consumo (2-4%). En el caso de asociatividad directa, el aumento del tamaño de la memoria *scratchpad* de 32K a 64K, produce una reducción del 24% en el número de ciclos y una ligera reducción del 3% en el consumo (siendo ese el mejor valor registrado en el apartado de consumo). En el caso de asociatividad 4 vías, la configuración con memoria *scratchpad* de 64K consigue reducir en un 22% el número de ciclos, consiguiendo el mejor resultado registrado en ese apartado para las tres configuraciones. Y al mismo tiempo reduce el consumo en un 6%. Podemos concluir, que en el caso de cuatro vías, la inclusión de una memoria *scratchpad* de 32K produce una gran reducción en el consumo, manteniendo invariable el número de ciclos, y que si incluimos una *scratchpad* de 64K, conseguimos además, una reducción en el número de ciclos. Por todo ello, la curva de Pareto que presentamos en el capítulo 4 (figura 4.8(c)) cambia sustancialmente. En la figura 5.18 se muestran los puntos de la antigua curva, correspondientes a las configuraciones sin memoria *scratchpad* (color azul oscuro), y los nuevos puntos tras incluir la memoria *scratchpad* (color rosa). Como se aprecia en la gráfica, los nuevos puntos forman una nueva curva a la que ahora no pertenecen las configuraciones con cache de tamaño 16K.

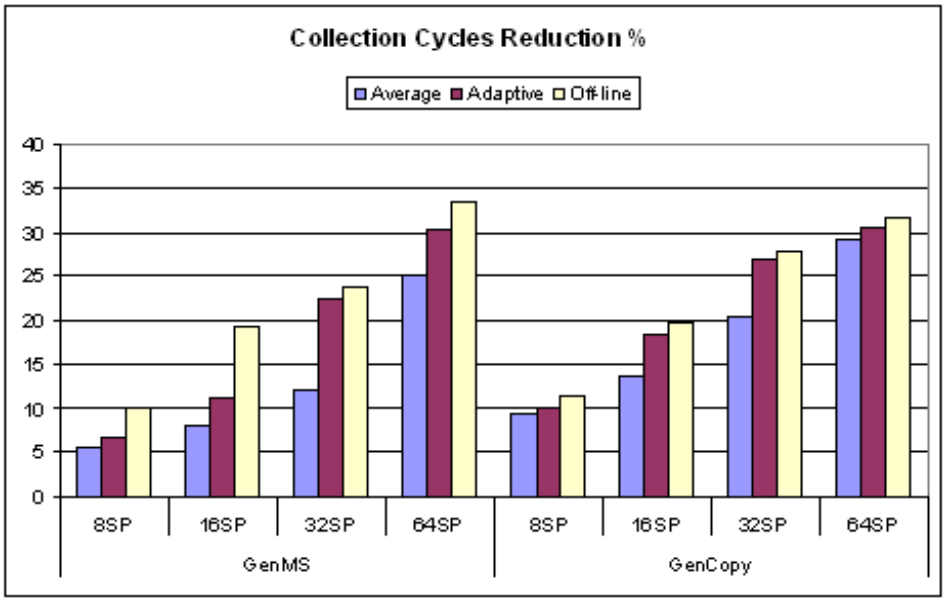


Figura 5.15: Porcentaje de reducción del número de ciclos del recolector para las tres estrategias que utilizan memoria *scratchpad*.

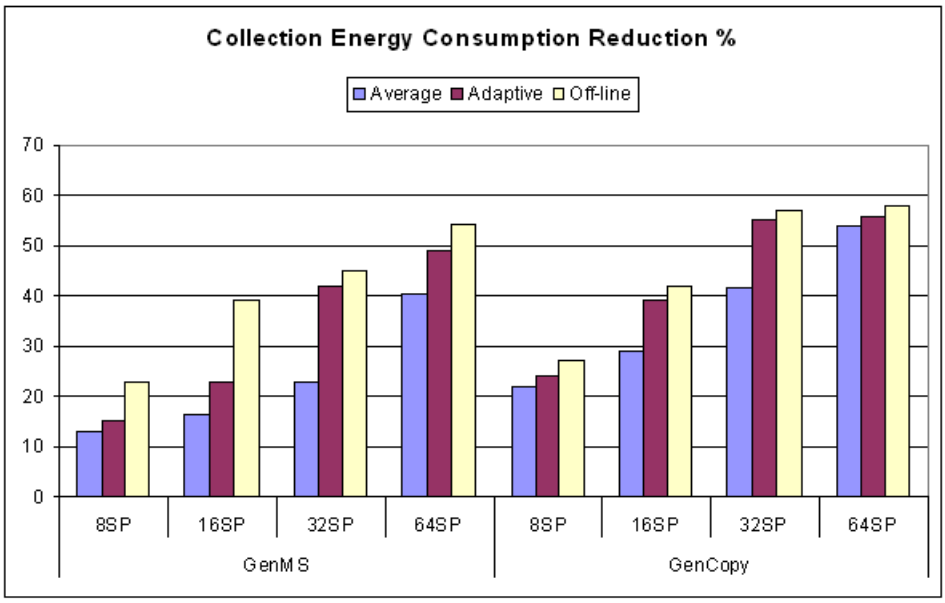


Figura 5.16: Porcentaje de reducción del consumo energético del recolector para las tres estrategias que utilizan memoria *scratchpad*.

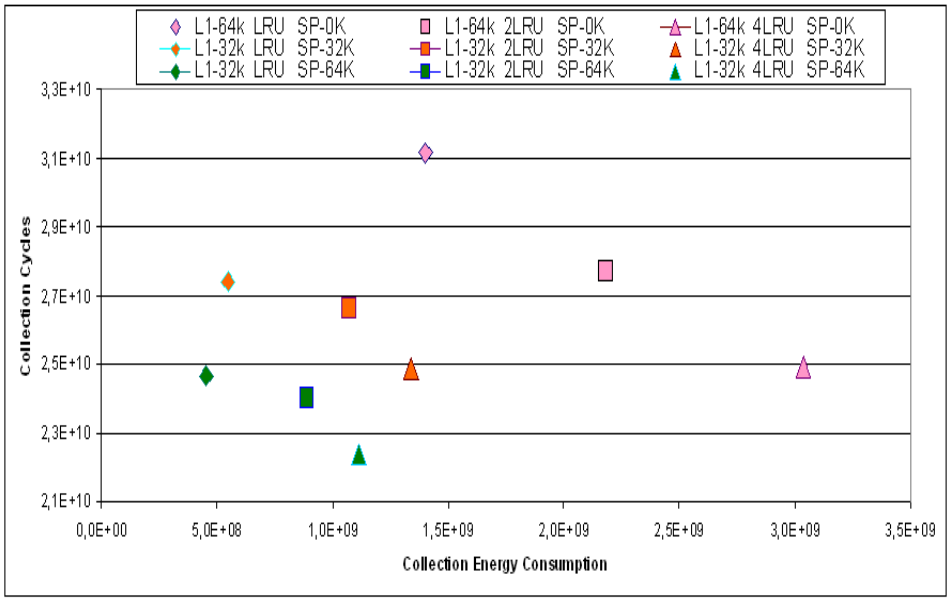


Figura 5.17: Consumo de energía frente a número de ciclos para las configuraciones sin *scratchpad*, y con tamaño de *scratchpad* de 32K y 64K.

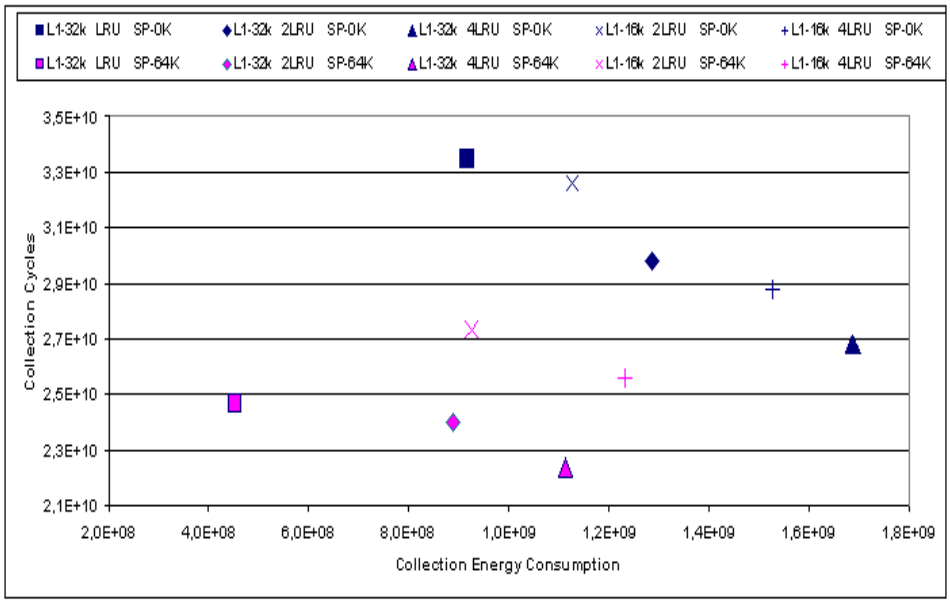


Figura 5.18: Curva de Pareto con y sin *scratchpad*.

5.5 Conclusiones

A partir de la información recopilada en el capítulo 4 y gracias a nuestra experiencia en el comportamiento de la máquina virtual de Java durante las fases de mutador y recolector, en este capítulo hemos presentado dos estrategias ideadas para la reducción del consumo de energía.

- La primera estrategia permite reducir el consumo estático de la memoria principal y está pensada para los recolectores generacionales, especialmente el generacional de copia puro. En nuestra técnica, la máquina virtual se encarga de controlar el modo de bajo consumo de los diferentes bancos de una memoria SDRAM. Para ello utiliza la información suministrada por el recolector de basura: tamaño de las generaciones e instantes de inicio y fin de las recolecciones. En los resultados de nuestras simulaciones, este mecanismo reduce a la mitad el consumo debido a las corrientes de fuga en una memoria de 16MB para el recolector generacional de copia. El dato más interesante de nuestros experimentos, es que con el aumento del tamaño de la memoria principal, nuestra estrategia consigue reducciones mayores. Como trabajo futuro, planeamos incluir un algoritmo compactador en la generación madura de GenMS que permita mejorar los resultados obtenidos en este recoelctor.

- La segunda estrategia, aplicable a todos los recolectores, reduce tanto el consumo dinámico como el tiempo total de recolección. Esta técnica consiste en utilizar una memoria *scratchpad* con el código de los métodos del recolector que son más accedidos durante la ejecución. En el capítulo hemos presentado tres formas posibles de seleccionar el contenido de la *scratchpad*. Los resultados experimentales de esta estrategia nos han proporcionado una nueva curva de Pareto con las configuraciones óptimas para el rendimiento y consumo. En esta nueva curva hemos comprobado que siempre es más rentable la inclusión de una memoria *scratchpad* con el código del recolector, en vez de un aumento ya sea del tamaño o de la asociatividad del primer nivel de la jerarquía de memoria. También hemos intentado ampliar esta técnica a la fase de mutador, pero no hemos encontrado niguna otra tarea de la máquina virtual cuyo peso en la ejecución sea comparable al recolector y obtenga resultados satisfactorios.

Capítulo 6

Recolector Generacional Adaptativo

6.1 Introducción

En la sección 4.5 se concluyó que dentro de los recolectores estudiados, los generacionales obtienen los mejores resultados, tanto a nivel de rendimiento como de consumo energético. Dentro de los dos recolectores generacionales (GenMS y GenCopy), y para el caso estudiado (tamaño de *heap* muy limitado), el recolector que gestiona la generación madura con la política de marcado y barrido (GenMS) obtiene una cierta ventaja.

En la capítulo 5 presentamos dos técnicas ideadas para reducir el consumo energético y aumentar el rendimiento de la máquina virtual, especialmente cuando ésta está implementada con un recolector generacional.

En el capítulo actual vamos a proponer una optimización de los recolectores generacionales a nivel algorítmico. Básicamente proponemos que la máquina virtual modifique dinámicamente el tamaño del espacio de reserva de sus generaciones basándose en información recopilada en tiempo de ejecución. De este modo conseguimos reducir el número total de recolecciones, la cantidad de memoria copiada, el tiempo de recolección y, claro está, el consumo de energía.

Nuestro recolector generacional adaptativo está especialmente indicado para los sistemas empotrados, pero puede emplearse con éxito para distintos tamaños de memoria. Por ello, primero mostraremos un estudio experimental para un rango de tamaños de

memoria más amplio del utilizado hasta ahora, y a continucación presentaremos los resultados dentro del tamaño de memoria principal habitual en este trabajo. Al tratarse de una técnica software, nuestro algoritmo puede usarse de forma paralela con las técnicas del capítulo 5 y al final de la sección 6.5 se muestran los resultados obtenidos tras aplicar conjuntamente las tres técnicas.

Dentro de los trabajos recientes de investigación en recolección de basura podemos citar como más relevantes a nuestro trabajo:

- En el artículo de Sachindran y Moss [SM03], se presenta una técnica de compactación para un recolector generacional híbrido con política de marcado y barrido en la generación madura. Este trabajo es completamente paralelo al nuestro y un recolector que implementase las dos técnicas, sumaría los beneficios de ambas y estimamos que, de hecho, sacaría más partido a nuestra estrategia.

- Otro trabajo paralelo al nuestro y que puede usarse conjuntamente es el de Harris [Har00]. El recolector genracional de Harris recopila información acerca del comportamiento de los objetos para buscar entre ellos candidatos a ser asignados directamente a la generación madura (*pretenuring*), sin pasar por la generación *nursery*.

- La propuesta de Stefanovic et al. [SMEM99] se inspira en la filosofía generacional y busca, al igual que nosotros, aprovechar mejor el espacio de memoria. En este recolector, el *heap* se divide en ventanas (*windows*) de igual tamaño en vez de generaciones, necesitando siempre tener una ventana reservada para la copia de objetos supervivientes. Lo interesante de esta propuesta es que al aumentar el número de ventanas el tamaño de éstas disminuye de forma acorde y por tanto el tamaño del espacio de reserva. El punto débil de esta estrategia es doble. Por un lado no puede reclamar objetos que pertenezcan a listas circulares o doblemenete enlazadas con referencias cruzadas entre ventanas. Por otro lado, al aumentar el número de ventanas crece también el coste de las barreras de escritura y la cantidad de datos que éstas generan. Este factor no hace aconsejable, en nuestra opinión, su utilización en plataformas con limitaciones de memoria, como son los sistemas empotrados.

El resto del capítulo se organiza del siguiente modo: en la sección 6.2 se muestran los datos experimentales que justifican nuestra propuesta. En la sección 6.3 se presenta

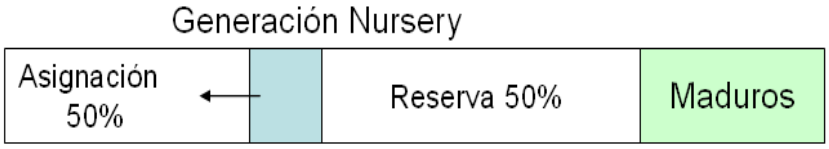
la idea central de nuestra estrategia y así en la sección 6.3.1 se plantean dos posibles algoritmos para su implementación. En las secciones 6.3.2 y 6.3.3 se discute acerca de distintos aspectos de la realización práctica de esta técnica. La aplicación de nuestra propuesta a la generación madura se plantea en la sección 6.4 y los resultados experimentales se muestran en la sección 6.5. El capítulo finaliza resumiendo nuestras conclusiones (sección 6.6).

6.2 El espacio de reserva

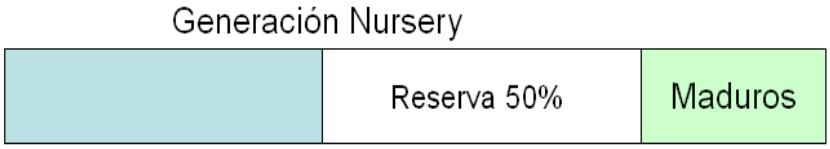
Recordamos que los recolectores generacionales, sección 2.5.4, pueden tener un tamaño de *nursery* fijo o flexible. El recolector con tamaño de *nursery* flexible, normalmente llamado recolector *Appel*, asigna toda la memoria disponible a la generación *nursery*. Tras la recolección todos los objetos vivos son trasladados a la generación madura, por ello, la estrategia con la que se implementa es la de copia e inevitablemente el espacio de la generación *nursery* se divide en dos mitades: una mitad donde se produce la asignación de objetos nuevos y la otra mitad que se reserva para la copia de objetos supervivientes a la recolección.

En la figura 6.1 podemos ver el funcionamiento del recolector *Appel* híbrido (GenMS) con dos generaciones. Tras una recolección cualquiera, toda la memoria que no está ocupada por la generación madura se destina a la generación *nursery*. La estrategia de copia divide ese espacio disponible en dos mitades idénticas, figura 6.1(a). La asignación de los objetos recién creados se lleva a cabo en una de esas mitades hasta agotarla, momento en que se llama al recolector, figura 6.1(b). Los objetos supervivientes se copian en el espacio de reserva y pasan a formar parte de la generación madura, figura 6.1(c). El espacio que no ocupan los objetos maduros es ahora la nueva generación *nursery*, figura 6.1(d).

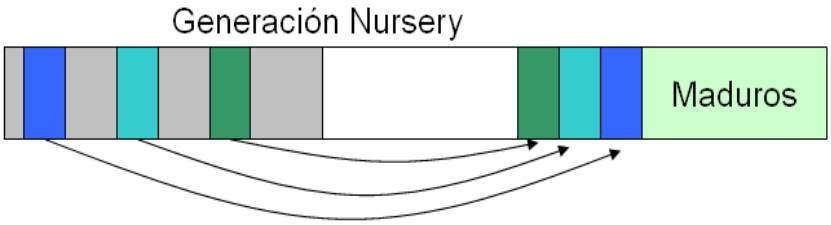
El tamaño del espacio de reserva es igual al de asignación para preveer el caso en el que todos los objetos de la generación *nursery* sobrevivan a la recolección. No obstante, la filosofía detrás de la estrategia generacional (”hipótesis generacional débil”, sección 2.5.3), nos dice que en la generación *nursery* esperamos tener un índice de mortalidad muy elevado. Por ello, es de suponer que gran parte de ese espacio reservado conservadoramente va a ser espacio malgastado en la mayoría de las recolecciones. En la figura 6.2 se muestra la ejecución de los cinco benchmarks más representativos



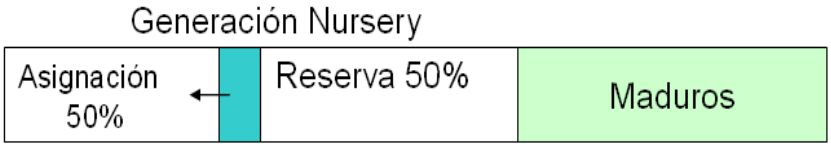
(a) Asignación en la generación Nursery



(b) Llamada al recolector



(c) Copia de los objetos vivos



(d) El espacio no ocupado por los maduros es la nueva generación *Nursery*

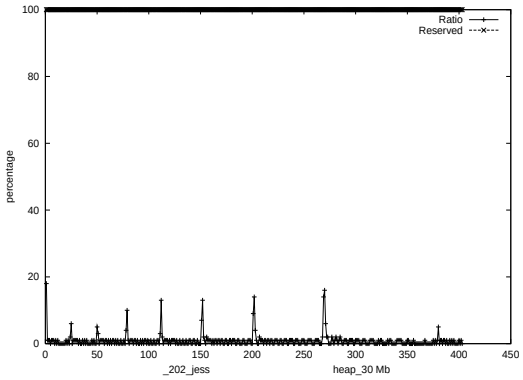
Figura 6.1: Funcionamiento del recolector generacional *Appel*

(desde el punto de vista de la recolección de basura) del SPECjvm98 (sección 3.6). En el eje de ordenadas se sitúan las recolecciones ordenadas cronológicamente. En el eje de abcisas se muestra el porcentaje que representa la memoria copiada en cada recolección relativa a la cantidad de memoria reservada para ello. Podemos ver que para jess(figura6.2(a)) y jack(figura6.2(b)) la memoria copiada (superviviente en el espacio de asignación) nunca supera el 20% de porcentaje de la memoria reservada, teniendo la mayoría de las recolecciones porcentajes muy inferiores. Para javac (figura 6.2(c)) nos encontramos con una mezcla heterogénea de porcentajes que oscilan entre un 40% y un 10% y que nunca supera la mitad del espacio de reserva. Para raytrace y mtrt (figuras6.2(d) y 6.2(e)), la situación es aún más dispar. Aunque hay recolecciones que alcanzan el 60% del espacio de reserva, la gran mayoría de las recolecciones (más del 98%) no tienen ningún, o prácticamente ningún, objeto superviviente. En la figura 6.2(f) se muestra el promedio de memoria del espacio de reserva que es utilizada para el conjunto de las recolecciones. De este modo se concluye que, aunque para garantizar el correcto funcionamiento del recolector es necesario reservar una cantidad de memoria equivalente a la utilizada para la asignación, en la mayoría de los casos este espacio de reserva es un gasto inútil. Nuestro objetivo es aprovechar inteligentemente esta oportunidad para optimizar el comportamiento de los recolectores generacionales.

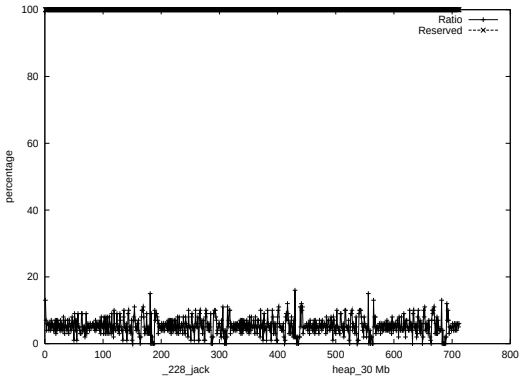
6.3 Adaptación de los parámetros del *Nursery*

Nuestra propuesta consiste en modificar la relación 50:50 entre el espacio de asignación y el espacio de reserva en los recolectores generacionales, tal como ilustra la figura 6.3. Puesto que el espacio de reserva se desaprovecha la mayor parte del tiempo parece lógico transferir parte de su memoria al espacio de asignación. De este modo se aprovecharían mejor los recursos disponibles. Para hallar la nueva relación entre los dos espacios se puede optar por tres estrategias:

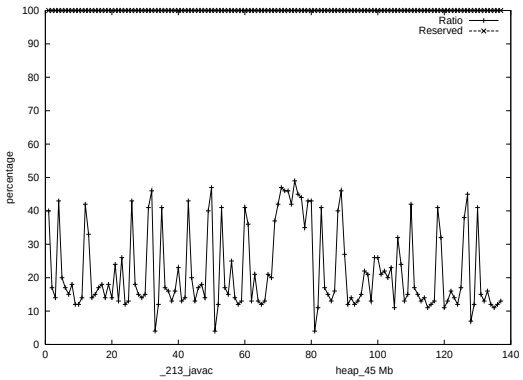
- Establecer una nueva relación fija para todos los casos. Esta nueva relación tendría que ser conservadora puesto que la aplicaríamos por igual a todos los benchmarks y durante todas las fases de la ejecución. Esta estrategia no conseguiría aprovechar los recursos para aquellos benchmarks con ratios de supervivencia muy bajos que pueden verse beneficiados por recortes drásticos en el tamaño del espacio de reserva. Del mismo modo fallaría para las aplicaciones que pasan por fases con



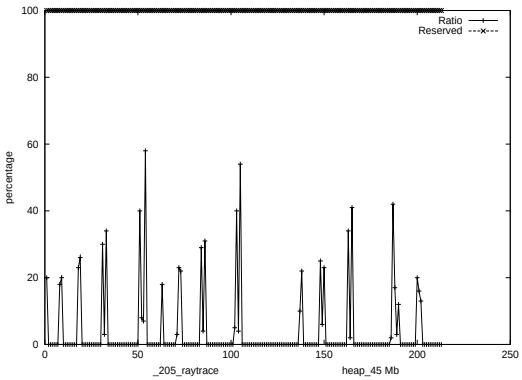
(a) _202_jess



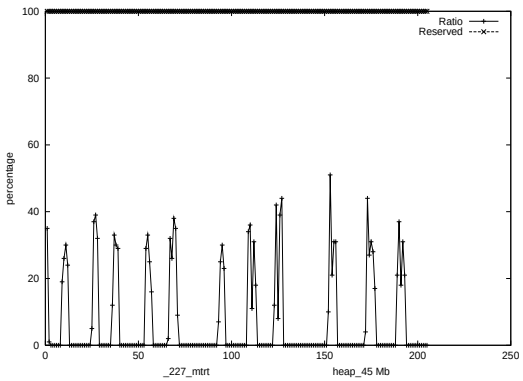
(b) _228_jack



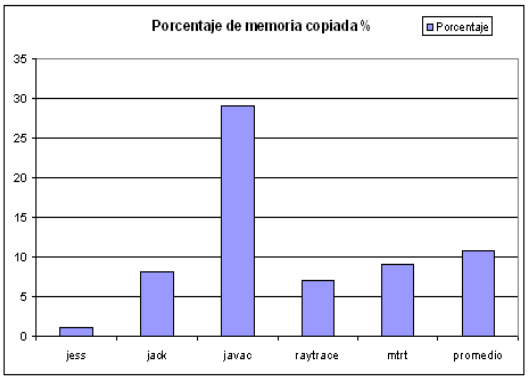
(c) _213_javac



(d) _206_raytrace

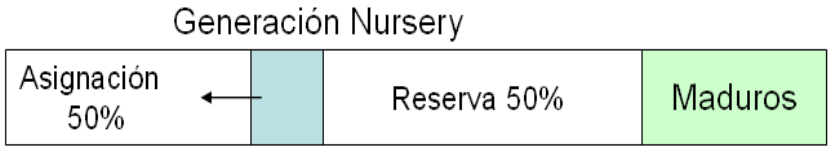


(e) _227_mtrt

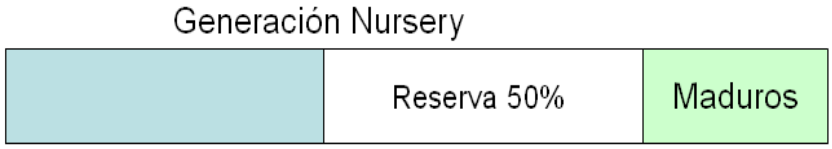


(f) Promedio

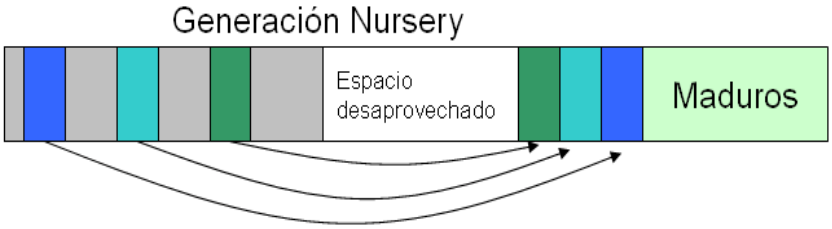
Figura 6.2: Porcentaje del espacio de reserva que se utiliza para la copia de objetos supervivientes tras la recolección



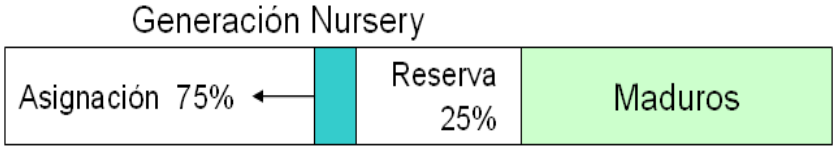
(a) Asignación en la generación Nursery



(b) Llamada al recolector



(c) Registro de la memoria copiada y del espacio desaprovechado



(d) Reducción del espacio de reserva a partir de los datos registrados durante las últimas recolecciones

Figura 6.3: Propuesta de modificación del tamaño del espacio de reserva en la generación *Nursery*

promedios de supervivientes muy dispares.

- Realizar un estudio off-line específico para cada benchmark y hallar la relación óptima para cada uno. El problema de esta opción es que el gasto del estudio previo es muy elevado si pensamos en aplicaciones con comportamientos dependientes de los datos de entrada, máquinas virtuales con diferentes tamaños de heap disponibles, etc. Además, la nueva relación, aunque específica para cada aplicación, siempre tendrá que ser conservadora y ceñirse a las fases de ejecución con mayores índices de memoria copiada y por tanto fallará en ejecuciones heterogéneas. Por todo ello, hemos desechado esta opción.

- Y por último, la opción por la que nos hemos decantado: desarrollar técnicas dinámicas de modo que el recolector pueda adaptarse durante cada fase de la ejecución y para cada benchmark particular. En nuestra propuesta la máquina virtual es la encargada de realizar un *profiling* del comportamiento de la aplicación en cada ejecución concreta, para un tamaño de heap y unos datos de entrada determinados. A partir de los datos registrados, el recolector puede modificar dinámicamente el tamaño del espacio de reserva de la generación *nursery* para aprovechar al máximo posible los recursos disponibles. Para el correcto funcionamiento de nuestra propuesta tenemos que garantizar que:

  - La instrumentación necesaria para facilitar al recolector la información que le permita modificar dinámicamente la nueva relación entre los espacios de asignación y reserva debe ser muy ligera para no interferir en exceso en la actividad de la máquina virtual y contrarrestar los beneficios que esperamos de esta técnica.

  - El recolector ha de disponer de una técnica de emergencia en caso de que cambios bruscos en el comportamiento de una aplicación provoquen que la máquina virtual se quede sin memoria de reserva para copiar objetos supervivientes a una recolección.

A continuación, en la sección 6.3.1, presentamos dos algoritmos adaptativos para modificar dinámicamente el tamaño del espacio de reserva de la generación *Nursery*.

6.3.1 Algoritmos adaptativos para reajustar el tamaño del espacio de reserva

Para la elección del tamaño del espacio de reserva hemos buscado estrategias muy simples que necesiten la menor cantidad de instrumentación posible. En la figura 6.4 se muestra el pseudo-código que implementa dos algoritmos dinámicos con filosofías distintas. En primer lugar y de forma común a los dos algoritmos, durante la preparación de la recolección se registran los tamaños del espacio de maduros y del espacio de asignación *nursery*, figura 6.4(a). Al finalizar la recolección, y antes de que el recolector devuelva el control a la máquina virtual, tiene lugar (si es necesario) la elección del nuevo tamaño del espacio de reserva. Esta fase difiere según el algoritmo, figura 6.4(b):

- El algoritmo ”conservador” fija la relación entre los dos espacios a partir de la cantidad más alta de objetos supervivientes registrada durante la ejecución.
- La estrategia ”promedio” es más agresiva y está enfocada a aprovechar al máximo las distintas fases de ejecución. Este algoritmo fija el tamaño del espacio de reserva basándose el promedio de memoria copiada de las últimas N recolecciones. En nuestros experimentos hemos variado el valor de N de 2 a 10 y no hemos apreciado diferencias significativas. En los resultados experimentales mostrados en este capítulo el valor de N es igual a cinco.

En ambos casos, al resultado obtenido se le suma un margen de seguridad. En nigún caso se permite que el espacio de reserva sea menor que un cierto porcentaje mínimo. En nuestros experimentos hemos variado el margen de seguridad desde un 10% a un 50% y no hemos encontrado diferencias significativas. En los resultados mostrados en este capítulo se ha empleado un margen de seguridad de 50% para la estrategia ”promedio” y un margen de 30% para la estrategia conservadora.

En la figura 6.5 podemos ver gráficamente el comportamiento de los dos algoritmos para el benchmark javac. En el eje de ordenadas se sitúan las recolecciones ordenadas cronológicamente. En el eje de abcisas se muestra el porcentaje que representa la memoria copiada en cada recolección relativa a la cantidad de memoria reservada previamente y el porcentaje que representa el espacio reservado para la copia. Se ha elegido el benchmark javac por ser el de comportamiento más variado, con numerosas recolecciones y porcentajes de supervivientes altos y bajos. Como se aprecia en la figura 6.5(a), la estrategia que modifica el tamaño del espacio de reserva basándose en

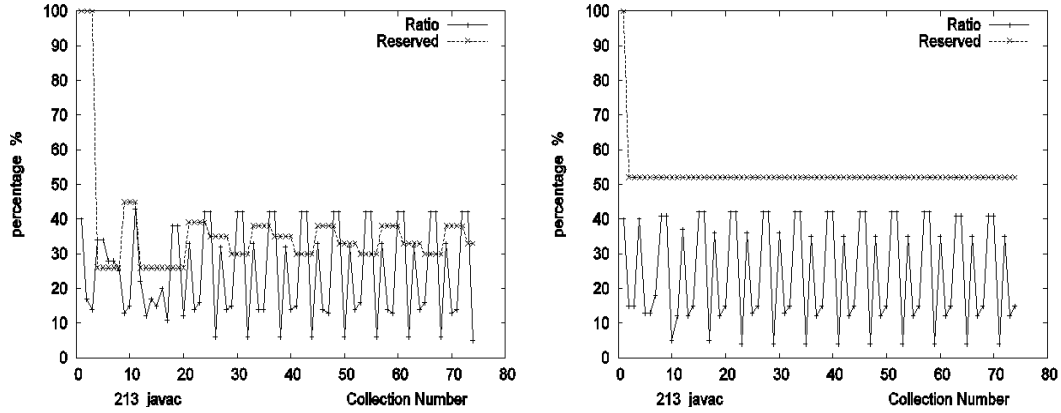
```
globalPrepare
-------------------------------------------------------------------------
recordAllocationNurserySizeBeforeCollection
recordMatureSizeBeforeCollection
if (biggestAllocationNurserySize<AllocationNursery SizeBeforeCollection)
  { biggestAllocationNurserySize=AllocationNurserySizeBeforeCollection)}
-------------------------------------------------------------------------
```

(a)

```
globalRelease
-------------------------------------------------------------------------
recordMatureSizeAfterCollection
copiedMemory = MatureSizeAfterCollection - MatureSizeBeforeCollection
 lastRatio = copiedMemory / allocationNurserySize
if (strategy = = Conservative){
        if ( lastRatio > worstRatioRecorded) {
                worstRatioRecorded = lastRatio
                newSecurityRatio = lastRatio  * securityMargin
                nurseryReserve= lastRatio + newSecurityRatio
                newNurseryThreshold =
                   biggestAllocationNurserySize * nurseryReserve /100
       }
 }
else if (strategy = = average) {
        sumRatio += lastRatio
        profileCounter ++
        if  (profileCounter = = N) {
                profileCounter = 0
                newAverageRatio = sumRatio/ N
                if (newAverageRatio< minimumReserve)
                        {newAverageRatio = minimumReserve}
                newSecurityRatio = newAverageRatio *  marginA
                nurseryReserve= newAverageRatio + newSecurityRatio
                newNurseryThreshold =
                      biggestAllocationNurserySize  *  nurseryReserve/100
        }
}
```

(b)

Figura 6.4: Pseudo-código para la implementación de los algoritmos adaptativos.



(a) Promedio de las últimas N recolecciones (b) Conservadora

Figura 6.5: Estrategias para decidir el tamaño del espacio de reserva

el promedio en las últimas N recolecciones realiza numerosas modificaciones e incurre en varios errores de estimación (como era previsible en una política agresiva). La estrategia más conservadora (figura 6.5(b)), sin embargo, sólo realiza una modificación del tamaño del espacio de reserva (a un 50% del tamaño del espacio de asignación) y no incurre en ningún error de previsión, aunque podríamos achacarle que siempre está reservando un espacio excesivo que en el mejor de los casos es superior en un 10% al estríctamente necesario.

En la sección 6.5 se presentan los resultados experimentales para ambas estrategias y se compara su rendimiento final.

6.3.2 El umbral que dispara la recolección global

A medida que la ejecución progresa y las recolecciones se producen, el tamaño de la generación madura aumenta y por tanto el espacio disponible para la generación *nursery* se reduce. Este menor espacio provoca que el tiempo entre recolecciones también se acorte. De modo que los objetos tienen menos tiempo para morir y el comportamiento de la aplicación, en términos de datos supervivientes, puede variar rápidamente como vimos en el capítulo 2 (subsección 2.5.4). Para contrarestar este efecto utilizamos dos técnicas, figura 6.4(b):

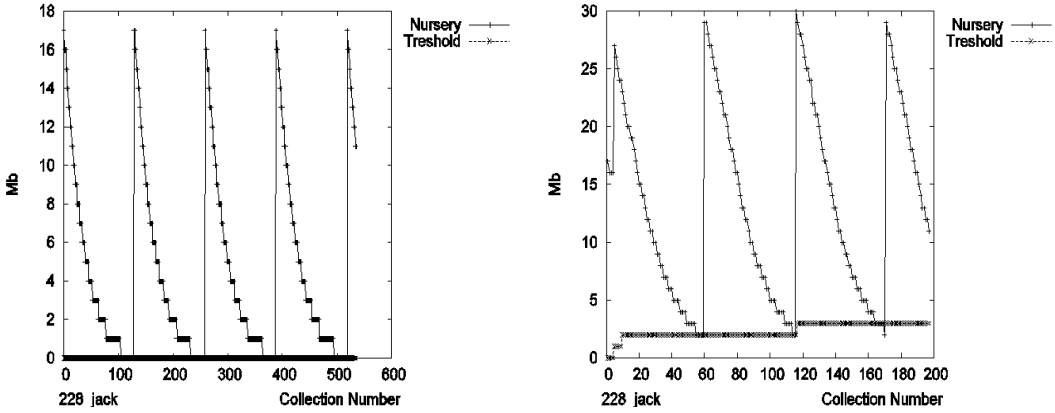
- La primera ya fué explicada en la subsección 6.3.1. Al finalizar la recolección se comprueba la cantidad de memoria en el espacio de reserva que ha quedado libre. Si esta cantidad cae por debajo de un cierto umbral el porcentaje que representa el espacio de reserva respecto al de asignación se incrementa en este valor.

- Sin embargo, ésta técnica por si sola no basta para evitar el incremento de recolecciones *nursery* a medida que el tamaño disponible para ésta generación disminuye. Por ello, cada vez que el recolector reajusta el porcentaje del espacio de reserva, también cambia el tamaño umbral de la generación *nursery* que dispara una recolección global. De este modo, además de evitar los fallos de estimación de nuestro recolector, se evita la producción de un gran número de recolecciones *nursery* con porcentajes altos de copia debido a que los objetos no han tenido un tiempo mínimo para completar su vida.

En la figura 6.6(a) se muestra la evolución del tamaño del espacio de asignación para el benchmark jack (aplicación con varias recolecciones globales) con un recoletor *Appel* GenMS y un heap de 40MB (este tamaño está escogido porque el comportamiento es muy ilustrativo). El umbral que dispara una recolección global está fijado en 512 KB (por defecto en JikesRVM). En el eje de ordenadas se sitúan las recolecciones a lo largo de la ejecución. Podemos ver que el tamaño máximo que alcanza el espacio de asignación es de 17MB, las recolecciones globales son cuatro y las recolecciones en la generación *nursery* sobrepasan las cinco centenas. En contraposición, en la figura 6.6(b) se muestra la misma evolución con nuestra política adaptativa promedio. La reducción del espacio de reserva permite que el tamaño máximo que experimenta el espacio de asignación casi alcance los 30MB. Con los distintos reajustes del espacio de reserva, el umbral cambia a 2MB primero y posteriormente a 3.5MB. En conjunto esto produce que el número de recolecciones globales, disparadas porque el tamaño del nursery es menor al tamaño umbral, se sitúe en tres y el total de recolecciones en la generación *nursery* esté por debajo de doscientas.

6.3.3 Recuperación cuando la memoria reservada es insuficiente

Al realizar una predicción del espacio de reserva que el recolector va a necesitar en una próxima recolección, cada algoritmo tiene una cierta probabilidad de incurrir en una estimación errónea. En ese caso, necesitamos dotar a la máquina virtual de un mecanismo



(a) Estático

(b) Porcentaje del espacio de reserva

Figura 6.6: Cambio dinámico del umbral que dispara una recolección global

que le permita liberar memoria en el heap para compensar el insuficiente tamaño del espacio de reserva. En la tabla 6.1 se muestra el número de predicciones incorrectas para cada una de las estrategias dinámicas. La estrategia más agresiva, basada en el promedio de las últimas recolecciones, es lógicamente la que incurre en un mayor número de fallos para aquellos benchmarks con un comportamiento más errático (observemos que el peor caso es javac con 12 fallos). Para estas aplicaciones, la estrategia conservadora obtiene un menor número de errores, aunque sigue incurriendo en fallos de predicción.

Nuestra estrategia de recuperación cuando la memoria reservada en el *nursery* es insuficiente se basa en dos supuestos:

- Si se trata de GenMS, suponemos que en el espacio de maduros hay muchos objetos que han muerto desde la última recolección global. Si se trata de GenCopy, además de ese factor, contamos con el espacio de reserva para la generación de maduros.

- En el "*remembered set*" (la pila donde se guardan las referencias desde el espacio de maduros al espacio *nursery*) hay muchos punteros pertenecientes a objetos muertos (lo que se conoce como "nepotismo"). Los objetos referenciados en el *nursery* desde el espacio de maduros por objetos muertos, probablemente también estén muertos. Por tanto, buscar espacio de copia para ellos es innecesario.

En el caso del recolector GenCopy, experimentalmente nunca se ha dado el caso de que la memoria reservada para el espacio de maduros no sea suficiente para satisfacer

la necesidad de ampliación del espacio de reserva del *nursery*. En caso de que ello ocurriera se seguiría la misma actuación que se sigue con GenMS. Antes de ver la fase de recuperación vamos a enumerar los pasos que sigue la máquina virtual en una recolección *nursery*:

- El recolector califica el área de los objetos *nursery* como ”movible”. Lo cual significa que al visitar el objeto este será copiado al espacio de reserva.
- La máquina virtual prepara el *root set* en una cola. Los componentes del *root set* son los nodos iniciales del grafo de referencias a través del heap. Los objetos referenciados desde el *root set* a la generación *nursery* son almacenados en una cola de objetos grises. Los objetos grises son objetos vivos cuya descendencia aún no ha sido visitada. Las referencias a objetos no pertenecientes al espacio *nursery* son ignaradas.
- Se procesa la cola de objetos grises. Cada nueva referencia se mete en la cola como un nuevo objeto gris para ser estudiado posteriormente.
- Finalmente, se procesa el *remembered set*. El *remembered set* es la cola que guarda los punteros que referencian la región *nursery* desde otras regiones. Es el resultado de la actuación de las barreras de escritura.

Si durante este proceso el recolector encuentra que no tiene suficiente memoria en el espacio de reserva para llevar a cabo una copia se inicia la fase de recuperación, cuyas etapas son:

- El recolector califica el área de los objetos *nursery* como no ”movible”. De este modo, los objetos del *nursery* serán sólo visitados, no copiados.
- Se calcula de nuevo el *root set*, mientras que el *remembered set* se desecha.
- Al visitar los objetos apuntados por el *root set* nos encontramos con dos casos. El objeto referenciado pertenece al espacio *nursery* o a cualquiera de las otras regiones del heap. Si estamos en el primer caso, el objeto se guarda en una cola para objetos grises del *nursery* además de la ya conocida cola de objetos grises. Si el objeto referenciado no pertenece al *nursery* se guarda en la cola de objetos grises y se marca como vivo.

- En el siguiente paso se procesa la cola de objetos grises. De nuevo, si su descendencia pertenece al *nursery* se introduce en las dos colas de objetos grises, si no, sólo en la cola de normal de objetos grises.
- Tras procesar la cola de objetos grises ya estamos seguros de que los objetos marcados como vivos en el espacio de maduros y en el espacio para objetos grandes son los únicos que van a sobrevivir la recolección en estas regiones y por tanto podemos liberar la memoria que ocupan el resto de objetos.
- Tras liberar la memoria ya podemos continuar con la recolección en la generación *nursery*.
  - Primero, se califica de nuevo el área de los objetos *nursery* como ”movible”.
  - Por último se procesa la cola de objetos grises del *nursery*, ahora copiando los objetos visitados.

En la tabla 6.2 se muestra la cantidad de memoria necesaria tras producirse un fallo en la estimación del tamaño del espacio de reserva, junto con la cantidad de memoria liberada tras el proceso de recuperación anteriormente descrito. Para esta tabla se ha empleado el benchmark javac con tamaño de heap de 45MB porque es el peor caso registrado en nuestros experimentos. El recolector es GenMS. En esta tabla se observa que el porcentaje de error nunca es superior al 15%. Y más importante, también se puede comprobar que el espacio liberado es siempre al menos 4 veces superior al espacio extra necesitado por la generación *nursery*, siendo en promedio unas veinte veces superior. En el caso de GenCopy, siempre fue suficiente con utilizar la memoria reservada para el espacio de maduros. Pensemos que el espacio de reserva de maduros es siempre superior a la posible cantidad de memoria liberada en éste.

6.4 Modificación del espacio de reserva en la generación madura

La modificación del tamaño del espacio de reserva de la generación *nursery* es una estrategia aplicable a los dos recolectores generacionales estudiados (GenMS y GenCopy). Es decir, es una técnica independiente de la política con la que el recolector gestiona la generación madura. Ahora bien, en el caso de que la generación

	Estrategia	
Benchmark	Agresiva	Conservadora
jess	0	0
jack	0	0
javac	12	2
raytrace	6	2
mtrt	6	1

Tabla 6.1: Sumario del número de fallos de predicción para cada estrategia con GenMS

Fallo número	error %	mem. necesaria (MB)	mem. liberada (MB)
1	8	2	8
2	2	0.5	6
3	12	0.5	30
4	3	0.5	18
5	12	1	19
6	4	0.5	18
7	12	1	19
8	4	0.5	18
9	12	1	19
10	4	0.5	18
11	12	1	18
12	4	0.5	18

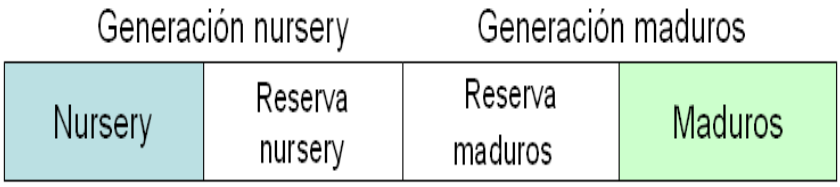
Tabla 6.2: Sumario de la cantidad de memoria necesitada tras una fallo en la predicción y la cantidad de memoria liberada en el espacio de maduros para javac con GenMS.

madura también se rija por el mecanismo de copia (GenCopy) el problema del espacio desaprovechado se ve incrementado por el espacio de reserva de la generación madura, figura 6.7(a). De hecho, para este recolector, y a medida que transcurren las recolecciones, la mayor parte del espacio de reserva pertenece a la generación madura. En la figura 6.7(b) podemos ver la distribución del heap con nuestra propuesta adaptativa. El espacio de reserva de la generación *nursery* ha disminuído pero seguimos teniendo un espacio de reserva de la generación madura equivalente a su espacio de asignación. En la figura 6.7(c) se muestra el caso óptimo con el que se consigue el mayor tamaño del espacio de asignación de la generación *nursery*. En este caso se ha reducido tanto el espacio de reserva de la generación *nursery* como el de la generación madura. Está claro que para sacar el máximo provecho de la filosofía de nuestra propuesta, en el caso de GenCopy, sería interesante ampliar su ámbito de aplicación a la generación madura. Sin embargo, la generación madura presenta características especiales que tenemos que tener en cuenta:

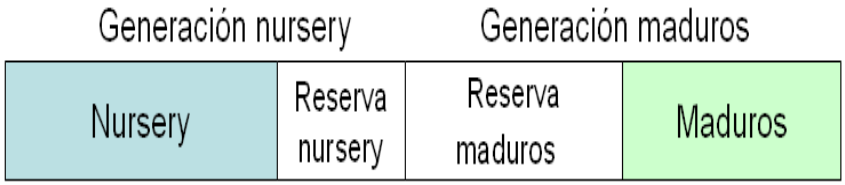
- El objetivo de la estrategia generacional es que el número de recolecciones que afectan a la generación madura (globales) sea muy inferior al número de recolecciones centradas en la generación *nursery* (menores). En promedio, en nuestros experimentos, encontramos casi doscientas recolecciones menores por cada recolección mayor. De modo que en una ejecución normal, la máquina virtual no va a disponer de suficientes recolecciones en la generación madura para poder aplicar algún criterio dinámico que modifique el tamaño del espacio de reserva basándose en el comportamiento registrado de la aplicación. Así que la extensión de nuestra idea al espacio de reserva de la generación madura tiene que aplicarse estáticamente, ya sea realizando un profiling off-line de cada benchmark o estableciendo un tamaño fijo general para todos los programas.

- En cualquiera de los dos casos anteriores tenemos que tener en cuenta el problema de la recuperación tras una estimación incorrecta. El mecanismo de recuperación de nuestra estrategia en la generación *nursery* se apoyaba en tres factores:

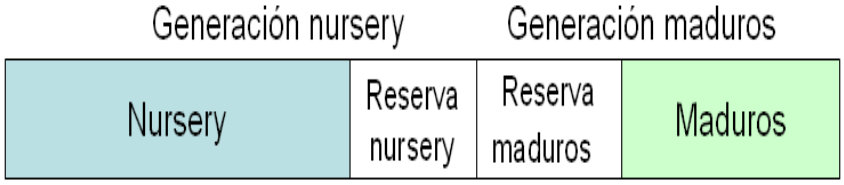
  - En el caso de GenCopy, si no disponíamos de suficiente memoria reservada en la generación *nursery* contábamos con el espacio reservado en la generación madura. Experimentalmente siempre resultó suficiente.



(a) *Appel*: el espacio de reserva equivale al espacio de asignación, para las dos generaciones



(b) Adaptativo I: se reduce el tamaño del espacio de reserva de la generación *Nursery*



(c) Adaptativo II: se reduce el tamaño del espacio de reserva de las dos generaciones

Figura 6.7: Funcionamiento de las distintas opciones para GenCopy

  - Para ambos recolectores generacionales, siempre podíamos liberar memoria muerta en el espacio de asignación de maduros. Memoria que siempre resultó cuatro o más veces superior a la necesitada para GenMS.
  - Al realizar nuestro mecanismo de recuperación, eliminamos los objetos supervivientes debidos al nepotismo (objetos en la generación *nursery* referenciados por objetos muertos de la generación madura).

Sin embargo, en el caso de la generación madura no contamos con ninguno de esos tres factores: no contamos con más memoria reservada, no podemos liberar memoria en una generación más madura (con excepción del espacio para objetos grandes), ni podemos contar con objetos falsamente vivos debido al nepotismo. Por ello, la elección de una nueva relación entre las mitades de la generación madura debe ser muy conservadora. Aún así no hemos querido renunciar al estudio de las posibilidades de aplicar nuestra filosofía a la generación madura.

En la tabla 6.3 se ha resumido el porcentaje de supervivientes a las recolecciones globales en el espacio de maduros y el espacio para objetos grandes. En la tabla se pueden observar tres grupos de comportamiento.

- El benchmark compress que no sólo asigna un número muy elevado de objetos grandes sino que además estos objetos grandes prácticamente son inmortales en su mayor parte. En la tabla podemos ver que el comportamiento de los objetos maduros es muy parecido y esto provoca el mayor ratio de supervivientes de todos los benchmarks. Estos hechos nos indican que este programa no recibiría ningún beneficio de aplicar nuestra estrategia. Por otro lado, este benchmark nunca se incluye en los estudios de gestión automática de memoria dinámica ya que no es un benchmark realmente orientado a objetos.

- Las aplicaciones con un número pequeño de recolecciones globales y con ratios de supervivientes bajos tanto en la generación madura como en los objetos grandes. Estos benchmarks pueden obtener grandes beneficios si se aplica una reducción grande del espacio de reserva. En el extremo óptimo se encuentra mpegaudio, que al tener un número muy pequeño de objetos asignados, no necesita de ninguna recolección global y en el que podemos rebajar drásticamente el espacio de reserva (aunque recordemos que este programa no es un ejemplo típico de programación orientada a objetos).

		Ratio %	
Benchmark	Número recolecciones	maduros	LOS
compress	10	90	85
db	1	25	50
mpeg	0	-	-
jess	2	35	0
jack	4	40	65
javac	5	45	40
raytrace	26	60	100
mtrt	24	70	70

Tabla 6.3: Sumario del porcentaje de supervivientes en la generación madura para el recolector GenCopy

- El último grupo lo compondrían los programas con un número grande tanto de recolecciones como de ratios de supervivientes. En nuestros resultados experimentales, estas aplicaciones también obtienen beneficios importantes al reducir el tamaño de reserva de la generación madura. Pero esta reducción ha de limitarse cuidadosamente para no incurrir en el error fatal de tener una cantidad menor de memoria reservada que la cantidad de memoria necesaria.

Es interesante señalar aquí el trabajo de McGachey et al [McG05] que como ampliación a nuestra propuesta han desarrollado un mecanismo de compactación en la generación madura. Este mecanismo de compactación permite fijar tamaños del espacio de reserva en la generación madura más agresivos. En caso de que que nuestro mecanismo de recuperación falle, la inclusión del compactador de McGachey soluciona la situación. Su trabajo es una gran aportación al nuestro desde el punto de vista formal, ya que garantiza un ámbito de aplicación de nuestra propuesta mucho mayor. Sin embargo, según nuestros experimentos, desde el punto de vista del rendimiento, el beneficio de tener reducciones más agresivas del espacio de reserva es muy limitado y se ve contrarestado por el elevado coste que supone la ejecución del algoritmo de compactación en caso de fallo. Por ello, nosotros creemos que la solución idónea es tener tamaños del espacio de reserva más conservadores e utilizar el compactador como un salvavidas para situaciones excepcionales.

6.5 Resultados experimentales

Uno de los principales factores a la hora de comprobar el comportamiento de un recolector generacional es la cantidad de memoria copiada desde la generación *nursery* a la generación madura. El movimiento de objetos a lo largo del heap es muy costoso (en términos de tiempo, instrucciones, accesos, etc), y la filosofía generacional supone una mejora en este sentido frente a la política de copia pura. En la figura 6.8 se muestra la cantidad total de memoria copiada de la generación *nursery* a la madura para el recolector GenMS. El tamaño del heap se ha variado desde 1.5 veces el tamaño mínimo necesario para la ejecución con el recolector *Appel* hasta 4.5 veces este mínimo. No se han incluido tamaños mayores de heap porque en ellos el rendimiento de los recolectores generacionales decrece frente al recolector de copia puro. El tamaño de los datos es el mayor posible (s100). A la vista de estas gráficas obtenemos las siguientes conclusiones:

- El aumento del tamaño del heap implica también un aumento del tamaño del espacio de asignación de la generación *nursery*. Esto significa que los objetos recién creados tienen más tiempo para completar su ciclo de vida (en términos de memoria asignada) antes de que se haga necesaria la recolección. Por ello, el porcentaje de memoria copiada para el recolector Appel disminuye claramente con el aumento del tamaño del heap.

- Las estrategias adaptativas obtienen siempre mejores resultados que el recolector clásico. La única excepción es para raytrace y tamaño del heap 3.5 veces el tamaño mínimo. En ese caso la estrategia adaptativa conservadora produce una recolección global más que el recolector Appel y eso repercute en un ligero aumento de la memoria copiada total. En los demás casos observamos siempre una disminución del porcentaje de la memoria copiada. En la figura 6.8(f) se resume la reducción en el total de memoria copiada y podemos ver que el promedio oscila entre un 19% y un 40%.

- Para jess, jack y javac (figuras 6.8(a), 6.8(b) y 6.8(c)) el comportamiento de las estrategias es muy similar, apenas observándose diferencias. Para raytrace y mtrt (figuras 6.8(d) y 6.8(e)) y tamaños menores de heap, los resultados del algoritmo ”promedio” son mejores, lo cual lo hace más idóneo para entornos con limitaciones de memoria. Las diferencias apreciadas en estas aplicaciones entre

los dos algoritmos se deben fundamentalmente al número de recolecciones globales producidas por fallos de estimación del tamaño necesario del espacio de reserva. Pensemos, que al tener un comportamiento tan errático la estrategia conservadora comete un considerable número de errores de estimación.

Otro factor importante a tener en cuenta a la hora de analizar el comportamiento de los recolectores generacionales es el número total de recolecciones producidas durante la ejecución. Pensemos que hay un coste inherente al cambio de mutador a recolector y viceversa. El recolector debe esperar a que todos los hilos en la ejecución de la máquina virtual se hayan parado en punto seguro. El contexto de cada hilo ha de guardarse y posteriormente, al acabar la recolección, recuperarse para volver a lanzar la ejecución. Este coste inevitable al lanzar la recolección sólo puede evitarse reduciendo el número de recolecciones. En la figura 6.9 se muestra el total de recolecciones para el recolector Appel clásico y nuestras dos estrategias adaptativas. Cotejando estas gráficas podemos concluir que:

- Se observa que la reducción en el número de recolecciones es muy significativa. La mayor reducción se da para los tamaños de heap menores, lo cual hace nuestra propuesta muy interesante para los entornos con limitaciones de memoria como los sistemas empotrados. En la figura 6.9(f) se resumen los porcentajes de reducción. Podemos ver que en promedio la reducción varía siempre entre valores ligeramente superiores al 40% y ligereamente inferiores al 70%. Para el tamaño más pequeño de heap tenemos una reducción del 60%. Y para algunos benchmarks encontramos reducciones superiores al 80%.

- Vimos antes que las diferencias entre los algoritmos de modificación del tamaño del espacio de reserva, a nivel de memoria total copiada, eran muy pequeñas (pero en algunos casos apreciables). En cuanto al número de recolecciones las diferencias entre algoritmos no son significativas.

- Podemos ver unos extraños picos en las gráficas del recolector clásico Appel, para los benchmarks raytrace y mtrt, que no tienen una contrapartida en las líneas de los algoritmos adaptativos. Estos picos son debidos a un gran número de recolecciones menores sin supervivientes con un tamaño de *nursery* ligeramente superior al umbral que dispara una recolección global. Este problema no se produce con nuestras

estrategias ya que el tamaño umbral de la generación *nursery* que dispara una recolección global es modificado dinámicamente.

- Por defecto, el código de la aplicación javac produce un número grande de recolecciones de basura (casi 50) que no son debidas al agotamiento del espacio de asignación. Con un tamaño de heap 4.5 veces el mínimo necesario, el recolector clásico no necesita más recolecciones que las requeridas por javac. Con nuestras propuestas adaptativas ese punto se alcanza antes, con un tamaño 3 veces superior al mínimo.

Ya que el registro de los factores citados supone un pequeño coste extra en el tiempo final de ejecución, se utilizaron experimentos independientes para las medidas de estas métricas y las medidas de tiempo. En la figura 6.10 se muestra el tiempo total de recolección, es decir, la suma de los tiempos de pausa producidos en cada una de las recolecciones. Puesto que nuestros algoritmos adaptativos consiguen reducir no sólo el número de recolecciones, sino también el tamaño total de la memoria copiada, no son de extrañar las claras reducciones que se aprecian en las gráficas. En la figura 6.10(f) podemos ver el resumen del porcentaje de reducción en el tiempo total de recolección. Esta reducción, en promedio, oscila entre un 17% y un 43%. Si comparamos estas gráficas con las gráficas de las métricas anteriores podemos observar dos grupos de comportamiento:

- En un grupo estarían los benchmarks con un total de memoria copiada pequeño en los que la reducción del tiempo de recolección se debe principalmente a la reducción en el número total de recolecciones. El más claro ejemplo es jess. Podemos observar la analogía entre las gráficas de reducción del número de recolecciones y de reducción del tiempo total de recolección.

- En otro grupo estarían las aplicaciones con tamaño total de memoria copiada alto, en cuyo caso la reducción del tiempo total de recolección viene liderada por la reducción en el porcentaje de memoria copiada. Como ejemplo podemos poner a raytrace.

La influencia de los recolectores generacionales en el tiempo global de ejecución va más allá del tiempo total de recolección y el número de recolecciones, pues tenemos que incluir el efecto del asignador de memoria, el coste de las barreras de escritura intergeneracionales y la localidad de los datos. En la figura 6.11 se muestra el porcentaje

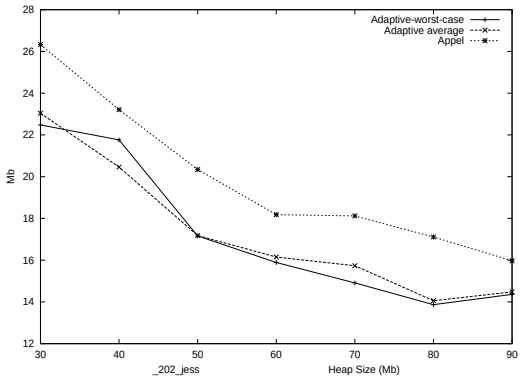
	Num. col.		Fallos pág.	
Tipo Recolector	totales	globales	mutador	recolección
GenMS	64	3.3	50882	357013
AdapMS	29	2.6	50012	223148
GenCopy	100	8.1	6579	457987
AdapCopy1	55	5.7	6314	283951
AdapCopy2	34	2.9	6178	242733

Tabla 6.4: Número de recolecciones y fallos de página en promedio para el subconjunto de benchmarks con mayor memoria asignada y un heap de 16MB. Los recolectores utilizados son los generacionales y sus respectivas versiones adaptativas.

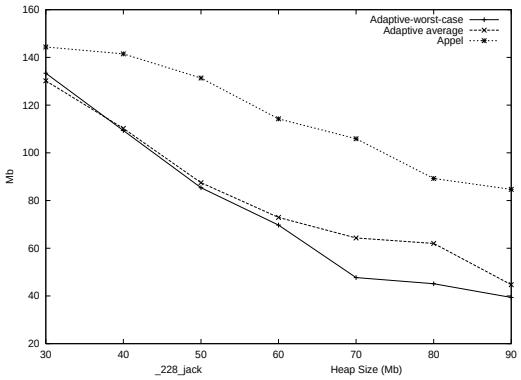
de reducción en el tiempo global de ejecución como resultado final de los distintos factores ya mencionados. En estas figuras se aprecia que la modificación dinámica del espacio de reserva de la generación *nursery* produce una reducción del tiempo global de ejecución. En la figura 6.11(f) podemos ver que, en promedio para los distintos tamaños de heap, la reducción varía entre un 2% y un 7%.

Podemos concluir, por tanto, que la modificación dinámica del espacio de reserva de la generación *nursery* produce una reducción en el número de recolecciones, la cantidad de memoria copiada, el tiempo global de recolección y, finalmente, en el tiempo total de ejecución. Además, nuestra propuesta está especialmente indicada en entornos con limitaciones de memoria. De hecho, podríamos interpretar nuestros resultados experimentales en el sentido de que con nuestra propuesta, la máquina virtual se comporta como si tuviera un mayor espacio de memoria disponible. Por ello, nuestro siguiente grupo de experimentos está enfocado al estudio de los algoritmos adaptativos dentro de una típica jerarquía de memoria propia de sistemas empotrados, al igual que los resultados del capítulo 4.

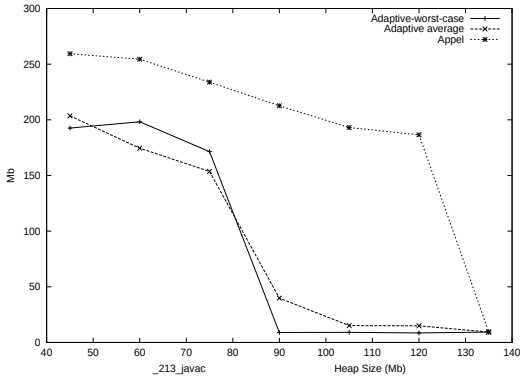
En la tabla 6.4 se muestra el número total de recolecciones y número de recolecciones globales junto con el número de fallos de página para los distintos recolectores generacionales tras obtener el promedio para los cinco benchmarks con mayor cantidad de memoria dinámica producida. AdapMS y AdapCopy1, son las versiones adaptativas de GenMS y GenCopy, cuando se modifica dinámicamente el tamaño del espacio de reserva de la generación *nursery*. El algoritmo usado es el conservador. AdapCopy2 modifica, además, de forma estática el tamaño del espacio de reserva de la generación madura. En



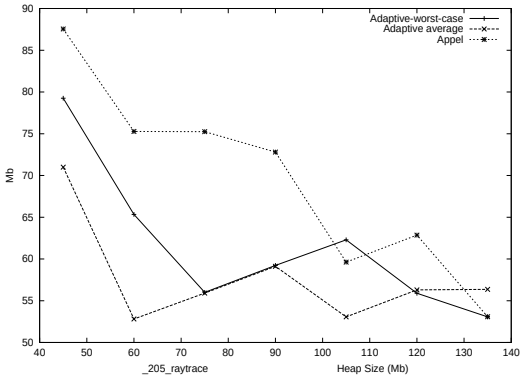
(a)



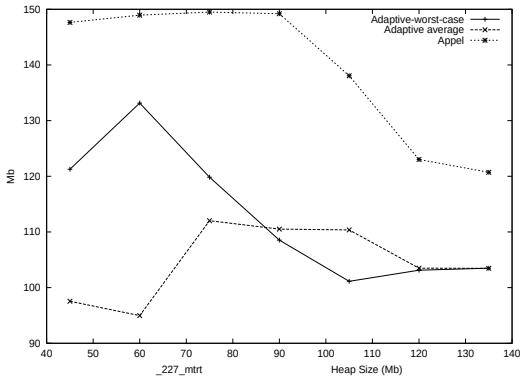
(b)



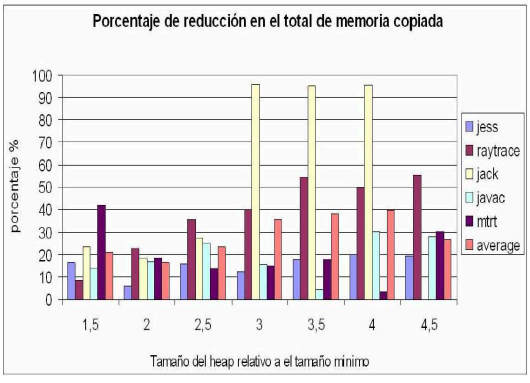
(c)



(d)

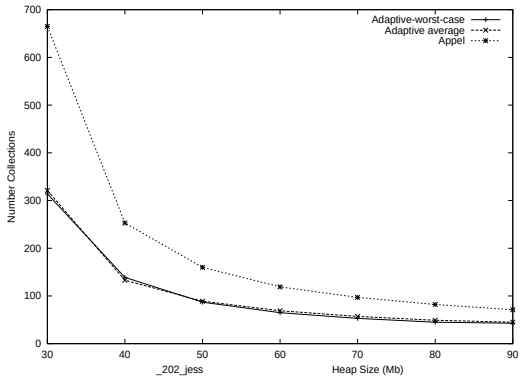


(e)

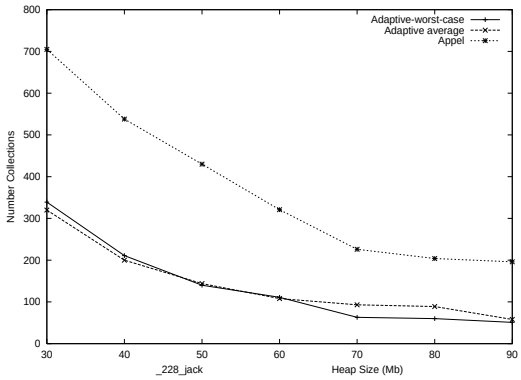


(f)

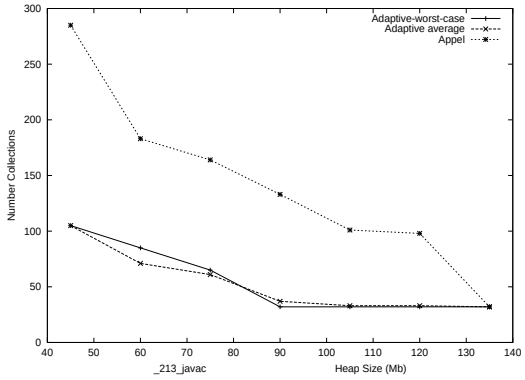
Figura 6.8: Cantidad total de memoria copiada por el recolector



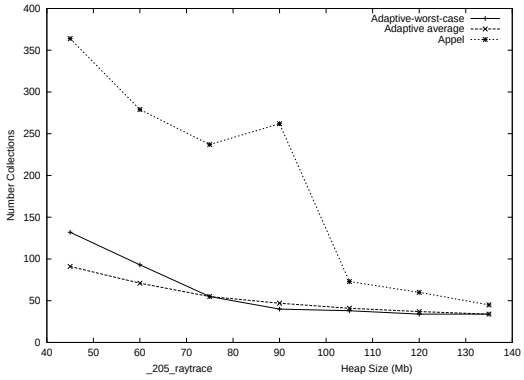
(a)



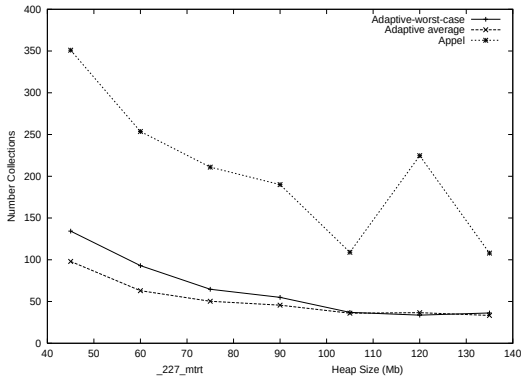
(b)



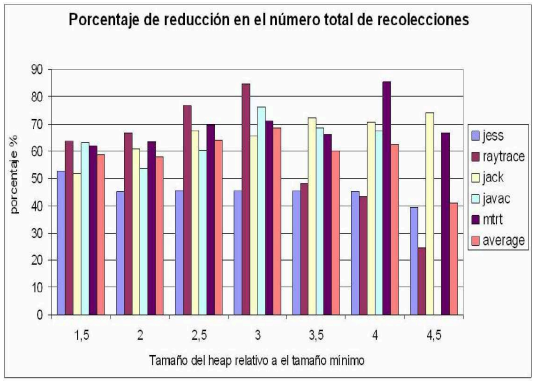
(c)



(d)

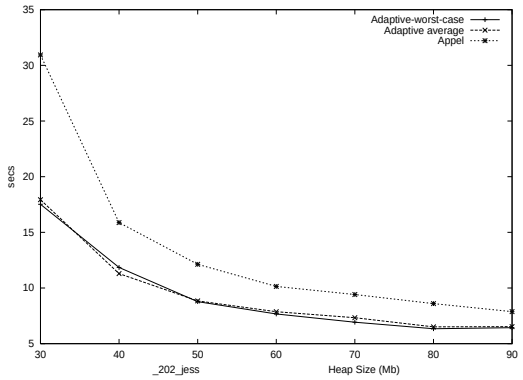


(e)

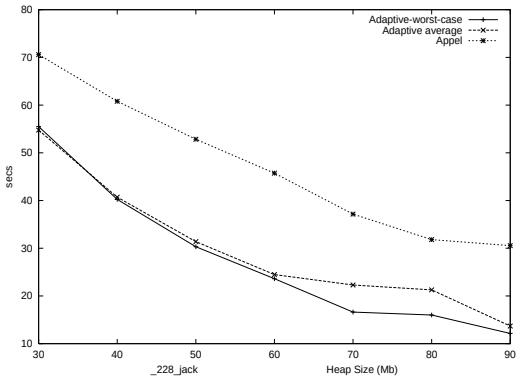


(f)

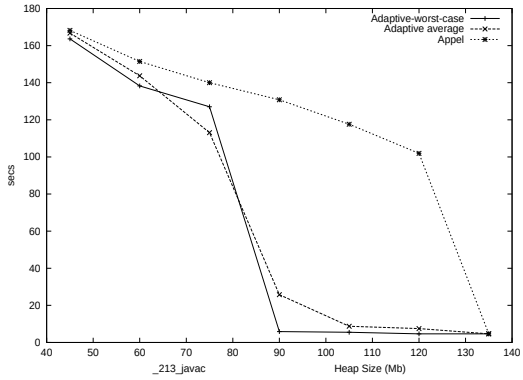
Figura 6.9: Número total de recolecciones



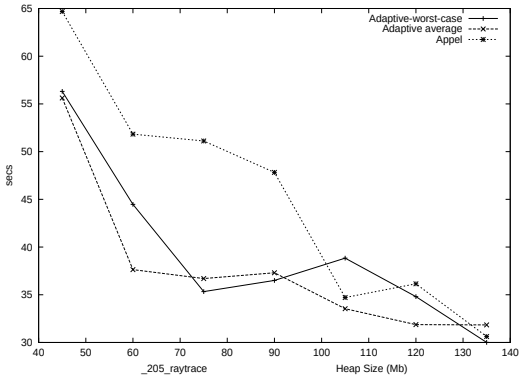
(a)



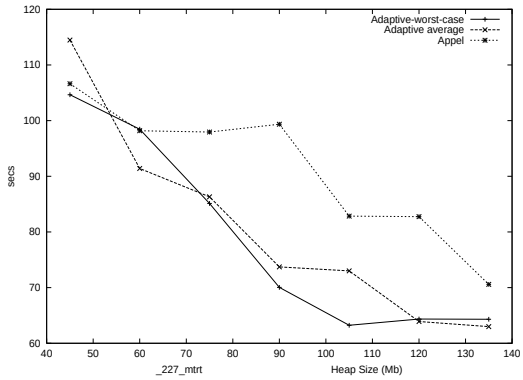
(b)



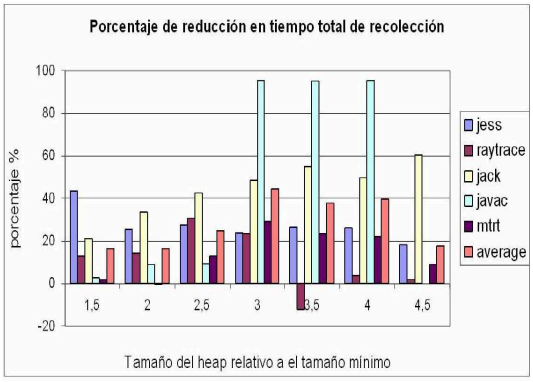
(c)



(d)

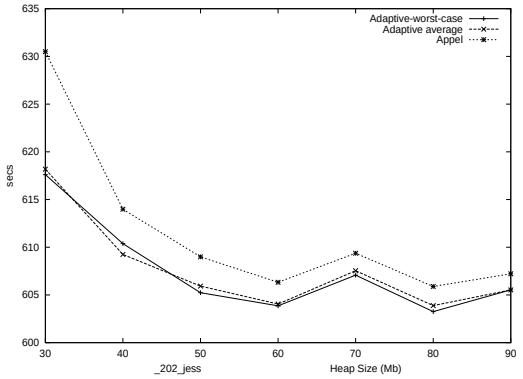


(e)

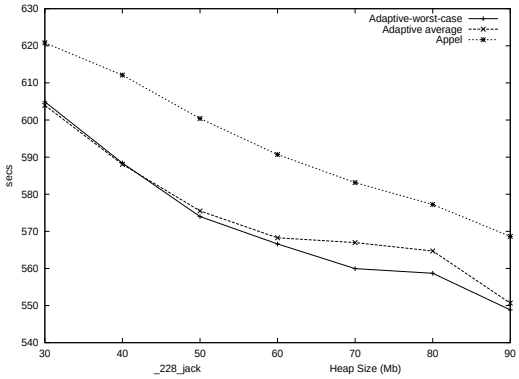


(f)

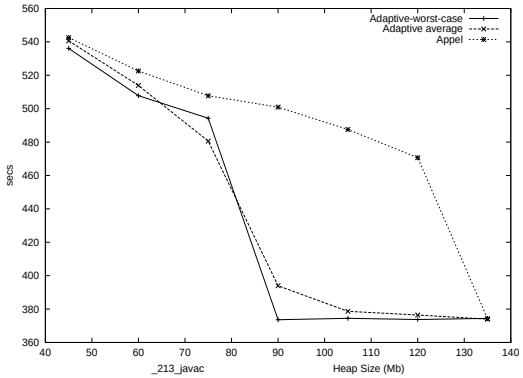
Figura 6.10: Tiempo total de recolección



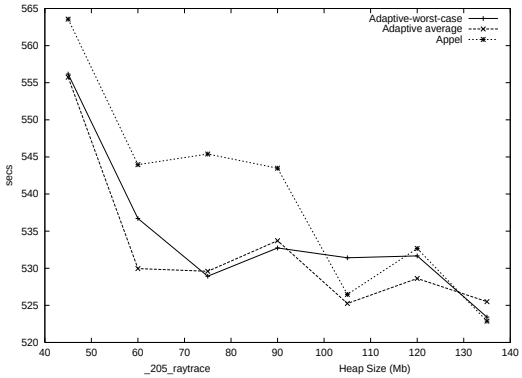
(a)



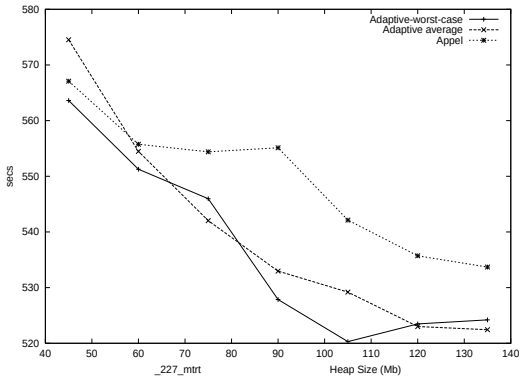
(b)



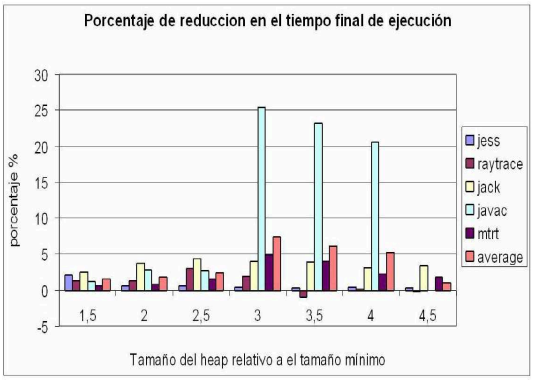
(c)



(d)



(e)



(f)

Figura 6.11: Tiempo total de ejecución

la tabla se aprecia que, como resultado de la disminución del número de recolecciones, las estrategias AdapMS y AdapCopy1 producen una reducción de los fallos de página durante la recolección por encima del 30%. En el caso de la estrategia AdapCopy2, la reducción supera el 40%. El menor número de recolecciones produce una menor interferencia a nivel de la jerarquíade memoria entre los contextos de ejecución del mutador y el recolector. Por ello, en la tabla, también podemos apreciar una ligera reducción en los fallos de página durante la fase del mutador para las versiones adaptativas.

En la figura 6.16 se resumen los porcentajes de reducción relativos al recolector generacional GenCopy de los distintos recolectores adaptativos y de GenMS. El tamaño del primer nivel de cache es 32K con asociatividad directa, pero como se aprecia en las gráficas 6.12 a 6.15, la reducción es similar para otros tamaños y asociatividades. Podemos ver que el recolector que obtiene los mejores porcentajes de reducción es el AdapMS, es decir, el recolector generacional adaptativo con política de marcado y barrido en la generación madura. Este recolector consigue una reducción en la energía total consumida por encima del 20% y una reducción en el número total de ciclos que supera el 30%. El recolector AdapCopy2, recolector generacional de copia puro que implementa la reducción del espacio de reserva tanto en la generación *nursery* como en la madura, obtiene reducciones próximas (tan solo un 2-3% por debajo) a AdapMS. El recolector adapCopy1, que sólo implementa la modificación dinámica del tamaño del espacio de reserva de la generación *nursery* consigue reducir la distancia que separa a GenMS de GenCopy en los sistemas con poca memoria disponible. Así, AdapCopy1 obtiene un resultado ligeramente inferior a GenMS en consumo de energía, pero logra una reducción en el número de ciclos superior a éste.

Las reducciones que las versiones adaptativas consiguen tanto en número de ciclos como en consumo de energía producen una nueva curva de Pareto dentro del espacio de diseño definido en el capítulo 4 (figura 4.8) que podemos ver en la figura 6.17. En esta figura se muestra el número de ciclos acumulados durante la recolección frente al consumo energético de ésta. En azul oscuro se muestran los puntos antiguos de la curva, correspondientes al recolector generacional clásico. En color rosa podemos ver los nuevos puntos correspondientes a las versiones adaptativas. Con un rombo se muestra la configuración que corresponde a un tamaño del primer nivel cache de 32K y asociatividad directa, y con un triángulo la configuración de igual tamaño y asociatividad 4 vías. Podemos ver que la reducción en número de ciclos conseguida por las dos versiones adaptativas

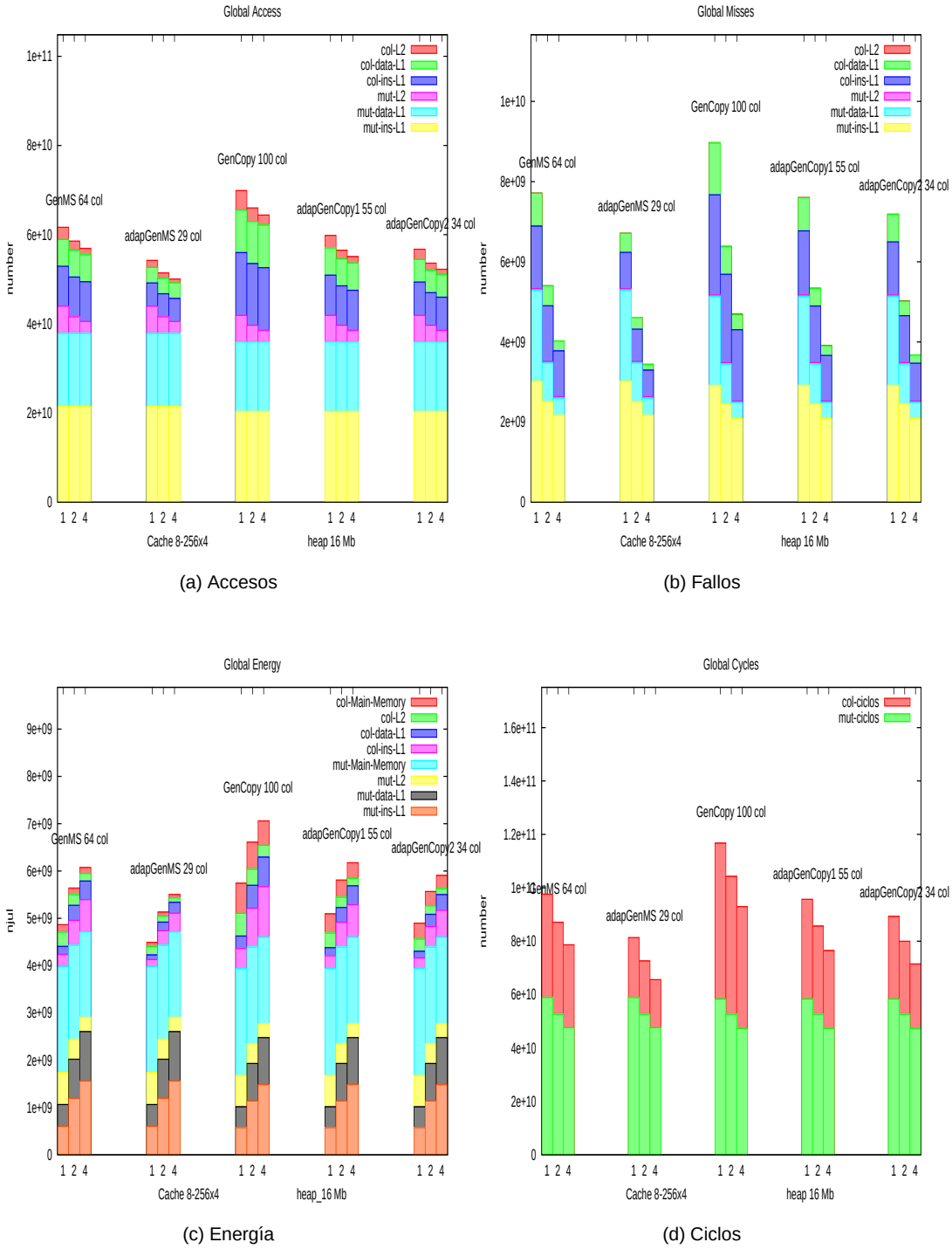
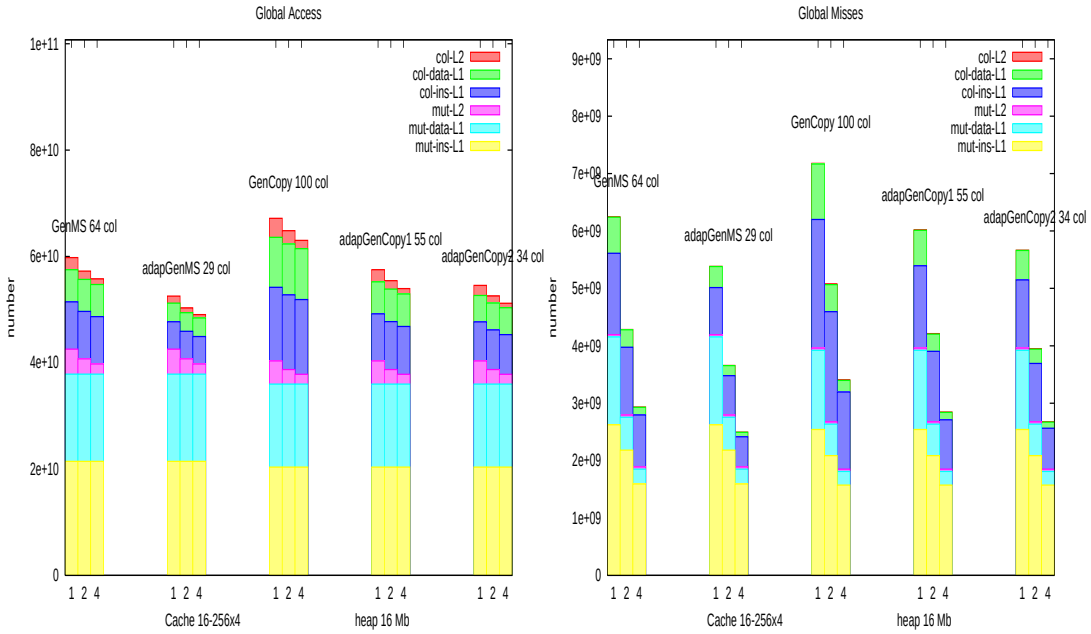


Figura 6.12: Comparación entre las diferentes asociatividades. Tamaño del primer nivel de cache es 8K. Promedio para el subconjunto de benchmarks representativos.



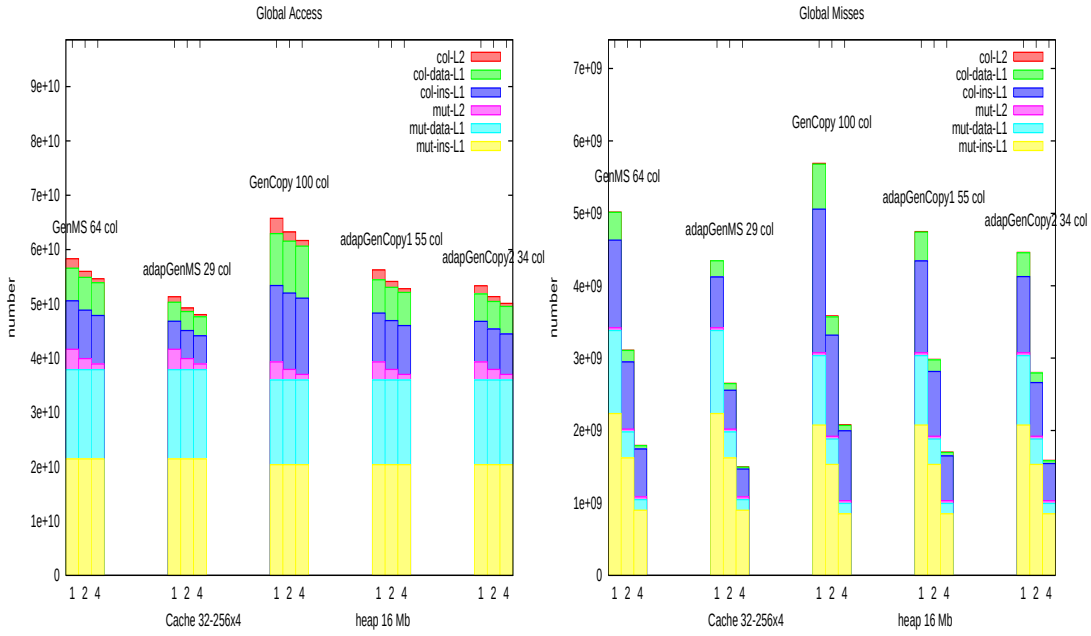
(a) Accesos

(b) Fallos

(c) Energía

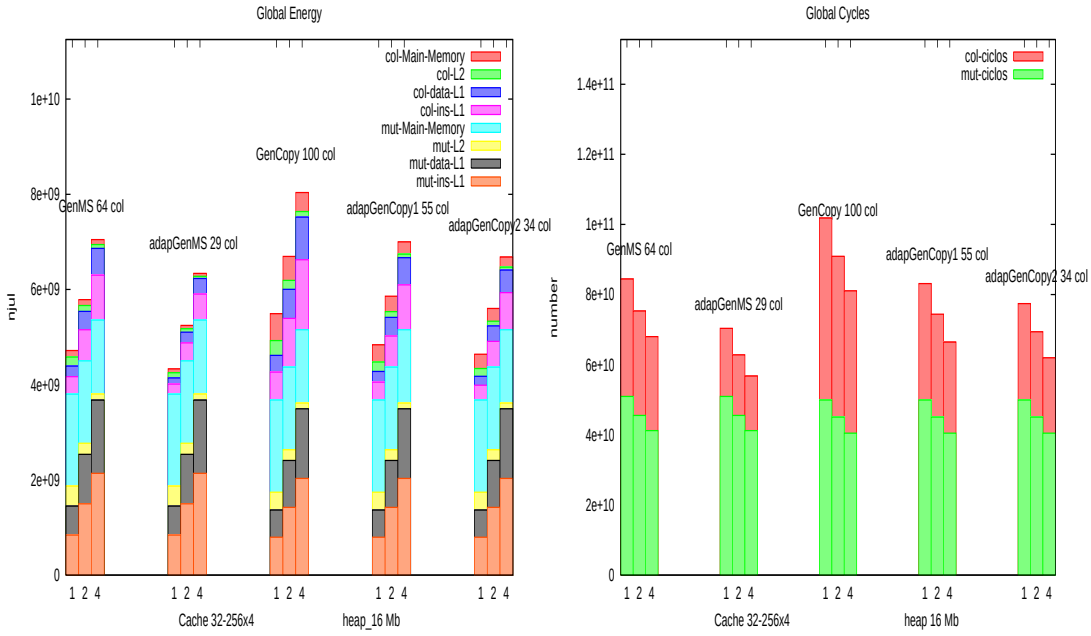
(d) Ciclos

Figura 6.13: Comparación entre las diferentes asociatividades. Tamaño del primer nivel de cache es 16K. Promedio para el subconjunto de benchmarks representativos.



(a) Accesos

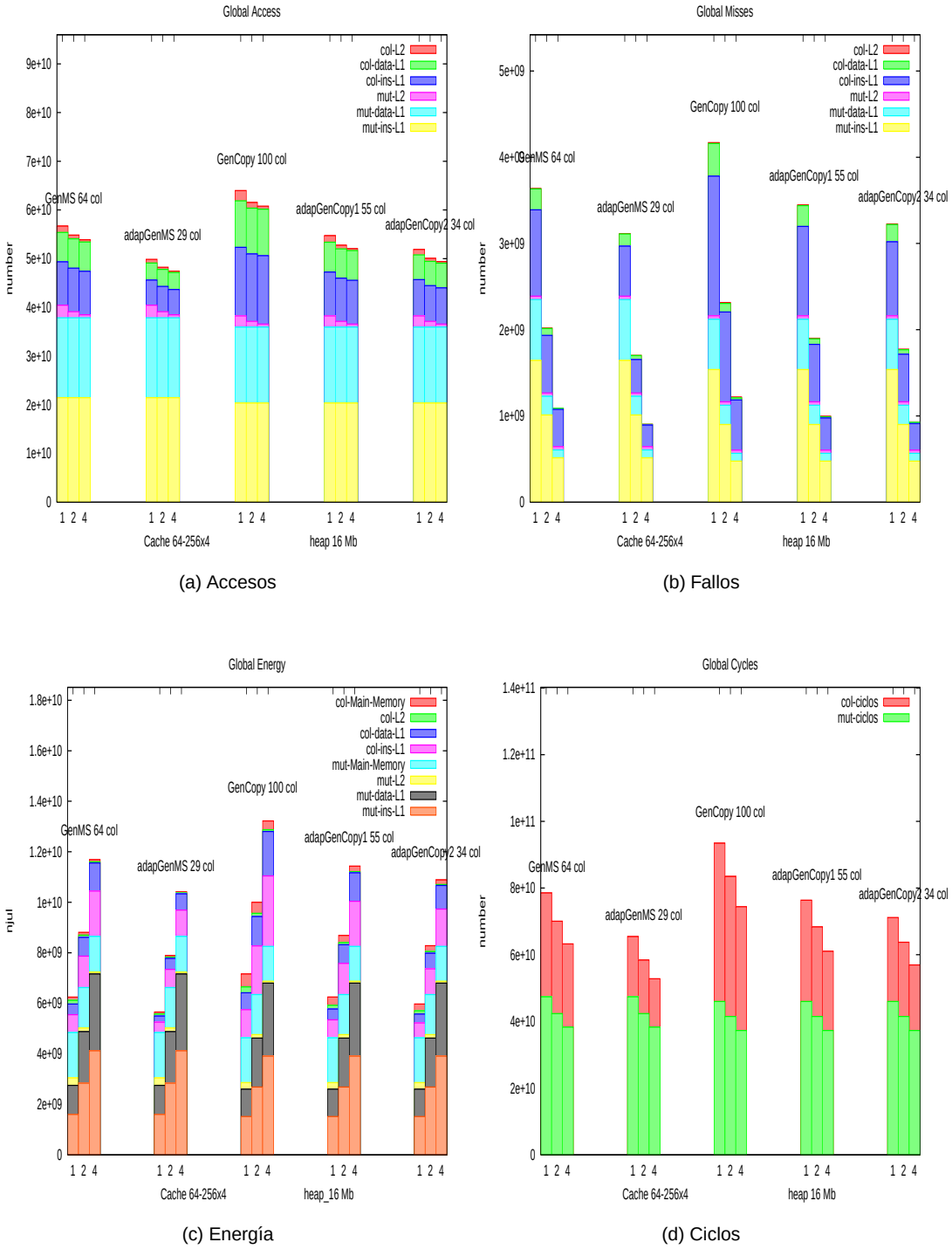
(b) Fallos



(c) Energía

(d) Ciclos

Figura 6.14: Comparación entre las diferentes asociatividades. Tamaño del primer nivel de cache es 32K. Promedio para el subconjunto de benchmarks representativos.



(a) Accesos

(b) Fallos

(c) Energía

(d) Ciclos

Figura 6.15: Comparación entre las diferentes asociatividades. Tamaño del primer nivel de cache es 64K. Promedio para el subconjunto de benchmarks representativos.

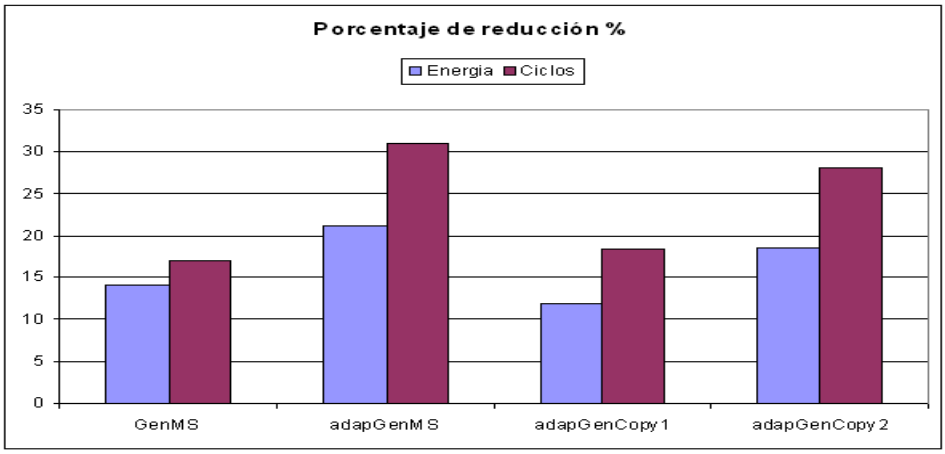


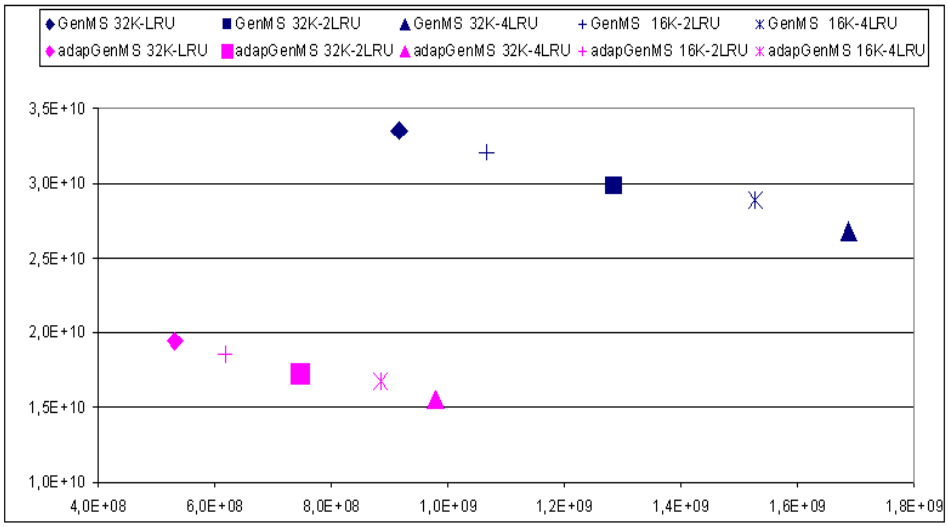
Figura 6.16: Porcentajes de reducción en número de ciclos y consumo energético para los recolectores generacionales adaptativos relativos al recolector GenCopy

hace que el punto correspondiente a asociatividad directa (rombo rosa) esté por debajo del punto de asociatividad 4 vías (triángulo azul) de las versiones clásicas. Del mismo modo, la reducción en el consumo de energía conseguida por AdapCopy2 hace que el punto correspondiente a asociatividad 4 vías y tamaño 32K (triángulo rosa) se sitúe a la izquierda del punto de asociatividad directa (rombo azul) de GenCopy. En el caso de AdapMS, los dos puntos están muy próximos, situándose el punto correspondiente a la versión adaptativa ligeramente a la derecha. La reducción conseguida por AdapMS respecto de GenMS es superior al 40% tanto para ciclos como para consumo para las distintas configuraciones con tamaño de cache de 32K. En el caso de AdapCopy2, los porcentajes de reducción son superiores a 45%.

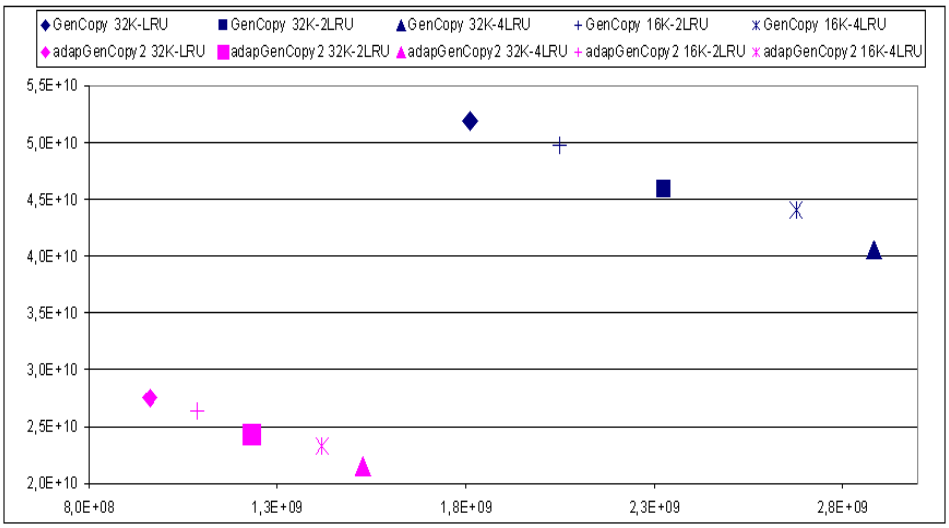
Por otro lado, en la figura podemos apreciar que la reducción en el número de ciclos conseguida por las versiones adaptativas es tan grande que ya apenas deja margen para la mejora producida por el aumento de asociatividad y tamaño del primer nivel cache. Aunque lo mismo puede decirse en cuanto al aumento de consumo energético. Por ello, los puntos en la nueva curva de Pareto están mucho más juntos que en la curva antigua.

Finalmente, hemos aplicado a los recolectores adaptativos las técnicas de optimización presentadas en el capítulo 5. Las reducciones conseguidas, tanto en número de ciclos como en consumo de energía, son menores que las conseguidas sobre las versiones clásicas de los recolectores generacionales. Ello es debido al menor margen de mejora que tienen nuestros recolectores adaptativos. Las nuevas curvas de Pareto se muestran en

la figura 6.18. La acción conjunta de las técnicas presentadas en estos dos capítulos, sobre el punto de la curva de Pareto con menor consumo (cache de 32K y asociatividad directa), producen una reducción final superior al 60% en el número de ciclos y por encima del 70% del consumo de energía durante la recolección para las dos versiones adaptativas de los recolectores generacionales. Esta reducción durante la fase de recolector se traduce, para GenCopy, en una reducción global (mutador y recolector) superior al 30% del total de ciclos de ejecución y del 19% del consumo de energía. En el caso de GenMS, las reducciones son de un 24% del número total de ciclos y de un 11% del consumo total de energía. En el caso del punto de la curva de Pareto con menor número de ciclos(cache de 32K y asociatividad 4 vías), las reducciones finales son aproximadamente del 60% tanto en ciclos como en consumo. Estas reducciones dentro de la fase de recolector se traducen, dentro de las cifras globales y para GenCopy, en el 30% del número de ciclos y 21% del consumo. Para el recolector GenMS son del 23% del total de ciclos y del 14% del consumo.

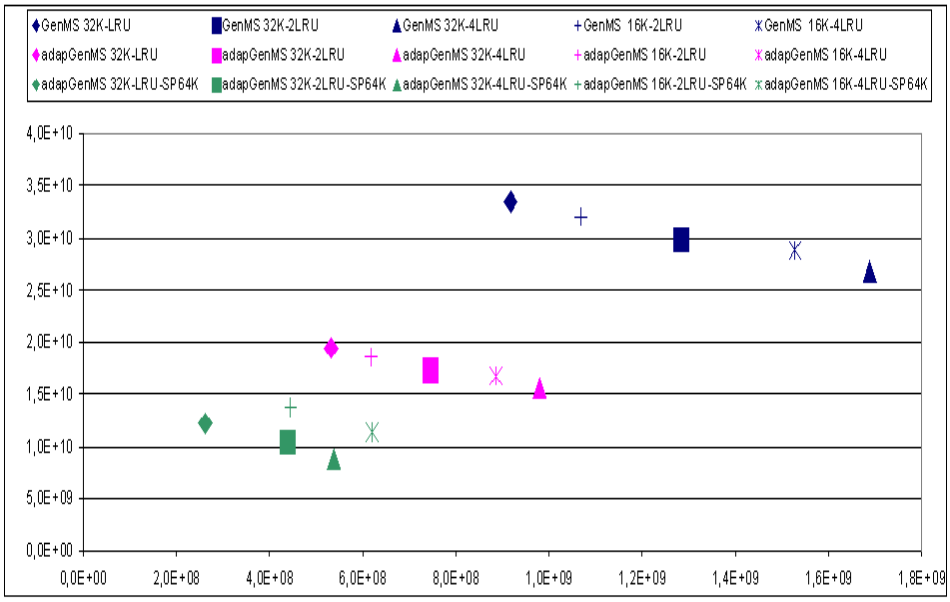


(a) GenMS

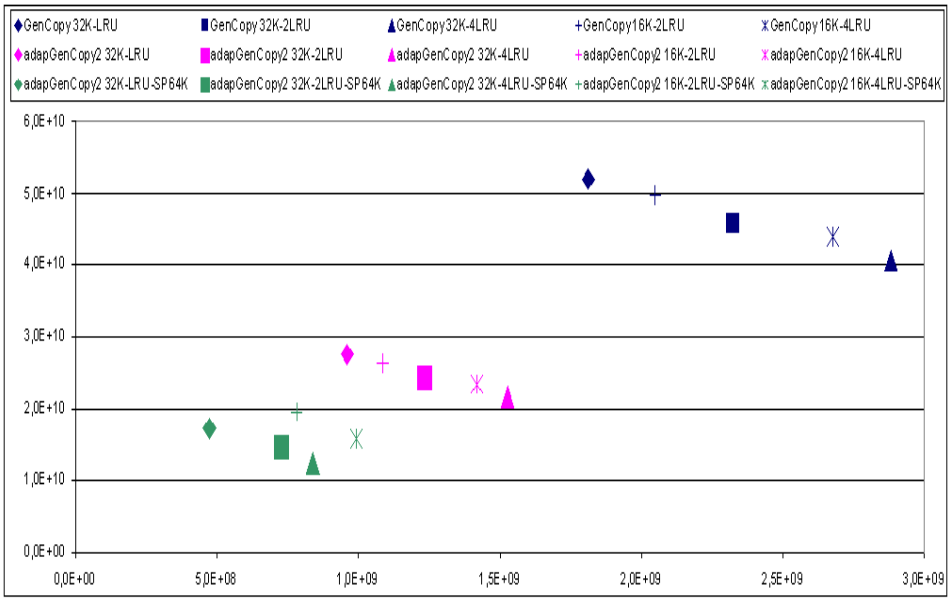


(b) GenCopy

Figura 6.17: Nuevas curvas de Pareto para los recolectores adaptativos. En el eje de abcisas se muestra el consumo energético (nanojulios). En el eje de ordenas el número de ciclos.



(a) GenMS



(b) GenCopy

Figura 6.18: Nuevas curvas de Pareto tras aplicar a los recolectores adaptativos las técnicas del capítulo 5. En el eje de abcisas se muestra el consumo energético (nanojulios). En el eje de ordenas el número de ciclos.

6.6 Conclusiones

En este capítulo hemos presentado una optimización a nivel algorítmico de la estrategia generacional con tamaño de *nursery* flexible. Nuestro recolector generacional adaptativo modifica dinámicamente el tamaño del espacio de reserva basándose en información obtenida en tiempo de ejecución. De este modo la generación *nursery* dispone de un mayor espacio de asignación. Con esta estrategia se consiguen reducir el número de recolecciones y la cantidad de memoria copiada, lo cual provoca una reducción sensible en el número de ciclos y en el consumo de energía. En nuestros resultados experimentales hemos comprobado que esta técnica consigue mejoras para distintos tamaños de memoria principal, pero es especialmente apropiada para plataformas con restricciones de memoria como los sistemas empotrados. La modificación del espacio de reserva permite a la máquina virtual funcionar como si dispusiese de un mayor tamaño de memoria principal.

Por otro lado, esta estrategia puede usarse paralelamente con las técnicas presentadas en el capítulo 5 consiguiendo reducciones superiores al 60% tanto en número de ciclos como en consumo de energía del recolector.

Capítulo 7

Extensión a sistemas distribuidos

7.1 Motivación

En los últimos años estamos asistiendo a un progresivo cambio en las prioridades de los fabricantes de ordenadores. La preocupación por el aumento de la frecuencia de reloj está dando paso a una búsqueda intensiva de nuevas arquitecturas multiprocesador construidas en un mismo circuito integrado [BDK+]. En el ámbito de los sistemas empotrados esta tendencia es muy agresiva, y hoy en día podemos encontrar sistemas en un rango que abarca desde los dos procesadores como el OMAP5910 de Texas Instruments [Prob], hasta cientos de cores como en el PC205 de Picochip [pAP]. Plataformas de gran éxito comercial como el IPhone 3GS de Apple [3GS] utilizan el multiprocesador ARM Cortex A5 [CA](evolución del Arm Cortex A8) así como el POWERVR SGX543MP [mpgIf] de Imagination Technologies, procesador gráfico con 16 núcleos. Por otro lado, recientes estudios en la universidad politécnica de Cataluña [Fer06] han demostrado que el rendimiento de las aplicaciones Java en un entorno multiprocesador puede mejorarse significativamente si se dota a la máquina virtual de Java con la capacidad para interarcionar con el sistema y gestionar los recursos disponibles. Ante esta situación el siguiente paso natural dentro de nuestra línea de investigación apunta a la gestión de memoria dinámica dentro de un sistema multiprocesador, y más específicamente a un sistema con memoria distribuida.

Ya hemos comentado que uno de los factores del éxito del lenguaje Java se debe a que la máquina virtual de Java ha liberado al programador de las tareas más arduas que encontramos en la creación del software como son la gestión de memoria dinámica o la compilación de ficheros ejecutables específicos para cada plataforma. En un entorno distribuido, estas tareas no hacen si no complicarse aún más, y el desarrollador de aplicaciones ve notablemente incrementados los quebraderos de cabeza que ya le acechan en un entorno monoprocesador. Por ello, sería muy deseable que la máquina virtual de Java fuera capaz de aprovechar los recursos de un sistema (asignando y reciclando memoria eficientemente) de un modo transparente al programador, sin incrementar la complejidad del lenguaje. De este modo, la ingenería del software podría separar completamente el desarrollo de una aplicación y la implementación específica para una plataforma concreta, incluyendo un sistema , figura 7.1. Sin embargo, la migración de una máquina virtual a un entorno distribuido conlleva serias dificultades. La gestión automática de memoria dinámica es una de las tareas más críticas en una máquina virtual monoprocesador y, como menciona Plainfosse [PS95], en un entorno distribuido la situación es aún más complicada. Añadamos a las dificultades ya vistas, inherentes al reciclaje de basura, el hecho de que hay que mantener actualizadas las referencias entre objetos que utilizan espacios de direcciones diferentes, y que esto hay que hacerlo de una forma escalable, eficiente y tolerante a fallos.

En la figura 7.2 podemos ver una situación habitual en la que los distintos objetos del grafo de relaciones de una aplicación están distribuidos en los nodos de un sistema de una forma poco eficiente. La red de comunicaciones del sistema tiene que soportar un elevado tráfico debido a dos factores:

- Los datos que los objetos necesitan comunicarse entre si.
- La actualización de las referencias globales entre objetos para que cada nodo pueda liberar la memoria una vez que los objetos estén muertos y nunca más vayan a ser accedidos.

Este capítulo se organiza de la siguiente forma: en la sección 7.1.1, haremos una descripción del trabajo relacionado previo. En la sección 7.2 se expone el nuevo entorno de experimentación necesario para la obtención de resultados empíricos. A continuación presentamos una propuesta, sección 7.3, para la asignación eficiente de los objetos en los nodos de un sistema , de modo que se minimize el tráfico de datos, basada en la

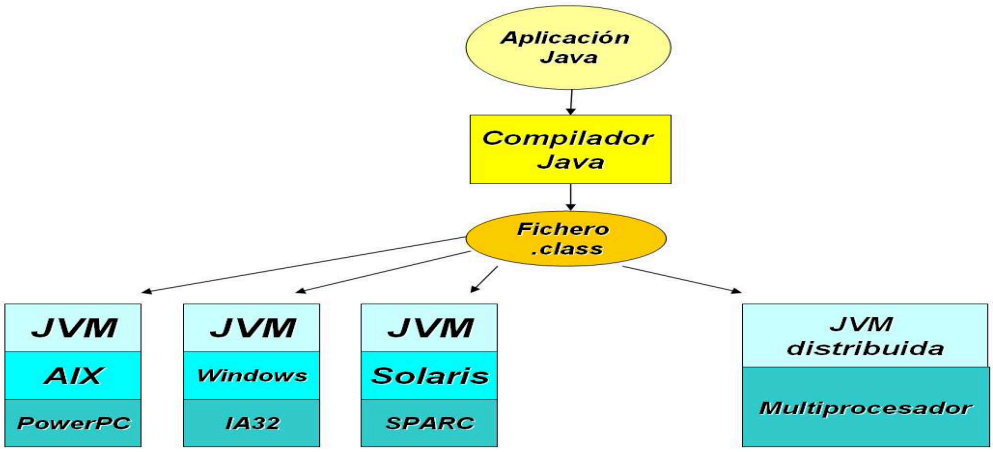


Figura 7.1: JVM capaz de aprovechar los recursos de un sistema multiprocesador de modo transparente al programador de aplicaciones monoprocesador

información suministrada por el recolector de basura. Además, hemos realizado un estudio experimental, sección 7.4, de distintos mecanismos de barrera como paso previo para la implementación de un recolector de basura global dentro de un entorno distribuido.

7.1.1 Trabajo previo

El trabajo más relevante que busca minimizar la cantidad de datos intercambiados entre los nodos de un sistema multiprocesador es el de Fang et al. [FWL03]. La estrategia de su propuesta se basa en reubicar dinámicamente los objetos dentro del sistema multiprocesador. Para ello y al tiempo que se ejecuta la aplicación, la máquina virtual ha de conocer el grafo de relaciones así como todos los cambios causados por la acción del mutador. A partir de una información detallada de las referencias entre objetos, esta propuesta plantea algoritmos de reubicación de los objetos a los nodos más convenientes. La principal debilidad de este enfoque es el reiterado uso de barreras que permitan registrar las referencias entre objetos y el espacio necesario para almacenarlas. Nuestra propuesta busca el mismo objetivo pero partiendo de una filosofía radicalmente distinta. Gracias al conocimiento exacto que tiene el recolector de basura del grafo de relaciones, se hace innecesario el registro detallado de sus cambios mientras actúa el mutador.

Además, una serie importante de trabajos se han desarrollado en el contexto tanto de máquinas virtuales distribuidas como de recolección de basura en un entorno distribuido. Plainfosse y Shapiro [PS95] describen una extensa colección de técnicas para la

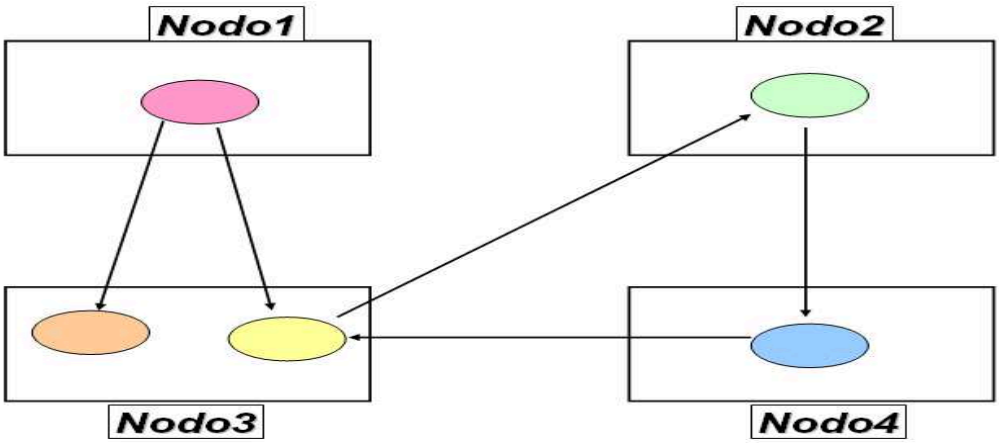


Figura 7.2: Grafo de relaciones entre objetos distribuidos en los nodos de un sistema multiprocesador

identificación de basura en heaps distribuidos. Conjuntamente, dentro del libro clásico de Jones sobre recolección de basura [Jon00], en el capítulo escrito por Lins, tenemos una perspectiva amplia de la recolección de basura distribuida y sus implicaciones en el diseño de las máquinas virtuales.

Un trabajo muy significativo dentro del tema de máquinas virtuales distribuidas es el trabajo realizado por Zigman et al. [ZS02], el entorno dJVM, el cual nos ha servido de base para la implementación de nuestra propuesta. La herramienta dJVM es una máquina virtual de Java distribuida en un sistema multiprocesador pero que presenta ante el programador la imagen de una máquina virtual monoprocesador. De este modo la complejidad inherente a un espacio de memoria distribuido es transparente al desarrollador de aplicaciones. El énfasis de su trabajo se centra en las distintas tareas de la JVM salvo en la recolección de basura en la que adoptaron un enfoque muy conservador: los nodos del sistema multiprocesador nunca reclaman objetos con referencias globales, directas o indirectas. Enfoques similares, aunque de perspectivas más limitadas que dJVM, son cJVM [IBMa] y Jessica [MWL00]. El primero, de Aridor et al. [AFT01], es también una máquina virtual de Java, construida sobre la infraestructura de un sistema multiprocesador. Este proyecto ha desarrollado nuevos modelos de objeto Java e hilo, que están ocultos al programador, y una estructura master-proxy para el acceso a objetos remotos. Jessica, de la universidad de Hong-Kong, utiliza un modelo de gestión de hilos master-slave. En este enfoque todos los hilos que están ejecutándose en los nodos del sistema multiprocesador

tienen una contrapartida en el nodo maestro, el cual se encarga de la entrada/ salida y la sincronización, siendo este factor un claro punto débil. Una mejora reciente es la propuesta por Daley et al [DSZ06], que consigue evitar la necesidad de la replicación de los hilos en un nodo maestro en determinadas circunstancias. Al igual que dJVM, cJVM y Jessica no han profundizado en la recolección de basura distribuida.

Dentro de otra línea de investigación está JavaParty [Zen], de la universidad de Karlsruhe. Este proyecto no implementa una máquina virtual que presente la imagen de una JVM monoprocesador al programador, liberándole de los aspectos más delicados y complejos de la programación. Este proyecto se desarrolla bajo la filosofía de dotar al lenguaje Java de extensiones para que el programador sea el responsable de la distribución y el acceso a los objetos remotos. Estas extensiones al lenguaje Java standard son posteriormente compiladas a lenguaje Java formal primero y después a bytecodes para su ejecución por la JVM. Un paso más alla en esta dirección, son Jackal [VBB99] e Hyperion [ABH+01]. En estas propuestas los bytecodes Java son traducidos a código C y posteriormente compilados a código nativo. De modo que durante la ejecución no hay ninguna participación de la máquina virtual de Java. El aspecto negativo de todos estos enfoques es que no aprovechan las dos grandes fortalezas de Java:

- La ejecución de una aplicación sobre cualquier hardware a partir de una única compilación.

- El programador está liberado de la gestión de memoria.

Dentro del tema de las técnicas de barrera, Pirinen [Pir98] realiza un análisis desde un punto de vista formal de mecanismos enfocados a la recolección incremental en el contexto monoprocesador. En la recolección incremental se identifican los objetos basura en un hilo paralelo a la acción del mutador. Así, nuestra investigación es una extensión de la suya, ya que en este trabajo presentamos el estudio experimental de diferentes mecanismos de barrera comparando sus rendimientos y en el contexto de la recolección de basura distribuida. Otro trabajo clásico es el de Blackburn et al. [BM02], el cual analiza el efecto de incluir las barreras de escritura en todas las secciones del código que las necesiten y cómo esto repercute en el tiempo de ejecución y su sobrecarga en el tiempo de compilación. Este último trabajo utiliza JikesRVM como máquina virtual y está restringido al ámbito monoprocesador.

7.2 Entorno de simulación

Nuestra propuesta se ha basado en la máquina virtual distribuida de Java (dJVM) de Zigman et al [ZS02], cuyas caracteristicas se discuten en la sección 7.2.1. Los benchmarks utilizados son las aplicaciones de la Standard Performance Evaluation Corp. (SPEC), secciones 7.2.2 y 3.6.

En nuestros experimentos hemos utilizado las posibles configuraciones dentro de un cluster con 8 nodos SMP, cada uno con 4 procesadores AMD OPteron a 2 GHz y 1024 MB de memoria RAM. El sistema operativo era Linux Red Hat 7.3. El hardware de comunicación entre nodos era ”fast Ethernet” utilizando el protocolo standard de red TCP/IP.

7.2.1 La máquina distribuida de Java: dJVM.

La máquina distribuida de Java, dJVM ([ZS02], [ZS03]), de la Universidad Nacional de Australia en Canberra, está implementada como una extensión de JikesRVM. Su principal objetivo es desarrollar una máquina virtual capaz de aprovechar los recursos de un sistema multiprocesador al ejecutar aplicaciones dirigidas a plataformas monoprocesador. Esta filosofía se puede apreciar de dos modos. El primero sería: dJVM puede ejecutar aplicaciones monoprocesador dentro de un sistema multiprocesador consiguiendo mejores rendimientos. O un segundo punto de vista sería que el desarrollador monoprocesador puede programar aplicaciones distribuidas sin ningún incremento de la complejidad.

EL diseño inicial de dJVM utiliza la versión 2.3.0 de JikesRVM con su compilador base. El gestor de memoria es el Java memory manager toolkit (JMTk). En esencia, dJVM es una máquina JikesRVM con una serie de cambios en determinados aspectos cruciales:

- Infraestructura: fundamentalmente para incluir los mensajes entre nodos, el proceso de inicialización en los distintos nodos y el uso de las librerias del sistema.
- Asignación de objetos, distribuyéndolos en los diferentes nodos del sistema multiprocesador.
- Modificaciones de la máquina virtual para que pueda realizar sus tareas habituales de forma remota, como por ejemplo la carga de clases, la invocación de métodos y el acceso a objetos.

La máquina virtual distribuida ha sido diseñada siguiendo el modelo maestro-esclavo. En dJVM un nodo hace las funciones de maestro y el resto son esclavos. El proceso de inicialización se lleva a cabo en el nodo maestro. Este nodo es el responsable de crear los canales de comunicación entre los nodos esclavos, la carga de clases y la gestión de los datos de forma global.

En dJVM todos los objetos tienen una instancia local pero son accesibles de forma remota. Esto se consigue al disponer de dos formas de direccionamiento:

- Cada objeto tiene asociado un identificador universal (UID) que claramente lo distingue del resto de objetos en el sistema multiprocesador.
- Desde el marco de referencia de un nodo específico, los objetos pueden ser locales e identificarse por un identificador de objeto (OID), o pueden ser remotos e identificarse por un identificador lógico (LID). El OID es la dirección que el objeto tiene en ese nodo. El LID sirve para encontrar el nodo en el cual está situado.

Para evitar la sobrecarga que supondría que el nodo maestro fuese el encargado de asignar y gestionar los identificadores universales de todos los objetos, en dJVM se utiliza un estrategia descentralizada. Cada nodo asigna el UID a cada uno de sus objetos dentro de un rango de direcciones que tiene asignado. Esta solución, aunque evita la mencionada sobrecarga, conlleva una complicación extra a la hora de actualizar los identificadores en caso de migraciones de objetos entre nodos. Esta situación proporciona mayor relevancia a nuestra propuesta que busca realizar una asignación eficiente de objetos que minimice la necesidad de migraciones posteriores.

JikesRVM, y por tanto dJVM, incluye una gran cantidad de código de comprobación por aserción (assertion checking code), que fue inhabilitado para nuestros experimentos.

La máquina virtual de Java distribuida permite escoger entre diferentes políticas de elección de nodo para la asignación de objetos. En nuestros experimentos la política Round Robin (RR) obtiene una ligera ventaja sobre las demás. Por ello la hemos escogido como valor de referencia.

7.2.2 jvm98 y Pseudo-jBB

La principal aplicación Java multihilo utilizada en nuestros experimentos pertenece también a la corporación SPEC y recibe la denomicación de SPECjbb2000 [SPE00b]. SPECjbb2000 es un programa Java que emula el funcionamiento de un sistema con tres

capas, figura 7.3. El funcionamiento de este sistema busca aislar las tres etapas clave en una típica aplicación de base de datos en una compañia de ventas. En la primera capa (”interfaz”) los usuarios plantean y reciben sus peticiones dentro de un entorno amigable, produciendo cada una de estas acciones un hilo. En la segunda capa se reciben las peticiones de los usuarios y se lleva a cabo todo el procesamiento lógico, cálculos, etc. Todos los datos necesarios para ese procesamiento se requieren de la tercera capa. En la tercera capa se almacenan los datos dentro de una estructura y se soporta un interfaz de comunicación con la segunda capa. En el caso de SPECjbb2000 los tres tiers se implementan dentro de la misma máquina virtual de Java poniendo el énfasis en la segunda capa. Los datos se almacenan dentro de una estructura de árboles binarios. Al inicio de la ejecución, la aplicación crea una serie de hilos. En ellos se producirán de forma aleatoria las transacciones que simulan la acción de los usuarios. Para poder utilizar resultados experimentales comparables en cuanto al tráfico de datos en el sistema hemos utilizado pseudo-SPECjbb2000. Pseudo-SPECjbb2000 ejecuta un número fijo de transacciones (120.000), en contraposición a SPECjbb2000 que se ejecuta durante un tiempo fijo y con un número indeterminado de transacciones.

Dentro del conjunto de aplicaciones que componen specjvm98 (sección 3.6), solamente _227_mtrt es un benchmark multihilo. Por defecto, dJVM solo distribuye objetos que implementan la interfaz ”runnable”. Esto implica que para poder tomar medidas del número y volumen de mensajes en el sistema hemos tenido que forzar la distribución de un gran número de objetos. Los resultados obtenidos no se pueden utilizar para comparar tiempos de ejecución frente a máquinas virtuales monoprocesador pero nos sirven para el estudio del tráfico de comunicaciones en el sistema o el efecto de los distintos mecanismos de barrera.

Para todas las aplicaciones se tomó el tamaño de datos máximo (s100). Cada experimento consistió en la ejecución de la aplicación diez veces seguidas sin liberar la memoria (flush) entre ejecuciones. Este proceso se repitió diez veces para finalmente obtener el promedio.

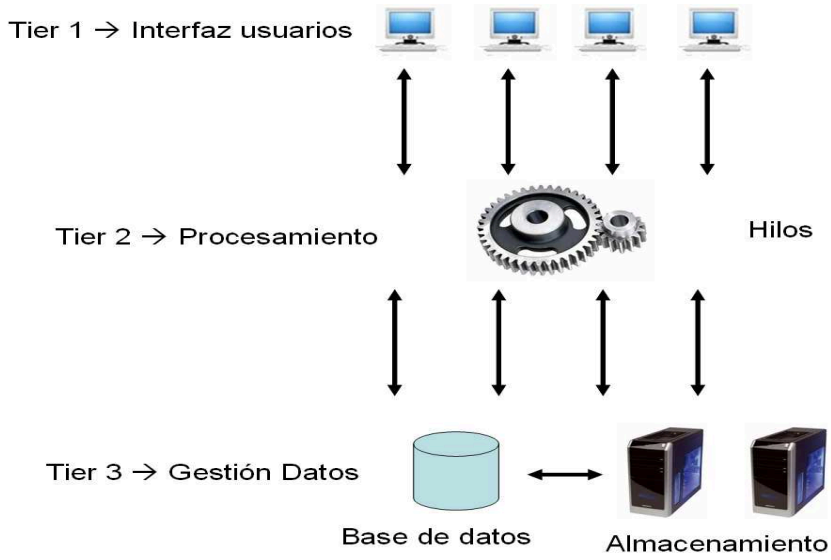
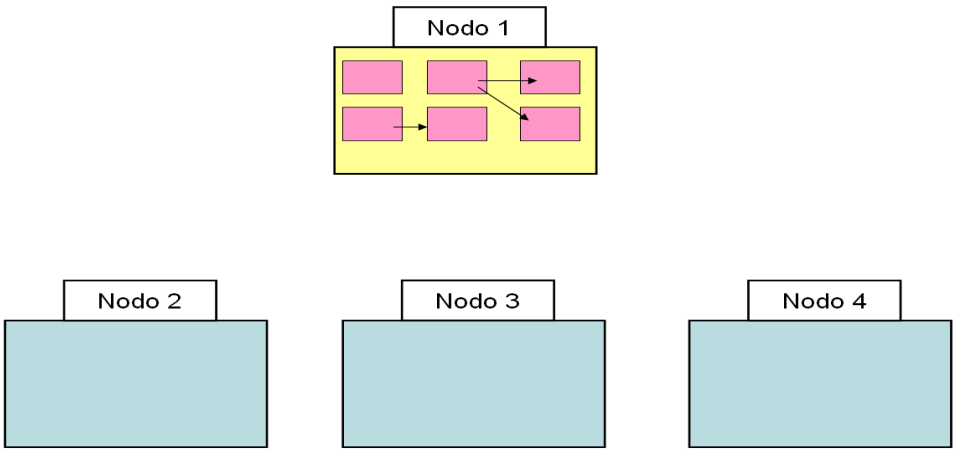


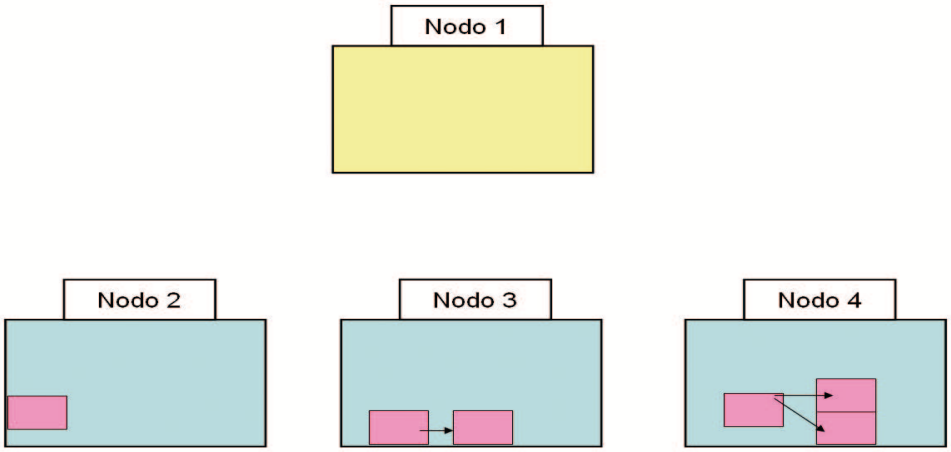
Figura 7.3: Sistema con 3 capas

7.3 Distribución de objetos basada en información del recolector de basura

El objetivo de nuestra propuesta es lograr una distribución de los objetos en los diferentes nodos de modo que se minimizen las comunicaciones del sistema. En una situación ideal cada objeto Java estaría en el mismo nodo que todos los objetos a los que él referencia o por los que él es referenciado a lo largo de su vida. Intentando acercarse a este objetivo el trabajo previo (sección 7.1.1) se ha preocupado en intentar conocer el grafo de relaciones de forma dinámica durante la ejecución de la aplicación Java, utilizando para ello costosas instrumentaciones que registran los cambios en las referencias y los almacenan en distintos tipos de tablas. Pensemos, además, que el grafo de relaciones evoluciona constantemente por la acción del mutador y que la mayoría de objetos Java tienen un tiempo de vida muy limitado, lo cual provoca que el coste necesario para la actualización exhaustiva de las referencias y el espacio necesario para almacenarlas va a contrarestar los beneficios de esta estrategia. Nuestro enfoque aborda el problema con una filosofía radicalmente distinta. Nuestra propuesta aprovecha el conocimiento exacto que la máquina virtual tiene del grafo de relaciones durante la recolección cuando ésta se lleva a cabo mediante un recolector de traza *"stop-the-world"*. Así, el funcionamiento de nuestra propuesta es:

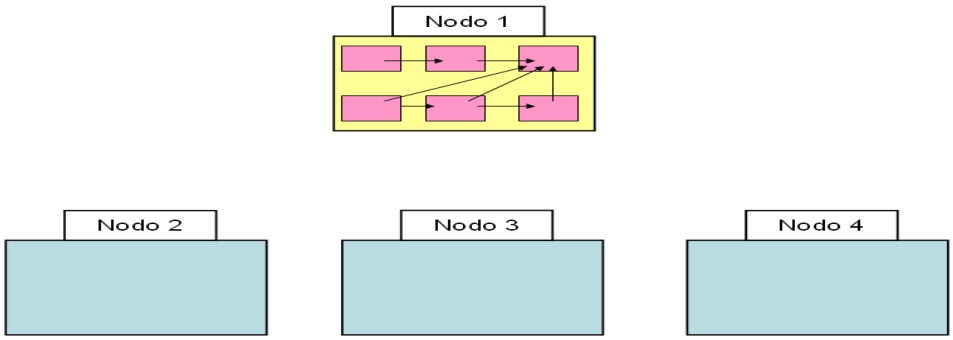


(a) Fase 1: asignación en el nodo principal

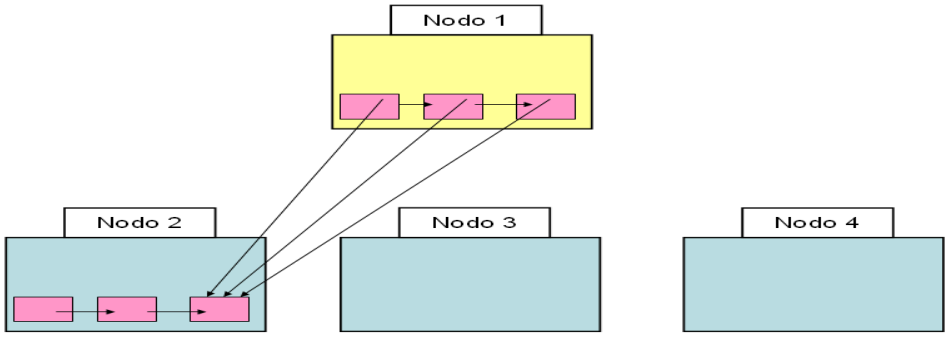


(b) Fase 2: distribución de objetos durante la recolección de basura

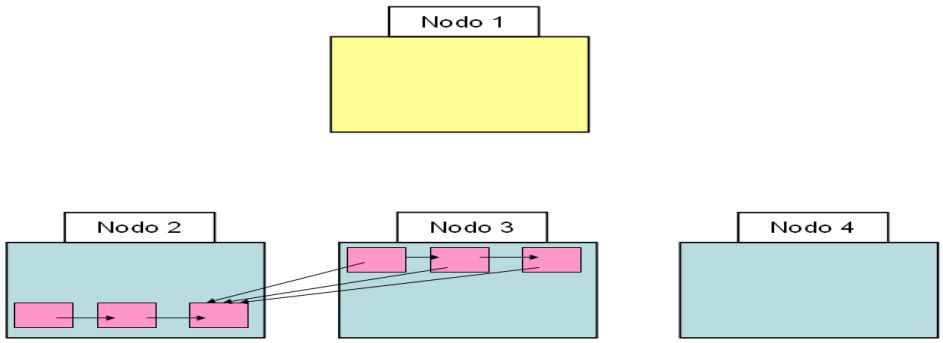
Figura 7.4: Funcionamiento de nuestra propuesta



(a) Fase 1: asignación en el nodo principal



(b) Fase 2: distribución del primer árbol durante la recolección de basura



(c) Fase 2: distribución del segundo árbol durante la recolección de basura

Figura 7.5: Funcionamiento de nuestra propuesta. Problema al encontrar un objeto referenciado por dos árboles.

- Todos los objetos nuevos son asignados a un nodo principal, figura 7.4(a).
- Cuando este nodo principal agota su memoria se inicia una recolección local de traza con política de recorrido ”primero en profundidad”. El recolector de traza es *stop-the-world*, es decir, la ejecución del mutador se suspende mientras se está produciendo la identificación de objetos vivos. De este modo la máquina virtual conoce todos los objetos que pertenecen a una rama del árbol de relaciones en el momento en que la ejecución fué cortada.
- Cada rama identificada del árbol es ahora movida a un nodo concreto del sistema. De modo que ahora tenemos en cada uno de los nodos objetos que han demostrado estar relacionados entre si con la esperanza de que en el futuro seguirán comportándose de la misma manera, figura 7.4(b).

En JikesRVM, por defecto, viene implementado un mecanismo de traza ”primero en anchura” que hubo que modificar para implementar nuestra propuesta. Recordemos que para el recorrido en anchura se utiliza una cola donde se han introducido los componentes del *root set* (ver sección 2.5.2). Procesar un objeto significa que el objeto se extrae de la cola, se recorre en busca de referencias a otros objetos y todos los objetos hallados son introducidos en la cola para su posterior procesamiento. Para implementar el recorrido en profundidad se incluyó una cola auxiliar adicional. De la primera cola sale un único objeto y siempre cuando la cola auxiliar está vacía. Cuando un objeto sale de la primera cola pasa a la cola auxiliar. El objeto es procesado cuando sale de la cola auxiliar y su descendencia se introduce en la cola auxiliar. Cuando la cola auxiliar está completamente vacía un nuevo objeto sale de la cola principal hacia la cola auxiliar y el proceso vuelve a comenzar.

Resulta evidente que nuestro sistema va a sufrir una penalización puesto que los objetos no son distribuidos al nacer. Nuestro sistema no se puede considerar distribuido hasta después de la primera recolección y por tanto estamos desaprovechando los recursos del sistema. No obstante, este desperdicio inicial de recursos se ve compensado a la larga por la acción conjunta de tres factores:

- La gran mayoría de objetos Java tienen un periodo de vida muy corto (hipótesis generacional débil, sección 2.5.3), y el gasto inherente a la distribución no va a verse compensado. Como la mayoría de estos objetos no van a sobrevivir a su primera recolección, nuestra estrategia ahorra ese gasto en términos de tráfico en el sistema puesto que no van a ser distribuidos.

- El recolector del nodo principal emplea la política de copia. El recolector de copia utiliza un asignador sencillo basado en un puntero incremental (bump pointer) pero necesita dividir la memoria disponible en dos mitades. En una mitad se realiza la asignación, la otra mitad es reservada para mover a ella los objetos supervivientes a la recolección (sección 2.5.2). Pero, puesto que todos los objetos que van a sobrevivir a la recolección en el nodo principal van a ser movidos a los otros nodos del sistema, no es necesario reservar espacio para ellos. Y, por tanto, toda la memoria disponible en el nodo principal se utiliza para la asignación.

- Y sobretodo, como factor principal, el objetivo buscado inicialmente con nuestra estrategia. Nuestros experimentos muestran que los objetos que durante la recolección mostraron que estaban relacionados entre si, tienden a seguir haciéndolo en el futuro y las comunicaciones entre objetos situados en diferentes nodos del sistema se ven drásticamente reducidas, y ello se traduce en una disminución del tiempo de ejecución.

En la implementación actual de nuestra propuesta, los objetos son asignados a un nodo conjuntamente con el árbol que primero los referencia. De modo, que cuando el recolector está recorriendo otro árbol y llega a un objeto que ya ha sido distribuido no realiza ninguna acción. Esta situación no es la ideal ya que el objeto es ubicado conjuntamente con el árbol que primero lo referencia, no con el árbol que tiene un mayor número de referencias a él como sería deseable, figura 7.5. Como trabajo futuro pensamos buscar algoritmos para ubicar los objetos pertenecientes a ramas compartidas basados en el número total de referencias que un objeto recibe de cada nodo.

7.3.1 Resultados experimentales

En la figura 7.6 se muestra la reducción en el número total de mensajes registrados entre los nodos del sistema al ejecutar nuestra propuesta frente a los registrados con la dJVM. Como ya se mencionó (sección 7.2.1), la política de elección de nodo para la asignación de nuevos objetos en dJVM es Round Robin. En esta figura se aprecia que la reducción en el total de mensajes para pseudo-jBB200 cuando el sistema consiste en 2 y 4 procesadores es superior al 50%. Al aumentar el número de nodos, el porcentaje de reducción disminuye aunque sin bajar del 25%. Esta disminución se explica por el incremento en el número de ramas que son compartidas por árboles que están ubicados en distintos nodos(ver

sección 7.3), y evidencia la necesidad en el futuro de trabajar en este sentido.

En el caso de los benchmarks pertenecientes a jvm98 el número de mensajes registrados para dJVM es muy alto y ello explica que la reducción para cualquier número de procesadores sea aún mayor que la registrada para pseudo-jbb2000. No obstante se aprecia la misma disminución en el porcentaje de reducción al aumentar el número de nodos por encima de 4.

En el caso de _227_mtrt, la reducción de mensajes de nuestra propuesta es menor para 2 y 4 procesadores, pero no disminuye tanto como en las otras aplicaciones al aumentar el numero de nodos.

Aunque el número de mensajes intercambiados entre los nodos es una métrica importante que nos permite estudiar la idoneidad de una política de elección de nodos, es más importante el volumen total de datos transmitidos a través de la red de interconexión pues ello nos proporciona un reflejo más preciso de la sobrecarga que proporciona cada política. En la figura 7.7 mostramos el porcentaje de reducción en el volumen total de datos intercambiados entre los nodos del sistema con nuestra propuesta respecto a la ejecución con dJVM. Estos resultados indican que nuestro enfoque consigue reducir el tráfico para las distintas configuraciones del sistema y los distintos benchmarks estudiados. De hecho, en el caso de pseudo-jbb2000 y los benchmarks monohilo que han sido forzados a distribuir sus datos, la reducción es siempre superior al 20%, teniendo su punto más alto para un sistema con cuatro nodos, en cuyo caso tenemos una reducción del volumen de datos intercambiados entre nodos superior al 40% y 50% respectivamente de nuestra propuesta respecto a dJVM. En el caso de _227_mtrt (recordemos que es una aplicación con dos hilos), la reducción para dos procesadores es muy pequeña, consiguiéndose mejores resultados al aumentar el número de nodos. Los porcentajes de reducción en el volumen de datos son menores que los porcentajes de reducción en el número de mensajes. Esto nos indica que la reducción en el tráfico se produce en gran medida en mensajes de actualización de referencias (propios de la máquina virtual), que tienen un peso menor en el volumen de datos total transmitido, y queda por tanto, bastante espacio para futuras optimizaciones que consigan una reducción mayor en los mensajes entre objetos.

En la figura 7.8 se muestra el porcentaje de reducción en el tiempo total de ejecución de nuestra propuesta relativa a dJVM. Nuestros resultados muestran que la reducción en el número de mensajes y en el total de datos transmitidos se traduce en una reducción

del tiempo final de ejecución. La única excepción es _227_mtrt cuando el sistema consiste en dos nodos (problema que no se muestra para el resto de configuraciones del sistema). Esta aplicación (compuesta por dos hilos) en un sistema de dos nodos funciona mejor con una política de ubicación sencilla, que sitúe cada hilo en uno de los nodos. En este caso nuestra estrategia complica la situación y no es capaz de aprovechar bien los recursos.

En general nuestra propuesta está pensada para sistema multiprocesadores de 4 o más nodos, y su funcionamiento con dos nodos no es el planeado. Pensemos que al tener solo dos nodos, uno de ellos se comporta como principal y por tanto sólo queda un único nodo para distribuir los objetos que sobreviven a la recolección de basura en el nodo principal. No obstante y a pesar de estos inconvenientes, nuestra propuesta consigue buenos resultados con dos nodos . Ello es debido a que evitamos distribuir objetos con tiempo de vida muy corto, se aprovecha al máximo el espacio de asignación del nodo principal y la distribución entre los dos nodos termina teniendo un caracter temporal que ayuda a disminuir el tráfico entre nodos. El carácter temporal a que nos referimos, es que los objetos están agrupados según el momento de su nacimiento: los objetos recien nacidos en el nodo principal y los objetos maduros en el segundo nodo.

A modo de resumen, podemos concluir que nuestra propuesta obtiene siempre mejores resultados que dJVM, aunque al igual que esta también sufre una clara degradación con el aumento del número de nodos. Estos resultados evidencian la necesidad de un mayor trabajo de investigación en el campo de la máquina virtual de Java distribuida para dotarla de una mayor competitividad. Para ello, en la sección 7.4 abordamos el estudio de los mecanismos de barrera en la máquina virtual distribuida como elemento crítico del rendimiento y como paso previo para la implementación de un mecanismo global de recolección de basura en un entorno distribuido.

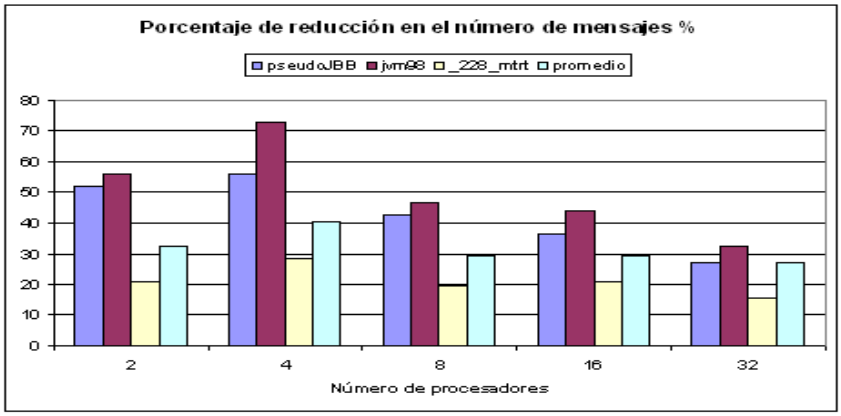


Figura 7.6: Reducción en el numero de mensajes entre nodos de nuestra propuesta relativa a dJVM

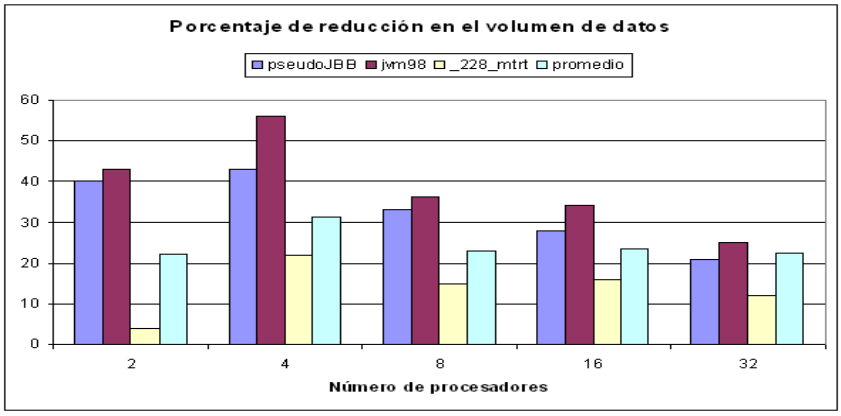


Figura 7.7: Reducción en el volumen de datos transmitidos de nuestra propuesta relativa a dJVM

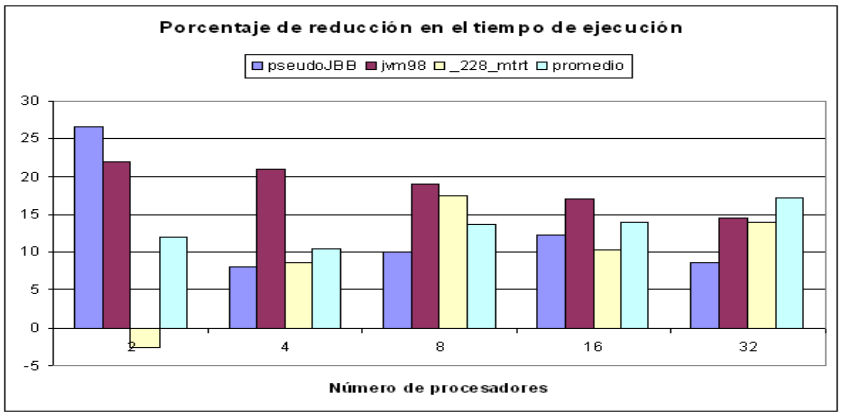


Figura 7.8: Reducción en el tiempo de ejecución de nuestra propuesta frente a dJVM

7.4 Análisis de las técnicas de barrera

En la sección 7.3 hemos presentado un algoritmo para la distribución de objetos en los nodos de un sistema para una máquina virtual distribuida basado en la implementación de un recolector de traza. Para aprovechar la mejora de rendimiento obtenida con nuestra propuesta, nuestro siguiente objetivo es desarrollar un mecanismo de recolección de basura global dentro de una máquina virtual distribuida a partir de la estrategia de traza. La recolección de basura basada en traza en un entorno distribuido contrasta con soluciones desarrolladas que podemos encontrar en la literatura ([JL96], [AFT01] y [ZS02]) que evitan reclamar objetos con referencias globales (directas o indirectas) o están basadas en la estrategia de cuenta de referencias. Recordemos que el recolector por cuenta de referencias (sección 2.4.2) no es completo, es decir, por si mismo no es capaz de reciclar estructuras de datos circulares. En un entorno distribuido, esta estrategia conlleva un elevado tráfico entre nodos para la actualización del registro del número de referencias de los objetos. Además, dentro del contexto de Java, la mayoría de estos mensaje entre nodos serían debidos a la actualización de objetos con un periodo de vida muy corto, lo cual no compensaría su elevado coste. La implementación de un recolector de traza evita los problemas anteriormente citados y además, permite aplicar nuestro algoritmo para la distribución de objetos en los nodos del sistema.

Conceptualmente nuestro recolector de traza distribuido funcionaría en dos fases diferenciadas:

- Primero, la fase de traza y marcado buscando objetos vivos. Esta fase sería global y supondría recorrer el grafo total de relaciones a través de todo el heap distribuido. Este recorrido global del grafo de relaciones nos permitiría reubicar los objetos supervivientes en el nodo más apropiado.
- Segundo, el reciclaje de los objetos muertos. Esta fase es local a cada nodo y se puede desarrollar a partir de un recolector de copia o de uno basado en la utilización de listas de distintos tamaños.

Para el estudio de los mecanismos de barrera, en la implementación actual de nuestro recolector, hemos desarrollado el trazado global del grafo de relaciones entre objetos, quedando como trabajo futuro el reciclaje de los objetos muertos con referencias globales.

Recordemos, sección 2.5, que el recolector de traza, para la identificación de objetos vivos, recorre recursivamente el grafo de relaciones marcando los objetos visitados, de

modo que en un instante determinado del recorrido cualquier objeto del grafo estaría incluido en alguno de los siguientes grupos:

- Objetos negros: objetos que han sido visitados, marcados como vivos y que no van a volver a ser visitados durante la traza actual. Estos objetos han sido completamente procesados o escaneados.
- Objetos grises: objetos que aún no han sido procesados pero que sabemos que están referenciados por algún otro objeto vivo. Estos objetos están vivos y han de ser escaneados en busca de referencias a otros objetos. El proceso de calificar un objeto como gris se conoce como ”sombreado”(shading).
- Objetos blancos: objetos que no han sido visitados y que por tanto son candidatos a ser considerados basura.

Al finalizar el recorrido del grafo de relaciones sólo tendremos objetos negros y blancos. El recorrido global del grafo de relaciones en un entorno distribuido conlleva inherentemente un tiempo de pausa muy elevado. Para minimizar en lo posible este tiempo de pausa necesitamos dividir el recorrido global en trazados parciales que o bien se ejecuten conjuntamente con el mutador o bien produzcan pequeñas pausas. Al igual que ocurría con la recolección generacional, un recorrido parcial del grafo de relaciones implica la utilización de algún mecanismo que registre los cambios en las referencias del resto de objetos del grafo a los objetos que están siendo trazados. En el contexto de la recolección de basura (incremental y generacional) sobre un monoprocesdor, el coste de las barreras es una de las principales (si no la principal) debilidades, y en un contexto distribuido este problema va a ser aún más crítico. De este modo la implementación de los mecanismos de barrera, que garantizan que ningún objeto vivo va ser considerado basura, es el principal talón de Aquiles de nuestro recolector global distribuido basado en la estrategia de traza. En esta sección vamos a analizar experimentalmente las distintas formas posibles de desarrollar los mecanismos de barrera como paso previo para el desarrollo completo de nuestro recolector.

En esencia, las barreras permiten a la máquina virtual registrar cambios en el grafo de relaciones. Las barreras pueden ser de dos tipos:

- Barrera de lectura: código encargado de interceptar instrucciones ”load” entre objetos.

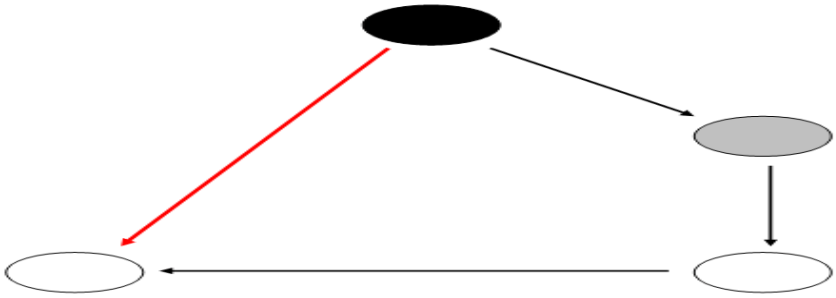


Figura 7.9: Invariante fuerte frente a invariante débil

- Barrera de escritura: código encargado de interceptar instrucciones ”store” entre objetos.

El objetivo de las barreras es conseguir que cuando se produzca el reciclaje de la basura no exista ninguna referencia de un objeto negro (vivo) a un objeto blanco (muerto). Esta propiedad [Pir98], que se conoce como ”invariante tricolor fuerte”, es de obligado cumplimiento al finalizar la fase de marcado. Mientras se está llevando a cabo el trazado del grafo de relaciones, podemos relajar el nivel de exigencia, y hablar de la ”invariante tricolor débil”. Esta invariante nos dice que todos los objetos blancos que son referenciados por un objeto negro, lo son tanbién a través de una cadena de referencias que tiene su origen en un objeto gris. Es decir, la invariante débil garantiza que al finalizar el trazado se alcanzará la invariante fuerte. Existen distintas técnicas para cumplir con estas invariantes. Cada una de ellas está basada en un tipo de barrera, ya sea lectura o escritura, o en ambas. En la figura 7.9 tenemos un ejemplo de un grafo de relaciones mientras está siendo recorrido por el recolector. En este caso no se ha preservado la invariante fuerte puesto que tenemos un objeto blanco que está siendo referenciado por un objeto negro. Sin embargo, sí se mantiene la invariante débil, ya que el objeto blanco es accesible a través de una cadena de referencias que tienen su origen en un objeto gris.

7.4.1 Espacio de diseño de decisiones ortogonales para la implementación de barreras

En la figura 7.10 mostramos nuestro espacio de diseño de decisiones ortogonales para la implementación de técnicas de barrera. En este espacio hemos clasificado cada una de las decisones más relevantes a la hora de desarrollar un mecanismo de barrera en árboles

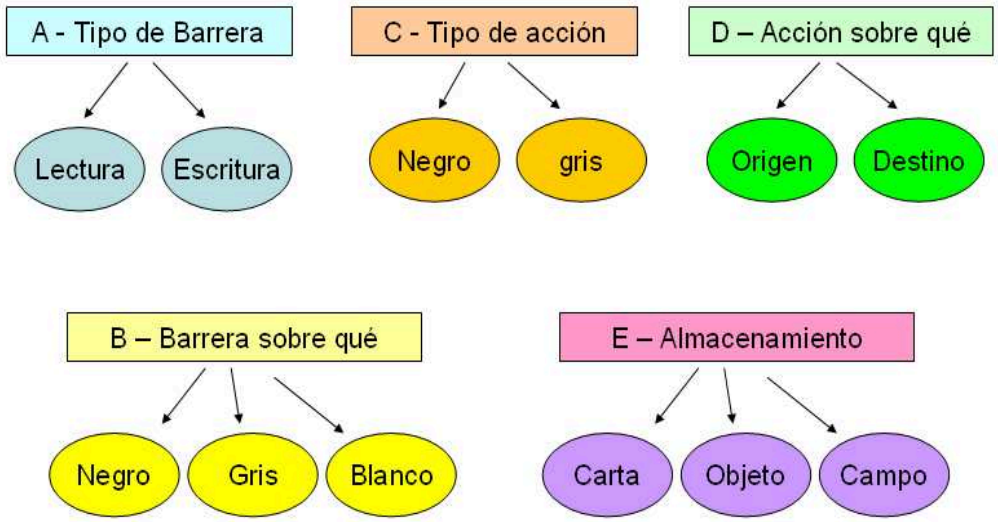


Figura 7.10: Espacio de diseño de decisiones ortogonales para la implementación de mecanismos de barrera

de decisión ortogonales. Ortogonal en este contexto significa que la elección de cualquier hoja de un árbol se puede combinar con cualquiera de las hojas de los otros árboles , y el resultado es un mecanismo de barrera válido (aunque no necesariamente un mecanismo óptimo para cualquier sistema y gestor de memoria con unas características concretas). Además, cualquier mecanismo de barrera puede ser definido como una combinación de elecciones en cada uno de los árboles de este espacio de diseño, del mismo modo que cualquier punto de un espacio geométrico puede ser representado para unos valores concretos de un conjunto de ejes ortogonales. Es importante señalar que la elección de los árboles ha sido tomada de modo que el recorrido por ellos no necesite de recorridos iterativos. Básicamente, cuando una elección ha sido tomada en cada uno de los árboles, tenemos definida una técnica de barrera. Los cuatro árboles de nuestro espacio de diseño son:

- A. Tipo de instrucción que la barrera se encarga de controlar.
  - *Read-barriers*: controlan instrucciones de lectura, *"loads"*,
  - *Write-barriers*: controlan instrucciones de escritura, *"stores"*.
- B. Tipo de objeto sobre el que se sitúan las barreras. En este caso el "tipo" del objeto está definido dentro del contexto del algoritmo de marcado tricolor de Dijkstra. Es decir, los objetos pueden ser: negros, blancos y grises.

- C. Cuando una barrera se dispara, ¿Qué clase de acción ”atómica” se produce?. El objeto se marca como negro (escaneado completamente) o gris (sombreado). En este estudio nos hemos centrado en la segunda opción, puesto que en un contexto distribuido, la realización de un escaneado atómico resulta extremadamente costosa.
- D. Cuando una barrera se dispara, ¿Sobre qué objeto se realiza la acción?: el objeto origen de la referencia o el objeto destino. Como ejemplo podemos decir que si un objeto negro referencia a uno blanco tenemos dos opciones:
  - *Origen*: marcar como gris el objeto negro.
  - *Destino*: marcar como negro el objeto blanco.
- E. Una vez que la barrera se ha ejecutado, ¿Cómo almacenamos el cambio producido en el grafo de relaciones?. Es decir, qué información nos permitirá en el futuro reconocer a un objeto vivo:
  - carta (*”Card marking”*). Este mecanismo utiliza una tabla en la que se divide la memoria en regiones de tamaño fijo (*”cards”*). Cuando una carta está marcada, el recolector necesita recorrer toda la región correspondiente en busca de objetos primero y posteriormente de punteros dentro de esos objetos. Esta solución es la que menor espacio de almacenamiento necesita, aunque también es la que implica un mayor coste durante la fase de búsqueda de objetos vivos.
  - Objeto. Se marca el objeto que contiene la referencia para su posterior procesamiento. El recolector posteriormente necesitará escanear el objeto completo en busca de punteros y visitar las direcciones a que apunten todos ellos. En cuanto a coste es una solución intermedia entre las otras dos opciones.
  - campo (*”Slot”*). Se almacena el campo del objeto que contiene la nueva referencia. El conjunto de estos campos es lo que se conoce como el *”Remembered Set”*. Esta opción es la que va a ocupar un mayor espacio de almacenamiento pero una menor carga para el recolector durante la fase de marcado de objetos vivos. Como esta ópción es la que va a sobrecargar la recolección con un menor tiempo de pausa, ha sido la escogida para este estudio enfocado a un sistema distribuido.

Aunque los árboles de decisión presentados son ortogonales, determinadas ramas de árboles específicos afectan de una manera muy directa la elección en otro de los árboles

desde el punto de vista de la coherencia de la solución final. Como ejemplo podemos citar los árboles C y D y el ejemplo visto anteriormente: un objeto negro que referencia a uno blanco. Aquí podemos marcar como negro el objeto destino o el objeto origen, pero esta última decisión resulta en una incoherencia. Este hecho, junto con las elecciones fijas ya mencionadas en los árboles C y E, nos ha llevado a el estudio de cuatro configuraciones:

- C1. Barrera de lectura sobre objetos grises.
- C2. Barrera de escritura sobre objetos negros.
- C3. Barrera de escritura sobre objetos grises y blancos.
- C4. Barrera de escritura sobre objetos grises conjuntamente con Barrera de lectura sobre objetos blancos.

Las dos primeras configuraciones mantienen la invariante *fuerte*, mientras que las dos últimas preservan la invariante *débil*.

7.4.2 Resultados experimentales

En la figura 7.11 se muestra la reducción en el número total de mensajes registrados entre los nodos del sistema como consecuencia de la acción de las barreras. Los resultados están normalizados respecto a la primera configuración (C1), barrera de lectura sobre objetos grises. Como se comprueba en la figura las otras tres configuraciones obtienen mejores resultados en todos los casos. En la figura 7.12 podemos observar que esta reducción en el número de mensajes causados por las barreras se traduce en una reducción clara en el tiempo total de ejecución. Así, la primera conclusión que obtenemos es que el mecanismo de barrera es causante de un número importante de mensajes en el sistema y la elección de un mecanismo no óptimo puede degradar claramente el rendimiento final.

Si estudiamos los resultados obtenidos por cada configuración podemos comprobar que el beneficio proporcionado por la configuración C2 es muy pequeño frente a los valores conseguidos por las configuraciones C3 y C4. Recordemos que C1 y C2 mantienen la invariante fuerte y que C3 y C4 la invariante débil. Así que este resultado nos lleva a la siguiente conclusión: desde el punto de vista del rendimiento final, la dicotomía entre invariante fuerte e invariante débil se resuelve a favor de la última. Además, si pensamos que C1 y C4 son técnicas que conllevan mecanismos de lectura mientras que C2 y C3 son

técnicas de escritura podemos concluir que las barreras de lectura son responsables de una mayor sobrecarga en el número de mensajes y consecuentemente en el tiempo final de ejecución.

A la vista de estos resultados, la configuración que causa una menor sobrecarga en el tráfico del sistema y un menor peso en el tiempo final de ejecución es la configuración C3, Barrera de escritura sobre objetos grises y blancos. La bondad de esta configuración se aprecia mejor en los benchmarks con una mayor asignación de datos como pseudo-jbb, mtrt o raytrace. A la vista de los datos experimentales, podemos concluir que las características de esta configuración, que implican mejores resultados respecto a las demás, son:

- Buscar satisfacer la invariante débil.
- Evitar implementar barreras de lectura.

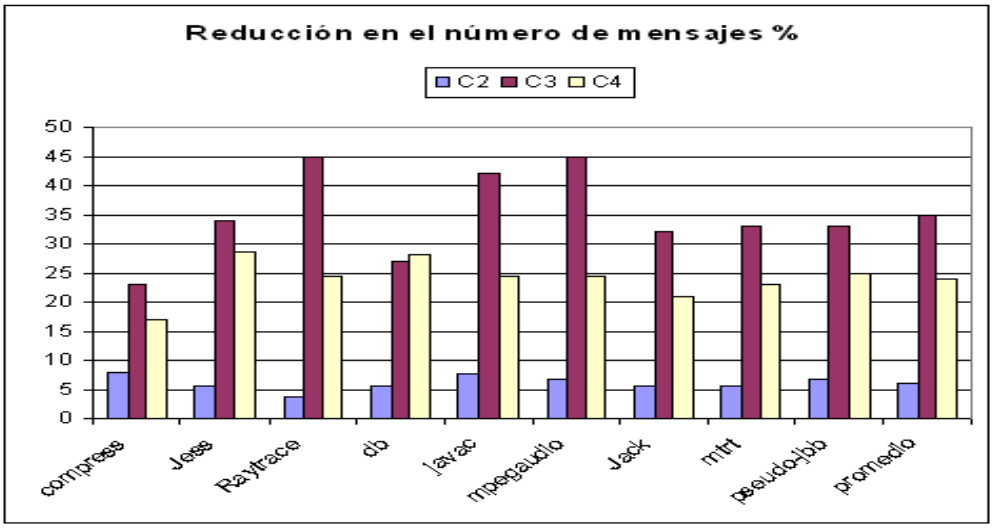


Figura 7.11: Reducción en el número de mensajes normalizada relativa a C1

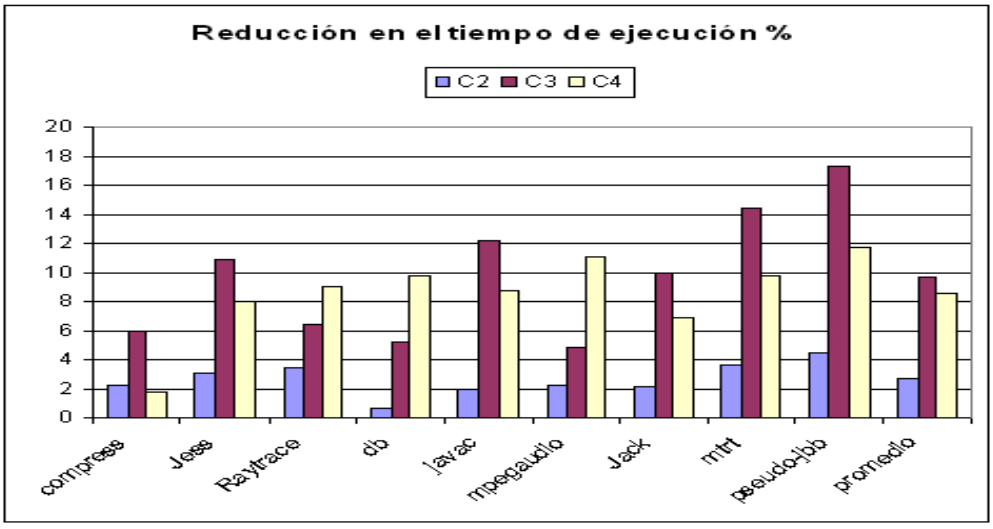


Figura 7.12: Reducción en el tiempo de ejecución normalizado frente a C1

7.5 Conclusiones

En los últimos años el diseño de software ha visto aumentada su complejidad enormemente debido a la aparición de los entornos distribuidos. Para cumplir con su ideario, la máquina virtual de Java ha de ser capaz de aprovechar los recursos que le proporciona un sistema multiporcesador pero presentando la imagen clásica de máquina virtual monoprocesador. De este modo no se exige al desarrollador de aplicaciones un mayor esfuerzo. Pero el desarrollo de esa máquina virtual distribuida de Java presenta serias dificultades. Dos de estas dificultades son la elección eficiente de un nodo específico del sistema para la asignación de cada objeto, y el desarrollo de un recolector de basura global.

En este capítulo hemos presentado una estrategia de ubicación de los objetos en los distintos nodos basada en el grafo de relaciones de la aplicación y que es llevada a cabo por el recolector de basura. Nuestros resultados experimentales muestran que nuestra propuesta proporciona una clara disminución tanto en el volumen de tráfico generado en el sistema, como el tiempo final de ejecución de las aplicaciones. Además, y como paso previo para la implementación de un recolector de basura global, hemos presentado un estudio cuantitativo de diferentes mecanismos de barrera. Nuestros resultados experimentales confirman que las distintas técnicas de barrera empleadas producen diferencias significativas en el número de mensajes que recorren el sistema. La elección incorrecta de la técnica de barrera puede degradar claramente el rendimiento del recolector de basura en un entorno distribuido.

Capítulo 8

Conclusiones

Al principio de esta tesis planteamos como principales objetivos de este trabajo:

1. Determinar experimentalmente la influencia de la recolección de basura cuando los programas Java son ejecutados por la máquina virtual en sistemas empotrados, tanto a nivel de rendimiento como de consumo energético.
2. Realizar una comparación de las estrategias de recolección actuales dentro del contexto de memoria limitada propio de los sistemas empotrados, para buscar la política que mejor se adecúe a ellos.
3. Encontrar técnicas de optimización, tanto a nivel algorítmico como a nivel de la interacción entre la máquina virtual de Java y la plataforma hardware subyacente, que reduzcan el consumo de potencia y los tiempo de pausa asociados a la recolección de basura.

Para la consecución de los objetivos uno y dos, en el capítulo 4 hemos presentado los resultados experimentales obtenidos tras la simulación completa de la máquina virtual de java ejecutando los distintos benchmarks de la corporación SPEC sobre distintas configuraciones de la jerarquía de memoria, todas ellas típicas de sistemas empotrados. Tras analizar los datos podemos concluir que:

- La elección del algoritmo de recolección puede significar una diferencia de hasta 20 puntos porcentuales en el número total de accesos a los distintos niveles de la jerarquía de memoria durante la ejecución completa. Y como el número de accesos

es el rasgo principal que marca el comportamiento de la máquina virtual en cuanto a fallos, número de ciclos y consumo de energía se refiere, podemos concluir que el algoritmo de recolección es una elección clave a la hora de implementar la máquina virtual de Java que va a determinar el rendimiento y el consumo final del sistema empotrado.

- Comparando los distintos recolectores, y dentro del espacio de diseño estudiado, encontramos que el recolector con un menor número de accesos, y por tanto con mejores valores de ciclos y energía, es el recolector generacional híbrido (GenMS), seguido de cerca por el recolector generacional de copia puro (GenCopy).

A medida que este estudio fué desarrollándose, presentamos los resultados en los workshops INTERACT-8 [VAOT04] y PARC-2004 [VAP+04]. Las conclusiones finales obtenidas, incluyendo la separación de los benchmarks en tres escenarios diferentes de comportamiento, fueron publicadas en el año 2005 en la revista Lecture Notes [VAC+05] (workshop PATMOS). Además, en este capítulo, también hemos presentado el espacio de diseño del consumo de energía frente al tiempo de ejecución, que nos ha proporcionado las mejores configuraciones (tamaño y asociatividad) de la jerarquía de memoria para cada uno de los recolectores estudiados. Estas configuraciones sirvieron de punto de partida para el desarrollo de las técnicas de optimización que nos planteamos como objetivo tercero.

En el capítulo 5 hemos presentado dos estrategias que utilizan información facilitada por el recolector de basura para que la máquina virtual de Java interactúe con la jerarquía de memoria buscando reducir el consumo de energía y el tiempo de recolección:

- La primera estrategia permite reducir el consumo estático de la memoria principal (tanto en la fase de mutador como de recolector) y está pensada para los recolectores generacionales, especialmente el generacional de copia puro. En nuestra técnica, la máquina virtual se encarga de controlar el modo de bajo consumo de los diferentes bancos de una memoria SDRAM. Para ello utiliza la información suministrada por el recolector de basura: tamaño de las generaciones e instantes de inicio y fin de las recolecciones. En los resultados de nuestras simulaciones, este mecanismo reduce a la mitad el consumo debido a las corrientes de fuga en una memoria de 16MB para el recolector generacional de copia. El dato más interesante de nuestros experimentos es que con el aumento del tamaño de la memoria principal, nuestra

estrategia consigue reducciones mayores.

- La segunda estrategia reduce tanto el consumo dinámico como el tiempo total de recolección. Esta técnica consiste en utilizar una memoria *scratchpad* con el código de los métodos del recolector que son más accedidos durante la ejecución. En el capítulo hemos presentado tres formas posibles de seleccionar el contenido de la *scratchpad*. Los resultados experimentales de esta estrategia nos han proporcionado una nueva curva de Pareto con las configuraciones óptimas para el rendimiento y consumo. En esta nueva curva hemos comprobado que siempre es más rentable la inclusión de una memoria *scratchpad*, con el código del recolector, en vez de un aumento ya sea del tamaño o de la asociatividad del primer nivel de la jerarquía de memoria. También hemos intentado ampliar esta técnica a la fase de mutador, pero no hemos encontrado niguna otra tarea de la máquina virtual cuyo peso en la ejecución sea comparable al recolector y obtenga resultados satisfactorios.

El espacio de diseño, así como las ideas preliminares de nuestras técnicas de optimización se presentaron en el simposium internacional, de la organización ACM, GLSVLSI [VAO09] en mayo de 2009. El desarrollo completo de las dos técnicas, junto con todos los resultados experimentales presentados en los capítulos 4 y 5, ha sido presentado en el Journal of Systems Architecture (editorial Elsevier) [VAO10]. El artículo ha sido aceptado y, en el momento de escribir esta tesis, está pendiente de publicación.

En el capítulo 6 hemos presentado una optimización a nivel algorítmico de la estrategia generacional con tamaño de *nursery* flexible. Nuestro recolector generacional adaptativo modifica dinámicamente el tamaño del espacio de reserva basándose en información obtenida en tiempo de ejecución. De este modo la generación *nursery* dispone de un mayor espacio de asignación. Con esta estrategia se consiguen reducir el número de recolecciones y la cantidad de memoria copiada, lo cual provoca una reducción sensible en el número de ciclos y en el consumo de energía. En nuestros resultados experimentales hemos comprobado que esta técnica consigue mejoras para distintos tamaños de memoria principal, pero es especialmente apropiada para plataformas con restricciones de memoria como los sistemas empotrados. La modificación del espacio de reserva permite a la máquina virtual funcionar como si dispusiese de un mayor tamaño de memoria principal.

La descripción de esta técnica adaptativa junto con sus resultados experimentales fueron presentados en la Conferencia Europea de Programación Orientada a Objetos (ECOOP) y, posteriormente, publicados en la revista Lecture Notes [VOT04] en el

año 2004. Diversos trabajos previos que buscaron optimizar diferentes parámetros de los recolectores generaciones fueron presentados en la conferencia internacional EUROMICRO [VOOT03] y en el workshop INTERACT [VOOT04]. Esta propuesta ha sido utilizada por P. McGachey [] como punto de partida para la realización de su Master Thesis en la Universidad de Purdue.

Al final del capítulo 6 se presenta los resultados experimentales obtenidos al aplicar conjuntamente las tres técnicas sobre los recolectores generacionales. La acción conjunta de nuestras estrategias consigue reducciones superiores al 60% tanto en número de ciclos como en consumo de energía en los recolectores generacionales, que, como hemos visto, son los más indicados para plataformas con limitaciones de memoria.

Dada la situación actual, en la que los fabricantes de microprocesadores están abandonando la reducción de la frecuencia de reloj como objetivo principal, en favor de una búsqueda intensiva de nuevas arquitecturas multicore construidas en un mismo circuito integrado, el siguiente paso natural dentro de nuestra línea de investigación apunta a la gestión de memoria dinámica dentro de un sistema multiprocesador. En el capítulo 7, hemos presentado una técnica para la asignación eficiente de los objetos en los nodos de un sistema con memoria distribuida, de modo que se minimize el tráfico de datos, basada en la información suministrada por el recolector de basura. Nuestros resultados experimentales muestran que nuestra propuesta proporciona una clara disminución tanto en el volumen de tráfico generado en el sistema distribuido, como el tiempo final de ejecución de las aplicaciones. Además, y como paso previo para la implementación de un recolector de basura global, hemos presentado un estudio cuantitativo de diferentes mecanismos de barrera. Nuestros resultados experimentales confirman que las distintas técnicas de barrera empleadas producen diferencias significativas en el número de mensajes que recorren el sistema multicore. La elección incorrecta de la técnica de barrera puede degradar claramente el rendimiento del recolector de basura en un entorno distribuido. Estos dos estudios han sido presentados en dos diferentes ediciones de la Conferencia Internacional en Computación Paralela PARCO, en los años 2005 [VAOC05] y 2007 [VAOT07].

Finalmente, podemos concluir que:

La gestión automática de memoria dinámica que realiza la máquina virtual de Java es una de las características de la filosofía Java que han hecho que este lenguaje sea el favorito de los desarrolladores e ingenieros de software de todo el mundo. Sin embargo,

como hemos visto, la recolección de basura es una tarea inherentemente compleja y en el caso de plataformas empotradas se convierte en un factor determinante tanto a nivel de rendimiento como de consumo energético. En este trabajo hemos utilizado el conocimiento del funcionamiento del recolector de basura para desarrollar técnicas que aprovechen la interacción entre la máquina virtual de Java y la plataforma hardware subyacente. Nuestros resultados experimentales muestran que esta línea de investigación es prometedora y debe ser profundizada en el futuro.

Apéndice A

Abreviaturas

A lo largo de esta tesis, y por motivos de espacio, en las tablas y figuras se utilizan la siguientes abreviaturas:

- TipoGC: tipo de recolector.
- mut: mutador
- col: recolector
- Asoc: asociatividad
  - LRU: asociatividad directa.
  - 2LRU: asociatividad 2 vías. Política de reemplazo LRU (Last Recent Used)
  - 4LRU: asociatividad 4 vías. Política de reemplazo LRU (Last Recent Used)
- i: instrucciones
- d: datos
- l2: segundo nivel de cache
- MM: memoria principal
- e: energía
- a: accesos
- m: fallos

- c: ciclos
- mi/ mut-ins-L1: primer nivel de cache para instrucciones mientras se está ejecutando el mutador
- md/ mut-data-L1: primer nivel de cache para datos mientras se está ejecutando el mutador
- ml2/ mut-l2: segundo nivel de cache mientras se está ejecutando el mutador
- mMM/ mut-Main-Memory: memoria principal mientras se está ejecutando el mutador
- ci/ col-ins-L1: primer nivel de cache para instrucciones mientras se está ejecutando el recolector
- cd/ col-data-L1: primer nivel de cache para datos mientras se está ejecutando el recolector
- cl2/ col-l2: segundo nivel de cache mientras se está ejecutando el recolector
- cMM/ col-Main-Memory: memoria principal mientras se está ejecutando el recolector
- sp: memoria scratchpad
- msp: memoria scratchpad mientras se está ejecutando el mutador
- csp: memoria scratchpad mientras se está ejecutando el recolector

Índice de Figuras

Índice de Tablas

Bibliografía

[3GS] I-Phone 3GS. http://www.apple.com/es/iphone/.

[6.7] Java NetBeans IDE 6.7. http://www.netbeans.org/.

[ABH+01] Gabriel Antoniu, Luc Boug, Philip Hatcher, Mark MacBeth, Keith Mcguigan, and Raymond Namyst. The hyperion system: Compiling multithreaded java bytecode for distributed execution, 2001. http://www4.wiwiss.fu-berlin.de/dblp/page/record/journals/pc/AntoniuBHMMN01.

[AFT01] Y. Aridor, M. Factor, and A. Teperman. A distributed implementation of a virtual machine for java. *Concurrency and Computation: practice and experience*, 13:221–244, 2001.

[AKB02] V. Agarwal, S. W. Keckler, and D. Burger. The effect of technology scaling on microarchitectural structures. Technical report, University of Texas at Austin. USA, 2002. Tech. Report TR2000-02.

[ALT] ALTERA. Powerplay early power estimators (epe) and power analyzer. http://www.altera.com/support/devices/estimator/pow-powerplay.jsp.

[Aus04] T. Austin. Simple scalar llc, 2004. http://simplescalar.com/.

[BCM04] S. M Blackburn, P. Cheng, and K. S. Mckinley. Myths and realities: The performance impact of garbage collection. In *In Proceedings of the ACM Conference on Measurement and Modeling Computer Systems*, pages 25–36. ACM Press, 2004.

[BDK+] S. Borkar, P. Dubey, D. Kuck, S. Khan, H. Mulder, S. Pawlowski, and J. Rattner. Platform 2015: Intel processor and software evolution for next decade. `http://tbp.berkeley.edu/~jdonald/research/cmp/borkar\_2015.pdf`.

[BM02] S. M. Blackburn and K. S. McKinley. In or out? putting write barriers in their place. In *IN ACM SIGPLAN INTERNATIONAL SYMPOSIUM ON MEMORY MANAGEMENT (ISMM)*. ACM Press, 2002.

[BW] J. Barton and J. Whaley. A real-time performance visualizer for java. `http://www.ddj.com/windows/184410509`.

[CA] ARM Cortex-A5. `http://www.arm.com/products/CPUs/ARM-Cortex-A5.html`.

[CSK+02] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Tuning garbage collection for reducing memory system energy in an embedded java environment. *ACM Transactions on Embedded Computing Systems*, 1:27–55, 2002.

[DSZ06] A. Daley, R. Sankaranarayana, and J. Zigman. Homeless replicated objects. In *In Proceedings of 2nd International Workshop on Object Systems and Software Architectures (WOSSA'2006)*, 2006.

[Eck00] B. Eckel. *Thinking in Java. Second Edition.* Prentice Hall. Pearson Education Company., 2000.

[EGB03] Lieven Eeckhout, Andy Georges, and Koen De Bosschere. How java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Anaheim. California. USA, 2003. ACM Press. New York. USA.

[EHE+92] A. H. Eliot, A. L. Hosking, J. Eliot, B. Moss, and D. Stefanovic. A comparative performance evaluation of write barrier implementations. In *In Proceedings of Conference on Object-Oriented Programming, Languages and Aplications. OOPSLA.*, 1992.

[Fer06] J. Guitart Fernandez. *Performance Improvement of Multithreaded Java Applications Execution on Multiprocessor Systems*. PhD thesis, Universidad Politécnica de Cataluña, 2006. http://www.tesisenxarxa.net/TDX-0303106-123000/.

[FH] E. Friedman-Hill. Jess. the rule engine for the javatm platform. http://herzberg.ca.sandia.gov/jess/.

[For82] C. L. Forgy. Rete: A fast algorithm for the many pattern/ many object pattern match problem. *Artificial Intelligence.*, 17(6):17–37, 1982.

[FWL03] W. Fang, C. Wang, and F. C. Lau. On the design of global object space for efficient multi-threading java computing on clusters. *Parallel Computing.*, 29(11-12):1563–1587, 2003.

[Har00] T. L Harris. Dynamic adaptive pre-tenuring. In *Proceedings of the 2000 ACM International Symposium on Memory Management*, 2000.

[Hei] R. Heisch. Db. a small database management program that performs several database functions on a memory-resident database. http://www.ibm.com.

[HJ06] S. Hu and L.K. John. Impact of virtual execution environments on processor energy consumption and hw adaptation. In *In* Proc. VEE*. 2006. Ottawa. Ontario. Canada.* ACM Press, 2006.

[Hom] NVIDIA Home. Physx. http://www.nvidia.com/object/physx_new.html.

[iAP] i.MX31: Applications Processor. http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=i.MX31.

[IBMa] IBM. Cluster virtual machine for java. http://www.haifa.il.ibm.com/projects/systems/cjvm/index.html.

[IBMb] IBM. Jikes compiler. http://jikes.sourceforge.net/.

[IBMc] IBM. The jikes rvm. http://oss.software.ibm.com/developerworks/oss/jikesrvm/.

[IBMd] IBM. Powerxcell 8i multicore processor. http://www.ibm.com/developerworks/power/cell/.

[IC04] M. J. Irwin and G. Chen. Banked scratch-pad memory management for reducing leakage energy consumption. In *In ICCAD*, 2004.

[Inca] Sun Microsystems. Inc. The connected device configuration (cdc). http://java.sun.com/javame/technology/cdc/.

[Incb] Sun Microsystems. Inc. Connected limited device configuration (cldc); jsr 139. http://java.sun.com/products/cldc/.

[Incc] Sun Microsystems. Inc. Java card platform specification 2.2.2. http://java.sun.com/javame/technology/l.

[Incd] Sun Microsystems. Inc. Java micro edition technology. http://java.sun.com/javame/technology/l.

[Ince] Sun Microsystems. Inc. Project sun spot. http://www.sunspotworld.com/.

[Inc03] Sun Microsystems. Inc. The source for java technology, 2003. http://java.sun.com.

[Ind] EE Times India. Powerplay early power estimators (epe) and power analyzer. http://www.eetindia.co.in/ART_8800398572_1800007_NT_b29a3cbf.HTM.

[JL96] R. Jones and R.D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons, 1996.

[Jon] R. Jones. The garbage collection bibliography. http://www.cs.kent.ac.uk/people/staff/rej/gcbib/gcbib.html.

[Jon00] R. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons. 4th edition., 2000.

[(JV] Squawk Java ME VM (JVM). https://squawk.dev.java.net/.

[Kaf05] Kaffe. Kaffe is a clean room implementation of the java virtual machine, 2005. http://www.kaffe.org/.

[Keg04] D. Kegel. Building and testing gcc/glibc cross toolchains, 2004. http://www.kegel.com/crosstool/.

[KTV+04] S. Kim, S. Tomar, N. Vijakrishnan, M. Kandemir, and M.J. Irwin. Energy-efficient java execution using local memory and object co-location, 2004. In *IEE Proc-CDT* on-line.

[Lab] Hewlett-Packard Labs. Cacti 4. http://www.hpl.hp.com/techreports/2006/HPL-2006-86.html.

[LIS] Lenguaje LISP. http://www.lisp.org/alu/home.

[Ltd] ARM. Ltd. The arm920t processor. http://www.arm.com/products/CPUs/ARM920T.html.

[McG05] P. McGachey. *An Improved Generational Copying Garbage Collector. Msc Thesis.* PhD thesis, Purdue University, December. 2005.

[McM01] Chuck McManis. Looking for lex and yacc for java? you don't know jack, 2001. http://www.javaworld.com/.

[mpgIf] POWERVR SGX543MP multi-processor graphics IP family. http://www.imgtec.com/News/Release/index.asp?NewsID=449.

[MWL00] Matchy J. M. Ma, Cho-Li Wang, and Francis C.M. Lau. Jessica: Java-enabled single-system-image computing architecture, 2000.

[NDB07] N. Nguyen, A. Dominguez, and R. Barua. Scratch-pad memory allocation without compiler support for java applications, 2007. In *Proc. CASES*.

[NVI] NVIDIA. Cuda zone. http://www.nvidia.com/object/cuda_what_is.html.

[oMAtUoT04] DSS. The University of Massachusetts Amherst and the University of Texas. Dynamic simple scalar, 2004. http://www-ali.cs.umass.edu/DSS/index.html.

[Paga] J. McCarthy's Home Page. http://www-formal.stanford.edu/jmc/.

[Pagb] R. Jones's Home Page. http://www.cs.kent.ac.uk/people/staff/rej/gc.html.

[pAP] PC205 pico Array Processor. http://www.picochip.com/products_and_technology/PC205_high_performance_signal_processor.

[PCC] PCCTS. Purdue compiler construction tool set. http://www.iis.fhg.de/audio.

[Per] Lenguaje Perl. http://www.perl.org/.

[Pir98] P. Pirinen. Barrier techniques for incremental tracing. In *Proceedings of International symposium on Memory Management*, Vancouver, Canada, September 1998.

[Proa] ARM1176JZ(F)-S Processor. http://www.arm.com/products/CPUs/ARM1176.html.

[Prob] OMAP5910 Dual Core Processor. http://focus.ti.com/docs/prod/folders/print/omap5910.html.

[Proc] Mushroom Project. http://www.wolczko.com/mushroom/index.html.

[PS95] D. Plainfossoe and M. Shapiro. A survey of distributed garbage collection techniques. In *Proceedings of International Workshop on Memory Management*, September 1995.

[Pyt] Lenguaje Python. http://www.python.org/.

[Rab03] J. Rabaey. *Digital Integrated Circuits: A Design Perspective*. Prentice Hall, Englewood Cliffs NJ, 2nd edition, 2003.

[Rub] Lenguaje Ruby. http://www.ruby-lang.org/es/.

[Sch] Lenguaje Scheme. http://www.schemers.org/.

[SHC+04] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of java application. In *USENIX 3rd Virtual Machine Research and Technology Symposium (VM'04)*, 2004.

[SM03] N. Sachindran and J. Eliot B. Moss. Mark-copy: Fast copying gc with less space overhead. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, 2003.

[SMEM99] D. Stefanovic, K. S. McKinley, J. Eliot, and B. Moss. Age-based garbage collection. In *In Proceedings of SIGPLAN 1999 Conference on Object-Oriented Programming, Languages and Applications*, pages 379–381. ACM Press, 1999.

[sof] Tiobe software. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.

[Sot00] J. Sotkes. Sound and vision: A technical overview of the emotion engine, 2000. `http://arstechnica.com/hardware/reviews/2000/02/ee.ars`.

[Sou04] Sourceforge. Jamvm - a compact java virtual machine, 2004. `http://jamvm.sourceforge.net/`.

[Sou05] Sourceforge. kissme java virtual machine, 2005. `http://kissme.sourceforge.net`.

[SPE99] SPEC. Specjvm98 documentation, March 1999. `http://www.specbench.org/osg/jvm98/`.

[SPE00a] SPEC. Spec documentation. 2000., 2000. `http://www.spec.org/`.

[SPE00b] SPEC. Specjbb2000 documentation, 2000. `http://www.spec.org/jbb2000/`.

[SSGS01] Y. Shuf, M. Serrano, M. Gupta, and J.P. Sing. Characterizing the memory behavior of java workloads: A structured view and opportunities for optimizations. In *In Proc. of SIGMETRICS'01*, June 2001.

[Tea] Multimedia Software Team. Mpeg (moving pictures experts group). `http://www.iis.fhg.de/audio`.

[tIa] Micron technologies Inc. `http://www.micron.com/`.

[tIb] Micron technologies Inc. System-power calculator. `http://www.micron.com/`.

[Ung82] D. Ungar. Generation scavening: A non-disruptive high performance storage reclamation. *Sigplan Notices.*, 19(5):157–165, 1982.

[url] Gnu classpath. `http://www.gnu.org/software/classpath/home.html`.

[VAC+05] J.M. Velasco, D. Atienza, F. Catthoor, K. Olcoz, F. Tirado, and J.M. Mendias. Energy characterization of garbage collectors for dynamic applications on embedded systems. *Lecture Notes on Computer Science*, 3728:69–78, 2005.

[Van] B. Vandette. Deploying java platform. standard edition (java se) in today's embedded devices. `http://developers.sun.com/learning/javaoneonline/2007/pdf/TS-2602.pdf`?

[VAO09] J. M. Velasco, D. Atienza, and K. Olcoz. Exploration of memory hierarchy configurations for efficient garbage collection on high-performance embedded systems. In *GLSVLSI '09: Proceedings of the 19th ACM Great Lakes symposium on VLSI*, pages 3–8, New York. NY. USA, 2009. ACM.

[VAO10] J. M. Velasco, D. Atienza, and K. Olcoz. Memory power optimization of java-based embedded systems exploiting garbage collection information. *Journal of Systems Architecture. JSA*, 2010.

[VAOC05] J.M. Velasco, D. Atienza, K. Olcoz, and F. Catthoor. Performance evaluation of barrier techniques for distributed tracing garbage collectors. In *In Proceedings of the Parallel Computing (PARCO)*, pages 549–556. Nic series Volume 33, 2005.

[VAOT04] J. M. Velasco, D. Atienza, K. Olcoz, and F. Tirado. Garbage collector refinements for new dynamic multimedia applications on embedded systems. In *In Proceedings of Workshop INTERACT-8*, pages 25–32. IEEE, 2004.

[VAOT07] J. M. Velasco, D. Atienza, K. Olcoz, and F. Tirado. Efficient object placement including node selection in a distributed virtual machine. In *In Proceedings*

of the Parallel Computing (PARCO), pages 509–516. Nic series Volume 38, 2007.

[VAP+04] J.M. Velasco, D. Atienza, L. Piñuel, F. Catthoor, F. Tirado, K. Olcoz, and J.M. Mendias. Energy -aware modelling of garbage collectors for new dynamic embedded systemss. In *In Proceedings of First workshop on Power-Aware Real-time Computing PARC*, 2004.

[VBB99] R. Veldema, R. Bhoedjang, and H. Bal. Distributed shared memory management for java. Technical report, 1999.

[VOOT03] J. M. Velasco, A. Ortiz, K. Olcoz, and F. Tirado. Dynamic tuning in generational collection. In *In Proceedings of the 29th EUROMICRO Conference. WIP Session*. E. Grosspietsch and K. Klöckner, 2003.

[VOOT04] J. M. Velasco, A. Ortiz, K. Olcoz, and F. Tirado. Dynamic management of nursery space organization in generational collectio. In *In Proceedings of Workshop INTERACT-8*, pages 33–40. IEEE, 2004.

[VOT04] J. M. Velasco, K. Olcoz, and F. Tirado. Adaptive tuning of the reserved space in an appel collector. *Lecture Notes on Computer Science*, 3086:543–559, 2004.

[Was] S. Wasson. Nvidia's geforce 8800 graphics processor. `http://techreport.com/articles.x/11211/1`.

[W.D75] Edsger W.Dijkstra. On-the-fly garbage collection: an exercise in multiprocessing, 1975. `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD492.html`.

[Wel84] T. A. Welch. A technique for high performance data compression. *IEEE Computer.*, 17(6):8–19, 1984.

[Zen] Matthias Zenger. Javaparty. `http://svn.ipd.uni-karlsruhe.de/trac/javaparty/wiki/`.

[ZS02] J. Zigman and R. Sankaranarayanara. djvm - a distributed jvm on a cluster. Technical report, 2002.

[ZS03] J. Zigman and R. Sankaranarayanara. djvm - a distributed jvm on a cluster. In *In Proceedings of 17th European Simulation Multiconference*, 2003.