MEJORA DEL RENDIMIENTO DE LAS APLICACIONES JAVA USANDO COOPERACIÓN ENTRE EL SISTEMA OPERATIVO Y LA MÁQUINA VIRTUAL DE JAVA

MEJORA DEL RENDIMIENTO DE LAS APLICACIONES JAVA USANDO COOPERACIÓN ENTRE EL SISTEMA OPERATIVO Y LA MÁQUINA VIRTUAL DE JAVA

Yolanda Becerra

Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya

Barcelona

Directores de tesis:

Toni Cortes

Jordi Garcia

Nacho Navarro

TESIS DEPOSITADA EN CUMPLIMIENTO
DE LOS REQUERIMIENTOS PARA OBTENER EL GRADO DE
Doctora en Informática

JULIO, 2006

RESUMEN

El uso de los entornos virtualizados de ejecución se ha extendido a todos los ámbitos y, en particular, se está utilizando para el desarrollo y la ejecución de aplicaciones con un alto consumo de recursos. Por lo tanto, se hace necesario evaluar si estas plataformas ofrecen un rendimiento adecuado para este tipo de programas y si es posibl aprovechar las características de estas plataformas para favorecer su ejecución.

El objetivo principal de este trabajo ha sido ha sido demostrar que es posible explotar las características de los entornos virtualizados de ejecución para ofrecer a los programas una gestión de recursos que se adapte mejor a sus características.

En este trabajo demostramos que el modelo de ejecución de este tipo de entornos, basado en la ejecución sobre máquinas virtuales, ofrece una nueva oportunidad para implementar una gestión específica de recursos, que permite mejorar el rendimiento de los programas sin renunciar a las numerosas ventajas de este tipo de plataformas como, por ejemplo, una portabilidad total del código de los programas.

Para demostrar los beneficios de esta estrategia hemos seleccionado como caso de estudio la gestión del recurso memoria para los programas de cálculo científico en el entorno de ejecución de Java. Después de un análisis detallado de la influencia que tiene la gestión de memoria sobre este tipo de programas, hemos visto que añadir en el entorno de ejecución una política de prefetch de páginas que se adapte al comportamiento de los programas es una posible vía para mejorar su rendimiento.

Por este motivo, hemos analizado detalladamente los requerimientos que debe cumplir esta política y cómo repartir las tareas entre los diferentes componentes del entorno de ejecución de Java para cumplir estos requerimientos.

Como consecuencia, hemos diseñado una política de prefetch basada en la cooperación entre la máquina virtual y el sistema operativo. En nuestra propuesta, por un lado, las decisiones de prefetch se llevan a cabo utilizando todo el conocimiento que la máquina virtual tiene sobre el comportamiento dinámico de los programas y el conocimiento que el sistema operativo tiene sobre las condiciones de ejecución. Por otro lado, el encargado de llevar a cabo las decisiones de gestión es el sistema operativo, lo que garantiza la fiabilidad de la máquina. Además, esta estrategia es totalmente transparente al programador y al usuario, respetando el paradigma de portabilidad de los entornos de ejecución virtualizados.

Hemos implementado y evaluado esta estrategia para demostrar los beneficios que ofrece al tipo de programas seleccionado y, aunque estos beneficios dependen de las características del programa, la mejora del rendimiento ha alcanzado hasta un 40 % si se compara con el rendimiento obtenido sobre el entorno original de ejecución.

AGRADECIMIENTOS

En primer lugar, quiero agradecer a mis directores de tesis, Toni Cortes, Jordi Garcia y Nacho Navarro, el esfuerzo que cada uno de ellos ha invertido en este trabajo y su nivel de implicación.

También quiero agradecer al Departament d'Arquitectura de Computadors y, en particular, a la línea de investigación de Computación de Altas Prestaciones y a su director Mateo Valero, el haberme proporcionado el entorno adecuado y los recursos necesarios para poder desarrollar este trabajo.

Toni Cortes y Jordi Garcia son los principales responsables de que este trabajo haya llegado a su fin, no sólo por su labor de dirección, sino también por su apoyo y la confianza que han depositado en mí. Gracias por no haber permitido que me rindiera hace casi seis años. Gracias por vuestros consejos, por vuestra paciencia y por sacar tiempo para este trabajo (muchas veces de donde no lo teníais). Gracias por esas reuniones de trabajo (atípicas) que, entre risas y bromas, han servido para encontrar el camino de salida de algún que otro laberinto (y para hacerme más fuerte). Es verdad que en las dificultades se descubren a los verdaderos amigos: vosotros lo sois.

Son muchos los motivos por los que quiero dar las gracias a David Carrera. El me ha ayudado con sus comentarios, siempre acertados, sobre este trabajo y con la instalación y la administración de las máquinas que hemos utilizado. Y, aún más importante, ha sido un apoyo fundamental en los momentos de desánimo. Gracias por tu paciencia y por esas rutas (y ese universo ficticio de fauna diversa) que siempre consiguen distraerme de las preocupaciones: sólo tu categoría personal supera tu capacidad profesional.

Afortunadamente, la lista de compañeros y amigos del departamento que me han ayudado en todo este tiempo es demasiado larga para ponerla de forma exhaustiva: muchas gracias

a todos. De forma especial tengo que agradecer el apoyo técnico y personal de dos de ellos. Ernest Artiaga, mi compañero más antiguo desde la época del colegio, pasando por la licenciatura, hasta llegar a los estudios de doctorado. En demasiadas ocasiones tu talento me ha sacado de un apuro: siempre silencioso... ¡pero cómo se nota si no estás! También Xavier Martorell, desde mis primeros días en el departamento, me ha ofrecido su amistad y su valioso soporte técnico. Gracias por tus comentarios siempre precisos: todo un lujo.

También quiero dar las gracias a mis amigos, por su inmensa paciencia y por su cariño. En especial gracias a Susana, María José y Pedro: mi familia elegida a dedo. Gracias por haber sido un hombro en el que llorar en los malos momentos y por haber reído conmigo en los buenos.

Y por último quiero dar las gracias a mi extensa familia porque recibir cariño incondicional por tierra, mar y aire hace que todo sea mucho más llevadero. En especial gracias a mi hermano José Antonio y a mi cuñada Yolanda, por la confianza que me da sentir que os tengo ahí. Y sobre todo, gracias a mis padres, José y Nieves: el ejemplo que siempre ha guiado mis pasos. Gracias por no escatimar muestras de cariño, por vuestra entrega y dedicación, y por la paciencia que habéis tenido estos años. Este trabajo os lo dedico porque, sin lugar a dudas, sin vosotros no habría sido posible.

Financiación

Este trabajo ha sido subvencionado parcialmente por el Ministerio de Educación Español y por la Unión Europea (fondos FEDER) bajo los contratos TIC2001-0995-C02-01 y TIN2004-07739-C02-01

A mis padres José y Nieves

ÍNDICE

LISTA DE FIGURAS

Capítulo 1

Capítulo 2

Capítulo 3

LISTA DE TABLAS

Capítulo 1

Capítulo 2

Capítulo 3

Capítulo 4

Capítulo 5

Capítulo 6

Capítulo 7

Capítulo 8

1

INTRODUCCIÓN

Los entornos virtualizados de ejecución han demostrado ser una opción muy atractiva para el desarrollo y la ejecución de programas. El interés por estos entornos de ejecución ha crecido al mismo tiempo que se descubría el potencial de Internet para la ejecución distribuida de aplicaciones y la posibilidad de explotar de forma global los recursos conectados a la red. Para poder aprovechar este potencial es necesario independizar los programas de las plataformas reales de ejecución, ya que el entorno formado por las máquinas conectadas a Internet es muy heterogéneo. Esta es precisamente una de las principales características de este tipo de entornos de ejecución: aislan a los programas de la plataforma real dotándolos de una portabilidad total, que permite que se ejecuten sobre cualquier sistema operativo y en cualquier arquitectura.

Sin embargo, la portabilidad no es la única ventaja derivada del uso de estos entornos. Por ejemplo, las facilidades ofrecidas por los lenguajes diseñados para el desarrollo de aplicaciones en este tipo de entornos (como Java o C#) son también un reclamo para los programadores.

Por todo ello, el uso de los entornos virtualizados se ha extendido a todos los ámbitos de la computación, para el desarrollo de programas con características muy diversas. Esto significa que estas plataformas de ejecución se están utilizando para programas con características muy diferentes a las que se consideraba durante su creación y, en concreto, se están utilizando para desarrollar aplicaciones con un alto consumo de recursos. Por consiguiente, es necesario estudiar si realmente son adecuadas y si existe alguna otra

característica de estos entornos que se pueda explotar para mejorar el rendimiento de este tipo de programas.

Una de las características más relevantes de estos entornos de ejecución viene dada por su modelo de ejecución. Este modelo se basa en la utilización de una capa de software o máquina virtual que, en tiempo de ejecución, convierte el código binario del programa en el código correspondiente a la plataforma real que está soportando la ejecución. Esto significa que la máquina virtual tiene acceso a toda la información sobre el comportamiento dinámico de los programas y, por lo tanto, es capaz de tomar las decisiones de gestión más adecuadas para este comportamiento. En este trabajo proponemos añadir a los entornos virtualizados de ejecución una gestión de recursos basada en la cooperación entre la máquina virtual y el sistema operativo, que aproveche la información que posee la máquina virtual para adaptar las decisiones de gestión a las características de cada programa.

1.1 MOTIVACIÓN

La gran diversidad de plataformas de ejecución ha hecho que la portabilidad de los programas haya sido siempre una característica muy apreciada por los desarrolladores de aplicaciones. Sin embargo, el auge de Internet y su utilización para la ejecución de aplicaciones de forma distribuida, han incrementado aún más el interés de esta característica, debido a la heterogeneidad de las máquinas conectadas a esta red.

Por este motivo han aparecido lenguajes, como Java [AGH00] o C# [AW02], en los que se ha definido el modelo de ejecución recuperando el concepto de máquina virtual, con el objetivo de independizar el código ejecutable de las aplicaciones de la plataforma real donde se ejecuta. Este modelo de ejecución consiste en generar, durante la fase de compilación, un código intermedio independiente de la máquina física. La ejecución de este código independiente se hace sobre una máquina virtual (la *Java Virtual Machine* [LY99], en el caso de los programa Java, o la *Common Language Infrastructure* [Ecm05], en el caso de los programas C#), que se encarga de convertir las instrucciones en las correspondientes a la plataforma real. De esta manera, se puede ejecutar el mismo código sobre

cualquier sistema operativo (SO) y en cualquier máquina física, sin necesidad de repetir su compilación, consiguiendo una portabilidad total.

Estos lenguajes en principio se utilizaban para implementar pequeños códigos que permitían exportar servicios a través de Internet. Sin embargo, su uso se ha extendido y diversificado a todos los ámbitos de la programación, hasta ser utilizados para el desarrollo de aplicaciones completas.

Existen varios motivos que han provocado esta diversificación. En primer lugar, las facilidades que ofrecen a los programadores estos lenguajes y sus plataformas de ejecución. Estas facilidades van más allá de la portabilidad e incluyen, por ejemplo, programación orientada a objetos, gestión automática de memoria, una comprobación estricta del código que lo hace robusto y seguro, etc. Además, el modelo de ejecución y la utilización de la máquina virtual como parte del entorno de desarrollo permite incluir otras características atractivas para los programadores. Por ejemplo, estos entornos garantizan la fiabilidad del código ejecutado, dan la posibilidad de integrar diferentes lenguajes de programación y dan facilidades para permitir la interoperabilidad entre diferentes componentes software [Ric00].

En segundo lugar, la expansión de Internet ha ofrecido una nueva opción para la ejecución de aplicaciones con un alto consumo de recursos. Esta opción consiste en considerar de forma global los recursos conectados a Internet para ejecutar de forma distribuida estas aplicaciones. De esta manera es posible explotar mejor estos recursos y, por ejemplo, obtener tiempo de cálculo de procesadores que de otra manera se encontrarían ociosos [CM92, AES97, KaMR02]. Las técnicas de *grid computing* van más allá y, entre otros objetivos, pretenden virtualizar el entorno formado por un conjunto de máquinas conectadas a Internet [FGT02, BFH03, KFH$^+$06].

Por lo tanto, actualmente las plataformas de ejecución basadas en máquinas virtuales se están utilizando para ejecutar programas con un comportamiento diferente al contemplado durante su diseño y, en particular, se están utilizando para la ejecución de programas con un alto consumo de recursos. Esto significa que es necesario estudiar si estas plataformas son realmente adecuadas para soportar la ejecución de los nuevos tipos de programas y

si es posible explotar alguna de sus características para mejorar el rendimiento obtenido por estos programas.

Esto es especialmente importante si tenemos en cuenta que, el método de ejecución basado en máquinas virtuales, tiene un coste asociado que disminuye el rendimiento de los programas con respecto a la ejecución basada en una compilación tradicional. Esta disminución no tiene un impacto significativo cuando se trata de pequeños códigos, como los que inicialmente se utilizaban como parte de las aplicaciones de Internet. Sin embargo, su importancia puede aumentar cuando los códigos ejecutados tienen mayor envergadura y, aunque el resto de las ventajas de la plataforma compensan esta disminución del rendimiento, es conveniente minimizar en lo posible sus efectos.

Esta es precisamente la motivación inicial de este trabajo: ¿es posible aprovechar las características de los entornos de ejecución basados en máquinas virtuales para soportar de forma eficaz la ejecución de los programas con un alto consumo de recursos, sin renunciar a las ventajas de este modelo de ejecución?

Una de las características que distingue a este entorno de ejecución de los entornos tradicionales basados en compilación es que, en tiempo de ejecución, las máquinas virtuales tienen acceso tanto al código como a los datos de los programas. Es decir, tienen más información sobre el comportamiento de los programas que los compiladores, que están limitados a la información estática, y que los sistemas operativos, que sólo pueden aproximar este comportamiento usando la información que le proporciona el hardware (mediante algunas excepciones o interrupciones) o los propios programas (mediante las llamadas a sistema).

Mediante esta información, las máquinas virtuales pueden completar una caracterización dinámica y exhaustiva de los programas, que las capacita para determinar qué decisiones de gestión de recursos se adaptan a las necesidades de cada programa. Es decir, este tipo de entorno de ejecución favorece la utilización de políticas de gestión específicas para el comportamiento de cada programa. Hay que decir que los sistemas operativos, tradicionalmente, han implementado políticas de propósito general para la gestión de los recursos de la máquina, ya que no disponen de suficiente información sobre el comportamiento de

los programas para ofrecer una gestión específica. Estas políticas en términos generales ofrecen un rendimiento aceptable en el caso medio. Sin embargo, es posible optimizar la gestión de recursos si para tomar las decisiones de gestión se tiene en cuenta el uso que cada programa hace de cada recurso.

Se han hecho muchas propuestas para proporcionar a los programas una gestión de recursos específica. Sin embargo, hasta ahora, ninguna de ellas ha podido satisfacer por completo los requerimientos de este objetivo (ver capítulo 7). Esto ha sido así porque, o bien no tenían acceso a una información completa sobre los programas (estrategias basadas en el análisis estático de los compiladores y estrategias que implementan políticas específicas dentro del SO), o bien no respetaban la fiabilidad del sistema o el coste de asegurarla limitaba demasiado el rendimiento de los programas (estrategias basadas en permitir que el programador aporte su propio código de gestión).

En este trabajo proponemos dotar al entorno de ejecución de una gestión específica de recursos basada en la cooperación entre la máquina virtual y el SO, que supera las limitaciones de los trabajos previos relacionados. En la estrategia que proponemos, por un lado la máquina virtual aporta todo el conocimiento que tiene sobre el comportamiento de los programas para guiar las decisiones de gestión de los recursos. Por otro lado, el SO comprueba que las condiciones de ejecución son las adecuadas para llevar a cabo dichas decisiones y, de ser así, las aplica garantizando la fiabilidad del sistema.

1.2 CASO DE ESTUDIO

Para probar los beneficios de la cooperación entre el sistema operativo y la máquina virtual en la gestión de recursos, hemos seleccionado como caso de estudio la gestión de memoria en el entorno de ejecución de los programas Java. Hay que destacar, sin embargo, que la idea es igualmente aplicable a otros entornos virtualizados de ejecución y a la gestión de otros recursos del sistema.

1.2.1 El lenguaje de programación Java y su plataforma de ejecución

El lenguaje de programación Java [AGH00] es un lenguaje orientado a objetos, que fue diseñado pensando en la ejecución de programas en un entorno distribuido heterogéneo, formado por sistemas empotrados con necesidades de tiempo real [GM96]. En este entorno, se necesitaba un lenguaje robusto, que garantizara la seguridad y que permitiera que los programas fueran totalmente independientes de la arquitectura, para poderlos ejecutar distribuidamente en cualquier nodo de la red heterogénea.

Sin embargo, las características de Java lo han convertido en un lenguaje de programación muy atractivo y su uso se ha extendido a todo tipo de aplicaciones y en entornos de ejecución muy diferentes al que originó su nacimiento. Por ejemplo, podemos encontrar proyectos que trabajan en el uso de Java para aplicaciones de comercio [CHL+00], minería de datos [MMGL99] o aplicaciones científicas [MMG+00].

La plataforma de ejecución de Java [Kra96] está formada por dos componentes principales: la *Java Virtual Machine* (JVM), que es la máquina virtual en la que se basa el modelo de ejecución, y la *Java Application Programming Interface* (Java API), que constituye el interfaz, independiente del SO, que ofrece a los programadores los servicios necesarios.

El modelo de ejecución de Java se basa en la JVM [LY99], que es la encargada de aislar los programas ejecutables de la máquina real donde se ejecutan, y garantiza que un binario de un programa Java se podrá ejecutar en cualquier sistema que disponga de la plataforma de ejecución de Java (ver figura 1.1).

Así pues, el compilador de Java (en la figura 1.1, *javac*), genera para cada fichero fuente (*.java*) el fichero correspondiente (*.class*) con el resultado de traducir el código escrito en Java a un lenguaje intermedio (*bytecode*), que es un código binario independiente del sistema real.

En tiempo de ejecución, la plataforma de Java carga en memoria el bytecode que constituye el binario del programa y, a medida que el programa invoque funciones de la Java API,

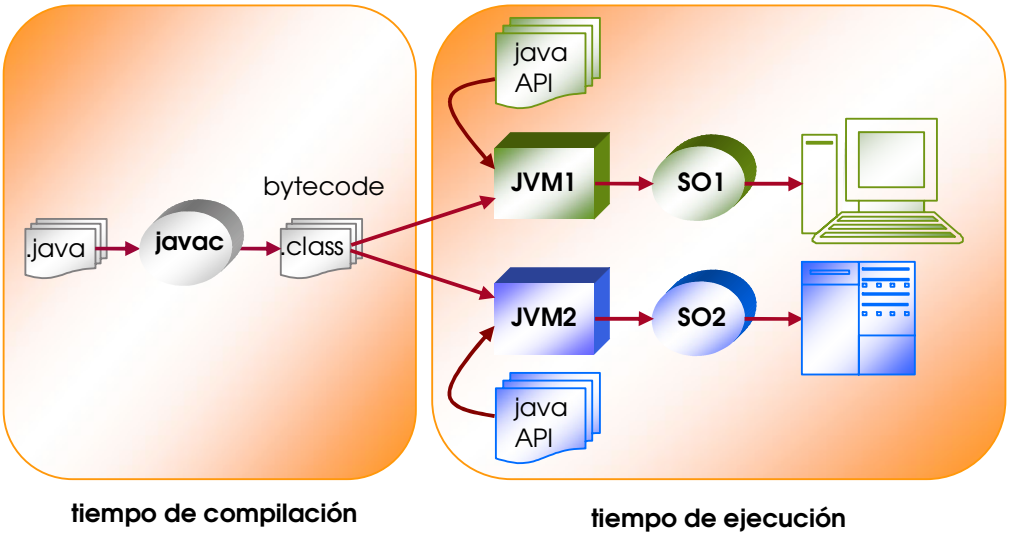

Figura 1.1 Modelo de ejecución de Java

se cargan dinámicamente los ficheros que las codifican. La JVM se encarga de ejecutar ese bytecode, convirtiéndolo en las instrucciones correspondientes al SO y a la máquina real que se está utilizando. En este punto hay que mencionar que la JVM ofrece dos posibilidades para realizar esta conversión: basarla en la interpretación del bytecode o utilizar un compilador al vuelo (compilador *just-in-time* (JIT)) que, en tiempo de ejecución, genera el código máquina correspondiente al bytecode.

1.2.2 Gestión de memoria en el entorno de ejecución de Java

La gestión del recurso memoria en el entorno de ejecución de Java está dividida entre los dos niveles de ejecución. En el nivel de usuario, la JVM toma todas las decisiones sobre la organización del espacio lógico de los programas Java, mientras que, en el nivel de sistema, el SO continúa implementando el resto de tareas relacionadas con la gestión de memoria como, por ejemplo, la protección entre los diferentes programas en ejecución o la implementación del mecanismo de memoria virtual.

Hay que destacar que la gestión del espacio de direcciones que ofrece la plataforma de ejecución de Java, constituye una de sus características más valoradas por los programadores ya que es simple, fiable, robusta, automática y, en muchos aspectos transparente al programador (en el capítulo 2 ofrecemos una descripción detallada de esta gestión). Por ejemplo, el programador no necesita controlar explícitamente las posiciones que ocupan sus datos en memoria ni existe el tipo de dato puntero a memoria, con lo cual desaparece una de las mayores causas de error de programación y una de las más complicadas de detectar y solucionar. También se evita que el programador tenga que gestionar la liberación de la memoria ocupada por los datos que ya no necesita, ya que esta liberación se hace automáticamente mediante la técnica de recolección de basura (*garbage collection*).

Sin embargo, dado que Java se está utilizando para desarrollar aplicaciones con características muy distintas a las que en un principio se consideraban, es necesario evaluar la influencia que esta gestión tiene sobre el rendimiento de estos nuevos tipos de programas.

Por ejemplo, durante el desarrollo inicial de esta plataforma se consideraba un tipo de programas que utiliza un conjunto de datos de trabajo (*working set*) pequeño. Es decir, son programas que pueden completar su ejecución sin utilizar el mecanismo de memoria virtual y, por lo tanto, con una participación reducida del SO en lo referente a la gestión de la memoria. Por este motivo, los esfuerzos durante el diseño de la plataforma se dedicaron a ofrecer al programador una buena gestión del espacio lógico de direcciones, sin considerar la posible interacción con las decisiones sobre gestión de memoria tomadas por el sistema operativo.

1.2.3 Entorno de trabajo

Para realizar este trabajo, en primer lugar es necesario determinar la plataforma a utilizar para el desarrollo de nuestras propuestas. El principal condicionante para esta selección es que necesitamos tener acceso al código fuente de los componentes de la plataforma, para poder modificarlos libremente e incorporar nuestras propuestas. Por este motivo, hemos seleccionado un SO y una máquina virtual cuyo código es de libre distribución. En concreto trabajamos sobre el SO Linux y sobre la versión del entorno de desarrollo de programas Java (*J2SDK Standard Edition*) que Sun distribuye para Linux. Durante el

desarrollo de este trabajo hemos utilizado diferentes versiones tanto del SO como de la JVM. En cada capítulo describimos la versión utilizada para el trabajo que ese capítulo presenta.

En cuanto al ámbito de los programas de prueba, nos hemos centrado en los programas de cálculo científico. Así, en las diferentes fases de este trabajo, hemos utilizado programas de prueba suministrados por el grupo de trabajo JavaGrande [JGF01] y códigos que forman parte del conjunto de programas Java NAS [FSJY03]. Además hemos implementado un benchmark sintético, para utilizar cuando el análisis de nuestras propuestas hacía necesario un código sencillo que nos permitiera un mayor control sobre su ejecución.

1.3 OBJETIVOS Y PLANTEAMIENTO DE ESTE TRABAJO

En este trabajo proponemos introducir en los entornos virtualizados de ejecución la posibilidad de ofrecer a los programas una gestión de recursos específica, aprovechando el conocimiento que tienen las máquinas virtuales sobre el comportamiento de los programas, haciendo que las decisiones de gestión sean compartidas entre el sistema operativo y las máquinas virtuales.

La tesis que vamos a demostrar en este trabajo es la siguiente:

> **Es posible aprovechar las características de las plataformas de ejecución basadas en máquinas virtuales para permitir que soporten de forma eficaz la ejecución de programas con alto consumo de recursos, sin renunciar a las ventajas que ofrece su modelo de ejecución.**

Para ello, nuestro objetivo principal es demostrar que:

> **El modelo de ejecución basado en máquinas virtuales presenta la oportunidad de ofrecer a los programas una gestión de recursos específica, que se adapta a su comportamiento dinámico sin comprometer la fiabilidad del sistema.**

El caso de estudio que hemos seleccionado es la gestión de memoria en el entorno de ejecución de los programas Java. Por lo tanto, para incorporar en el entorno de ejecución de Java una gestión de memoria adaptada al comportamiento de los programas, el primer paso es analizar el uso que estos programas hacen de la memoria y cómo se ven afectados por las decisiones que se toman sobre la gestión de este recurso. Las conclusiones de este análisis deben indicar si es posible mejorar el rendimiento de los programas, modificando las políticas de gestión de memoria existentes o incluyendo alguna nueva política más adecuada. En cualquier caso, hay que garantizar que las decisiones que se tomen tendrán en cuenta el comportamiento particular de cada programa y, además, se respetará la fiabilidad del sistema.

Así pues, los objetivos aplicados de este trabajo son:

- **Analizar detalladamente la gestión de memoria que ofrece el entorno de ejecución de Java y las posibles interacciones entre las tareas que se desarrollan en cada nivel de ejecución** (ver capítulo 2).
- **Evaluar el uso de la memoria que hacen los programas Java y la influencia que tiene la gestión de memoria sobre su rendimiento** (ver capítulo 3).
- **Proponer los cambios y las políticas de gestión de memoria adecuadas para mejorar el rendimiento de los programas** (ver capítulo 3).
- **Introducir estos cambios en el entorno de ejecución de Java** (ver capítulos 4, 5 y 6):
  - **Diseñando la estrategia que deben seguir las políticas de gestión que se van a introducir en el entorno de ejecución, para que sus decisiones se puedan adaptar al comportamiento de cada programa**.

- **Implementando estas políticas mediante un código eficiente, que favorezca la eficacia de las políticas de gestión.**
- **Y evaluando la implementación para demostrar que el entorno de ejecución modificado con nuestras propuestas realmente mejora el rendimiento de las aplicaciones.**

2

GESTIÓN DE MEMORIA PARA LOS PROGRAMAS JAVA

En este capítulo presentamos la gestión de memoria que ofrece el entorno de ejecución de Java. Esta gestión aparece distribuida entre los dos niveles de ejecución (nivel de usuario y nivel de sistema), ya que la JVM se encarga de implementar la organización y el mantenimiento del espacio de direcciones de los programas, mientras que el SO conserva el control del resto de tareas de gestión de memoria.

En la sección 2.1 describimos las tareas involucradas en la gestión de memoria y cómo se reparten entre los dos componentes principales del entorno de ejecución (SO y JVM). A continuación la sección 2.2 describe la implementación que hace el SO de las tareas de gestión de memoria que lleva a cabo y la sección 2.3 describe la implementación de las tareas que ejecuta la JVM. Cierra el capítulo las conclusiones que hemos extraído del estudio que sobre la gestión de memoria del entorno.

2.1 DIVISIÓN DE TAREAS ENTRE EL SO Y LA JVM

Las tareas de gestión de memoria de un proceso se pueden separar en dos grandes grupos.

Por un lado, es necesario gestionar el espacio lógico de direcciones del proceso. Es decir, hay que permitir la reserva y la liberación de memoria en tiempo de ejecución, y mantener qué regiones del espacio de direcciones son válidas y qué permisos de acceso tienen asociados.

Por otro lado, hay que implementar la traducción de direcciones, manteniendo la relación entre las direcciones lógicas y las posiciones físicas que realmente ocupan. Como parte de esta tarea se implementa la protección entre los accesos a memoria física de los diferentes procesos que están en ejecución y se implementa el mecanismo de memoria virtual. Este mecanismo permite que parte de los espacios de direcciones de los procesos en ejecución se mantengan fuera de memoria física, en el almacenamiento secundario (área de *swap*), ya que se garantiza su carga en memoria física siempre que el proceso necesite acceder a esas zonas.

En el caso del entorno de ejecución de Java, la JVM releva al SO de las tareas de gestión del espacio lógico de direcciones de los programas. Para ello, al principio de la ejecución le pide al SO que valide el tamaño máximo de memoria que se va a dedicar para almacenar los objetos del programa. A partir de ese momento, la JVM gestiona esa memoria, decidiendo las posiciones que ocupan en cada momento los objetos dentro de este espacio y liberando la memoria ocupada por objetos que dejan de estar en uso. Sin embargo, el SO sigue implementando la gestión de la traducción de direcciones y del mecanismo de memoria virtual, lo cual garantiza la fiabilidad e integridad del sistema.

Esto significa que la gestión de memoria de un proceso aparece separada entre los dos niveles de ejecución: la gestión del espacio lógico de direcciones se lleva a cabo en el nivel de usuario y la asociación con el espacio físico continúa ejecutándose en el nivel de sistema (ver figura 2.1).

Hay que destacar que ambos niveles de gestión se ejecutan independientemente, aunque las decisiones y la ejecución de uno de ellos pueden afectar al rendimiento de las decisiones del otro nivel. En el diseño de la JVM no se tuvo en cuenta el efecto de esta interacción ya que se pensaba en un entorno de ejecución en el que apenas se utilizaba el mecanismo de memoria virtual. Recordemos que el objetivo inicial era ejecutar programas de tiempo real para dispositivos empotrados, y este tipo de programas no deben usar la memoria virtual, porque eso ralentizaría su ejecución y podría ser que no cumplieran con las restricciones de tiempo. Por ese motivo, es necesario estudiar el efecto de esta interacción para aquellos programas Java que sí necesitan utilizar el mecanismo de memoria virtual.

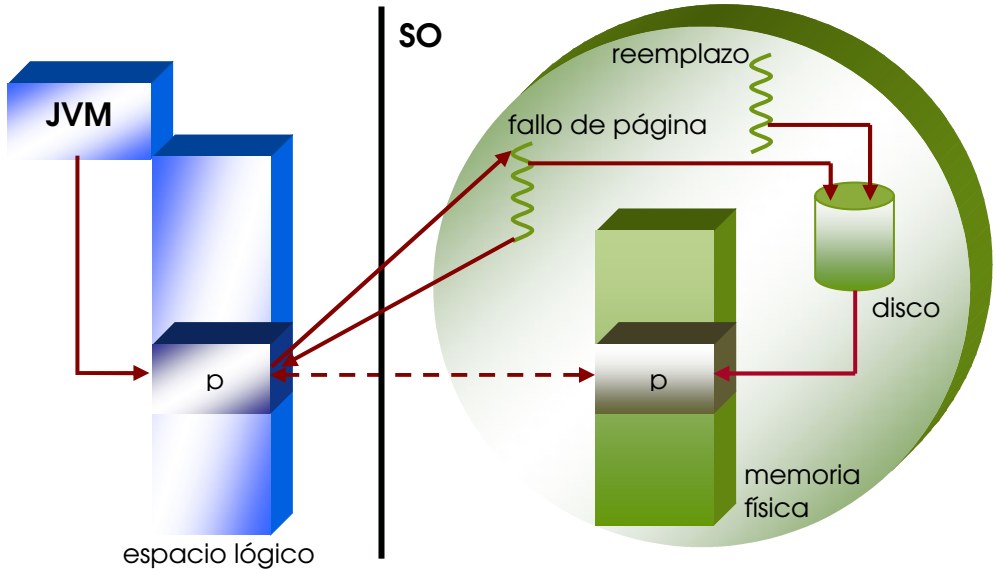

Figura 2.1 Gestión de memoria en el entorno de ejecución de Java

A continuación vamos a describir brevemente los aspectos más relevantes de ambos niveles de gestión en nuestro entorno de trabajo, así como las posibles interacciones que pueden aparecer entre ellos.

2.2 GESTIÓN DE LA MEMORIA VIRTUAL EN LINUX

La gestión de memoria virtual en Linux se basa en paginación bajo demanda. Es decir, si un proceso accede a una página de su espacio lógico de direcciones que no tiene asociada una página física, el hardware genera una excepción (fallo de página) que Linux resuelve para que el proceso pueda completar el acceso a memoria.

Para poder resolver el fallo de página, Linux debe reservar una página física y asociarla a la página lógica, actualizando la tabla de páginas del proceso. Además, si la página lógica contiene información, entonces es necesario recuperar esa información y escribirla en la página física reservada. Hay que decir que el método utilizado para recuperar los datos de la página depende del tipo de región al que pertenezca y de si se trata del primer acceso o

no. Por lo tanto, en función de las tareas involucradas en la resolución del fallo de página, podemos distinguir tres situaciones distintas:

- Fallo debido al primer acceso a una página de memoria anónima: las regiones de memoria anónima son las que no están respaldadas por ningún dispositivo lógico y su contenido será el almacenado por el programa durante la ejecución. Es decir, en el momento de la creación se consideran vacías. Por lo tanto, para resolver este tipo de fallo de página, Linux sólo reserva una página física libre y la asocia a la página lógica.
- Fallo debido al primer acceso a una página de memoria *mapeada*: en Linux es posible mapear el contenido de un dispositivo lógico sobre una región del espacio de direcciones y acceder a ese dispositivo a través de la región que lo mapea. Para este tipo de regiones, el fallo de página debido a un primer acceso debe localizar los datos correspondientes en el dispositivo lógico asociado para cargarlos en la página física que se reserve.
- Fallo debido al acceso a una página ya en uso: cuando no se trata del primer acceso a una página, significa que esa página lógica ha tenido que ser expulsada al área de swap. Por lo tanto, es necesario localizar los datos en el área de swap y cargarlos en memoria física.

Hay que destacar que el tiempo dedicado a resolver un fallo de página es muy diferente si es necesario realizar una carga de información en memoria física (como los dos últimos casos) o si es posible resolverlos sin ningún acceso a un dispositivo lógico (como en el caso del primer fallo de página sobre memoria anónima). Por este motivo, Linux denomina fallos de página *hard* o *major* a aquellos que involucran una operación de carga, y fallos de página *soft* o *minor* a los que se resuelven sin esa carga asociada.

En este punto, es interesante mencionar que, además del primer acceso a una página de memoria anónima, hay otras excepciones de fallo de página generadas por el hardware que Linux clasifica como fallos soft. Son accesos que tampoco requieren la carga de la información, bien sea porque ya está presente en memoria física y sólo es necesario actualizar la tabla de páginas, o porque ya se encuentra en proceso de ser cargada debido a

una solicitud previa. Por ejemplo, Linux implementa la optimización *copy on write* para gestionar el uso de la copia de memoria que se da, por ejemplo, durante la creación de nuevos procesos. Esta técnica se basa en permitir la compartición de la memoria copiada mientras ningún proceso intente escribir en la región, momento en el que la copia se hace realmente efectiva. Para detectar esta situación, Linux marca en la tabla de páginas que las páginas involucradas son de sólo lectura. De esta manera, un acceso de escritura sobre una de esas páginas provoca una excepción de fallo de página que, si la página ya está cargada, se puede resolver efectuando la copia en una nueva página física y actualizando la tabla de páginas con la nueva asociación y con el permiso de escritura ya activado. Es decir, es un fallo de página que no necesita ninguna carga para ser resuelto y que, por tanto, forma parte de los fallos de página soft.

2.2.1 Algoritmo de reemplazo

Un aspecto importante relacionado con la memoria virtual es el algoritmo de reemplazo. Este algoritmo es el encargado de decidir qué páginas del espacio lógico de direcciones de un proceso son expulsadas al área de swap. El objetivo es mantener en memoria física las páginas activas, es decir, las páginas que los procesos están utilizando. De esta manera, se intenta minimizar el número de fallos de página por accesos a páginas que se encuentran en el área de swap, ya que el acceso a disco necesario para solventar estas excepciones ralentiza la ejecución de los programas.

El algoritmo utilizado por Linux es una aproximación del LRU (*Least Receantly Used*). El algoritmo LRU se basa en el hecho de que el pasado reciente de un proceso aproxima su futuro inmediato. Así, selecciona como páginas víctimas aquellas que hace más tiempo que no se referencian, suponiendo que serán las que el proceso tarde más en necesitar. Para poder implementar este algoritmo sería necesario actualizar la información sobre el uso de la memoria para cada referencia a una página, lo que añadiría un alto coste al tiempo de ejecución. Por ese motivo, los sistemas reales implementan algoritmos que aproximan el comportamiento del LRU. En el caso de Linux, divide la memoria en dos listas diferentes: las páginas activas y las páginas inactivas. Cada vez que es necesario liberar páginas, primero recorre la lista de páginas inactivas, reactivando aquellas páginas que han sido referenciadas (se mueven a la lista de páginas activas) y liberando las páginas que no lo

han sido (ver figura 2.2). Hay que decir que, si antes de completar la liberación de una página, el proceso intenta referenciarla de nuevo, el fallo de página provocado es un fallo de página soft que Linux resuelve asociando de nuevo la página lógica con la página física que conserva los datos. Una vez finalizado el tratamiento de la lista de páginas inactivas, el algoritmo de reemplazo recorre la lista de páginas activas actualizando el contador de actividad de cada página, que sirve para decidir cuándo se mueve la página a la lista de páginas inactivas: si la página ha sido referenciada desde la última ejecución del algoritmo, se incrementa su contador de actividad y en caso contrario se decrementa.

Figura 2.2 Reemplazo de memoria en Linux

2.2.2 Prefetch de páginas

También con el objetivo de minimizar el número de fallos de página, Linux implementa una política muy simple de carga anticipada (*prefetch*). Esta política está limitada por la poca información que tiene el sistema sobre los accesos a memoria de los procesos, que no le permite implementar una predicción de accesos futuros elaborada. Por este motivo, el prefetch de Linux simplemente, para cada fallo de página, solicita del disco la carga de un número determinado de páginas consecutivas a partir de la que ha provocado el fallo. El número de páginas cargadas con antelación es un parámetro del sistema, global para

todos los procesos y configurable por el administrador de la máquina. Linux no actualiza la tabla de páginas del proceso con las páginas cargadas con antelación hasta que ocurre la primera referencia. Es decir, esta carga anticipada también involucra fallos de página soft. Hay que destacar que esta política sólo puede favorecer a las aplicaciones que acceden de forma secuencial al espacio de direcciones. Es más, el prefetch de páginas erróneas puede perjudicar el rendimiento de la máquina ya que se están cargando páginas no necesarias que pueden provocar la expulsión de páginas que sí son útiles para los procesos. Por este motivo, las versiones actuales de Linux sólo utilizan este mecanismo si el sistema de memoria no está sobrecargado.

2.3 GESTIÓN DEL ESPACIO LÓGICO DE DIRECCIONES EN JAVA

La JVM es la encargada de gestionar el espacio lógico de direcciones de los programas Java. En esta gestión hay que destacar tres aspectos importantes: la estructuración del *heap* del programa, la reserva de nuevos objetos y la liberación de objetos no necesarios.

El heap es la zona del espacio de direcciones donde la JVM almacena los objetos de los programas. Al inicio de la ejecución la JVM pide al SO que valide suficiente memoria lógica para soportar el tamaño máximo de heap que el programa puede utilizar, aunque inicialmente no sea necesario todo ese espacio. De esta manera, la JVM pasa a administrar ese espacio lógico, decidiendo en cada momento qué cantidad de memoria es adecuada para que el programa se ejecute. Tanto el tamaño máximo del heap como el tamaño inicial que se ofrece al programa son parámetros de la ejecución que puede decidir el usuario al lanzar el programa, aunque ambos tienen valores por defecto.

La reserva de memoria en Java se hace bajo petición de los programas. Cuando el programa crea un nuevo objeto, la JVM primero comprueba si hay suficiente espacio en el heap, y, en ese caso, busca na zona apropiada para el nuevo objeto. Esto significa que la JVM sabe exactamente las direcciones de memoria asociadas a cada objeto y, por lo tanto, las direcciones involucradas en cada acceso. Como consecuencia de la reserva de un nuevo objeto, es posible que la JVM decida hacer crecer el espacio utilizado por el heap, siempre

respetando los límites establecidos por el tamaño inicial y el tamaño máximo al inicio de la ejecución.

En cuanto a la liberación de memoria, está basada en el mecanismo de *garbage collection* y, por lo tanto, es automática y transparente al usuario [JL96a]. Cuando la JVM detecta que la ocupación del heap hace necesario liberar memoria, inicia el proceso de garbage collection, buscando aquellos objetos que ya no están en uso y liberando la memoria que ocupaban. Además, Java también ofrece a los programadores un *method* para provocar la ejecución explícita de la liberación de memoria.

Todas estas tareas de gestión pueden influir en el rendimiento de la memoria virtual de la máquina de dos posibles maneras. Primero, por las decisiones que se toman durante la gestión. Por ejemplo, agrupar en las mismas páginas aquellos objetos que se utilizan al mismo tiempo puede reducir el número de fallos de página de los programas. Y, segundo, por la propia ejecución de los algoritmos de gestión. Tanto el algoritmo de reserva como el algoritmo de liberación pueden implicar recorridos del heap, y estos recorridos, por un lado, pueden provocar nuevos fallos de página y, por otro lado, pueden alterar la información que utiliza el algoritmo de reemplazo de memoria virtual para seleccionar las páginas víctimas.

Por este motivo, es necesario estudiar la influencia de esta interacción en el rendimiento de los programas Java que requieren el uso de la memoria virtual. Si esta interacción es responsable de una penalización significativa sobre el rendimiento, entonces mejorar la interacción entre ambos niveles puede ser un camino para aumentar el rendimiento de estos programas, sin renunciar a las ventajas de la plataforma de ejecución.

A continuación describimos en detalle la implementación de las tareas de gestión en dos máquinas virtuales diferentes. Las dos máquinas que vamos a describir vienen suministradas por el entorno de desarrollo de programas Java de Sun que hemos utilizado (Java 2 SDK Standard Edition para Linux). Este entorno de ejecución permite que el usuario pueda seleccionar la JVM que quiere utilizar para la ejecución de sus programas. Estas máquinas virtuales son la JVM *classic*, versión 1.2.2 (sección 2.3.1), y la JVM *HotSpot*, versión 1.3.1 (sección 2.3.2).

2.3.1 Gestión del heap en la JVM *classic*

El heap que implementa la JVM *classic* está totalmente contiguo en memoria lógica y está dividido en dos zonas separadas. En la primera se sitúan las cabeceras de los objetos (*handles*) y en la segunda los objetos propiamente dichos (ver figura 2.3). Cada cabecera contiene las características del objeto al que representa y la dirección donde se encuentra. Esta separación de cabeceras y datos facilita los posibles cambios de posición de los objetos supervivientes, que pueden ser necesarios para eliminar la fragmentación externa del heap, después de una liberación de memoria. Sin embargo, acceder a un objeto en esta JVM requiere como mínimo dos accesos a memoria: uno para obtener su dirección y otro para acceder realmente a los datos.

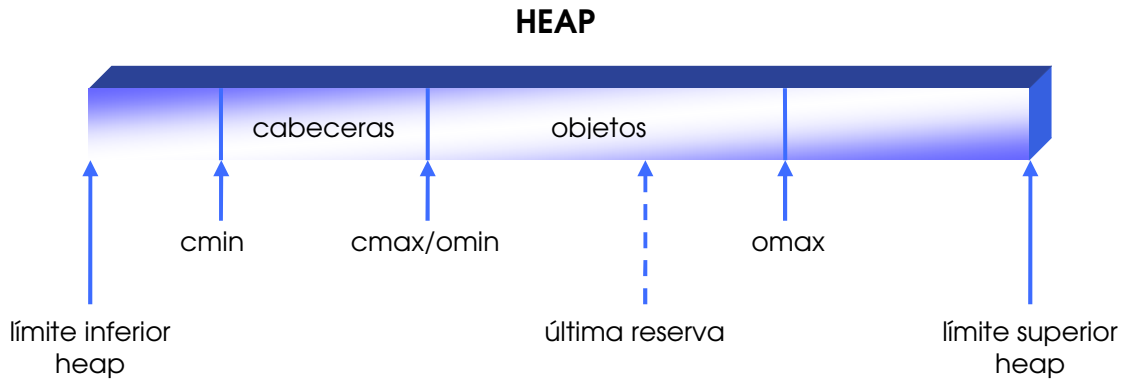

Figura 2.3 Organización del heap en la JVM *classic*

La reserva de memoria se hace de forma secuencial y cíclica. Cuando el programa instancia un nuevo objeto, la JVM busca en el heap suficiente memoria contigua para situarlo, empezando en el punto donde finalizó la última reserva de memoria. La búsqueda acaba cuando se encuentra una región de tamaño suficiente o cuando se alcanza el punto inicial de la búsqueda. Este método cíclico tiene como objetivo reaprovechar los huecos aparecidos por las liberaciones previas de objetos en desuso. Hay que destacar que este algoritmo, en el peor de los casos recorre toda la zona de datos y, por lo tanto, es una potencial fuente de fallos de página.

La liberación de memoria se hace mediante el algoritmo de garbage collection *mark and sweep with compact* [JL96b]. Este algoritmo se divide en tres fases principales:

- Fase de marcado (*mark*): durante esta fase se recorre la zona de handles para detectar y marcar aquellos objetos que están referenciados (objetos *vivos*) y que, por lo tanto, no se pueden liberar.
- Fase de barrido (*sweep*): una vez marcados los objetos vivos, el algoritmo accede a la zona de objetos para liberar la memoria que ocupan los objetos que no se están marcados (objetos *muertos*). Nótese que en esta fase se está accediendo a zonas de memoria que en realidad ya no están en uso y, por lo tanto, se está alterando la información sobre el comportamiento del programa que recibe el SO.
- Fase de compactación (*compact*): una vez liberada la memoria ocupada por los objetos muertos, el algoritmo comprueba si el grado de fragmentación externa del heap aconseja compactar la memoria libre, para evitar que el rendimiento del algoritmo de reserva degenere. El algoritmo de compactación puede involucrar varios recorridos de la zona de objetos, ya que intenta mover cada objeto vivo al primer hueco libre de la zona de objetos, con suficiente espacio para soportarlo. Por lo tanto, esta fase también puede alterar la información sobre el uso de la memoria que hace el programa. Además, al recorrer el heap varias veces, es una potencial fuente de fallos de página, que se añadirían a los fallos de página propios del código del programa.

Una vez finalizada la liberación de objetos muertos, la JVM comprueba si es conveniente aumentar el tamaño utilizable del heap, siempre respetando los límites del tamaño máximo. Esta última operación es muy simple, ya que únicamente requiere modificar el valor de los límites del heap.

En la figura 2.4 representamos el algoritmo ejecutado para liberar memoria cuando no es posible satisfacer la reserva de memoria para un nuevo objeto.

```
reserva_ok = reserva(&ultima_reserva, tamaño);
if (!reserva_ok) {
    marcar_objetos_refereciados();
    compactar = barrer_objetos_no_marcados();
    ultima_reserva = omin;
    if (compactar) {
        compactar_heap();
        ultima_reserva = ultimo_objeto();
    }
    if (necesita_expandir)
        expandir_heap();
    reserva_ok = reserva(&ultima_reserva, tamaño);
}
```

Figura 2.4 Algoritmo de la liberación de memoria: marcado y barrido con compactación

2.3.2 Gestión del heap en la JVM *HotSpot*

Cuando se implementó la JVM *HotSpot*, la plataforma Java ya se estaba utilizando para una amplia gama de aplicaciones diferentes. Por este motivo se intentó favorecer a aquellas aplicaciones con mayor consumo de recursos. Este esfuerzo se ve reflejado en la implementación del modelo de memoria y del garbage collector [Sun01].

En esta versión de la JVM los objetos ya no aparecen separados de sus cabeceras, lo que elimina el problema de la doble indirección necesaria para acceder a los datos. Además se ha conseguido reducir la cabecera de la mayoría de los objetos a sólo dos bytes, en lugar de los tres bytes necesarios en la versión *classic*, lo que disminuye la cantidad de memoria necesaria para albergar la misma cantidad de objetos.

En la versión 1.3.1 de *HotSpot*, que es la que hemos utilizado en este trabajo, tanto la organización del heap como el método de reserva de nuevos objetos depende en gran medida del algoritmo de garbage collection utilizado. En esta versión, el usuario puede escoger entre dos garbage collectors generacionales diferentes.

Los garbage collectors generacionales [SMM99, WLM92] están pensados para favorecer la localidad de referencia y se basan en la suposición de que la mayoría de objetos Java tienen una vida corta. Por este motivo, la zona de objetos se divide en dos partes: la zona de objetos de reciente creación (*generación joven*) y la zona de objetos donde se van almacenando los objetos que llevan más tiempo en ejecución (*generación vieja*) (ver figura 2.5). Así, estos garbage collectors acceden frecuentemente a la zona de los objetos jóvenes para eliminar los que ya no están en uso (*minor collections*). Cada vez que un objeto sobrevive a una recolección se incrementa su edad. Cuando la edad de un objeto llegue a un determinado límite se considera que es un objeto viejo y se mueve a la zona de objetos viejos. Sólo cuando una minor collection no consigue liberar suficiente memoria, se recurre a una limpieza más profunda de la zona de objetos, que considera también la zona de objetos viejos (*major collection*).

Por lo tanto, si se cumple la premisa en la que se basan estos garbage collectors y la mayoría de objetos mueren jóvenes, entonces se reduce la cantidad de memoria que el garbage collector tiene que acceder para liberar los objetos muertos. Es decir, se limitan las posibles interacciones con el mecanismo de memoria virtual.

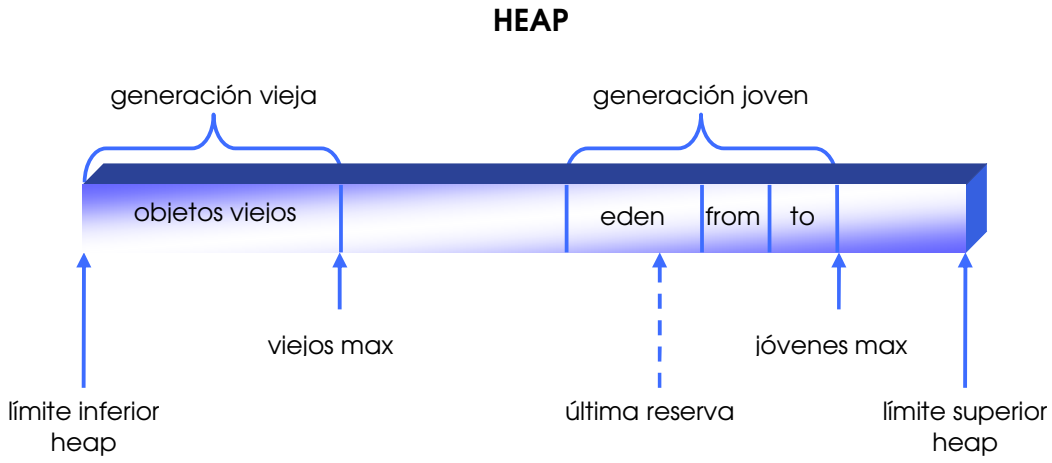

Figura 2.5 Organización del heap en la JVM *HotSpot*

Los dos garbage collectors que ofrece la versión 1.3.1 de *HotSpot* coinciden en el algoritmo utilizado para las minor collections, que es *mark and copy*. Para ello, la zona de objetos jóvenes a su vez está divida en 3 zonas diferentes (ver figura 2.5). La zona que se usa para crear objetos nuevos (*eden*), una zona que se utiliza para almacenar los supervivientes y en la que se reservan objetos que no han cabido en el *eden* (*from*) y una zona que se utiliza durante la copia (*to*). Durante la minor collection se van copiando en la zona *to* todos los objetos de *eden* y *from* que aún están vivos. Una vez finalizada la liberación, la zona *from* y la zona *to* intercambian los papeles, mientras que la zona *eden* queda vacía y preparada para alojar a los nuevos objetos que se creen a continuación (ver figura 2.6).

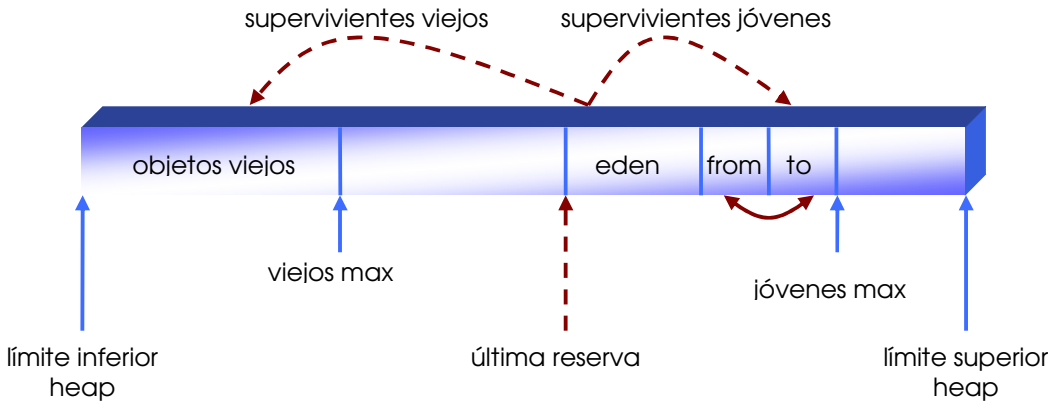

Figura 2.6 Liberación de memoria de la generación joven en la JVM *HotSpot*

Las diferencias entre los dos garbage collectors ofrecidos radican en el método usado para las major collections. Aunque los dos métodos usan el algoritmo *mark and sweep with compact*, explicado en la sección 2.3.1 (ver figura 2.4), ambos se diferencian en la zona sobre la que se ejecuta el algoritmo. En el garbage collector que se utiliza por defecto, cada major collection se hace sobre toda la zona de objetos viejos. La alternativa a esta opción, es solicitar una limpieza *incremental*, en cuyo caso cada major collection se hace sólo sobre una porción de la zona de objetos viejos. Esta alternativa está orientada a disminuir el tiempo de pausa de los programas, necesario mientras dura la limpieza de objetos. Para utilizar la opción del garbage collector incremental, es necesario dividir el

heap en diferentes zonas (*trenes*), de manera que cada ejecución de una major collection analiza una *tren* diferente.

En cuanto a la reserva de nuevos objetos siempre se hace en la zona de la generación joven, de forma consecutiva. La copia de los objetos supervivientes también se hace a continuación del último objeto. Hay que decir que, durante la copia de los objetos jóvenes supervivientes, se intenta situar próximos en el heap los objetos que están relacionados, es decir, los objetos que se referencian entre sí, con el objetivo de agrupar los objetos que tienen una alta probabilidad de ser accedidos con proximidad en le tiempo. En cuanto a los objetos que deben ser movidos a la zona de los objetos viejos, también se copian a continuación del último objeto de la zona. En el caso del garbage collector incremental, se tiene que decidir además si el objeto que pasa a la generación vieja se incluye en el último *tren* existente, o si es necesario crear un *tren* nuevo (esta decisión afecta a la cantidad de objetos que se analizan en cada major collection).

2.4 CONCLUSIONES

En este capítulo hemos presentado brevemente cómo se gestiona el uso de la memoria en el entorno de ejecución de Java. En este entorno, el SO sigue siendo el responsable de gestionar la asociación entre las direcciones del espacio lógico del programa y la posición física que ocupan. Sin embargo, la JVM es la encargada, desde el nivel de usuario, de gestionar el espacio lógico de direcciones de los programas de forma transparente al SO. Esta separación de la gestión de memoria entre los dos niveles de ejecución tiene como ventaja que la JVM puede adaptar las decisiones sobre el espacio de direcciones al comportamiento del programa, intentando, por ejemplo, favorecer la localidad de referencia del programa. Además, al no requerir la participación del sistema en cada nueva reserva, puede mejorar el rendimiento de los programas que necesiten crear un gran número de objetos durante su ejecución. Sin embargo, es necesario estudiar la interacción que tienen entre sí las decisiones tomadas de forma independiente en los dos niveles de ejecución, ya que, durante el diseño del lenguaje Java, no se tuvo en cuenta el efecto de esta interacción. Hay que decir que, las versiones más actuales de JVM, como la JVM *HotSpot*, tienen una implementación más cuidadosa de la gestión del espacio lógico, que pretende favorecer el

rendimiento de la memoria virtual. A pesar de ello, en el trabajo relacionado no existe ningún estudio detallado sobre el efecto que tiene esta separación de la gestión de memoria entre los dos niveles sobre el rendimiento de los programas que hacen un uso intensivo del mecanismo de memoria virtual. En el capítulo 3 presentamos un análisis minucioso de varios programas que necesitan del uso de la memoria virtual para poder completar su ejecución, y vemos cómo se puede mejorar el rendimiento de este mecanismo desde la gestión de memoria implementada como parte de la JVM.

3

EVALUACIÓN DEL USO DE LA MEMORIA VIRTUAL DE LOS PROGRAMAS JAVA

En este capítulo presentamos la evaluación que hemos hecho del uso de la memoria de algunos programas Java que tienen un consumo importante de recursos. El objetivo de esta evaluación es comprobar si estos programas se ven afectados por el rendimiento de la memoria virtual y obtener las causas de esta influencia. Además, queremos determinar si es posible mejorar el rendimiento de los programas Java mediante alguna política de gestión de memoria que aproveche las características propias del entorno de ejecución de Java.

Para ello hemos contado y clasificado los fallos de página provocados por los programas, así como el tiempo involucrado en estos fallos de página. De esta manera podemos cuantificar la penalización que el uso de la memoria virtual añade al rendimiento de los programas. Nuestra clasificación nos permite distinguir entre los fallos de página provocados por el propio código del programa y los fallos de página debidos a la ejecución del código de gestión. Además, para completar la caracterización y así acotar nuestro estudio, hemos separado los fallos de página en función de la zona del espacio de direcciones accedida (ver sección 3.2).

Por otro lado, también hemos obtenido una cota máxima del rendimiento de estos programas, simulando su ejecución sobre un sistema de gestión de memoria perfecto. Esta cota, aunque sea inalcanzable, nos permite estimar si existe margen de mejora en el sistema de memoria (sección 3.3).

Por último, hemos hecho un análisis del tipo de objetos que utilizan los programas (sección 3.4). El objetivo de este análisis es completar la información sobre el uso de la memoria que hacen los programas, para así facilitar las propuestas de mejora de la gestión de memoria.

3.1 ENTORNO DE TRABAJO

En esta sección vamos a describir el entorno de trabajo que hemos utilizado para la evaluación del uso de la memoria.

Como ya se ha explicado en el capítulo 1 el SO sobre el que hemos desarrollado este trabajo es Linux. La versión del kernel que hemos utilizado para esta evaluación es la 2.2.12. En cuanto a la máquina virtual de Java, hemos utilizado tanto la JVM *classic* como la JVM *HotSpot*, ambas suministradas por Sun como parte del entorno de desarrollo *J2SDK Standard Edition*. Inicialmente, hemos evaluado la influencia de la memoria virtual en el rendimiento de los programas utilizando la JVM *classic* (versión 1.2.2). Una vez hecha esta evaluación hemos comparado los resultados con la ejecución sobre la JVM *HotSpot* (versión 1.3.1), lo que nos ha permitido validar las conclusiones que habíamos obtenido sobre la JVM *classic*. Para ambas máquinas virtuales, hemos utilizado la versión basada únicamente en la interpretación de los programas, sin utilizar compilación al vuelo, para facilitar la introducción en las JVM de nuestras herramientas de medida.

Todos los experimentos de este evaluación los hemos ejecutado sobre un PC con un procesador Pentium III a 500 Mhz con 128Mb de memoria física.

3.1.1 Programas de prueba

El grupo JavaGrande [JGF01] trabaja sobre el uso de Java para computación de altas prestaciones. Este grupo suministra un conjunto de programas de prueba, para que se utilicen en la comparación del rendimiento de entornos Java cuando ejecutan programas con gran consumo de recursos (*aplicaciones grandes*) [BSW+00].

En este conjunto de programas hay algunos núcleos de uso frecuente en computación de altas prestaciones y algunas aplicaciones completas. Para cada programa se suministran tres versiones, con datos de entrada de diferentes tamaños, identificadas, de menor a mayor tamaño, como SizeA, SizeB y SizeC.

De todo este conjunto de programas, hemos seleccionado los que tienen un consumo de memoria suficiente para que sea necesario el uso de la memoria virtual. La tabla 3.1 resume para cada uno de los benchmarks que hemos utilizado el tamaño de entrada correspondiente a cada versión (hemos marcado en negrita los datos correspondientes a la versión que utilizamos en los experimentos). A continuación describimos brevemente estos benchmarks junto con la versión que utilizamos en los experimentos (para una descripción completa se puede consultar [JGF01]).

Programas	**Unidades de la entrada**	**SizeA**	**SizeB**	**SizeC**
Crypt	Bytes	3.000.000	20.000.000	**50.000.000**
HeapSort	Enteros	1.000.000	5.000.000	**25.000.000**
MonteCarlo	Muestras	**10.000**	60.000	n.a.
FFT	Números complejos	**2.097.152**	8.388.608	16.777.216
Sparse	Doubles	50.000x50.000	100.000x100.000	**500.000x500.000**

Tabla 3.1 Tamaños de entrada de los benchmarks

- Crypt: Este es un kernel que implementa la encriptación y desencriptación de un array, usando el algoritmo *International Data Encryption* (IDEA). Para este kernel hemos seleccionado el tamaño mayor (SizeC), que trabaja sobre un array de 50.000.000 bytes.
- HeapSort: Este kernel ordena un array de enteros mediante el algoritmo de ordenación *heap*. El tamaño que hemos utilizado es el mayor (SizeC), que tiene como entrada un array de 25.000.000 enteros.
- MonteCarlo: Este programa consiste en una aplicación completa que implementa una simulación financiera basada en técnicas Monte Carlo, para derivar precios en base de unos datos históricos. Para esta aplicación sólo se suministran dos versiones y hemos elegido la más pequeña (SizeA), que genera 10.000 elementos de la serie temporal, porque consume suficiente memoria para presionar al sistema de memoria.

- FFT: Este kernel implementa una transformada de Fourier sobre un conjunto de números complejos. Hemos usado la versión más pequeña de este programa, ya que ésta consume suficiente memoria (SIZEA). Esta versión trabaja sobre 2.097.152 números complejos.
- SPARSE: Este kernel multiplica 200 veces una matriz dispersa por un vector denso. La matriz está almacenada comprimiendo las filas. Para este kernel hemos seleccionado la versión de mayor tamaño (SIZEC), en la que la matriz tiene 500.000 x 500.000 doubles.

3.2 EVALUACIÓN DEL RENDIMIENTO DE LA GESTIÓN DE MEMORIA

Hemos hecho tres grupos de experimentos para evaluar el rendimiento de la gestión de memoria actual. El primer grupo nos ha permitido obtener el tiempo que el sistema dedica a gestionar fallos de página para los programas de prueba. De esta manera, cuantificamos la relevancia que tiene la memoria virtual sobre el rendimiento de los programas. Los otros dos grupos de experimentos nos han servido para determinar el tipo de accesos más afectado por el uso de la memoria virtual, clasificándolos en función de la zona del espacio de direcciones accedida y del tipo de código en ejecución. El resultado de esta clasificación ha servido para acotar nuestros objetivos para la mejora de la gestión de memoria.

3.2.1 Clasificación de los fallos de página

Para poder realizar esta evaluación hemos definido la siguiente clasificación para los fallos de página.

En primer lugar, hemos separado los fallos de página que implican un acceso a disco (hard) de aquellos fallos de página que se pueden resolver sin ningún acceso a disco (soft). Ejemplos de tipos de acceso que provocan un fallo de página soft son el primer acceso a una página sin datos, un acceso que se refiere a una página que ya está siendo cargada (por ejemplo, como consecuencia de una operación de prefetch), o un acceso a una página que

está en proceso de ser expulsada al área de swap pero que todavía no se ha hecho efectiva la escritura en disco y, por lo tanto, es posible reactivarla. Hacemos esta distinción porque el tiempo necesario para resolver estos dos tipos de fallos de página es muy diferente, ya que el tiempo de acceso a disco domina claramente sobre el resto de tiempo necesario para la resolución de un fallo de página.

Para obtener la zona del espacio de direcciones más afectada por los fallos de página hemos distinguido entre tres zonas diferentes: zona de objetos, zona de cabeceras y accesos fuera del heap. En este último grupo se engloban fallos provocados por accesos al código del programa, a las pilas de los flujos de ejecución, etc.

Por último hemos separado los fallos de página provocados durante la ejecución de código de gestión de los provocados durante la ejecución del código del programa. Además, nuestra clasificación considera tres tipos de tareas de gestión y distingue entre los fallos de página provocados por la reserva de nuevos objetos, los provocados durante las fases de marcado y barrido de la liberación de objetos muertos y los que se dan durante la fase de compactación del heap.

3.2.2 Metodología para la recolección de datos

Para poder efectuar la clasificación de los fallos de página hemos necesitado modificar tanto el kernel de Linux como la JVM.

En el kernel de Linux hemos añadido los contadores que nos permiten separar los diferentes fallos de página y el tiempo empleado en su resolución, así como el código necesario para manipular esos contadores. Cada vez que un programa de prueba provoca un fallo de página, el kernel de Linux actualiza los contadores correspondientes a su tipo. Para calcular el tiempo invertido en resolver cada fallo de página hemos utilizado el contador de ciclos que posee el procesador sobre el que hemos medido los programas de prueba.

Sin embargo el kernel de Linux carece de suficiente información para distinguir entre todos los tipos de fallos de página que hemos definido y sólo es capaz de separar fallos de página soft y hard. Dado un fallo de página, Linux sólo conoce el proceso que lo ha

provocado, pero no sabe el tipo de código que estaba ejecutando. De la misma manera, Linux tampoco sabe el uso concreto que se le está dando a una dirección involucrada en un fallo de página, sólo puede saber el espacio de direcciones al que pertenece y el tipo de acceso permitido sobre esa dirección.

Por este motivo, además de los contadores, hemos añadido al kernel de Linux unas variables de configuración, que contienen la información necesaria para que Linux complete la clasificación de los fallos de página. Y hemos modificado la JVM para que se encargue de mantener estas variables con los valores adecuados.

Las variables de configuración contienen los siguientes datos:

- Identificador del proceso que se quiere monitorizar
- Límites de las zonas del espacio de direcciones. Para mantener estos límites hemos implementado seis variables diferentes, que nos permiten dividir el espacio de direcciones en las tres zonas distintas que queremos considerar. Dos de las variables delimitan la zona de objetos, otras dos variables limitan la zona que contiene las cabeceras de los objetos, y las otras dos separan el heap del programa del resto del espacio de direcciones.
- Tipo de código que está en ejecución, es decir, si se trata del código del programa o si se trata de código de alguna de las tareas de gestión (reserva de nuevos objetos, fase del garbage collector de marcado y barrido o fase de compactación del heap).

Cada vez que hay un fallo de página, el kernel de Linux consulta estas variables para determinar, primero, si lo ha provocado el proceso que se está monitorizando y, segundo, el tipo de fallo de página y, por lo tanto, los contadores asociados.

Como ya hemos explicado en el capítulo 2, la JVM decide la organización del espacio lógico de direcciones de los programas Java y, por lo tanto, conoce exactamente las direcciones que forman parte en cada momento de cada zona. Además la JVM también controla el tipo de código que se ejecuta en cada momento. Es decir, tiene toda la información necesaria para configurar la clasificación de fallos de página.

Para que la JVM pueda modificar las variables de configuración, es necesario definir un interfaz entre la JVM y el kernel que lo permita. Hemos implementado este interfaz mediante el mecanismo que Linux ofrece para interactuar con los gestores de dispositivos (ver figura 3.1). Este mecanismo permite que se incorporen fácilmente nuevos dispositivos así como el código específico que los manipula, sin necesidad de modificar el kernel del SO. El código específico forma parte de gestores externos, y su interfaz con los programas de usuario debe ser el genérico que define Linux para el acceso a cualquier dispositivo. Así, hemos creado en nuestro sistema de ficheros un nuevo dispositivo lógico que representa a los contadores y hemos implementado el código del gestor encargado de manipular este nuevo dispositivo. En la figura 3.1 mostramos un esquema del funcionamiento del mecanismo.

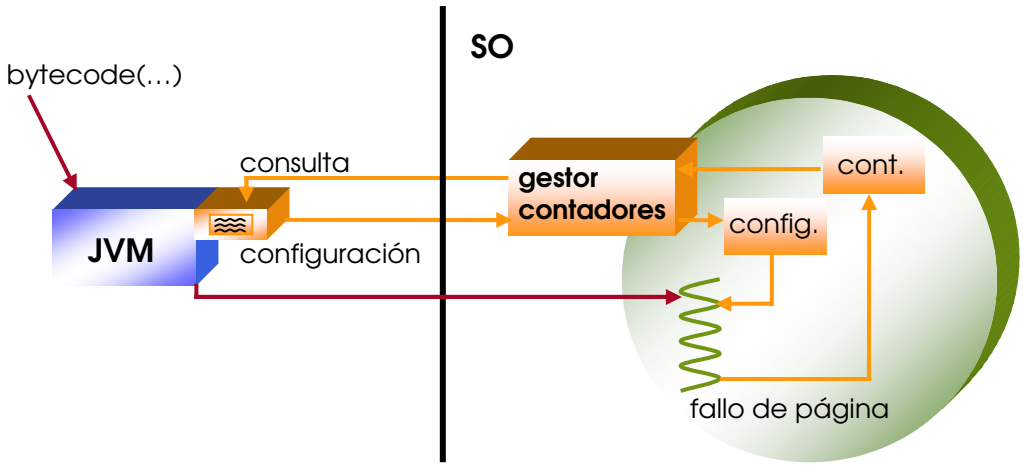

Figura 3.1 Esquema de funcionamiento del recuento de fallos de página

El gestor de los contadores implementa únicamente tres funciones del interfaz y que describimos brevemente a continuación.

- OPEN: recibe como parámetro el dispositivo lógico asociado a los contadores. Esta función inicializa los contadores y registra al proceso que la utiliza como proceso para el que se debe evaluar la memoria virtual:

```
countFd=open(" /dev/contadores ", O_RDONLY)
```

- IOCTL: esta llamada a sistema está pensada para la configuración de los dispositivos y permite que se utilice para diferentes tareas. El segundo parámetro sirve para indicar el tipo de tarea solicitada, y en el tercero se pueden pasar los datos necesarios para la tarea. En nuestro caso la utilizamos para las tareas descritas a continuación.

  - Activación y desactivación del recuento: estas opciones permiten suspender momentáneamente el recuento de fallos de página y reanudarlo cuando interese, para poder analizar, en caso necesario, zonas concretas de la ejecución de los programas. Para esta funcionalidad, el segundo parámetro de la llamada IOCTL contiene la operación que interesa realizar (activar o desactivar), mientras que el tercer parámetro no se utiliza:

    ```
    ioctl(countFd, UPDATE_ACT_STATE, 0)
    ```

  - Modificación de la variable que indica el tipo de código en ejecución: para llevar a cabo esta tarea, en el segundo parámetro de la llamada IOCTL indicamos que ha cambiado el tipo de código en ejecución, y en el tercer parámetro se informa sobre el código que inicia la ejecución:

    ```
    ioctl(countFd, UPDATE_CODE, code_type)
    ```

  - Modificación de las variables que contienen los límites del espacio de direcciones: mediante estas opciones es posible mantener actualizadas las variables del kernel que definen los límites de las zonas el heap. El segundo parámetro de la llamada IOCTL indica el límite que se quiere modificar y el tercer parámetro el nuevo valor para la variable:

    ```
    ioctl(countFd, LIMIT_ID, limit_addr)
    ```

  - Consulta de los valores de los contadores: la implementación que hemos hecho para la operación de consulta requiere que, mediante el segundo y el tercer parámetro, se indique el tipo de fallo de página para el que se quiere consultar los datos (zona del espacio de direcciones y código involucrado), y el tipo de información que se quiere obtener (número de fallos de página o tiempo de resolución), y la llamada devuelve como valor de retorno el contador solicitado. Para el caso de los contadores del número de fallos de página, cada llamada a IOCTL devuelve el valor de uno de ellos. Para los contadores de tiempo de resolución, para cada tipo de fallo de página es necesario utilizar dos llamadas a IOCTL, ya que el tipo de datos que devuelve esta llamada (*int*) es menor que el tipo de datos del contador

de tiempo (*long long*) y, por lo tanto, es necesario obtener por separado la parte baja y la parte alta de su valor.

```
ioctl(countFd, COUNTER_ID|ZONA_ID, CODE_ID)
```

- CLOSE: se libera el uso de los contadores:

```
close(countFd)
```

En la figura 3.2 mostramos un ejemplo del uso del interfaz para manipular los contadores. Cuando la JVM inicia la ejecución utiliza la llamada OPEN sobre el dispositivo lógico de los contadores para registrarse como proceso que va a ser monitorizado. Además debe utilizar la llamada IOCTL para completar la configuración del recuento, es decir, para registrar los límites de las zonas del heap y para indicar el tipo de código que está en ejecución. A partir de ese momento, utiliza la llamada IOCTL cada vez que es necesario modificar alguna variable de configuración. Por ejemplo, si se modifica alguno de los límites del heap, o cada vez que pasa a ejecutar código de gestión. Por último, antes de finalizar la ejecución, utiliza la llamada IOCTL para consultar el valor de los contadores y la llamada CLOSE para liberar el uso de los contadores.

Ejecución de los experimentos para la evaluación

Las medidas que presentamos para evaluar el rendimiento de la gestión de memoria son el resultado de la ejecución real de los programas de prueba sobre el entorno de ejecución. Por lo tanto, hay que tener en cuenta que existen factores externos que pueden influir en los resultados, como por ejemplo, la ejecución de los procesos de gestión del SO. Para suavizar los efectos de estos factores externos, presentamos la media de los resultados de varias ejecuciones. Además, para que todos los programas se ejecuten en las mismas condiciones, hemos reiniciado la máquina antes de cada batería de experimentos.

Hemos ejecutado los programas sobre diferentes tamaños de memoria física, para poder evaluar su comportamiento bajo diferentes condiciones de ejecución. Esto se puede conseguir a través de un parámetro de arranque del kernel de Linux, con el que se puede indicar la cantidad de memoria física disponible en el sistema. De esta manera, aunque la máquina disponía de 128Mb reales de memoria RAM, los experimentos se han hecho

```
    . . .
countFd = open(/dev/contadores ", O_RDONLY);
/* configuración de los límites de la zona de Objetos */
ioctl(countFd, TOP_OBJECTS, maxHeapAddr);
ioctl(countFd, BOTTOM_OBJECTS, minHeapAddr);
    . . .
    /* registrar el resto de límites del espacio de direcciones */
    . . .
/* el programa inicia la ejecución */
ioctl(countFd, UPDATE_CODE, program);
/* activa el recuento */
ioctl(countFd, UPDATE_ACT_STATE, 0);
    . . .
    /* si cambian los valores de configuración actualizar las variables */
    . . .
    /* al final de la ejecución desactivar recuento y consultar los valores de los contadores */
ioctl(countFd, UPDATE_ACT_STATE, 0);
hard_pf_heap_program = ioctl(countFd, HARD_PF | HEAP, program);
hard_pf_heap_management = ioctl(countFd, HARD_PF | HEAP, jvm);
. . .
hi_time = ioctl(countFd, HI_HARD_PF_TIME | HEAP, program);
lo_time = ioctl(countFd, LO_HARD_PF_TIME | HEAP, program);
hard_pf_time_heap_program = (hi_time≪32)—lo_time;
. . .
close(countFd);
    . . .
```

Figura 3.2 Ejemplo de uso del interfaz de manipulación de los contadores

además sobre 64Mb, 32Mb y 16Mb, lo que nos ha permitido analizar la influencia de la disminución de memoria sobre el comportamiento de los programas.

Otro parámetro de ejecución que puede influir en los resultados es el tamaño del heap. Como ya se ha dicho, el usuario tiene la opción de decidir el tamaño inicial y el tamaño máximo del heap, y la JVM adapta el tamaño actual al más adecuado para cada momento de la ejecución, respetando siempre los límites fijados. El tamaño máximo debe ser el suficiente para soportar todos los objetos que están en uso simultáneamente. Pero, además de esa restricción, el tamaño del heap también puede influir en el rendimiento de la gestión del espacio de direcciones. Un tamaño de heap pequeño puede aumentar la frecuencia necesaria de limpieza de memoria. Además puede ralentizar el proceso de reserva de

memoria, porque se puede complicar la localización de espacios libres adecuados para satisfacer las reservas. Para estos experimentos, hemos seleccionado como tamaño máximo de heap para cada programa el suficiente para que se ejecute, sin entrar en otro tipo de consideraciones. En cuanto al tamaño mínimo dejamos que la JVM utilice el tamaño mínimo que decide por defecto. De esta manera, emulamos el comportamiento de un usuario estándar, que no tiene por qué conocer los detalles de implementación de la JVM que utiliza para ejecutar sus programas.

3.2.3 Importancia del tiempo de gestión de fallos de página

El primer paso que hemos dado en la evaluación es determinar la relevancia que tiene la memoria virtual sobre el rendimiento de los programas de prueba. Para ello hemos medido el tiempo que el SO debe dedicar a resolver los fallos de página de los programas.

En las figuras 3.3 y 3.4 presentamos los resultados de este experimento. Las abscisas de las dos gráficas representan los diferentes tamaños de memoria física que hemos considerado, mientras que el eje de las coordenadas representa el tiempo de ejecución, separando el tiempo de fallo de página del resto de tiempo de la aplicación. La figura 3.3 representa el tiempo en valor absoluto (expresado en horas de ejecución), mientras que en la figura 3.4 el tiempo aparece expresado en porcentaje con respecto al tiempo total de ejecución. Sólo mostramos el tiempo dedicado a resolver fallos de página hard , porque hemos visto que el efecto de los fallos de página soft se puede ignorar.

En las gráficas se puede ver que para CRYPT el tiempo de cálculo domina claramente el tiempo de ejecución, y hace que el tiempo dedicado a resolver fallos de página sea totalmente insignificante. Además, en la figura 3.3 se puede ver que el tiempo de fallo de página se mantiene estable ante los cambios en el tamaño de memoria y, por lo tanto, el tiempo de ejecución es prácticamente el mismo para todos los tamaños de memoria que hemos evaluado.

Para el resto de programas de prueba se puede observar en la figura 3.3 que, cuando la memoria física deja de ser suficiente para albergar todo el espacio de direcciones de los programas, el tiempo dedicado a resolver fallos de página es considerable. Además, en

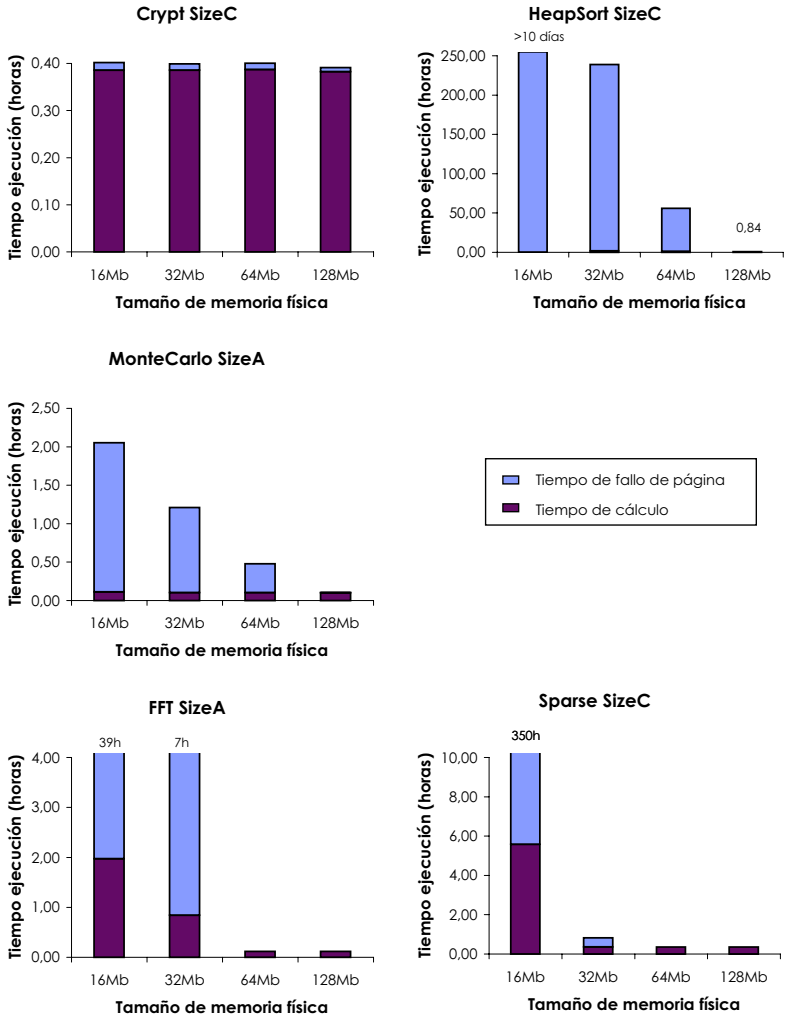

Figura 3.3 Tiempo dedicado a resolver fallos de página

la figura 3.4 se ve que, en esta situación, el tiempo de fallo de página representa en la mayoría de los casos más de un 90 % del tiempo total de ejecución.

El tiempo de ejecución de HEAPSORT y de MONTECARLO aumenta para cada disminución en el tamaño de memoria física, debido al aumento del tiempo necesario para resolver los fallos de página. Es más, excepto para la ejecución sobre 128Mb, el tiempo de fallo de página de las dos aplicaciones domina por completo el tiempo de ejecución. En el caso

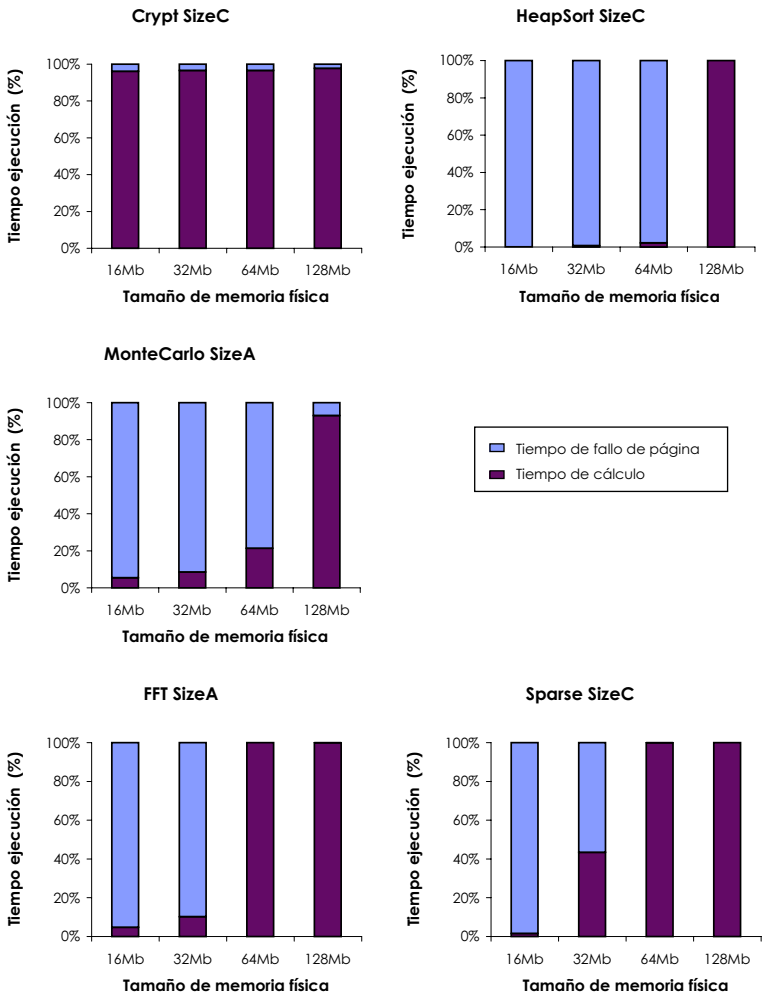

Figura 3.4 Porcentaje de tiempo dedicado a resolver fallos de página

de HeapSort este incremento del tiempo es más evidente. Si se ejecuta sobre 128Mb, el working set del programa cabe en memoria y por lo tanto no provoca ningún fallo de página, y el tiempo de ejecución es 50 minutos. Sin embargo, si se ejecuta sobre 64Mb, el tiempo de ejecución pasa a ser de más de 2 días, con casi un 98 % del tiempo dedicado a resolver fallos de página. Y si se ejecuta sólo con 16Mb entonces el tiempo de ejecución supera los 10 días con un tiempo de cálculo insignificante comparado con el tiempo de fallo de página. En el caso de MonteCarlo, aunque el tiempo absoluto no sufre un incremento tan considerable, también se observa que el porcentaje de tiempo dedicado a

resolver fallos de página se multiplica, y pasa de ser nulo, en el caso de la ejecución sobre 128Mb, a representar un 78,6 % sobre el tiempo total del programa, si la ejecución se hace sobre 64Mb, y un 94,6 % si se hace sobre 16Mb.

SPARSE y FFT necesitan menos cantidad de memoria para albergar todos sus datos, y se ejecutan sobre 128Mb y sobre 64Mb sin provocar fallos de página. Sin embargo, si la cantidad de memoria deja de ser suficiente para almacenar su working set, para ambas aplicaciones el tiempo de fallo de página se incrementa considerablemente y, como consecuencia también lo hace el tiempo total de ejecución.

En el caso de SPARSE, si comparamos la ejecución sobre 32 Mb con la ejecución sobre 128Mb, podemos observar en la figura 3.3 que el tiempo total de ejecución se ha multiplicado por un factor de 2,3, y en la figura 3.4 vemos que el porcentaje de tiempo dedicado a resolver fallos de página deja de ser nulo y pasa a ser de un 40 %. Si la comparación la hacemos de la ejecución sobre 16Mb con la ejecución sobre 128Mb, el factor multiplicativo del tiempo de ejecución pasa a ser 990, ya que el programa pasa de ejecutarse en únicamente 21 minutos sobre 128Mb a necesitar aproximadamente 350 horas para completar su ejecución sobre 16Mb. Además, en la figura 3.4 se ve que para la ejecución sobre 16Mb el porcentaje de tiempo de fallo de página de SPARSE convierte en insignificante el resto de tiempo de su ejecución.

Para FFT la diferencia en el tiempo de ejecución sobre 128Mb y sobre 32Mb viene afectada por un factor multiplicativo de 70, ya que pasa de una ejecución de 6 minutos sobre 128Mb a una ejecución de aproximadamente 7 horas sobre 32Mb (ver figura 3.3), y el porcentaje de tiempo dedicado a resolver fallos de página pasa de ser nulo a representar un 90 % del tiempo total (ver figura 3.4). En cuanto a la comparación entre la ejecución sobre 128Mb y la ejecución sobre 16Mb, el tiempo de ejecución se multiplica por un factor de 347 y pasa a ser aproximadamente 39 horas sobre 16Mb, con un porcentaje dedicado a resolver fallos de página del 95 % sobre el tiempo total de ejecución.

Además, tanto para SPARSE como para FFT se observa un incremento en el tiempo que no se dedica a resolver fallos de página. Este incremento se debe al aumento de cálculo necesario para gestionar el uso de la memoria virtual.

Los resultados de este experimento nos muestran que, cuando la memoria física no es suficiente para soportar el espacio de direcciones de los programas, la memoria virtual se convierte en un aspecto crítico para el rendimiento. Por lo tanto, analizando las causas de esta penalización, se podrá ver si es posible optimizar la ejecución de los programas en este tipo de escenarios.

3.2.4 Distribución de los fallos de página en el espacio de direcciones

El objetivo de este experimento es determinar qué regiones del espacio de direcciones de las aplicaciones tienen más accesos que involucran fallos de página. Como ya se ha dicho, hemos considerado tres posibles zonas: objetos, cabeceras de objetos, y resto del espacio de direcciones.

La tabla 3.2 resume los resultados significativos de este experimento para cada tamaño de memoria física que estamos considerando, y contiene el porcentaje que representan los fallos de página provocados por los accesos a objetos sobre el total de los provocados por los programas. Recordemos que en la subsección 3.2.3 hemos visto que la ejecución de los programas FFT y Sparse sobre 64Mb y 128Mb y la de HeapSort sobre 128Mb se completan sin provocar fallos de página y, por lo tanto, para este experimento no aplica ejecutar estas dos aplicaciones sobre estos dos tamaños de memoria física (en la tabla 3.2, las casillas correspondientes a estos resultados aparecen marcadas como n/a).

Programas / Tam. mem.	**16Mb**	**32Mb**	**64Mb**	**128Mb**
Crypt SizeC	96.99 %	97.50 %	98.05 %	97.73 %
HeapSort SizeC	100 %	100 %	99.99 %	n/a
MonteCarlo SizeA	90.29 %	95.81 %	96.96 %	95.60 %
FFT SizeA	99.96 %	99.93 %	n/a	n/a
Sparse SizeC	100 %	99.90 %	n/a	n/a

Tabla 3.2 Fallos de página en los accesos a objetos (%)

El resultado de este experimento nos muestra que la mayoría de los fallos de página provocados por los programas evaluados se deben a accesos a la zona de objetos. Se puede ver en la tabla 3.2 que el porcentaje menor es de un 90 % mientras que para el resto de ejecuciones supera siempre el 95 %.

Esto significa que ni los accesos a las cabeceras de los objetos ni los accesos a código o pilas son relevantes para el rendimiento de la memoria virtual. Por lo tanto, de cara a buscar fuentes de mejora para este rendimiento podemos centrarnos sólo en la zona de objetos y en los accesos involucrados.

3.2.5 Origen de los fallos de página

El objetivo de este experimento es distinguir entre los fallos de página provocados por el código de las aplicaciones de los provocados por la ejecución del código de gestión de la JVM. De esta manera, se puede determinar la penalización que la interacción entre el código de gestión de la JVM y la memoria virtual está añadiendo al rendimiento de los programas.

Como ya se ha dicho en la subsección 3.2.1, en esta clasificación hemos considerado tres tipos de tareas: reserva de memoria, liberación de memoria (marcado y barrido) y compactación del heap.

Sin embargo, esta clasificación no es suficiente para determinar por completo el impacto que la ejecución de la JVM tiene sobre los programas. Por ejemplo, no nos permite separar los fallos de páginas que, provocados por el código del programa, son debidos a la ejecución del código de gestión, que puede haber causado la expulsión anticipada de las páginas que el programa necesitaba a continuación.

Para evaluar la importancia de este efecto, ejecutamos cada programa de prueba de dos maneras diferentes. La primera, es la que ya se ha utilizado en los experimentos anteriores y emula el tipo de ejecución que un usuario estándar utilizaría. La segunda manera intenta reducir al máximo la ejecución del código de gestión de la JVM. Esta limitación se consigue lanzando los programas con un tamaño inicial de heap igual que el tamaño máximo, es decir, el suficiente para albergar los objetos vivos del programa. De esta manera, no será necesario que se ejecute en ningún momento la liberación de memoria ni la compactación del heap. Además, la reserva de memoria pasa a ser una tarea muy simple, ya que consiste en asignar siempre la posición contigua a la asignada en la última operación de reserva. Por último, para tratar las posibles llamadas explícitas que los programas hagan

al garbage collector, hemos substituido el garbage collector de la máquina por un código vacío, que retorna inmediatamente al ser invocado (*garbage collector nulo*).

Si el código del programa provoca menos fallos de página al reducir de esta manera el código de gestión de la JVM, podremos concluir que la ejecución del código de gestión de la JVM está perjudicando el uso de la memoria virtual que hace el código del programa.

A la hora de comparar los resultados de los dos tipos de ejecución hay que tener en cuenta que la ejecución con gestión de memoria mínima puede provocar que el espacio de direcciones de los programas sea mayor, ya que en ningún momento se liberan objetos que ya no son necesarios. Como consecuencia, puede ser que la cantidad de memoria física necesaria para la ejecución sea mayor y que el número de fallos de página del programa aumente. Si se da esta situación, la única conclusión que podemos extraer de esta comparación, es que los beneficios obtenidos por la ejecución del código de gestión de la JVM son mayores que la posible influencia negativa que su ejecución pueda tener sobre el uso de la memoria virtual del programa.

En la figura 3.5 se muestran los resultados que hemos obtenido, tanto para la ejecución normal (las columnas de la gráfica marcadas como *gc*) como para la ejecución con gestión de memoria mínima (las marcadas como *gcnulo*). Sólo aparecen los fallos de página debidos a los accesos a objetos ya que, como hemos mostrado en la sección 3.2.4, ésta es la región más afectada por los fallos de página.

Lo primero que hay que destacar es que, para todos los programas excepto para MonteCarlo, el código de los programas es la fuente de la mayoría de los fallos de página provocados. Además, el resultado de ejecutarlos con la gestión de memoria habitual o con la gestión de memoria mínima es prácticamente el mismo.

Para entender este comportamiento hemos contado el número de veces que estos programas necesitan que se libere memoria durante la ejecución con la gestión de memoria habitual, y hemos visto que este número es muy bajo (varía entre tres, para FFT y HeapSort, y siete, para Sparse). Este es el motivo por el que el rendimiento de los programas apenas se ve afectado si no se ejecuta esta tarea. Por este motivo, también, una mejora

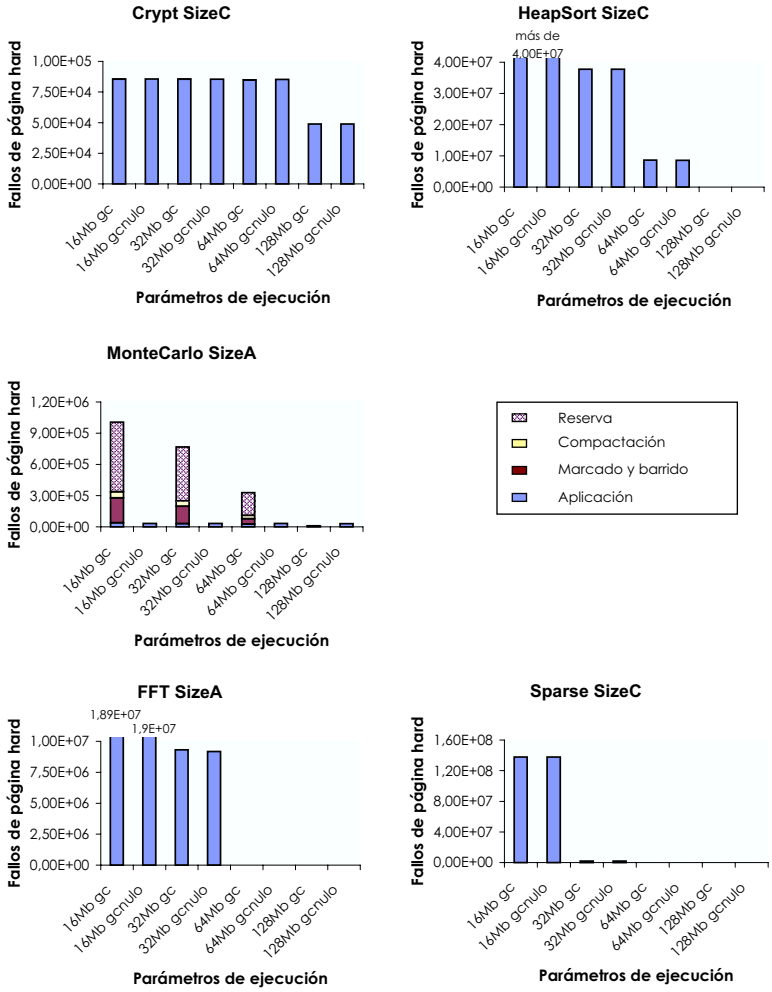

Figura 3.5 Clasificación de los fallos de página según su origen

del rendimiento del mecanismo de liberación de memoria no influiría en el rendimiento de estos programas.

Respecto a la tarea de reserva de memoria, estos resultados no nos permiten asegurar que no está influyendo en el rendimiento de los programas. Como ya hemos dicho, si no se liberan objetos, el proceso de reserva es muy simple y no añade sobrecarga a la ejecución. Sin embargo, durante este proceso se decide la distribución de los objetos en el heap, y esa decisión puede influir sobre la efectividad de la memoria virtual. Desde el punto de

vista del rendimiento de la memoria virtual, la decisión adecuada sobre la situación de los objetos sería la que agrupara en las mismas páginas los objetos que están en uso al mismo tiempo, para conseguir que las decisiones que el SO toma a nivel de página se correspondan también con el uso a nivel de objeto. Para poder estudiar la influencia de esta decisión, hemos analizado el tipo de objetos que estos programas reservan y cómo se utilizan (ver sección 3.4).

Comportamiento particular de MonteCarlo

El programa MonteCarlo se comporta diferente al resto de programas que hemos ejecutado. Este programa sí que necesita una participación importante de la liberación de memoria y, por lo tanto, los resultados son muy diferentes si se ejecuta el programa con la gestión de memoria habitual o con la gestión de memoria mínima.

En primer lugar, si analizamos la ejecución con gestión de memoria mínima, vemos que se consigue que todos los fallos de página sean debidos a la ejecución del código del programa. Además, este número de fallos de página se mantiene estable aunque se reduzca la cantidad de memoria física disponible. Esto se debe al patrón de accesos del programa: su working set, los datos que utiliza el programa al mismo tiempo, ocupa poco espacio de memoria, y los fallos de página se producen sólo durante los cambios de working set.

Por otro lado, en la ejecución con la gestión de memoria habitual el número de fallos de página se incrementa considerablemente con cada reducción en el tamaño de la memoria física. Sin embargo, todo el incremento se debe a fallos de página provocados por el código de gestión, mientras que los fallos de página debidos al código del programa se mantienen estables. Esto significa que la ejecución del código de gestión no afecta a los fallos de página provocados por el código del programa. A continuación se analiza los efectos de cada una de las tareas de gestión sobre el rendimiento final del programa.

- Reserva de memoria: esta es la tarea que provoca más fallos de página cuando la memoria física disponible es reducida. Esto se debe a que implementa una búsqueda cíclica por toda la zona de objetos, empezando a partir del punto donde se hizo la última reserva. Este recorrido de la zona de objetos finaliza cuando encuentra suficiente

espacio contiguo para albergar al nuevo objeto o cuando se alcanza el punto inicial de la búsqueda, en cuyo caso se inicia la liberación de memoria. Por lo tanto, en el peor de los casos, si no se encuentra el espacio adecuado, puede implicar un recorrido completo de toda la zona de objetos, con los fallos de página que eso implica.

- Marcado y barrido: la ejecución de este algoritmo de liberación de memoria está generando alrededor del 30 % de los fallos de página del código de gestión. Durante la fase de marcado, recorre la zona de cabeceras de los objetos para marcar los objetos que no están en uso. La fase de barrido es la que accede a la zona de objetos, y sólo accede a aquellos objetos marcados, para liberar la memoria que ocupan. Además, se ejecuta en menos ocasiones que la tarea de reserva, ya que sólo se lanza cuando la reserva no ha sido capaz de satisfacer la creación de un objeto nuevo y como respuesta a una única llamada explícita que hace el programa al final de su ejecución.
- Compactación del heap: por último, los fallos de página provocados por la compactación del heap representan menos del 10 % del total de los fallos de página del código de gestión. Esta fase debe recorrer varias veces todo el heap, para mover los objetos y eliminar la fragmentación externa del heap. Sin embargo, MONTECARLO sólo requiere una ejecución de esta compactación. Por este motivo, aunque es un algoritmo susceptible de provocar una gran cantidad de fallos de página, para este programa su impacto es menos importante que el resto de tareas de gestión.

Finalmente, comparando los resultados entre ambos tipos de ejecución, podemos observar que el número de fallos de página provocados por el código del programa es ligeramente superior si utilizamos la gestión de memoria mínima. Esto es debido a que la liberación de memoria consigue reducir el tamaño del área de objetos y, de esta manera, la cantidad de memoria física necesaria para soportarla.

3.2.6 Validación de los resultados en la JVM *HotSpot*

HotSpot es otra implementación de JVM que, junto con la JVM *classic*, Sun suministra con el entorno de de ejecución J2SDK. Esta JVM ha sido diseñada pensando en la ejecución de programas con mayor consumo de recursos.

Hemos ejecutado los programas de prueba también sobre la JVM *HotSpot* para comprobar el efecto que tiene sobre su rendimiento una gestión de memoria más elaborada y para validar las conclusiones que hemos extraído de la ejecución sobre la JVM *classic*.

Para ello, hemos comparado la cantidad de fallos de página provocados sobre la JVM *HotSpot* y los provocados sobre la JVM *classic*, separando los fallos fallos de página debidos al código de gestión de los fallos de página debidos al código del programa. Como ya se ha explicado en el capítulo 2, la gestión de memoria implementada en *HotSpot* es muy diferente de la implementada en la JVM *classic*, por este motivo, no hemos utilizado la subdivisión de tareas de gestión que hemos considerado durante la evaluación de la ejecución sobre la JVM *classic* en la sección 3.2.5.

La JVM *HotSpot* ofrece dos posibles configuraciones: una para aplicaciones servidor y otra para aplicaciones cliente. Hemos seleccionado la configuración para aplicaciones servidor porque es la que está pensada para trabajar con mayor demanda de recursos.

Además, como ya se ha explicado en la sección 2.3.2, la JVM *HotSpot* permite que el usuario decida si la limpieza sobre la zona de objetos *viejos* debe ser incremental o no. En este experimento hemos lanzado los programas con las dos opciones, para comprobar si influían de alguna manera en el resultado.

En la gráfica 3.6 mostramos los resultados que hemos obtenido, tanto para la ejecución con limpieza incremental (*HS-incgc* en la gráfica), como para la ejecución con limpieza total (*HS-noincgc*), comparados con los resultados de la ejecución sobre la JVM *classic*. Sólo presentamos los resultados para la cantidad de memoria física que hace necesario el uso de la memoria virtual para cada programa.

Por un lado, podemos observar que, excepto para MonteCarlo, el comportamiento de los tres tipos de ejecución es muy similar. Como ya se ha dicho en la sección 3.2.5, estas aplicaciones apenas requieren la ejecución de la liberación de memoria, lo cual también simplifica la reserva de memoria. Por lo tanto, la ejecución del código de estos programas son los causantes de la mayoría de los fallos de página para las tres opciones y, aunque la JVM *HotSpot* implementa una gestión de memoria más cuidadosa, esta mejora apenas

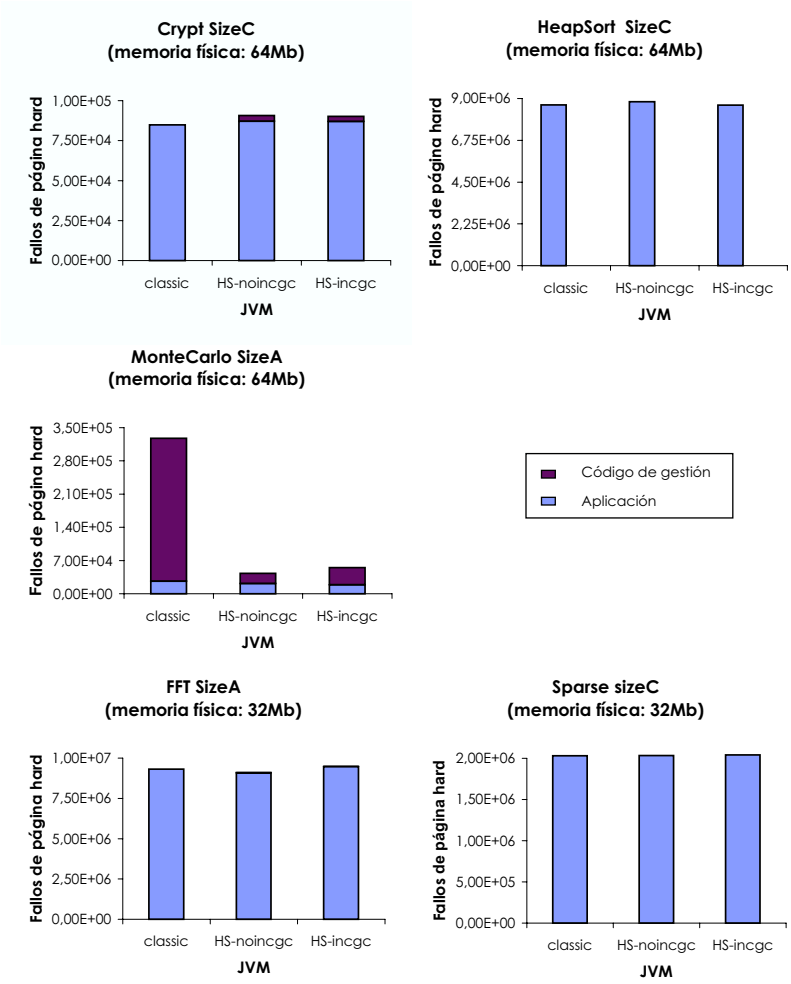

Figura 3.6 *HotSpot* vs. *classic*

influye en los resultados. Hay que recordar que la gestión de memoria de la JVM puede influir en el rendimiento de la memoria virtual no sólo por su propia ejecución sino también a través de la política que decide la situación de los objetos en el espacio de direcciones. Aunque *HotSpot* intenta adaptar la situación de los objetos en el heap para favorecer la localidad de referencia y agrupar en las mismas páginas objetos relacionados, esta política tampoco consigue mejorar el rendimiento con respecto a la ejecución sobre la JVM *classic*.

En cuanto al programa MonteCarlo, que sí requiere de una participación activa del garbage collector, se puede ver que se beneficia de las mejoras introducidas en la gestión de memoria de *HotSpot*, y la ejecución sobre esta JVM reduce substancialmente el número de fallos de página provocados por el código de gestión, aunque sigue superando al número de fallos de página provocados por el código del programa.

Por lo tanto, después de ejecutar los programas de prueba sobre la JVM *HotSpot*, hemos comprobado que únicamente uno de ellos (MonteCarlo) se ha visto beneficiado por la gestión de memoria mejorada que ofrece *HotSpot*. De todas maneras, la ejecución del código de gestión sigue penalizando el rendimiento de MonteCarlo en mayor medida que el código propio del programa. El resto de programas, se comportan de manera similar en ambas JVM, independientemente del modelo de liberación de memoria utilizado en la JVM *HotSpot*. Es decir, mejorar las políticas de gestión del espacio de direcciones no ha servido para mejorar el uso de la memoria virtual que estos programas hacen y, por lo tanto, es necesario buscar otras posibles vías para optimizar su ejecución.

3.3 RENDIMIENTO ÓPTIMO DE LA GESTIÓN DE MEMORIA EN JAVA

El objetivo de la evaluación que hemos realizado es determinar qué aspectos de la gestión de memoria en un entorno Java son susceptibles de mejora. Para ello es necesario medir el rendimiento de la gestión actual, y determinar los aspectos que son más críticos para el rendimiento de los programas. Pero también es necesario determinar si este rendimiento es mejorable o si por el contrario no es posible ofrecer una mejor gestión a los programas.

En esta sección presentamos los resultados que hemos obtenido al simular la ejecución de los programas de prueba sobre un sistema de memoria óptimo. Este sistema de memoria ofrece el mejor rendimiento teórico, dado un patrón de accesos y una cantidad de memoria física. El rendimiento lo medimos en términos de fallos de página, y nos centramos únicamente en los producidos por accesos a la zona de objetos ya que los resultados indican que es la zona más afectada.

Aunque no es factible implementar la gestión óptima que proponemos sobre un sistema real, nos sirve para obtener una cota superior del rendimiento de la gestión de memoria y nos permite estimar el margen de mejora que existe para el rendimiento de esta gestión.

3.3.1 Modelo de gestión de memoria óptima

La gestión de memoria óptima en un entorno Java sería la compuesta tanto de una gestión óptima de la memoria virtual como de una gestión óptima del espacio de direcciones del proceso.

La gestión óptima de la memoria virtual es la que minimiza el número de intercambios con el área de swap y consigue mantener en memoria las páginas que están en uso. Esto es posible combinando la paginación bajo demanda con el algoritmo de reemplazo óptimo. Este algoritmo de reemplazo selecciona como páginas víctimas aquéllas que el programa va a tardar más tiempo en referenciar. De esta manera, los accesos al área de swap son sólo los inevitables. Este algoritmo de reemplazo no es implementable en un sistema real, ya que requiere conocer los accesos futuros de los programas para poder seleccionar adecuadamente las páginas víctimas.

En cuanto al espacio de direcciones, los programas Java ejecutados sobre memoria virtual tendrán una gestión óptima si ésta consigue que los objetos que ocupan la memoria física en cada momento sean los que se están utilizando en ese instante y, además, consigue que, si un objeto tiene que ser expulsado de memoria física, sólo se almacene en el área de swap si todavía está *vivo*. Esto sólo sería posible utilizando una política de asignación de memoria que cambiara de posición dinámicamente los objetos en función de su uso y un algoritmo de liberación de memoria perfecto.

La política de asignación de memoria óptima debería, pues, estar vinculada al algoritmo de reemplazo y al mecanismo de paginación bajo demanda. Cada vez que el algoritmo de reemplazo seleccione una página víctima, la política de asignación de memoria lógica debería mover a esa página los objetos que se va a tardar más tiempo en referenciar. De la misma manera, cada vez que sea necesario cargar una página almacenada en el área de

swap, se debería mover a esta página todos los objetos del área de swap que se vayan a necesitar antes.

El algoritmo de liberación de memoria perfecto sería aquel capaz de detectar inmediatamente qué objetos dejan de ser necesarios, para liberar la memoria que ocupan y evitar además movimientos innecesarios al área de swap.

Aunque el coste de implementar esta gestión del espacio de direcciones no es asumible por un sistema real, la simulación de esta gestión, combinada con la gestión óptima de la memoria virtual, cumple con el objetivo de ofrecer una cota superior del rendimiento del sistema de memoria.

3.3.2 Implementación del modelo de gestión de memoria óptima

Para obtener el rendimiento del modelo de gestión de memoria óptima hemos implementado un simulador, que tiene como datos de entrada la traza que describe el uso de la memoria del programa que se quiere evaluar (ver figura 3.7), y que ofrece como resultado el número de accesos al área de swap inevitables.

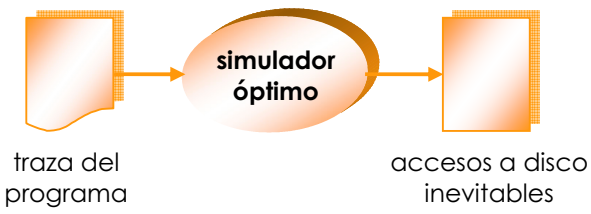

Figura 3.7 Simulación de la gestión óptima de memoria

La traza de entrada debe contener para cada objeto creado por el programa el instante de su creación y su tamaño, para poder simular en cada momento su situación en el espacio de direcciones. Además, para poder simular el uso de la memoria virtual, la traza debe describir todos los accesos a los objetos, es decir, el instante de cada referencia y, para poder tratar con objetos que ocupan más de una página, la posición concreta del objeto accedida. Por último, para poder implementar el algoritmo de reemplazo óptimo, la agrupación de objetos en páginas y la liberación perfecta de memoria, la traza debe contener también el instante de la próxima referencia de cada uno de los objetos.

Para poder generar una traza con esta información hemos tenido que modificar la JVM, introduciendo en el tratamiento de los *bytecodes* de creación y de acceso a objetos el código necesario para registrar los datos de la traza en un fichero de salida.

A continuación describimos brevemente estas modificaciones, así como la implementación del simulador.

Adquisición de datos sobre el comportamiento de los programas

La información que necesita el simulador de gestión óptima de memoria está asociada a la creación de nuevos objetos y a todos los accesos a objetos.

Para cada *bytecode* de creación de objetos (listados en la tabla 3.3) registramos en el fichero de trazas los datos que describen esta operación. Es decir:

- Cabecera del objeto, que permitirá identificar al objeto durante toda la simulación.
- Dirección inicial asignada, que permitirá localizar al objeto en el espacio de direcciones.
- Tamaño del objeto, para usar en el algoritmo de asignación de memoria, es decir, durante la creación y en cada cambio de situación del objeto en el espacio de direcciones.
- Instante de su próxima referencia, para simular el algoritmo de asignación de memoria, los intercambios con el área de swap y la liberación de memoria.

Bytecode	**Descripción**
new	Creación de objeto
newarray	Creación de array de escalares
anewarray	Creación de array de referencias
multianewarray	Creación de array multidimensional

Tabla 3.3 Bytecodes de creación de objetos

En cuanto a los *bytecodes* de acceso (ver tabla 3.4), necesitamos registrar la siguiente información:

- Cabecera del objeto, que lo identifica durante la simulación.
- Posición accedida, para determinar la página de memoria involucrada en el acceso, necesario si el objeto ocupa varias páginas.
- Instante de su próxima referencia, necesario para simular la asignación de memoria, los intercambios con el área de swap y la liberación de memoria.

Bytecode	**Descripción**
iload/istore lload/lstore fload/fstore dload/dstore aload/astore	Lectura/escritura sobre escalares
iaload/iastore laload/lastore faload/fastore daload/dastore aaload/aastore baload/bastore caload/castore saload/sastore	Lectura/escritura sobre arrays
getfield/putfield	Lectura/escritura sobre campos de objeto

Tabla 3.4 Bytecodes de acceso a objetos

La figura 3.8 contiene el pseudocódigo que describe el código que hemos añadido a la JVM para generar la traza con el uso de la memoria del programa. Básicamente, registramos en un fichero una línea con los datos de cada nueva creación o nuevo acceso, indicando el tipo de línea mediante el *bytecode*. Para registrar el campo que contiene la siguiente referencia al objeto, mantenemos siempre cuál ha sido su último acceso registrado. De esta manera, dado un acceso podemos retroceder en el fichero hasta la línea del acceso

previo, actualizar su campo de siguiente referencia para que sea el acceso actual, y marcar el acceso actual como último acceso al objeto.

```
if (generacion_trazas_activada) {
    registrar(codigo_operacion);
    registrar(objeto.cabecera);
    if (codigo_operacion == creacion) {
        registrar(objeto.direccion_inicial);
        registrar(objeto.tam_objeto);
        marcar_esta_como_ultima_referencia(objeto);
    }
    else {
        registrar(desplazamiento_acceso);
        marcar_esta_como_ultima_referencia(objeto);
        acceso_previo=buscar_acceso_previo(objeto);
        accceso_previo.proximo_acceso = instante_actual
    }
}
```

Figura 3.8 Algoritmo para generar las trazas

Hay que tener en cuenta que los programas de prueba tienen un alto consumo de memoria y millones de referencias a objetos. Esto significa que guardar una línea para cada acceso implicaría una traza de gran tamaño, lo que dificultaría y ralentizaría su procesado posterior, además de hacer necesaria una gran cantidad de espacio de disco disponible. Por este motivo, durante el registro de la traza, hemos filtrado información que no aporta datos relevantes para la simulación (ver figura 3.9).

El filtro que hemos implementado afecta únicamente a la información sobre los accesos a objetos, y consiste en eliminar referencias consecutivas a un objeto que no influyen sobre las decisiones de gestión. En particular hemos adoptado el siguiente criterio: si dado un conjunto de referencias consecutivas a un objeto, la cantidad de memoria cargada entre la primera de ellas y la última es menor que la cantidad de memoria de la máquina, entonces las referencias intermedias no son registradas en el fichero de trazas.

A continuación demostramos intuitivamente que eliminar estas referencias del fichero de trazas no modifica el resultado del simulador de memoria óptima.

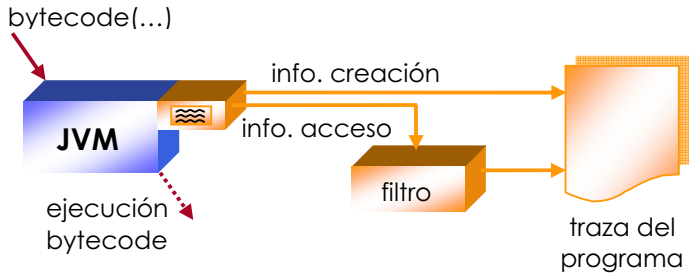

Figura 3.9 Esquema de la generación de trazas

Consideremos la siguiente secuencia de referencias de un programa:

$$Ref = \{O_1, \overbrace{X_i, \cdots, \underbrace{O_2, \cdots, Y_j}_{N}}^{L}, O_3, \cdots, Z_k, O_4\} \qquad \text{donde } L < M < N$$

Siendo O_1, O_2, O_3 y O_4 cuatro accesos consecutivos al objeto O y M la cantidad de memoria física de la máquina. Suponiendo que la cantidad de memoria necesaria para satisfacer las referencias entre O_1 y O_3 es L, y N es la cantidad de memoria implicada entre O_1 y O_4.

Después de la referencia O_1 el objeto O pasa a estar cargado en memoria física. La referencia O_2 será relevante para el resultado de la simulación si es la responsable de que el objeto permanezca en memoria hasta que ocurra la referencia O_3. Es decir, si eliminar la referencia O_2 del fichero de trazas puede hacer que la simulación del algoritmo de reemplazo seleccione a O como objeto víctima entre las referencias O_1 y O_3. Pero esto sólo podría ocurrir si a partir de O_1 todos los objetos cargados en memoria son referenciados antes de que ocurra O_3. Sin embargo, partiendo de la base de que la cantidad de memoria cargada entre O_1 y O_3 es menor que la cantidad de memoria física, si la memoria está llena seguro que hay objetos cargados que no se han referenciado entre O_1 y O_3, y por lo

tanto son los candidatos a ser expulsados de memoria en este intervalo de referencias, independientemente de la presencia o no de O_2 en la traza.

Sin embargo, si filtramos también el acceso O_3, los dos accesos consecutivos a O pasan a ser O_1 y O_4. Entre estos dos accesos se está utilizando más memoria (N) que la memoria física del sistema, y ya no es posible asegurar que el comportamiento del simulador sea equivalente al que tendría en caso de disponer también de la información sobre O_3. Esta equivalencia depende de la cadena concreta de referencias y de las cantidades de memoria involucradas y, por lo tanto, la decisión de filtrarlo requiere un análisis de cada situación particular. Para simplificar el algoritmo de filtrado y evitar hacer este análisis para cada situación hemos adoptado la opción conservadora de grabar siempre este tipo de referencias.

Es decir, dado un conjunto de referencias que involucren tantas páginas diferentes como memoria física haya en el sistema, para cada uno de los objetos guardamos el primer y el último acceso que se hace a ese objeto dentro de ese conjunto.

$$Ref = \{\overbrace{O_1, X_i, \cdots, O_2, \cdots, O_3, Y_j, \cdots, W_l,}^{M} \cdots, Z_k, O_4\} \qquad \text{registramos } O_1, O_3 \text{ y } O_4$$

Aunque con este algoritmo es posible que conservemos algunos accesos no relevantes para la simulación, el porcentaje de filtrado que hemos obtenido es suficiente para hacer de las trazas una información fácilmente manipulable.

Implementación del simulador

El simulador que hemos implementado recibe como parámetro el fichero de trazas que describe el comportamiento de los programas, y la cantidad de memoria física libre. Como resultado devuelve el número de fallos de página que requieren acceso a disco y que son inevitables.

En la figura 3.10 mostramos el pseudocódigo del simulador.

```
if (!objeto.presente) {
    tam_necesario = objeto.tam_objeto;
    if (tam_necesario > memoria_libre) {
        memoria_libre += liberar_objetos_inactivos();
        while (tam_necesario > memoria_libre) {
            objeto_victima = seleccionar_objeto_presente_referenciado_mas_tarde();
            objeto_victima.presente=FALSO;
            memoria_libre+=objeto_victima.tam_objeto;
        }
    }
    if (! primer_acceso(objeto))
        fallos_de_pagina_hard += (tam_necesario/TAM_PAGINA);
    objeto.presente = CIERTO;
    memoria_libre-= objeto.tam_objeto;
}
actualizar_informacion_sobre_proxima_referencia()
if (ultima_referencia(objeto))
    objeto.inactivo = CIERTO;
```

Figura 3.10 Simulador para la gestión de memoria óptima

Mientras haya memoria física disponible, se simula la carga de páginas a medida que se referencian. Cuando la memoria libre no es suficiente para satisfacer una carga, entonces primero se libera la memoria ocupada por objetos no necesarios (aquellos para los que ya se ha tratado su último acceso). De esta manera simulamos la liberación perfecta de objetos.

Si con la memoria obtenida por esta liberación no es suficiente, entonces se continúa con la liberación de memoria mediante la simulación del algoritmo de reemplazo, hasta obtener la suficiente memoria libre. Este algoritmo de reemplazo libera la memoria ocupada por aquellos objetos que tardarán más tiempo en ser referenciados. De esta manera se simula el reemplazo de memoria óptimo y el movimiento de los objetos en el espacio de direcciones para agrupar en las páginas víctimas los que tardarán más tiempo en ser necesarios.

En el momento de cargar un objeto, si no es la primera referencia al objeto se debe actualizar el contador de fallos de página. Para simular la carga conjunta de objetos

utilizados al mismo tiempo, incrementamos ese contador con la fracción de página ocupada por el objeto cargado.

3.3.3 Evaluación del rendimiento de la gestión óptima de memoria

En la figura 3.11 comparamos los resultados de la ejecución real de los programas con los resultados que hemos obtenido de la simulación sobre el sistema de gestión de memoria óptimo. Como ya hemos dicho, medimos el rendimiento de ambos sistemas mediante los fallos de página provocados por los accesos a la zona de objetos.

En las gráficas presentamos los resultados para los diferentes tamaños de memoria que hemos considerado durante toda la evaluación, aunque los tamaños relevantes para esta simulación son los que hacen necesaria la utilización de la memoria virtual. Se puede ver que en estos casos, para todos los programas de prueba, el rendimiento de la ejecución real está muy lejos de acercarse al rendimiento óptimo.

Este resultado hace que tenga sentido plantearse la búsqueda de alternativas de gestión que mejoren el rendimiento de la memoria ya que, aunque el rendimiento de la gestión óptima es inalcanzable, el margen existente para la mejora es considerable.

3.4 EVALUACIÓN DEL TIPO DE OBJETOS

Para completar la evaluación del uso de la memoria que hacen los programas Java y determinar las posibles mejoras que se pueden introducir en su gestión, hemos analizado los objetos que estos programas crean durante su ejecución.

Además, este análisis nos va a permitir determinar si la política de asignación de memoria utilizada por la JVM está influyendo de alguna manera en el rendimiento de los programas. Aunque en las secciones 3.2.5 y 3.2.6 hemos visto que, para cuatro de los cinco programas evaluados, la ejecución del código de gestión de la JVM apenas influye en el rendimiento de la memoria virtual, los resultados obtenidos no nos permiten descartar que las decisiones

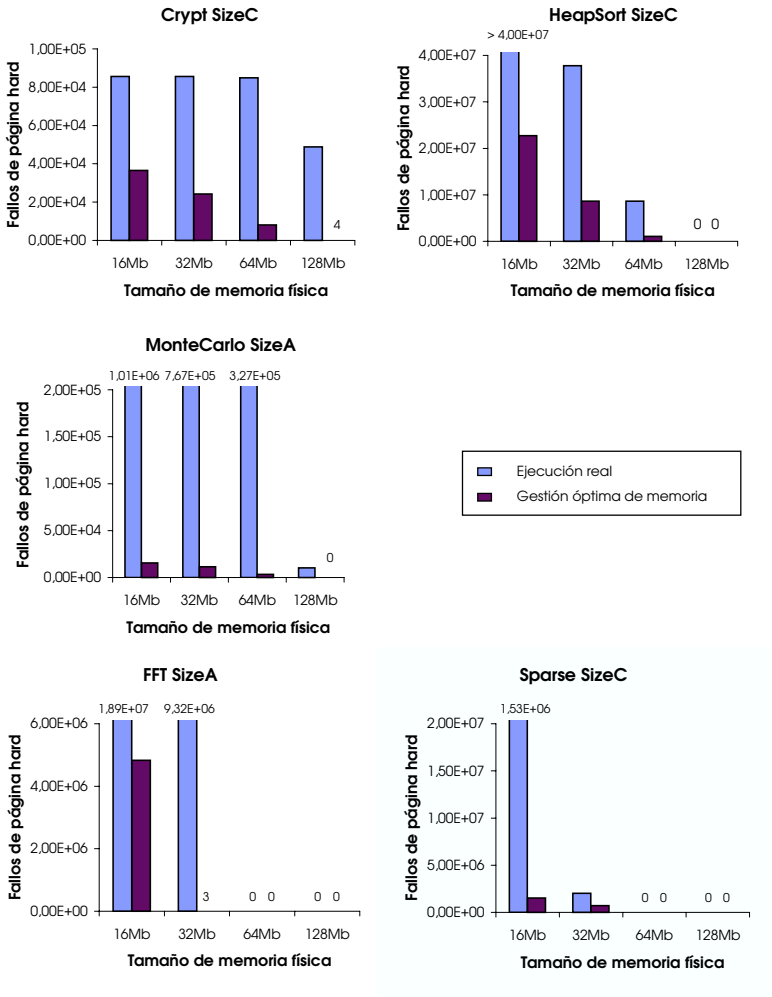

Figura 3.11 Ejecución real vs. gestión óptima

sobre la situación de los objetos en el heap no estén penalizando el rendimiento del sistema de memoria.

El primer paso para este análisis es saber qué tipo de objetos crean estos programas, cuál es su tamaño, y qué uso se hace de ellos. Esto permite decidir si es posible obtener una agrupación diferente de los objetos en páginas que sea más favorable al rendimiento de la memoria virtual.

Esta información se puede obtener generando una traza durante la ejecución de los programas, que registre todas las creaciones de objetos que hace el programa. Para cada nueva reserva guardamos en el fichero de traza el objeto implicado, su dirección inicial, su tipo y su tamaño. De esta manera, se puede analizar la distribución de objetos en el heap y cómo va evolucionando durante la ejecución.

Con la información de esta traza hemos clasificado a los objetos en función de su tamaño, considerando tres posibles grupos. El primer grupo es el de los objetos cuyo tamaño es menor que 4Kb (el tamaño de una página de memoria en el procesador sobre el que trabajamos). El segundo grupo es el de los objetos cuyo tamaño es menor que 64Kb (16 páginas). Y el tercer grupo es el de los objetos mayores de 64Kb.

En la figura 3.12.a mostramos la cantidad de objetos de cada grupo que los programas reservan y en la figura 3.12.b, mostramos la cantidad de memoria ocupada por cada uno de los grupos de objetos. Sólo mostramos los resultados para los programas de prueba que necesitan poca participación de las tareas de gestión de la JVM.

Se puede observar que los programas de prueba reservan pocos objetos mayores de 64Kb: uno en los casos de HEAPSORT y FFT, tres en el caso de CRYPT y cinco para SPARSE. Sin embargo, prácticamente el 100 % de la zona de objetos está ocupada por este tipo de objetos para los cuatro programas de prueba. Esto significa que los objetos implicados en el uso de la memoria virtual ocupan más de una página y, por lo tanto, la decisión sobre cómo agrupar objetos en páginas no les afecta. Es decir, la política de situación de objetos en el heap tampoco está influyendo en el rendimiento de la memoria virtual.

Un posible camino para mejorar el rendimiento de la memoria en estos programas es aplicar la técnica de *prefetch* en los accesos a estos objetos de grandes dimensiones. Para que esta técnica sea efectiva, es necesario anticipar las próximas referencias a memoria, para cargar esas páginas con antelación y de forma solapada con el cálculo y, de esta manera reducir el número de fallos de página de los programas.

La información de la traza que hemos generado indica que estos objetos de grandes dimensiones son de tipo *array* y los trabajos previos nos dicen que este tipo de objetos se

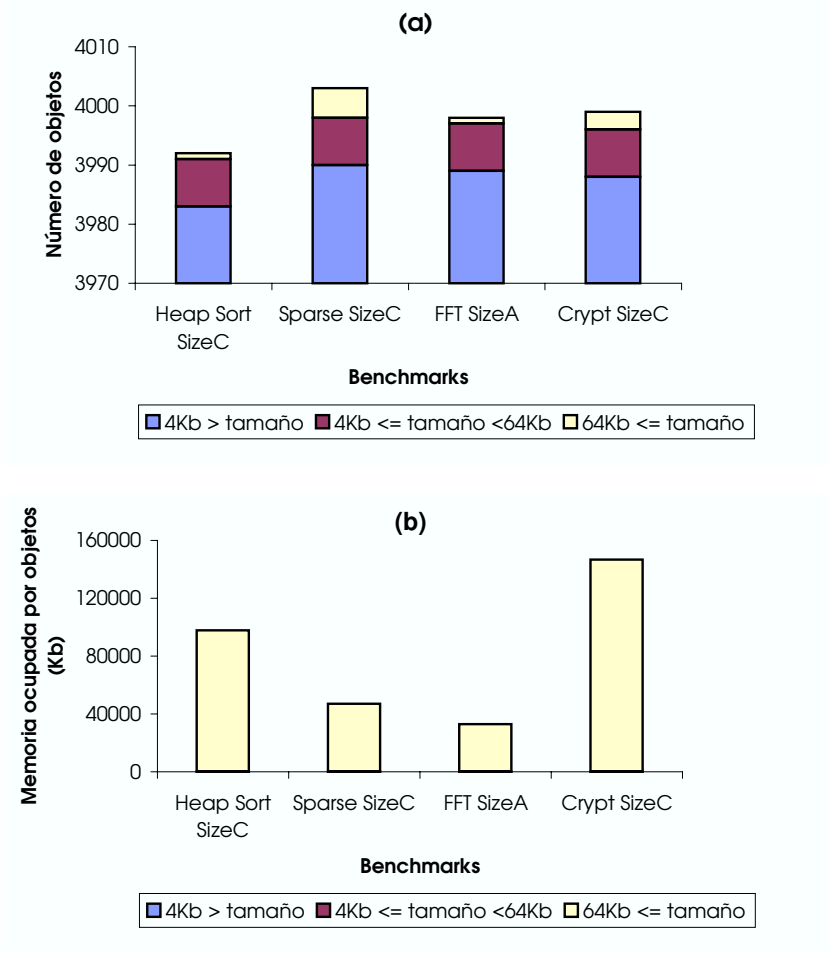

Figura 3.12 Clasificación de objetos por tamaño

suelen utilizar con un patrón de acceso regular y, por lo tanto, predecible [GG97], lo que facilita la implementación de un prefetch efectivo.

Un análisis más detallado del uso que los programas de prueba hacen de los objetos nos ha mostrado que, la mayor parte de las instrucciones que acceden a los vectores de grandes dimensiones lo hacen mediante un patrón regular. La tabla 3.5 resume el tipo de acceso que se utiliza en cada uno de los programas y a continuación lo describimos brevemente.

- HEAPSORT SIZEC: este programa utiliza un único vector de grandes dimensiones. Se trata del array que recibe como datos de entrada que tiene 25.000.000 enteros, es decir, ocupa 95,37Mb. De los accesos que este programa hace sobre el vector de entrada, la mayor parte son aleatorios y dependen del contenido del vector. Unicamente dos de las instrucciones utilizan un patrón predecible, basado en una separación entre accesos consecutivos (*stride*) de valor -1. Hay que decir que el número de veces que se ejecutan estas dos instrucciones hace que su influencia sobre el comportamiento del programa se pueda ignorar.

- SPARSE SIZEC: este programa trabaja sobre 5 arrays de grandes dimensiones. Dos de ellos se recorren de forma aleatoria, y son la matriz dispersa que se utiliza como entrada y el vector resultado de la multiplicación. Ambos arrays tienen 500.000 doubles (ocupan 3,81Mb cada uno) y la posición a la que se accede en cada momento viene determinada por el contenido de otros dos vectores. Estos vectores de índices se recorren de forma secuencial y tienen 2.500.000 elementos de tipo entero (cada uno ocupa 9,53Mb). El quinto array es el que se utiliza para almacenar el vector denso que se multiplica con la matriz dispersa. Este vector tiene 2.500.000 doubles (ocupa 19,07Mb) y se recorre de forma secuencial.

- FFT SIZEA: el único vector de grandes dimensiones que utiliza este programa es el que recibe como entrada. Este vector tiene 4194304 doubles (32Mb), que codifican los números complejos sobre los que trabaja el programa. Este programa utiliza un patrón regular para acceder al vector, y los accesos se hacen desde instrucciones que forman parte de bucles anidados (hasta tres niveles de anidación como máximo). El stride utilizado por cada instrucción en el acceso al vector depende del bucle anidado al que afecte el cambio de iteración. Es decir, las instrucciones que se utilizan como parte de tres bucles anidados, pueden utilizar tres niveles diferentes de stride: uno para cada cambio de iteración del bucle más interno, otro diferente cuando el cambio de iteración afecta además al segundo bucle, y el tercer stride cuando se cambia la iteración del bucle exterior. Además, para algunas de las instrucciones de este programa, el stride utilizado en el bucle más interno depende de la iteración de los bucles externos que se esté ejecutando. Es decir, algunos de los strides utilizados son dinámicos.

- CRYPT SIZEC: este programa, además del vector de entrada, utiliza otros dos vectores donde almacena el resultado de encriptar los datos de entrada y el resultado de

desencriptarlo. Cada vector tiene 50.000.000 elementos de tipo *byte* (en total suman 143,05Mb) y los tres se recorren de forma secuencial.

Programas	**Número de arrays**	**Memoria ocupada**	**Tipo de acceso**
HeapSort SizeC	1	95,37Mb	Strided y aleatorio
Sparse SizeC	2	7,62Mb	Aleatorio
	3	38,13Mb	Secuencial
FFT SizeA	1	32Mb	Strided dinámico (3 niveles)
Crypt SizeC	3	143,05Mb	Secuencial

Tabla 3.5 Uso de los arrays de grandes dimensiones

Por lo tanto, dotar al entorno de ejecución de un prefetch de memoria que aproveche la predictibilidad de los accesos strided sobre los arrays para cargar con antelación sus páginas, parece una alternativa viable para mejorar el rendimiento de la memoria virtual de estos programas Java.

3.5 OPORTUNIDADES DE MEJORA

En este capítulo hemos presentado una evaluación completa y novedosa en el entorno de ejecución de Java, del rendimiento del sistema de memoria. En esta evaluación hemos obtenido la penalización que involucra el uso de la memoria virtual en los programas de computación de altas prestaciones escritos en Java, distinguiendo entre la penalización debida al código del programa y su patrón de acceso de la debida al código de gestión necesario en el entorno de ejecución de Java.

Hay que decir que, aunque esta evaluación la hemos llevado a cabo sobre la versión de la JVM basada en la interpretación, las conclusiones que hemos extraído son independientes del modelo de ejecución que se utilice, ya que este modelo no afecta a las características de los objetos que han determinado las conclusiones de la evaluación.

Las principales conclusiones que hemos extraído de la evaluación son:

- La memoria virtual influye de forma significativa en el rendimiento de los programas cuando éstos trabajan sobre un conjunto de datos mayor que la memoria disponible.

- La zona del espacio de direcciones que más influye en la pérdida de rendimiento es la zona que contiene los objetos.
- Existen programas de cálculo numérico que se ejecutan sin apenas participación del código de gestión de la JVM que manipula el espacio de direcciones, y que sólo deben la penalización a la ejecución de su propio código. Para este tipo de programas, no existen trabajos previos sobre cómo mejorar su rendimiento, ya que todos los trabajos hechos sobre la gestión de memoria en Java se han centrado en optimizar las tareas de gestión que implementa la JVM (ver capítulo 7).
- El rendimiento de los programas evaluados está muy lejos del rendimiento óptimo que se obtendría con la ejecución sobre un sistema de memoria perfecto. Por lo tanto, el amplio margen de mejora invita a trabajar en posibles optimizaciones para el sistema de gestión de memoria en Java.
- Estos programas ocupan la mayor parte del espacio de direcciones con pocos objetos de gran tamaño y reservados al inicio de la ejecución. Por lo tanto no es viable optimizar el rendimiento de la memoria virtual mejorando las técnicas de gestión del espacio de direcciones, que apenas están influyendo. Además, estos objetos grandes son de tipo array que típicamente tienen un patrón de accesos predecible.

Por todo ello, en este trabajo proponemos mejorar el rendimiento del sistema de memoria dotando al entorno de ejecución de Java de un mecanismo de prefetch de páginas, que aproveche las características propias del entorno de ejecución para adaptar sus decisiones al comportamiento de los programas, y sea capaz de cargar, anticipadamente y de forma solapada con el tiempo de cálculo, las páginas que el programa accede.

4

MEJORA DEL RENDIMIENTO DE LOS PROGRAMAS JAVA MEDIANTE EL PREFETCH DE MEMORIA

En el capítulo 3 hemos visto que implementar una política de prefetch efectiva en el entorno de ejecución de Java puede mejorar el rendimiento de los programas de cálculo científico.

Esto nos ha llevado a seleccionar esta política como caso práctico para evaluar nuestra propuesta. Por este motivo, hemos diseñado e implementado, dentro del entorno de ejecución de Java, una política de prefetch que, utilizando las características propias del entorno, adapta sus decisiones al comportamiento de los programas.

Antes de diseñar el mecanismo de prefetch es necesario analizar las tareas involucradas en el prefetch y cuáles son los requerimientos impuestos por el entorno de ejecución para que estas tareas sean efectivas.

También es necesario analizar las facilidades que ofrece cada componente del entorno de ejecución para desarrollar cada una de las tareas de prefetch. De esta manera, se puede asignar cada una de ellas al componente más adecuado para cumplir sus requerimientos. Además, partimos de la base de que, ante varias estrategias que ofrezcan un rendimiento equivalente, el método ideal es el que exija menos modificaciones sobre el entorno de trabajo, ya que esto facilita la utilización de este mecanismo en diferentes plataformas.

En este capítulo presentamos este análisis previo. Vemos cómo el entorno de ejecución de Java ofrece una oportunidad en la asignación de tareas, que no existe en los entornos de ejecución basados en compilación, y que permite adaptar las decisiones de gestión al

comportamiento de los programas, de una forma eficiente y totalmente transparente al usuario y al programador.

Por último, presentamos una visión general de la propuesta de prefetch, que desarrollaremos en profundidad en los siguientes capítulos.

4.1 TAREAS Y REQUERIMIENTOS PARA UN PREFETCH EFECTIVO

El objetivo de las técnicas de prefetch de páginas es mejorar el rendimiento del sistema de memoria reduciendo el número de fallos de página provocados por los programas y, de esta manera, reduciendo el tiempo de bloqueo implicado en la carga en memoria física de las direcciones referenciadas mientras estaban almacenadas en el área de swap.

Estas técnicas se basan en determinar con anticipación las referencias a memoria que hacen los programas para iniciar su carga antes de que se efectúen, si es que se encuentran en el área de swap, y realizarla en paralelo con la ejecución del código del programa. Si cuando el programa accede a las direcciones involucradas en el prefetch, éstas ya están cargadas en memoria física, se consigue solapar por completo el tiempo de cálculo con el tiempo de carga, con la potencial mejora que ese solapamiento puede tener en el tiempo total de ejecución.

Así pues, el prefetch de memoria se puede dividir en dos grandes tareas. La primera tarea consiste en seleccionar las páginas que se deben cargar con antelación; la segunda en cargar las páginas seleccionadas de forma asíncrona con la ejecución del programa, para que el solapamiento de carga y cálculo sea real.

El primer requisito para que una política de prefetch sea efectiva es que la selección de páginas sea acertada y consiga anticipar las próximas referencias a memoria. Esto es necesario no sólo para poder solapar el mayor número de cargas posibles sino, también, para evitar sobrecargar el sistema con lecturas de disco erróneas. Hay que tener en cuenta que, las cargas erróneas de páginas pueden ser las responsables de expulsar al área de

swap páginas que sí son necesarias para el programa. Esto significa que estos errores en la selección, no sólo penalizan el rendimiento por la lectura de disco de las selecciones equivocadas sino que, además, pueden ser los responsables de aumentar, innecesariamente, el número de intercambios con el área de swap realizados para satisfacer las referencias del programa.

La precisión de esta selección depende de la cantidad de información disponible sobre el comportamiento dinámico de los programas. Para poder automatizar la selección de páginas de prefetch es necesario obtener un patrón con los accesos a memoria realizados por el programa, que se puede aplicar para predecir los accesos futuros. Cuánta más información se tenga sobre estos accesos, más preciso podrá ser el patrón generado y, consecuentemente, más precisas serán las predicciones efectuadas.

Para que el prefetch sea efectivo no es suficiente con detectar las próximas referencias a memoria, también es necesario solicitar su carga con la antelación adecuada (*distancia de prefetch*). Esta distancia se debe tener en cuenta durante la selección de páginas de prefetch y tiene que ser la suficiente para completar la carga antes de que el programa referencie esa página. Pero este no es el único condicionante ya que, además, es necesario evitar que la página precargada sea expulsada de memoria antes de ser utilizada, situación que podría darse si se utilizara una distancia de prefetch demasiado alta.

Para poder determinar el valor adecuado de la distancia de prefetch es necesario, una vez más, conocer el comportamiento del programa (código ejecutado antes de la referencia). Sin embargo, este no es el único factor que influye en este parámetro. También hay que tener en cuenta las características del sistema, tanto del hardware como del software, y las condiciones de ejecución. Así, por ejemplo, influye la velocidad del procesador, el tiempo necesario para acceder a disco, las políticas de planificación de recursos implementadas por el SO, la cantidad de procesos en ejecución y su comportamiento, etc.

En cuanto a la carga de páginas, es necesario disponer de un mecanismo asíncrono, que permita solicitar la carga de memoria y que el programa continúe en paralelo con su ejecución, sin esperar a que esta carga concluya. La carga de memoria requiere acceder al disco y a memoria física y, como todo acceso al hardware, es primordial que respete

la fiabilidad del sistema, y que no arriesgue la correcta ejecución del resto de programas de la máquina. Además, este mecanismo también debe detectar si las condiciones de ejecución son las adecuadas para efectuar la carga anticipada o si, por el contrario, el sistema está tan sobrecargado que hacerlo perjudicaría al rendimiento de los programas y, por lo tanto, es mejor desestimar el prefetch.

Otro aspecto a tener en cuenta en la implementación del prefetch, es que su ejecución debe ser eficiente, de manera que no oculte los beneficios de la carga solapada. Para ello, además de implementar las tareas de prefetch mediante un código eficiente, es importante aplicar optimizaciones como, por ejemplo, desactivar el prefetch para aquellos casos en los que no se puede anticipar las próximas referencias a memoria, o evitar la solicitud de carga para aquellas páginas que ya están presentes en memoria.

En el caso de las aplicaciones Java existe un requerimiento adicional, que no aparece en los entornos de ejecución basados en la compilación. Este requerimiento es garantizar la portabilidad de los programas, ya que éstos deben poder ejecutarse en cualquier máquina sin tan siquiera ser compilados de nuevo. Por lo tanto, añadir prefetch al entorno de ejecución debe respetar este paradigma. En este sentido, hay que recordar que la eficacia del prefetch depende de la máquina sobre la que se ejecuta y de las condiciones de ejecución. Por lo tanto, es deseable que esta técnica pueda auto configurarse en tiempo de ejecución, adaptándose a las características del momento.

Teniendo claras las tareas de prefetch y los requerimientos para que estas tareas sean eficaces, el siguiente paso consiste en determinar qué componente del entorno de ejecución es el más adecuado para cumplir los requerimientos de cada tarea. En las secciones 4.2 y 4.3 presentamos esta discusión para cada una de las grandes tareas involucradas en el prefetch: selección de páginas y lectura anticipada.

4.2 SELECCIÓN DE PÁGINAS DE PREFETCH

La tarea de seleccionar las páginas para el prefetch requiere un profundo conocimiento sobre los accesos a memoria que hacen los programas.

La información sobre qué direcciones de memoria accede el programa permite generar un patrón de accesos que se puede utilizar para predecir los accesos futuros del programa. Cuanta mayor sea la granularidad de esta información mayor precisión tendrá el patrón generado y, por lo tanto, más precisas serán las predicciones obtenidas a partir de esta predicción.

Pero si, además de las direcciones involucradas en los accesos, se tiene una información completa sobre estos accesos (objeto almacenado en la dirección, características del objeto e instrucción que efectúa el acceso), entonces es posible mantener patrones de acceso independientes en función de las características de cada instrucción y de cada objeto, mejorando aún más la precisión de estos patrones y simplificando su generación automática.

Con esta caracterización completa, es posible, por ejemplo, tratar de una forma muy simple con instrucciones que utilizan patrones de acceso muy diferentes. Por ejemplo, un programa puede tener instrucciones cuyos accesos son aleatorios y, por lo tanto, no siguen un patrón regular y predecible, mientras que otras instrucciones pueden utilizar un patrón totalmente predecible (en la tabla 3.5 del capítulo 3 podemos ver que los programas HeapSort y Sparse tienen este comportamiento). En esta situación, generar un patrón global es complicado, ya que los accesos aleatorios introducen ruido difícil de aislar. Sin embargo, distinguiendo la instrucción que efectúa cada acceso, es posible detectar y aislar los accesos aleatorios del resto de instrucciones, de manera que sólo esa función sea tratada como impredecible.

Es más, también es posible utilizar funciones de predicción específicas para cada tipo de acceso que tengan en cuenta sus características. En particular, esto facilita la optimización de evitar ejecutar el código de prefetch para aquellas instrucciones que no siguen un patrón de accesos regular ya que, una vez detectada la situación, se les puede asociar funciones de prefetch nulas, lo cual permite desactivar el prefetch únicamente para las instrucciones que no se pueden beneficiar de su ejecución.

Por lo tanto, la automatización de la selección de páginas de prefetch se debe incluir en el componente del entorno de ejecución que tenga más información sobre las características

de los accesos de los programas, ya que, cuanto más completa sea la caracterización de estos accesos, más simple y efectiva podrá ser la tarea.

4.2.1 Limitaciones del SO para la selección de páginas

La única información que tiene el SO sobre los accesos a memoria de los programas es la proporcionada por la excepción de fallo de página. Esto significa que, de todos los accesos que hace un programa, sólo será informado sobre los que se refieren a páginas que en el momento del acceso no estaban presentes en memoria física. Esta limitación, reduce la cantidad de datos que el SO puede utilizar para generar los patrones de acceso. Hay que destacar que hacer un trap al sistema para cada acceso del programa tendría un coste inaceptable que, en ningún caso, podría ser compensado por los beneficios del prefetch.

Además, para poder caracterizar estos accesos, el SO sólo dispone de la información que él mantiene sobre el espacio de direcciones de los programas y la información proporcionada por el hardware durante la excepción de fallo de página.

La única información que tiene el SO sobre el espacio de direcciones de los programas es la que describe qué regiones son válidas y qué permisos de acceso tiene cada una de ellas, pero no conoce las características del contenido de cada región, ya que es algo decidido en el nivel de usuario de forma transparente al SO.

Por otro lado, la información proporcionada por el hardware ante un fallo de página es únicamente la necesaria para resolver dicha excepción. Es decir, identificador del proceso (para localizar la información sobre su espacio de direcciones), dirección accedida al provocar el fallo de página (para poder resolver el fallo determinando si el acceso es correcto y, en ese caso, cargando en memoria la página accedida), y dirección que ocupa la instrucción que ha provocado el fallo (para reanudarla, si el SO resuelve con éxito la excepción).

Es decir, ante un fallo de página el SO no puede determinar ni el tipo de instrucción que se estaba ejecutando, ni el objeto sobre el que estaba accediendo ni las características del objeto. Con esta escasez de información sobre el comportamiento previo del programa es

complicado determinar su patrón de accesos y, por lo tanto, anticipar cuáles serán sus próximas referencias a memoria.

Por este motivo, las únicas políticas de prefetch implementadas dentro de los SO han sido políticas sencillas y genéricas, que intentan favorecer comportamientos habituales en los programas (como por ejemplo el acceso secuencial).

4.2.2 Limitaciones del compilador para la selección de páginas

El compilador, desde el nivel de usuario, puede extraer más información sobre los accesos a memoria de los programas que la disponible en el nivel de sistema.

A la hora de traducir el código fuente, el compilador puede analizar el código y sus accesos a memoria para determinar qué patrón siguen y añadir en el ejecutable generado las operaciones necesarias para seleccionar las páginas que se quieren cargar con antelación.

Sin embargo, hay que tener en cuenta que este análisis sólo puede ser estático y, por lo tanto, carece de cualquier dato dependiente de la ejecución como, por ejemplo, el comportamiento dependiente del valor de los parámetros de la ejecución.

En tiempo de compilación se desconocen también las características de la máquina sobre la que se va a ejecutar el programa y las condiciones de ejecución que habrá en ese momento, factores que influyen en la distancia de prefetch y, por tanto, también determinan la selección de páginas. Hay que tener en cuenta que una compilación dependiente de la plataforma física no respetaría el paradigma de portabilidad de Java.

Una posibilidad para adaptar las decisiones tomadas por el compilador a las características de cada ejecución sería implementar las operaciones de prefetch dentro de librerías dinámicas y mantener una versión diferente de estas librerías para cada plataforma física donde se quisiera ejecutar el programa. De esta manera, en tiempo de ejecución, el código de prefetch podría acceder a la información sobre las condiciones de ejecución para completar las decisiones de prefetch. Sin embargo, esta opción tampoco es suficiente ya que

implicaría poder acceder al código fuente en la fase de compilación para enlazarlo con las librerías correspondientes. Es decir, esta técnica sólo podrían utilizarla aquellos usuarios que tuvieran acceso al código fuente de los programas, lo que, en general, no se puede suponer que vaya a ocurrir.

4.2.3 Superación de las limitaciones mediante la JVM

La JVM, como el compilador, tiene acceso a todo el código y datos de los programas Java. Esto se debe a que la JVM es la encargada de ejecutar cada instrucción del programa. Por lo tanto, para cada una de ellas sabe si va a acceder a memoria, cuál es el objeto destino del acceso y cuál es la posición concreta del objeto. Además, la JVM gestiona las características de todos los objetos, incluida la posición que ocupan en memoria, y por ello es capaz de determinar la dirección involucrada en el acceso.

Es decir, la JVM dispone de una información completa sobre todos los accesos, que le permite realizar una caracterización detallada del comportamiento del programa. Esta caracterización la puede utilizar para generar un patrón minucioso sobre los accesos del programa y para determinar las funciones de predicción más adecuadas.

Además, toda esta información se obtiene en tiempo de ejecución. Esto significa que, a diferencia del compilador, la caracterización se realiza sobre el comportamiento dinámico de los programas.

Otra ventaja sobre la opción de extraer esta información mediante el compilador, es que en el caso de la JVM estos datos se obtienen de forma totalmente transparente al programador de las aplicaciones y al usuario que las ejecuta, y sólo necesita el código generado por un compilador estándar. Es decir, ni siquiera es necesario disponer del código fuente de los programas para poder obtener esta información.

Por lo tanto, en el entorno de ejecución de Java, la JVM es el componente que dispone de más información sobre el comportamiento de los programas, y eso la convierte en la candidata ideal para contener el código de selección de páginas de prefetch.

Así pues, nuestra propuesta consiste en modificar la JVM para que, para cada instrucción de acceso a memoria, utilice la información sobre la referencia para actualizar el patrón de accesos del programa y, luego, aplique ese patrón para seleccionar las páginas que se deben cargar con anticipación.

Hay que destacar que, como el análisis que hace la JVM es en tiempo de ejecución, es posible adaptar las decisiones de prefetch a las características de la plataforma real de ejecución y de las condiciones de cada instante, sin comprometer la portabilidad de los programas. Por lo tanto podrá utilizar estas características, por ejemplo, para determinar el valor adecuado de la distancia de prefetch y aplicarlo en la selección de páginas de prefetch. Además, tener acceso a esta información puede ser útil de cara a optimizar la ejecución del código de prefetch y, por ejemplo, evitar solicitar la carga anticipada de aquellas páginas seleccionadas que ya se encuentren presentes en memoria física.

Predicción a nivel de instrucción

Como ya hemos dicho, conocer la instrucción y el objeto relacionados con cada acceso, permite una mayor granularidad en la predicción, al poder tener patrones de acceso independientes en función de las características de las instrucciones y de los objetos.

Las ventajas de poder tener patrones independientes se ven claramente en el caso de los accesos a objetos de tipo array. Normalmente los arrays se utilizan desde instrucciones que forman parte de un bucle. Esto significa que cada una de estas instrucciones se ejecuta varias veces (tantas como indique la variable de control del bucle) y que, por lo tanto, se puede asociar un patrón independiente al conjunto de accesos de todas sus ejecuciones. En este caso, la granularidad de los patrones de acceso es máxima, obteniendo una predicción a nivel de instrucción y el consiguiente aumento de precisión en la selección de páginas.

El análisis que hemos efectuado sobre el comportamiento de los programas que se van a beneficiar del prefetch, nos ha demostrado que los accesos que están provocando los fallos de página son los asociados a arrays de grandes dimensiones, que se realizan desde instrucciones que forman parte de bucles (ver la sección 3.2.1 y la sección 3.4 del capítulo 3).

Por lo tanto, en este trabajo nos hemos centrado en intentar optimizar el acceso a este tipo de objetos, y la solución que hemos adoptado consiste en implementar la predicción de accesos futuros a nivel de instrucción. Es decir, para cada instrucción de acceso a arrays, se intenta detectar el patrón de accesos que sigue, para utilizarlo en la predicción de los accesos que realizará en sus próximas ejecuciones.

Sin embargo, asociar un patrón de accesos a cada instrucción puede no ser suficiente porque este patrón puede depender de las características del objeto que se está accediendo y las instrucciones pueden acceder a varios objetos durante la ejecución del programa. Por ejemplo, consideremos el caso de una función de multiplicación de matrices. Las instrucciones que recorren las matrices seguirán un patrón con varios strides, donde el valor de cada stride dependerá del tamaño de cada columna y de cada fila y, por lo tanto, dependerá de las características de las matrices que reciba como parámetro.

Por lo tanto, la predicción a nivel de instrucción debería considerar también los objetos utilizados por cada una y la influencia que pueden tener sobre el patrón de sus accesos.

Además, para poder completar la selección de páginas de prefetch, es necesario tener en cuenta la distancia de prefetch. Es decir, además de predecir las referencias a memoria de las próximas ejecuciones de una instrucción hay que determinar el instante apropiado para solicitar su carga anticipada. Esto dependerá del tiempo necesario para completar su carga y del momento previsto para la ejecución de las iteraciones de la instrucción que las referencian. A su vez, estos tiempos vienen determinados por el código del programa, las características de la plataforma de ejecución y las condiciones presentes en la ejecución.

Como la JVM conoce todo el código del programa en ejecución puede estimar fácilmente el tiempo de ejecución entre dos iteraciones de cada instrucción. En cuanto al resto de factores, se pueden averiguar, o aproximar de forma bastante precisa, desde el nivel de usuario. Sin embargo, para definir el método para obtener este valor hay que tener muy en cuenta su tiempo de cálculo, ya que es importante no ralentizar innecesariamente la ejecución del mecanismo de prefetch. Por este motivo, puede ser necesario substituir parte de la información real sobre la ejecución por heurísticas que la aproximen.

Otro aspecto a considerar durante la selección de páginas de prefetch es el estado de las páginas seleccionadas. Puede ser que algunas de estas páginas seleccionadas ya estén presentes en memoria y no sea necesario solicitar su carga. Hay que evaluar el tiempo involucrado en estas solicitudes y, si es relevante sobre el tiempo total, el mecanismo de prefetch deberá comprobar el estado de las páginas antes de solicitar su carga y filtrar de la selección aquéllas que ya estén presentes en memoria.

Desde el nivel de usuario no se tiene acceso a la información sobre el estado de las páginas. Por lo tanto, para que la tarea de selección de páginas pueda filtrar estas peticiones innecesarias, es necesario dotarla de un método para obtener esta información. Para ello contamos con dos opciones. La primera es utilizar algún tipo de heurística que permita aproximar el estado de las páginas desde el nivel de usuario. La segunda opción consiste en modificar el SO para que exporte esta información. Para tomar esta decisión, hay que comparar, por una parte, el número de solicitudes innecesarias que se evitan con ambos métodos y, por otra parte, el tiempo necesario para aplicar el filtro en los dos casos ya que, para que esta optimización tenga sentido, es necesario que el método utilizado tenga una sobrecarga menor que el de solicitar páginas para prefetch.

4.3 CARGA ASÍNCRONA Y ANTICIPADA

En cuanto a la tarea de cargar en memoria las páginas seleccionadas, como cualquier acceso al hardware, tiene el requerimiento primordial de respetar la fiabilidad del sistema. Por este motivo, la solución más adecuada es que se lleve a cabo desde el nivel sistema.

El otro aspecto relacionado con la carga es cómo solicitar desde el nivel usuario que el SO inicie la carga de las páginas seleccionadas.

La carga anticipada debe ser asíncrona con respecto a la ejecución de los programas. Es decir, una vez solicitada la carga, el programa debe continuar la ejecución sin bloquearse hasta que la carga finalice. En los SO actuales, la gestión de memoria virtual sólo dispone de un mecanismo que provoca la carga de memoria desde nivel de usuario, y es el fallo de página. El fallo de página es la excepción que se produce cuando un programa accede a

una dirección de memoria que en ese momento no está presente en memoria física. Como consecuencia, el SO bloquea al programa y carga en memoria la página accedida. Hasta que esta carga no finaliza, el programa no puede continuar la ejecución. Es decir, es un mecanismo síncrono.

Por lo tanto, como el entorno de ejecución no dispone de un interfaz para iniciar la carga asíncrona de páginas residentes en el área de swap, es necesario añadirle un mecanismo que lo permita. En este punto nos planteamos dos alternativas.

Una solución es adaptar, desde el nivel de usuario, el mecanismo ya existente y dotarlo de asíncrona con respecto a la ejecución del programa. Esto se puede conseguir añadiendo a la JVM un nuevo flujo de ejecución. Este flujo puede acceder a las páginas seleccionadas para provocar los fallos de página que desencadenarán su carga en memoria física. Mientras este flujo espera bloqueado a que finalice la carga de las páginas, el flujo del programa puede continuar asíncronamente con su ejecución. Esta solución consigue que la carga anticipada sea transparente al SO, en el sentido de que el SO no distingue si las páginas solicitadas son por carga anticipada o por fallo de página real. Por lo tanto, al no modificar el SO, favorece el objetivo de minimizar el número de cambios introducidos en el sistema y, de esta manera, la portabilidad del mecanismo.

La otra solución es añadir al SO un nuevo interfaz que permita que los programas soliciten la carga asíncrona y que, una vez solicitada, continúen la ejecución concurrentemente con la carga de la página. Esta opción implica modificar el SO para incluir este nuevo sevicio. Sin embargo, con esta alternativa, se involucra al SO en el mecanismo de prefetch y se le permite que distinga entre lecturas de disco relacionadas con prefetch del resto de lecturas, información que puede utilizar al aplicar las políticas de gestión de acceso a disco.

Como ya hemos dicho, la estrategia que requiera menos modificaciones en el entorno de ejecución es la que más facilita su introducción en diferentes plataformas de ejecución. Por este motivo, en nuestro diseño inicial hemos adoptado la primera solución, con el objetivo de implementar un mecanismo totalmente transparente al SO (ver capítulo 5). Este primer diseño nos ha permitido evaluar los aspectos débiles de la estrategia de prefetch transparente al SO y determinar aquéllos puntos es los que es aconsejable una mayor

participación del SO para dotar al mecanismo de una mayor estabilidad y eficiencia. Como consecuencia de este análisis, nuestra propuesta final propone modificar el SO para implementar un estrategia de prefetch basada en la cooperación entre el SO y la JVM, y adopta la solución de implementar un interfaz dedicado para el prefetch (ver capítulo 6).

En cualquier caso, antes de llevar a cabo la carga anticipada de una página, es necesario evaluar si las condiciones de ejecución son favorables para la utilización de prefetch, ya que existen situaciones en las que la ejecución del prefetch es incapaz de beneficiar a los programas y puede incluso perjudicar su rendimiento. Por ejemplo, si el sistema de memoria está sobrecargado, cargar de forma anticipada una página puede provocar que se expulsen páginas que todavía estén en uso y que se vuelvan a referenciar incluso antes de que se acceda a la página que provocó su expulsión.

Por lo tanto, la implementación de esta tarea necesita utilizar la información sobre la carga del sistema para descartar las peticiones de prefetch no adecuadas. El método para obtener esta información dependerá de la estrategia que se utilice para solicitar la carga. El caso de la estrategia transparente al SO requiere consultar esta información desde el nivel de usuario y, en este caso, hay que definir un método para obtener estos datos o para aproximarlos sin comprometer la eficiencia del mecanismo. En la estrategia alternativa, que involucra al SO en la solicitud de carga, como el código del SO tiene acceso directo a esta información, no es necesario definir ningún nuevo método y, simplemente, antes de iniciar una carga anticipada, debe acceder a sus estructuras de datos para descartarla si las condiciones de ejecución desaconsejan utilizar prefetch.

4.4 VISIÓN GENERAL DE LA PROPUESTA DE PREFETCH

En este capítulo hemos analizado los requerimientos que debe cumplir la estrategia de prefetch para ser añadida en el entorno de ejecución de los programas Java, y cómo esta técnica se puede beneficiar de las características propias del entorno para optimizar el rendimiento del sistema de memoria. Este análisis nos ha permitido obtener un primer

esbozo de la estrategia de prefetch, decidiendo qué componente del entorno de ejecución debe encargarse de cada tarea involucrada en el prefetch, para maximizar su rendimiento.

En esta sección resumimos las principales conclusiones de este análisis, presentando una visión general de la estrategia de prefetch que proponemos añadir al entorno de ejecución de Java (ver figura 4.1).

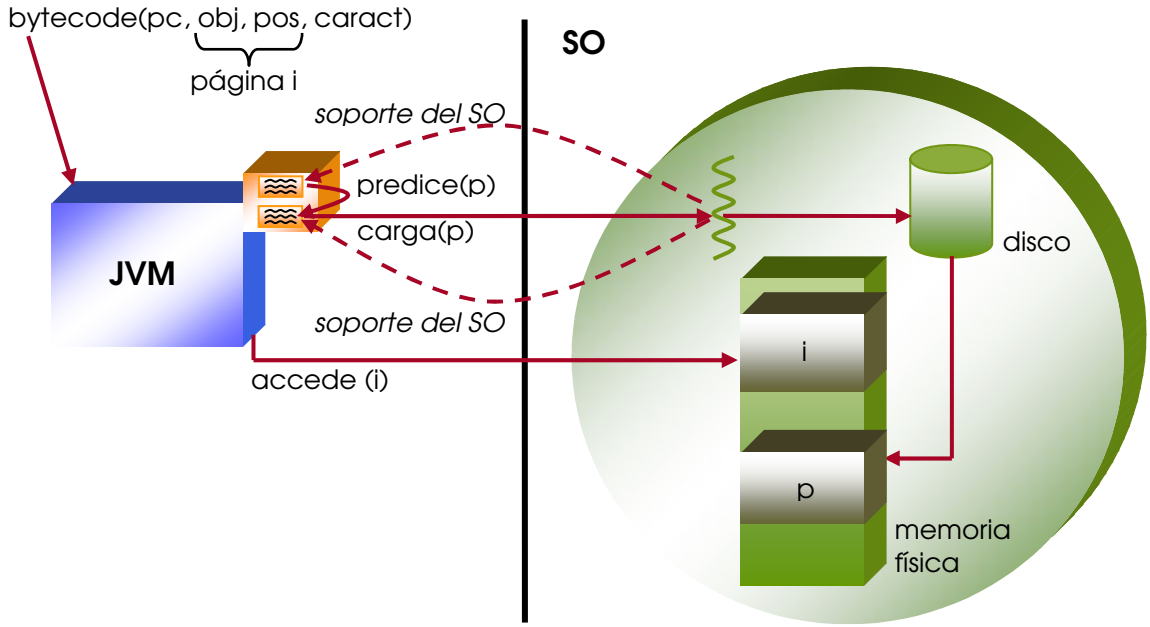

Figura 4.1 Visión general del mecanismo de prefetch

Proponemos que la JVM se encargue de la tarea de selección de páginas de prefetch. Para ello, para cada bytecode de acceso a memoria, actualiza la información que tiene sobre el patrón de accesos del programa y, utilizando este patrón, se encarga de predecir cuáles serán las próximas referencias a memoria (ver figura 4.1). Hay que destacar que toda la información que tiene la JVM sobre los accesos a memoria de las instrucciones permite obtener un patrón de accesos muy minucioso, y, por lo tanto, permite predecir de forma muy precisa los próximos accesos de las instrucciones.

Además, esta estrategia permite mantener diferentes patrones de acceso en función de las características de la instrucción y del objeto accedido, lo cual también simplifica la obtención del patrón de accesos y optimiza la ejecución de la tarea de selección. En el caso de los accesos a arrays la selección que proponemos se ejecuta a nivel de instrucción. Es decir, para cada instrucción se mantiene un patrón de accesos independiente, que también tiene en cuenta el objeto accedido, y que se utiliza para predecir las referencias de esa instrucción en sus próximas ejecuciones.

Para completar la selección de páginas de prefetch, es necesario decidir la distancia de prefetch adecuada y aplicar el filtro para eliminar peticiones de prefetch innecesarias. Para ello, existen dos alternativas: se pueden utilizar heurísticas que aproximen el estado de la memoria o se puede modificar el SO para que exporte esta información al nivel de usuario. Es necesario evaluar las dos alternativas para determinar si el uso de heurísticas es suficiente para obtener un buen rendimiento o si, por el contrario, es necesario modificar el SO para poder acceder a la información real sobre el estado de la memoria. Por este motivo, en la figura 4.1 hemos querido representar también la posible participación del SO en la tarea de selección de páginas.

En cuanto a la carga en memoria de las páginas seleccionadas, proponemos que sea el SO en el encargado de llevarla a cabo (ver figura 4.1). De esta manera, se garantiza que los accesos al hardware involucrados respetan la integridad de la máquina y la protección del resto de procesos en ejecución. Sin embargo, nos hemos planteado dos posibilidades para la solicitud de esta carga. La primera opción es que la JVM aproveche algún mecanismo ya existente en el sistema para provocar la carga asíncrona de las páginas seleccionadas. Esta opción obtiene una carga anticipada transparente al SO y, por lo tanto, el SO no puede colaborar para optimizar el mecanismo. La segunda opción consiste en modificar el SO con un interfaz dedicado para la carga asíncrona de memoria. Con esta opción, el SO puede participar también en las decisiones de carga anticipada, por ejemplo, descartando las operaciones de prefetch si las condiciones de ejecución así lo recomiendan. En la figura 4.1 también hemos representado la posible participación del SO en la solicitud de carga.

Nuestra propuesta supera las desventajas de las estrategias de prefetch propuestas en los entornos tradicionales basados en compilación en los siguientes aspectos:

- Selecciona de forma precisa las páginas de prefetch. La JVM es el componente del entorno de ejecución capaz de predecir los próximos accesos con mayor precisión, ya que dispone de más información sobre los accesos:
  - Conoce todos los accesos, a diferencia del SO, que sólo es informado de los accesos involucrados en fallos de página.
  - Conoce todas las características de los accesos, a diferencia del SO, que sólo tiene la información proporcionada por la excepción de fallo de página.
  - Y además en tiempo de ejecución, no como el compilador que sólo puede hacer un análisis estático de los programas.
- Respeta el paradigma de portabilidad de los programas Java. A diferencia del compilador, es capaz de adaptar las decisiones de prefetch, en tiempo de ejecución, a las características de la plataforma física y a las condiciones de ejecución sin necesidad de regenerar el ejecutable del programa y de forma transparente al usuario.
- Es transparente al programador y al usuario. Cualquier programa se puede beneficiar de esta técnica sin participación del programador, ni del usuario y sin requerir siquiera el código fuente.
- Respeta la fiabilidad del sistema, ya que el SO es el encargado de llevar a cabo la carga de las páginas.

Como consecuencia del análisis que presentamos en este capítulo, hemos visto factible la implementación de un mecanismo eficaz de prefetch totalmente transparente al SO. Por este motivo, nuestro diseño inicial evita cualquier modificación del SO y utiliza sólo el interfaz existente, junto con las heurísticas necesarias para suplir, si es necesario, la falta de información exacta.

En el capítulo 5 presentamos este diseño inicial, su implementación y su evaluación. Los resultados de la evaluación nos muestran que, aunque con esta estrategia hemos obtenido un prefetch eficaz, es posible mejorar su estabilidad y eficiencia si se añade al mecanismo cierto grado de cooperación con el SO.

En el capítulo 6 presentamos el diseño del prefetch cooperativo en el que se basa nuestra propuesta final. En este diseño introducimos los cambios necesarios en el interfaz del SO para conseguir un mecanismo más estable y poder eliminar algunas heurísticas de las que, aunque han demostrado ser una buena aproximación de la realidad en los programas que hemos evaluado, no es posible asegurar su eficacia para el resto de programas. Mediante esta estrategia obtenemos un mecanismo eficaz y estable, que demuestra que es posible mejorar el rendimiento de los programas Java implementando la cooperación entre el SO y la JVM para obtener una gestión de recursos adaptada al comportamiento del programa.

5

PREFETCH GUIADO POR LA JVM Y TRANSPARENTE AL SO

En este capítulo presentamos el diseño y la implementación de un prefetch a nivel de usuario, que se ejecuta de forma totalmente transparente al SO. En esta estrategia, la JVM utiliza la información que tiene sobre el comportamiento de los programas para predecir de forma precisa las próximas referencias a memoria y solicitar su carga anticipada. Además, para completar una selección eficiente de páginas de prefetch sin modificar el SO, aproxima mediante heurísticas la información sobre el estado de la memoria. Una vez seleccionadas las páginas, la JVM solicita su carga aprovechando el mecanismo ya existente para resolver los fallos de página de los procesos, de manera que, aunque el SO es el encargado de leer la página del área de swap, no es necesario modificarlo con ninguna nueva funcionalidad.

En las secciones 5.1 y 5.2 presentamos una discusión sobre los aspectos más relevantes que hay que tener en cuenta durante la implementación de las dos tareas principales del prefetch (selección de páginas y carga asíncrona), y en la sección 5.3 mostramos una visión general de la estrategia. A continuación, explicamos detalladamente la implementación de ambas tareas así como los puntos críticos de esta implementación para obtener una estrategia eficiente (sección 5.4). Completa este capítulo los resultados que hemos obtenido de la evaluación de la estrategia de prefetch transparente al SO (sección 5.5) y las conclusiones que hemos extraído durante el desarrollo y la evaluación de esta estrategia (sección 5.6).

5.1 SELECCIÓN DE PÁGINAS DE PREFETCH

Como hemos visto en el capítulo 4, la JVM es el componente del entorno de ejecución más adecuado para predecir las próximas referencias a memoria que los programas van a realizar. Por lo tanto la estrategia de prefetch que proponemos se basa en introducir el código de selección dentro de la JVM. La selección de páginas que proponemos consiste en utilizar la información sobre los accesos a memoria de cada instrucción para predecir las próximas referencias del programa, aplicar a las referencias predichas la distancia de prefetch adecuada y filtrar del conjunto aquellas páginas que ya se encuentran presentes en memoria física.

En este trabajo nos estamos centrando en el tratamiento de los arrays de grandes dimensiones que provocan el uso intensivo de la memoria virtual. Por este motivo, proponemos una selección de páginas de prefetch a nivel de instrucción. Es decir, para cada instrucción predecimos las referencias a memoria que realizará en sus próximas ejecuciones. Esto implica mantener para cada una de ellas la información sobre su patrón de accesos, teniendo en cuenta, además, el objeto que está accediendo en cada momento. Además, como hemos visto en la sección 3.4 del capítulo 3, normalmente el patrón de acceso a un array suele ser strided, por lo que el algoritmo de predicción que hemos implementado está orientado a detectar los strides de acceso de las instrucciones. Esta focalización en el uso de los arrays nos permite también ligar la ejecución de la selección de páginas sólo a las instrucciones que ejecutan bytecodes de acceso a arrays, sin afectar al tratamiento del resto de bytecodes (ver figura 5.1).

Hay que destacar que esta simplificación no cierra las puertas a aplicar nuestra propuesta de prefetch a otro tipo de aplicaciones en un trabajo futuro. Recordemos que tener la información sobre las instrucciones que realizan los accesos a memoria, nos permite utilizar varias funciones de predicción en función de las características del objeto o de la instrucción. Por lo tanto, sería posible mantener la predicción strided para los bytecodes de acceso a vectores y asociar al resto de bytecodes de acceso a memoria un algoritmo de predicción más genérico capaz de detectar su patrón de acceso.

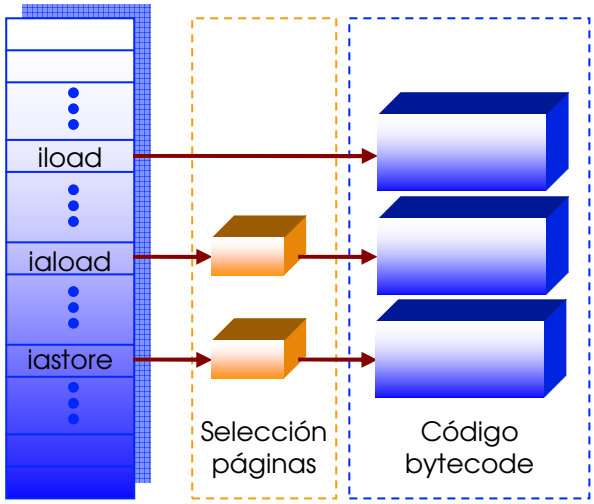

Figura 5.1 Predicción a nivel de instrucción

5.1.1 Efectividad de la predicción a nivel de instrucción

Para aproximar la efectividad de la selección de páginas a nivel de instrucción hemos implementado un prototipo de predicción. Este prototipo consiste en modificar la JVM para que cada instrucción de acceso a array, antes de realizar el acceso correspondiente, prediga y acceda también a la dirección que referenciará en su próxima ejecución. Hay que destacar que este acceso a la página predicha provoca un fallo de página, si la página se encuentra en el área de swap y, por lo tanto, la carga en memoria anticipada a la referencia real del proceso. Sin embargo, no es una operación de prefetch ya que la carga se hace de forma síncrona, es decir, el proceso se bloquea hasta que se completa el tratamiento de fallo de página y, por tanto, la lectura de disco asociada.

El algoritmo de predicción que hemos implementado es muy sencillo y es capaz de detectar un patrón con varios strides. Para ello, calcula el stride utilizado por cada instrucción (guardando siempre la dirección accedida por la ejecución previa de la instrucción) y almacena los valores de cada stride obtenido, el número de accesos consecutivos que se realizan usando cada uno de ellos, y el orden de las transiciones entre los diferentes strides.

En este prototipo, pues, cada instrucción de acceso a array realiza dos accesos a memoria: primero, el que se corresponde con la dirección predicha y, luego, el correspondiente a la instrucción en curso. Para aproximar la tasa de aciertos de la predicción, hemos contado por separado los fallos de página producidos por el código de predicción y los fallos de página producidos por el código del programa. Así, es posible obtener el número teórico de fallos de página que el algoritmo de predicción podría evitar. Para este recuento, hemos adaptado el gestor de dispositivo implementado para la evaluación del uso de memoria, descrito en el capítulo 3, al nuevo criterio de clasificación de fallos de página.

La plataforma que hemos utilizado para este experimento es la misma utilizada para la evaluación del capítulo 3: PC con un procesador Pentium III a 500 Mhz y 128Mb de memoria física, la versión 2.2.12 del kernel de Linux y la versión 1.3.1 de la JVM *HotSpot* de Sun.

En la figura 5.2 mostramos el resultado de este experimento para los cuatro programas que, según los resultados presentados en el capítulo 3, son candidatos a mejorar su ejecución mediante prefetch: CRYPT, HEAPSORT, FFT y SPARSE. Hemos ejecutado cada programa sobre la cantidad de memoria física que hace necesario el uso de memoria virtual para completar su ejecución. Cada gráfica presenta el número de fallos de página de cada programa tanto en el entorno de ejecución original (en la figura, *Sin predicción*) como en el entorno modificado con el prototipo de predicción (en la figura, *Predicción (instr-obj)*). Los fallos de página en el entorno modificado aparecen divididos en dos tipos: los provocados por el código de predicción y los provocados por el código del programa. La situación ideal sería que en el entorno modificado el código del programa no provocara ningún fallo de página y todos fueran debidos a la ejecución del código de predicción.

Se puede observar que, para tres de los cuatro programas evaluados (CRYPT, FFT y SPARSE), el resultado de este prototipo se acerca bastante al ideal y, aunque la ejecución del código de predicción no elimina por completo los fallos de página del código del programa, consigue reducirlos considerablemente. Así, el porcentaje de fallos de página del programa no eliminados es de 9,7 % para el peor caso (FFT), de 3,46 % para SPARSE y de 1,17 % en el mejor caso (CRYPT).

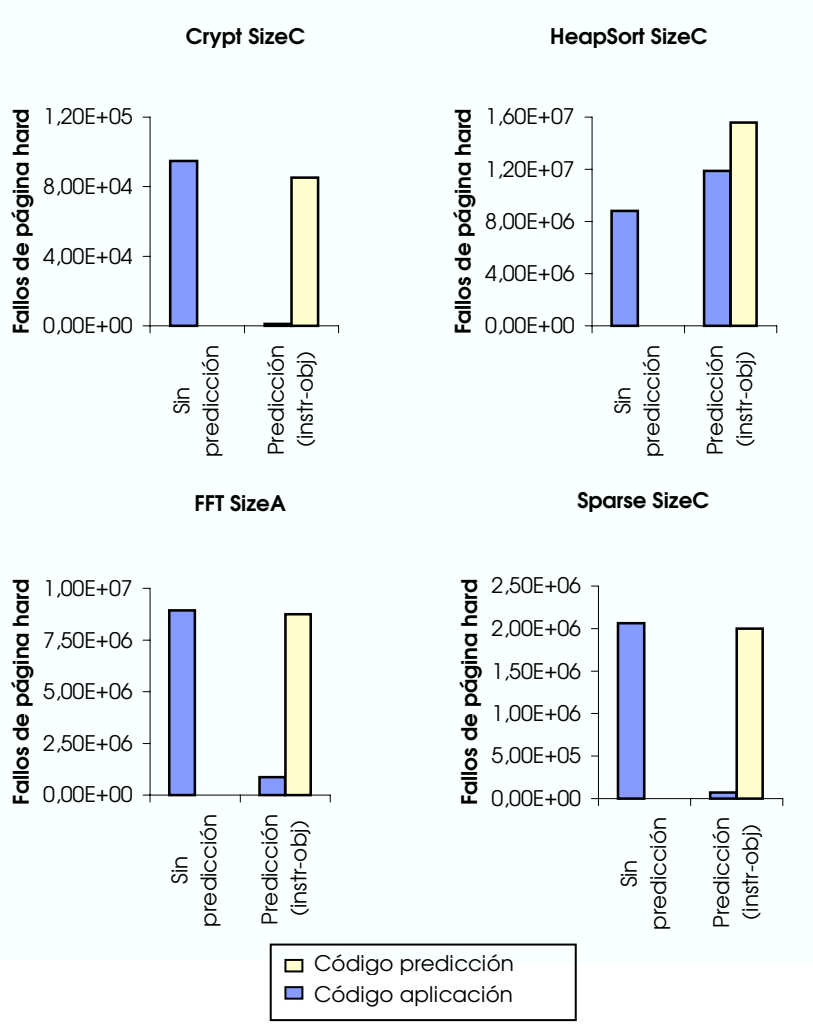

Figura 5.2 Efectividad de la predicción a nivel de bytecode

En cuanto a HeapSort, en la figura 5.2, podemos observar que los fallos de página provocados por el código de predicción no consiguen evitar los fallos de página del programa, ya que se deben a predicciones erróneas. Es más, al cargar la memoria con páginas no necesarias, se incrementa el número de fallos de página del programa, lo cual penaliza el rendimiento del sistema. Este comportamiento se debe a que este programa no utiliza un patrón de accesos regular y, por lo tanto, no es posible anticipar cuáles serán sus accesos

futuros. La implementación final del código de predicción debe ser capaz de detectar esta situación y desactivar el prefetch, para no perjudicar el rendimiento del sistema.

Hemos querido aprovechar el prototipo para comparar la precisión de la estrategia a nivel de instrucción con la que se podría obtener mediante una caracterización parcial de los accesos. En particular, hemos implementado dentro del prototipo una predicción a nivel global, en la que el algoritmo de predicción se comporta como si no tuviera la información sobre la instrucción que está realizando el acceso ni sobre el objeto situado en la dirección accedida y calcula los strides en función de los últimos accesos a arrays. Hay que destacar que la predicción a nivel global aproxima el tipo de predicción que podría hacer el SO, ya que éste sólo dispone de información sobre la dirección accedida. Es más, la información del SO es incluso menor ya que sólo es informado sobre los accesos que provocan fallos de página y en este prototipo se usa la información sobre todos los accesos.

Además de esta comparación entre las dos estrategias extremo, hemos evaluado también la tasa de aciertos que se tendría manteniendo el patrón de accesos sólo a nivel de instrucción (sin tener en cuenta el objeto que utiliza en cada momento), y manteniéndolo sólo a nivel de objeto (sin tener en cuenta desde qué instrucción se accede al objeto). El objetivo de la evaluación de estas estrategias intermedias es determinar si es posible optimizar la selección de páginas reduciendo la cantidad de datos manejada, sin renunciar a la precisión obtenida mediante la caracterización completa de los accesos.

En la figura 5.3 presentamos los resultados de emular estos tipos de predicciones sobre los tres programas con patrón de accesos predecible: CRYPT, FFT y SPARSE. Podemos observar que, para los tres programas, la predicción que tiene un peor comportamiento (la que evita menos fallos de página del programa) es la que se ejecuta sin tener en cuenta ni el objeto accedido ni la instrucción en ejecución (en la figura 5.3, *Predicción global*). Es decir, intentar predecir los próximos accesos usando sólo la información disponible en el SO ofrece una baja tasa de aciertos.

Los tres programas también coinciden en comportarse de forma similar tanto si se mantiene el patrón a nivel de instrucción y teniendo en cuenta el objeto accedido (en la figura 5.3, *Predicción (instr-obj)*), como si se mantiene únicamente a nivel de instrucción (en la

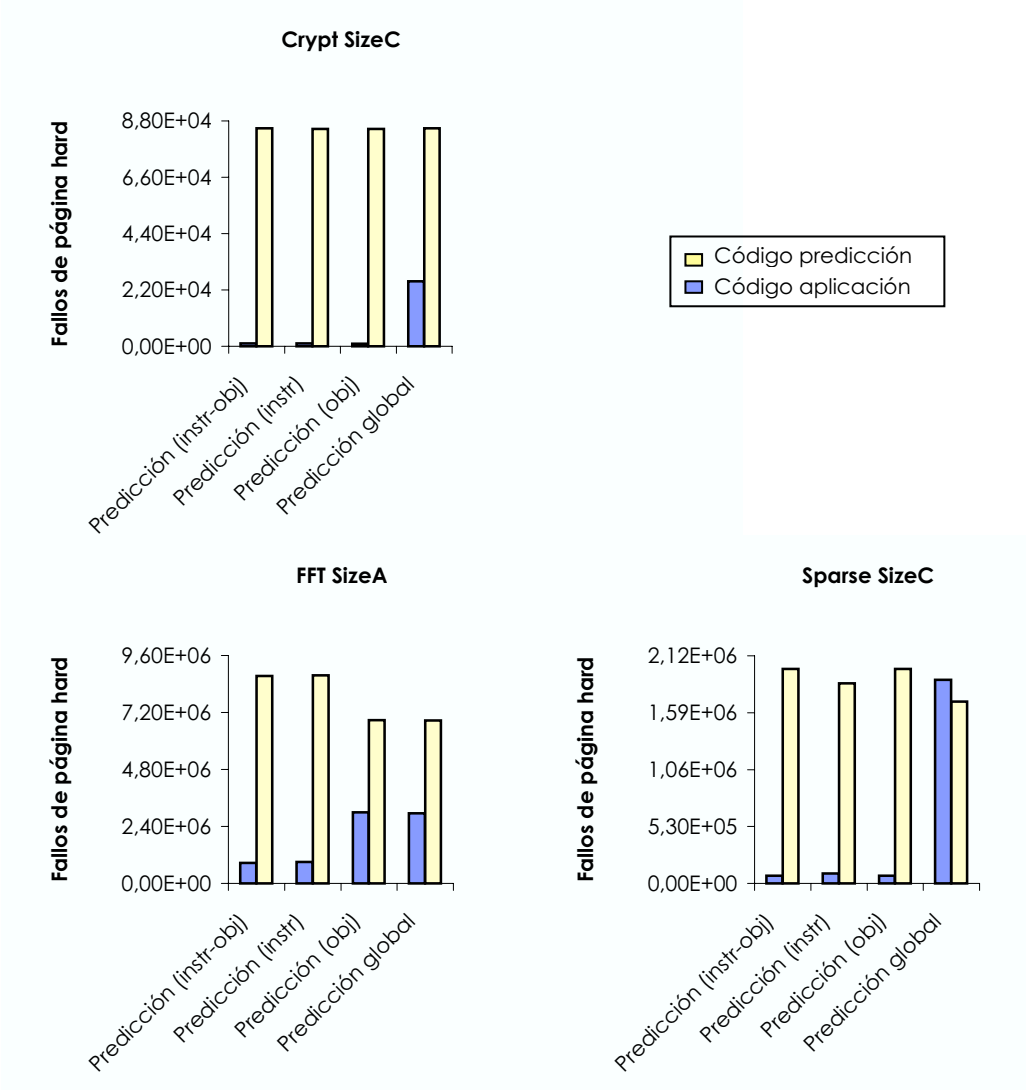

Figura 5.3 Información sobre accesos y eficacia de la predicción

figura 5.3, *Predicción (instr)*). Esto es debido a que, para estos tres programas, en la mayor parte de casos una instrucción sólo accede a un objeto, y por lo tanto no hay variaciones en el patrón obtenido en ambos casos. El único caso en el que hay instrucciones que acceden a varios objetos lo encontramos en el programa CRYPT, en el que existe una función que se ejecuta dos veces y cada una de ellas sobre objetos diferentes. Por lo tanto, para este programa, podría esperarse alguna diferencia entre ambas estrategias de predicción. La explicación para que esto no sea así la encontramos en el patrón de accesos de la función.

Esta función ejecuta un único recorrido secuencial (stride 1) sobre cada objeto, ambos del mismo tamaño. Es decir, la caracterización correcta de los accesos de esta función, consistiría en mantener dos patrones independientes idénticos: stride 1 que se aplica tantas veces como elementos tiene el objeto. Sin embargo, si no se distingue entre los objetos accedidos, se mantiene un único patrón para cada instrucción que utiliza dos strides. El primero es el que se aplica para recorrer secuencialmente los dos arrays. El segundo es el que se detecta al cambiar el array que se está recorriendo, es decir, después de usar tantas veces el primer stride como elementos tiene un array, y su valor es la separación entre los dos arrays. Hay que decir que este segundo stride sólo se intentará aplicar una vez, al terminar el recorrido del segundo array, momento en el que se acaba la segunda y última invocación de la función. Por este motivo, sólo se realiza una predicción errónea.

Generar los patrones de acceso considerando únicamente el objeto, sin tener en cuenta la instrucción desde la que se accede (en la figura 5.3, *Predicción (obj)*), puede ser efectivo si todas las instrucciones que lo utilizan lo hacen con el mismo patrón. En el caso de CRYPT esto es así, ya que todas las instrucciones acceden de forma secuencial a los objetos. El programa SPARSE accede siempre secuencialmente a tres de sus objetos mientras que otros dos pueden ser accedidos secuencialmente (durante la inicialización) o de forma aleatoria (durante el bucle de la multiplicación). Sin embargo, debido al tamaño de estos dos objetos y al uso que se hace de ellos, su impacto sobre el rendimiento de la memoria virtual es muy pequeño, por lo que no se se aprecia diferencia entre la predicción considerando sólo el objeto o considerando tanto el objeto como la instrucción que accede. En cuanto a FFT, este programa accede desde varias instrucciones y con diferentes patrones de acceso al array de gran tamaño que utiliza. Por este motivo, usar sólo la información sobre el objeto para generar el patrón de accesos no es suficiente para predecir de forma correcta los accesos futuros.

Como conclusiones del análisis de este prototipo inicial de predicción podemos decir que, independientemente del código y de los datos del programa, una predicción global es la peor opción para captar el patrón de accesos de los programas. Esta conclusión nos lleva a confirmar que implementar la tarea de selección de páginas de prefetch dentro del SO no es una opción adecuada, dado que, con la información que éste tiene sobre los accesos a

memoria de los programas, sólo podría implementar una estrategia de selección de páginas basada en una predicción global.

En cuanto a la ganancia de tener la predicción a nivel de instrucción y objeto, con respecto a tenerla sólo a nivel de instrucción o sólo a nivel de objeto, dependerá del tipo de uso que las instrucciones hagan de los objetos. En cualquier caso, teniendo en cuenta sólo uno de los dos parámetros (objeto o instrucción) no se mejora la tasa de aciertos de usar ambos parámetros.

5.1.2 Caracterización del patrón de accesos

La JVM tiene toda la información necesaria para poder caracterizar el patrón de acceso de las instrucciones y, en el caso de los accesos strided a arrays lo puede hacer de una manera muy simple.

Para determinar el stride utilizado por una instrucción en el acceso a un array es suficiente con mantener almacenada la dirección referenciada por su ejecución previa sobre ese array y calcular la separación (*stride*) que esa dirección tiene con respecto a la dirección que está referenciando en la ejecución actual de la instrucción.

Si las instrucciones utilizaran sólo un stride en sus accesos, la caracterización consistiría únicamente en calcular el stride que separa los accesos de sus dos primeras ejecuciones sobre el array.

Sin embargo, para ser capaces de tratar con instrucciones que utilizan varios strides en los accesos a un vector, como ocurre, por ejemplo, en las instrucciones que forman parte de bucles anidados, la JVM tiene que ser capaz de detectar diferentes strides y de determinar cuál debe usar en cada momento para predecir los próximos accesos.

Es decir, los parámetros que sirven para caracterizar el patrón de accesos strided son el número de strides utilizados y el valor de cada uno, el número de veces consecutivas que se aplica cada uno de ellos y el orden de utilización entre ellos.

La caracterización se puede dar por finalizada al detectar una transición que completa un ciclo entre los diferentes strides y comprobar que ese ciclo representa el patrón que la instrucción aplica repetitivamente en el resto de sus accesos.

Por ejemplo, consideremos el algoritmo de multiplicación de matrices almacenadas en memoria por filas. En este caso, cada matriz fuente es accedida mediante una instrucción que utiliza tres strides diferentes (ver figura 5.4).

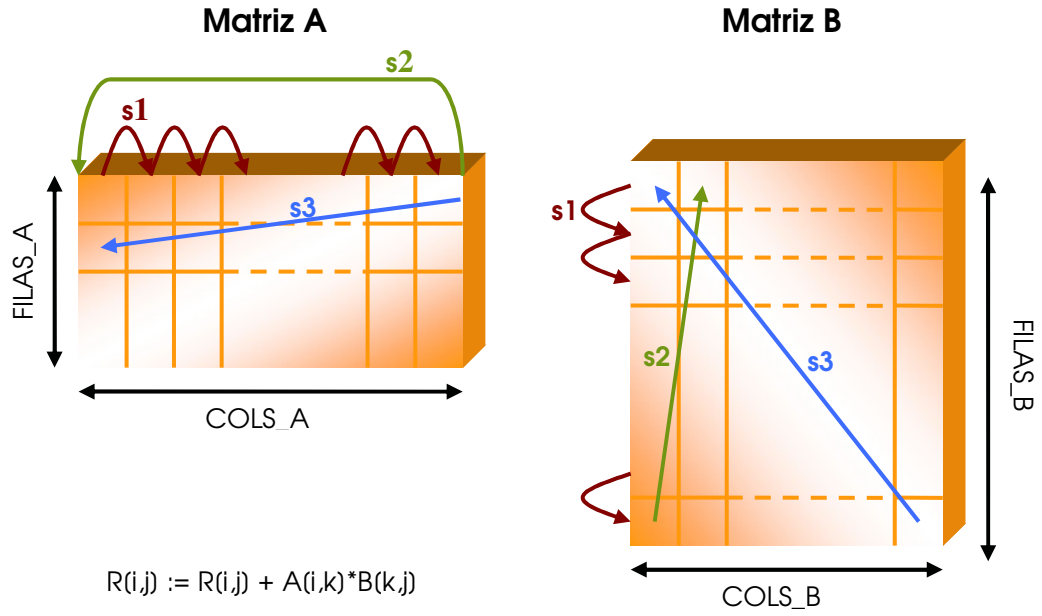

Figura 5.4 Patrón de accesos en la multiplicación de matrices

Para acceder a la matriz que se recorre por filas (matriz A en la figura 5.4), se utiliza un primer stride para recorrer de uno en uno todos los elementos de una fila (s1 == 1). Después de completar el acceso a cada fila, utiliza un segundo stride que sirve para repetir el uso de la fila. Es decir, en cada ejecución de la instrucción que sea múltiplo del número de elementos de una fila se utiliza como stride el valor adecuado para situarse de nuevo al inicio de la fila (s2 == -(COLS_A - 1)). El número de recorridos que se realizan sobre cada fila viene determinado por el número de columnas de la segunda matriz fuente (la que es recorrida por columnas). Después de este número de recorridos, se cambia la fila fuente utilizando un tercer stride. Suponiendo que las matrices se almacenan por filas, su

valor es el mismo que para el stride de primer nivel (s3 == 1). Este patrón de accesos se puede representar mediante el grafo de estados que aparece en la figura 5.5.

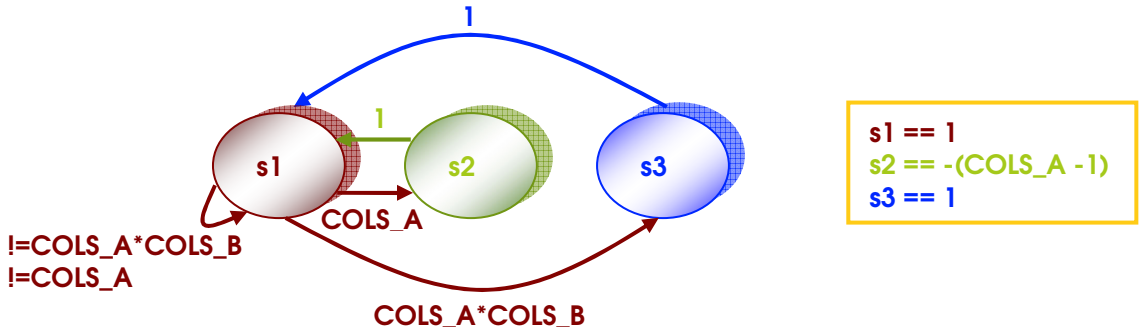

Figura 5.5 Grafo del recorrido por filas en la multiplicación de matrices

En cuanto a la matriz que se recorre por columnas (matriz B en la figura 5.4), el primer stride es el que se utiliza para recorrer todos los elementos de una columna. Su valor depende del número de elementos de la fila (s1 == COLS_B). El segundo stride es el que permite cambiar de columna, y su valor, depende del tamaño de cada fila y del de cada columna (s2 == -((COLS_B * FILAS_B) - 1)). Se utiliza después de haber recorrido toda una columna, es decir, en cada ejecución de la instrucción múltiplo del número de elementos de la columna. El tercer stride es el se utiliza cuando se inicia el cálculo de la siguiente fila destino, es decir, después de haber utilizado tantas veces el primer stride como elementos tiene la matriz, y sirve para volver a recorrer toda la matriz. Su valor es el tamaño de la matriz, pero con signo negativo (s3 == -((COLS_B * FILAS_B) - 1)). En la figura 5.6 representamos el grafo de estados que representa el patrón que siguen los accesos a la matriz recorrida por columnas.

5.1.3 Consideraciones para la optimización de la selección de páginas

Como ya se dijo en el capítulo 4, un requerimiento importante para que la técnica de prefetch sea efectiva es que el tiempo involucrado en su ejecución sea el menor posible, para que no oculte los beneficios de la carga anticipada.

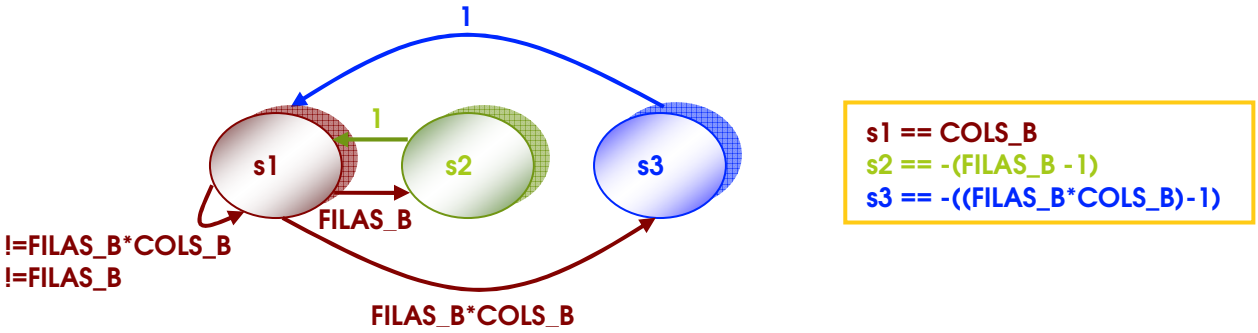

Figura 5.6 Grafo del recorrido por columnas en la multiplicación de matrices

Existen varios aspectos relacionados con la selección de páginas que permiten la simplificación del código ejecutado para realizar esta tarea y que, por lo tanto, se deben tener en cuenta en su implementación.

El primer aspecto es el que se refiere a las instrucciones que siguen un patrón de accesos irregular y, por lo tanto, impredecible. Estas situaciones se deben detectar durante la fase de generación del patrón de accesos para poder desactivar la ejecución del código de predicción para esa instrucción y, de esta manera, evitar la ejecución de un código incapaz de mejorar el rendimiento del programa.

Otro aspecto a tener en cuenta para optimizar la ejecución es evitar selecciones redundantes de páginas de prefetch. El objetivo del algoritmo de predicción es seleccionar las páginas que vamos a cargar con antelación. Esto significa que lo que nos interesa determinar es el stride en unidades de página, en lugar de unidades de elementos del vector, para saber cuales son las páginas que se referenciarán después de la actual.

Además, si una instrucción tiene como predicción la misma página en varias ocasiones diferentes, y entre las dos predicciones la página no ha sido expulsada de memoria, entonces la segunda predicción es redundante y, por lo tanto, se podría evitar la ejecución del código que la ha seleccionado. El código necesario para prever esta situación de forma genérica es complicado, y no tiene sentido implementarlo si el objetivo es minimizar el

coste del código de predicción. Sin embargo existen situaciones particulares en las que se da esta redundancia y que son muy sencillas de determinar.

Por ejemplo, si la ejecución de una instrucción accede a la misma página que su ejecución anterior, calculando el stride en unidades de página, el resultado de la predicción será el mismo y, por lo tanto, no es necesario ejecutar ese código.

También existen patrones de acceso que permiten detectar de forma sencilla predicciones idénticas y aproximar de forma bastante exacta si son redundantes o no. Se trata de las instrucciones que dividen los vectores en subconjuntos de elementos, y cada uno de estos subconjuntos constituyen el working set de la instrucción durante un periodo de tiempo, en el que accede de forma repetitiva a los elementos del working set. Si dos de los working sets caben en memoria, sólo es necesario ejecutar el código de predicción para el primer acceso a cada elemento del working set, para predecir el elemento del siguiente working set que se va a utilizar.

Este tipo de patrón se da con frecuencia en las operaciones sobre matrices cuando la unidad de trabajo es la fila o la columna. Por ejemplo, si las matrices están almacenadas por filas el mismo conjunto de páginas albergará varias columnas consecutivas. Mientras se acceda a esas columnas, se estará usando el mismo working set de páginas de forma repetitiva, hasta que la columna objetivo forme parte del siguiente working set de páginas (ver figura 5.7).

Un patrón de accesos que represente estrictamente este comportamiento tendría como primer stride el que se usa para recorrer todos los elementos de una columna y como segundo stride el que se usa para cambiar de columna. Sin embargo, si es posible mantener en memoria los dos working sets consecutivos al mismo tiempo, se puede optimizar el código de predicción utilizando únicamente un stride que permita predecir la página equivalente del siguiente working set (la que contiene los elementos de la misma fila en el siguiente conjunto de columnas) y aplicar este stride únicamente durante el recorrido de la primera columna de cada working set.

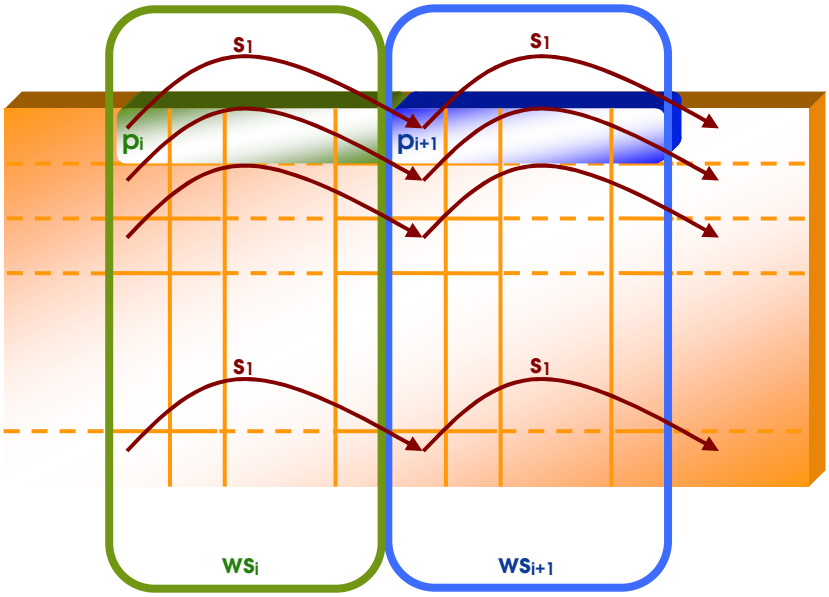

Figura 5.7 Predicción basada en working sets

Para poder aplicar esta simplificación en el patrón es necesario, además de detectar el uso de working sets con estas características, comprobar que es posible mantener dos de ellos en memoria. Hemos basado esta simplificación en dos heurísticas.

La primera heurística es para decidir si es posible mantener simultáneamente en memoria dos working sets de una instrucción. Consiste en comprobar, una vez detectado el uso de working sets, si los dos últimos working sets cargados permanecen en memoria. Si es así se puede asumir que para el resto del recorrido se mantendrá este comportamiento, y por lo tanto, se puede aplicar el patrón de accesos simplificado.

La segunda heurística es necesaria para determinar si un conjunto de páginas está cargado en memoria o no. Habitualmente esta información sobre el estado de las páginas no está disponible desde el nivel de usuario. Para poder utilizar estos datos sin modificar el SO, y así conseguir una estrategia de prefetch totalmente transparente al sistema, hemos implementado un bitmap en el nivel de usuario que representa de forma aproximada el estado de las páginas del heap de la aplicación. Hay que decir que esta heurística

servirá también para poder filtrar de las páginas seleccionadas para prefetch las que se encuentran presentes en memoria física, y así evitar las peticiones innecesarias de carga.

Respecto al resto de selecciones redundantes de páginas que no se pueden evitar de una forma eficiente, sí que se puede evitar la solicitud de su carga utilizando esta segunda heurística y consultando el bitmap que aproxima el estado de las páginas del heap. Es decir, antes de hacer efectiva la solicitud de carga de una página seleccionada se debe comprobar en el bitmap si ya está presente en memoria física y, si es así se debe filtrar del conjunto de páginas seleccionadas. De esta manera, aunque no somos capaces de evitar la ejecución del código de predicción que ha generado estas solicitudes, sí que evitamos la sobrecarga asociada al intento innecesario de carga.

5.2 CARGA ASÍNCRONA Y ANTICIPADA

El acceso al área de swap, para leer de forma anticipada las páginas seleccionadas por el código de predicción, permite completar el prefetch de memoria.

Como ya hemos visto en el capítulo 4, este acceso a disco debe realizarlo el SO, para garantizar de forma sencilla y eficiente la fiabilidad de la máquina. Por lo tanto, es necesario definir el mecanismo que se puede utilizar para solicitar esta lectura desde el nivel de usuario.

Para evitar cualquier modificación en el SO, y así conseguir una estrategia de prefetch totalmente transparente al SO, hay que utilizar el mecanismo de fallo de página que, aunque es un mecanismo síncrono, es el único interfaz que desde el nivel de usuario provoca la carga en memoria de páginas que están en el área de swap.

Debido a la sincronía de este mecanismo, el flujo de ejecución del programa no puede encargarse de provocar los fallos de página que desencadenen la carga anticipada, ya que ésta se debe realizar en paralelo con la ejecución del programa. Por este motivo, hemos añadido un nuevo flujo de ejecución en la JVM (*prefetcher*) que únicamente se encarga de solicitar las páginas seleccionadas. Es decir, este nuevo flujo accede a cada dirección que

interesa cargar con antelación, provocando el fallo de página que desencadena su carga y bloqueándose hasta que la carga concluye, mientras el flujo de ejecución del programa continúa la ejecución.

Además del flujo de prefetch, hemos añadido las estructuras de datos y el código necesario para implementar la comunicación y la sincronización necesarias entre el prefetcher y el flujo de ejecución del programa.

El flujo de ejecución debe comunicar al prefetcher las páginas que ha seleccionado para cargar con anticipación. El mecanismo seleccionado para la comunicación tiene que respetar la asíncronía necesaria para que la carga sea efectiva. Es decir, el flujo de ejecución del código del programa no debe bloquearse tampoco para hacer efectivo este paso de información. Este objetivo se puede conseguir fácilmente implementando la comunicación entre ambos flujos mediante un buffer circular compartido, donde la JVM vaya dejando las direcciones que interesa cargar con antelación, y el prefetcher las vaya recogiendo para solicitar del SO su carga.

La implementación de este mecanismo de comunicación lleva asociada la necesidad de decidir el comportamiento de ambos flujos si se da el caso de que no haya peticiones pendientes y si ocurre que el buffer se llena porque la velocidad de generación de peticiones de la JVM es demasiado alta comparada con la velocidad con la que se pueden resolver esas peticiones.

En el caso de que el prefetcher no tenga peticiones pendientes, para no consumir CPU innecesariamente, lo más adecuado es bloquearlo hasta que la JVM tenga preparada nuevas solicitudes. Por lo tanto, cuando el código de selección de páginas de prefetch introduzca una nueva petición en el buffer, deberá comprobar si el prefetcher estaba bloqueado para, en ese caso, desbloquearlo y permitir que continúe con las solicitudes de carga.

Para tratar el caso del buffer lleno, hay que tener en cuenta que el código del programa no debería bloquearse como consecuencia de una solicitud de carga anticipada, para conseguir el solapamiento entre carga y cálculo. Teniendo en cuenta, además, que otro

de los requisitos es obtener un código eficiente, la única opción es descartar alguna de las solicitudes de carga anticipada. Sin embargo, la pérdida de peticiones de prefetch es una situación que se debería evitar en la medida de lo posible, ya que se pierden posibles puntos de optimización del rendimiento. Por este motivo, es importante dimensionar de forma adecuada el buffer de peticiones.

El mecanismo de solicitud de carga descrito en esta sección se puede implementar mediante un código muy simple. Sin embargo, para que la implementación sea eficaz, debe tener en cuenta varios aspectos de bajo nivel, como por ejemplo, la influencia que puede tener la política de planificación de flujos del SO, la influencia en el rendimiento de los cambios de contexto necesarios entre los dos flujos, o el coste de ejecutar las operaciones de sincronización.

Además, hay que tener en cuenta que estamos implementando una política de prefetch totalmente transparente al SO. Es decir, el prefetcher, desde el nivel de usuario, toma decisiones que forman parte de la gestión de memoria de un programa y que, por lo tanto, interaccionan con las decisiones de gestión de memoria que el SO sigue tomando sin tener en cuenta este nuevo código. Esto hace necesario analizar esta interacción para comprobar su efecto sobre el rendimiento final del programa.

5.3 VISIÓN GENERAL: PREFETCH A NIVEL DE USUARIO

En la figura 5.8 presentamos una visión general de la estrategia de prefetch implementada por completo en el nivel de usuario.

En esta estrategia hemos modificado únicamente la JVM, de manera que el prefetch es totalmente transparente al SO. Además, cualquier programa ejecutado sobre la JVM puede beneficiarse del prefetch, sin que sea necesario ni tan siquiera recompilarlo.

Para cada bytecode de acceso a array se ejecuta el código de selección de páginas que, si es necesario, actualiza la información sobre el patrón de accesos de la instrucción usando

los datos del acceso actual y de la predicción anterior, y se encarga de decidir si es adecuado cargar anticipadamente alguna página. De ser así, dejará la dirección de la página seleccionada en el buffer compartido con el flujo de prefetch y continuará con la ejecución normal del bytecode. Por su parte, el flujo de prefetch accede a este buffer para recoger las solicitudes de prefetch, y las lleva a cabo simplemente accediendo a la dirección involucrada. Si la página está almacenada en el área de swap, el acceso provocará un fallo de página que bloqueará al prefetcher hasta que la rutina de atención al fallo de página complete la carga.

El código de selección de páginas no dispone de una información exacta sobre las páginas presentes en memoria, por lo tanto puede ser que se hagan solicitudes de prefetch para páginas ya presentes. En este caso, el acceso de prefetch se convierte en un acceso a memoria normal, y por lo tanto no implica accesos a disco innecesarios.

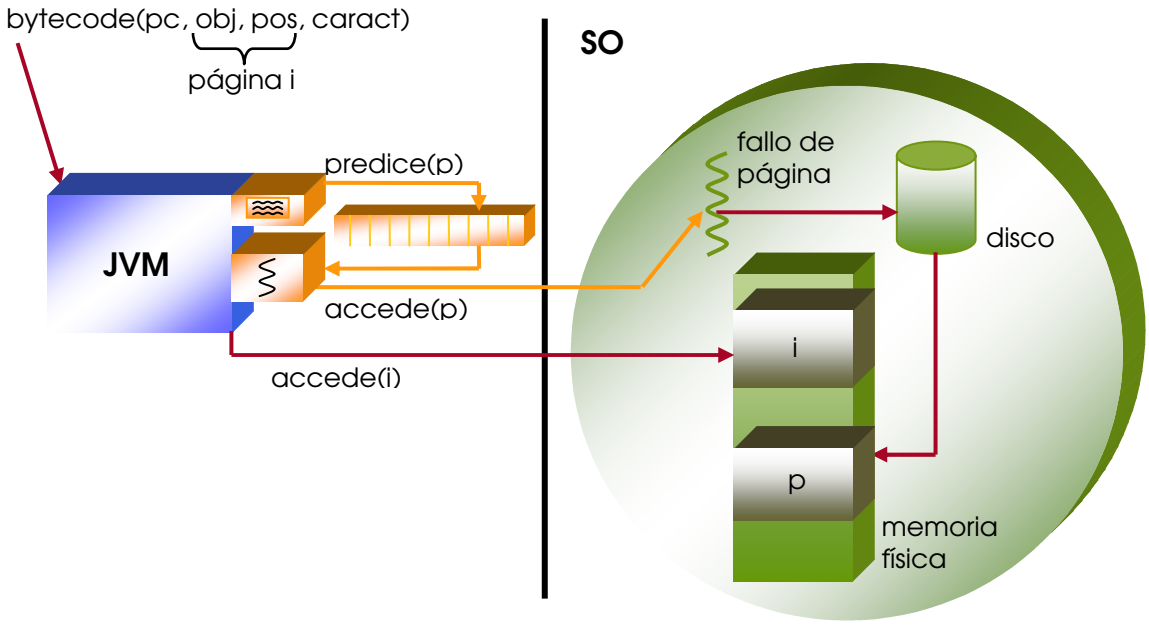

Figura 5.8 Visión general del prefetch transparente al SO

5.4 IMPLEMENTACIÓN DEL PREFETCH A NIVEL DE USUARIO

En esta sección describimos de una forma detallada la implementación del prefetch a nivel de usuario. Los algoritmos y las estructuras de datos involucradas en el prefetch son a priori sencillos. Sin embargo, para obtener una implementación eficaz hemos tenido que ser muy cuidadosos por dos motivos principales.

El primer motivo es que el código de prefetch, presumiblemente, se va a ejecutar con mucha frecuencia y, por lo tanto, es primordial que sea eficiente. Hay que tener en cuenta que recorrer un array de grandes dimensiones requiere millones de ejecuciones del bytecode de acceso. Por lo tanto, la repercusión de cualquier línea de código añadida al tratamiento del bytecode viene multiplicada por ese factor.

El segundo motivo es que estamos implementando una optimización del uso de recursos de forma transparente al SO, lo que implica que el SO reacciona ante este código de la misma manera que ante cualquier código de usuario, sin tener en cuenta su objetivo de gestión, lo que le puede llevar a tomar decisiones de gestión que perjudiquen la ejecución del prefetch o que entren en conflicto con las decisiones de prefetch y, por lo tanto, que perjudiquen su eficacia.

Durante la fase de implementación hemos analizado el tiempo de prefetch invertido en la ejecución de los programas, para determinar los posibles puntos de optimización, tanto los debidos a la ejecución del propio código de prefetch como los debidos a la interacción con el SO. Este análisis nos ha permitido ajustar la implementación hasta obtener un prototipo de prefetch eficaz capaz de mejorar el rendimiento de los programas tipo objetivo de nuestra propuesta.

La implementación que presentamos se ha hecho sobre la versión 2.4.18 del kernel de Linux. Hemos introducido el código de prefetch en la JVM *HotSpot*, versión 1.3.1, que se distribuye con la Java 2 SDK Standard Edition de Sun para Linux. De las dos posibles configuraciones de esta JVM hemos seleccionado la configuración *server*, que está sintonizada para favorecer la ejecución de aplicaciones con alto consumo de recursos. Además, para

facilitar la fase de implementación, hemos modificado únicamente el código del intérprete de la JVM. Sin embargo, las modificaciones que hemos introducido como parte del tratamiento de cada bytecode, podrían formar parte también del código generado por el *JIT* asociado a cada bytecode, consiguiendo que el código de prefetch se ejecutara igualmente.

En la sección 5.4.1 detallamos todos los aspectos relacionados con la implementación de la tarea de selección de páginas de prefetch y en la sección 5.4.2 nos centramos en la tarea de carga de las páginas seleccionadas.

5.4.1 Implementación de la selección de páginas

Para poder introducir el código de selección de páginas en la JVM hemos modificado el tratamiento de aquellos bytecodes que acceden a arrays. En la tabla 5.1 se muestra cuáles son estos bytecodes junto con una breve descripción.

Bytecode	**Descripción**
iaload/iastore	Lectura/escritura sobre un array de enteros
laload/lastore	Lectura/escritura sobre un array de longs
faload/fastore	Lectura/escritura sobre un array de floats
daload/dastore	Lectura/escritura sobre un array de doubles
aaload/aastore	Lectura/escritura sobre un array de referencias
baload/bastore	Lectura/escritura sobre un array de bytes
caload/castore	Lectura/escritura sobre un array de caracteres
saload/sastore	Lectura/escritura sobre un array de shorts

Tabla 5.1 Bytecodes de acceso a arrays

En Java, el acceso a un array es traducido a un bytecode diferente en función del tipo del array y del tipo de acceso que se intenta realizar. Así, cada tipo de array tiene su propio bytecode de lectura de un elemento y su propio bytecode de escritura, con un código específico asociado para su tratamiento. En todos los casos, en el momento de la ejecución de uno de estos bytecodes, la pila de operandos de la JVM contiene la dirección base del array, el índice del elemento sobre el que se quiere efectuar el acceso y, en el caso de los bytecodes de escritura, el valor que se quiere almacenar en esa posición del vector. Además, durante el tratamiento del bytecode también se conoce la posición de ese bytecode dentro del código del programa (es decir, un *program counter lógico*) y se tiene acceso a todas las características del objeto como, por ejemplo, su tamaño.

El código que hemos introducido forma parte del tratamiento de cada bytecode y se ejecuta previamente al tratamiento original. Lo primero que comprueba es que el tamaño del array objeto del acceso sea superior al umbral establecido. Si no se trata de un array *grande* se continúa con el tratamiento original del bytecode.

La siguiente comprobación se utiliza para reducir la cantidad de código de predicción ejecutada, y forma parte de las heurísticas que utilizamos para evitar selecciones redundantes de páginas. Esta comprobación consiste en comparar la página del acceso de la instrucción actual con la que accedió en su ejecución previa sobre el mismo objeto: si la página es la misma, se asume que el código de predicción seleccionará la misma página de prefetch y, por lo tanto, no es necesario ejecutarlo y se continúa con el tratamiento del bytecode.

Si el acceso supera estos dos filtros, entonces se invoca a la función de predicción correspondiente, pasándole como parámetros los datos necesarios para describir el acceso: instrucción desde la que se realiza (el program counter lógico), la dirección base del array, el tamaño de cada elemento del array y el índice del elemento accedido. Cuando la función de predicción retorna, se continúa con la ejecución original del bytecode (ver figura 5.9).

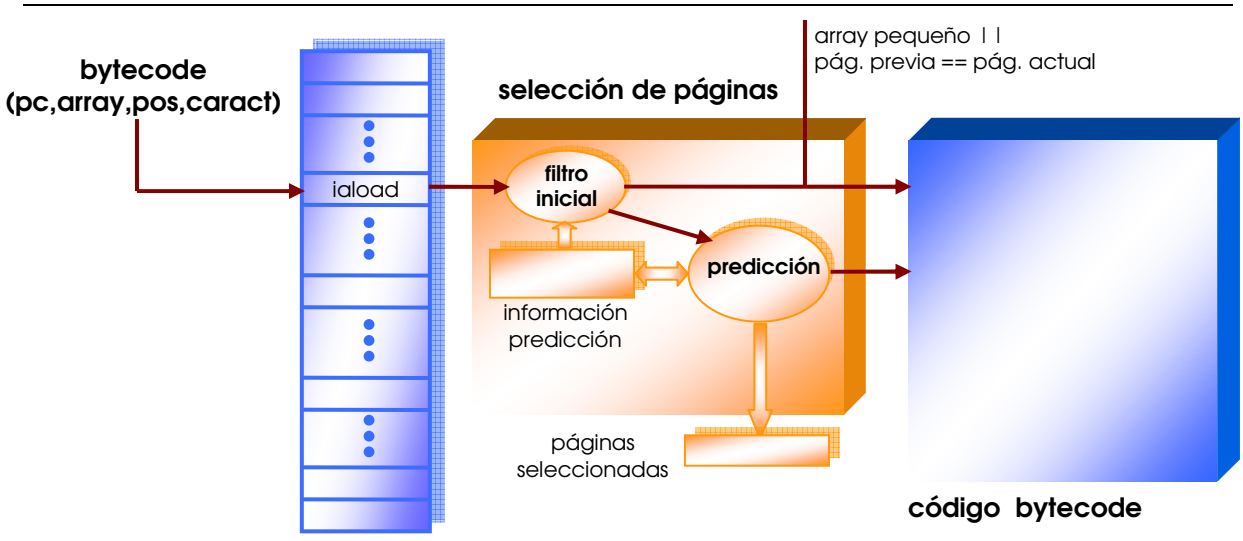

Figura 5.9 Visión general de la implementación de la selección de páginas

En este punto hay que comentar que la JVM que estamos modificando, inicia el tratamiento de cada bytecode con un código que se ha escrito en ensamblador, con el objetivo de que el ejecutable generado para la JVM sea más optimizado que el generado por el compilador. Sólo para aquellos bytecodes que requieren un tratamiento complejo se ha primado el objetivo de la portabilidad del código de la JVM y se ha utilizado un lenguaje de alto nivel para implementar las funciones que completan el tratamiento y que se llaman desde el código inicial. El tratamiento adicional que hemos introducido para los bytecodes de acceso a arrays sigue el mismo criterio: tiene los dos filtros iniciales implementados en ensamblador y sólo si es necesario ejecutar el código de predicción se invoca a una función escrita en alto nivel. De esta manera optimizamos el código necesario para descartar la predicción de las instrucciones que no superan los filtros: primero, porque el código en ensamblador que hemos escrito está más optimizado que el generado por el compilador y, segundo, porque evitamos ejecutar todo el código necesario para gestionar la llamada a la función y el correspondiente salto al código de la misma.

Para implementar la predicción a nivel de instrucción, necesitaríamos almacenar, para cada par instrucción-objeto los datos necesarios para predecir los accesos de esa instrucción sobre ese objeto. Sin embargo, para esta implementación hemos optado por asociar para cada instrucción los mismos datos de predicción independientemente del objeto al que esté accediendo. Los experimentos que hemos realizado previamente nos han mostrado que para nuestros programas de prueba es suficiente mantener esta información de predicción. Esto es así porque para la mayoría de estos programas cada instrucción utiliza sólo un objeto. Además, para los casos en los que no es así, el cambio de objeto se puede tratar como un nuevo stride cuyo valor sea la separación entre el objeto anterior y el actual (ver los resultados que se muestran en la figura 5.3, sección 5.1.1). Esta simplificación permite optimizar, sobre todo, la localización de la información que se debe usar en cada predicción.

Los datos de predicción incluyen, no sólo el valor de los parámetros que determinan el patrón de accesos de la instrucción sino también, la función de predicción utilizada para seleccionar las futuras referencias de la instrucción sobre el objeto. De esta manera somos capaces de utilizar una función de predicción adaptada a las características de la instrucción y del objeto.

Para cada uno de los datos necesarios, hemos definido un array que contiene su valor para cada instrucción (ver figura 5.10). Se podría haber organizado toda esta información en un único array de estructuras, donde cada campo de la estructura fuera uno de los datos necesarios. Sin embargo, hemos observado que el compilador presente en nuestro entorno para compilar la JVM genera un código menos eficiente cuando el acceso a todos los datos de la predicción requieren acceder a campos de una estructura. Esta disminución de la eficiencia en el acceso a los datos adquiere una gran importancia cuando se trata de accesos que se realizan millones de ocasiones.

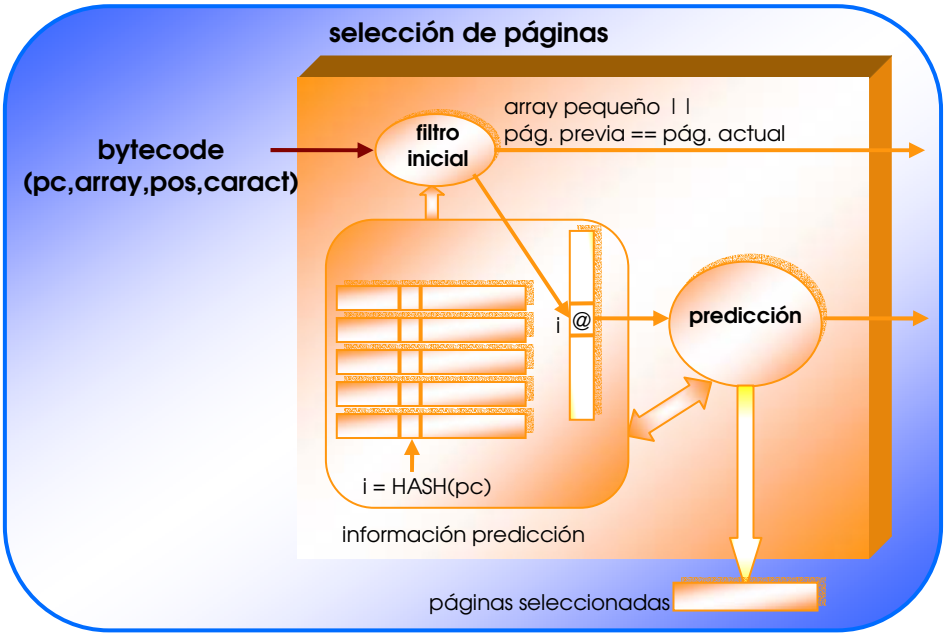

Figura 5.10 Organización de la información de predicción

Con los datos de la predicción organizados en arrays, dada una instrucción es necesario localizar de forma eficiente sus datos. Hemos implementado la indexación de estos arrays mediante una función de hash. El comportamiento ideal de la función de hash es el que no genera ninguna colisión, es decir, el que establece una biyección entre instrucción y posición del array, ya que de esta manera se minimiza el coste de la indexación. Experimentalmente, hemos obtenido una función de hash que exhibe este comportamiento ideal

para todos los programas que hemos evaluado. Esta función obtiene el índice del array basándose el valor del program counter lógico (ver figura 5.11).

```
#define HASH(pc) ((pc & 0xF000) ≫ 1 ) | (pc & 0x7FF)
```

Figura 5.11 Función de hash para localizar la información de predicción

Implementación del algoritmo de predicción

En la implementación de la predicción, hemos simplificado el algoritmo de generación de patrones de acceso limitando el número máximo de strides considerados a tres. Por lo tanto, el algoritmo que hemos implementado es capaz de captar el patrón de acceso de las instrucciones que forman parte de bucles de hasta tres niveles de anidación como máximo. La generalización de este algoritmo no sería complicada. Es más existen muchas propuestas en la literatura sobre predicción del comportamiento de los programas dedicadas a este propósito [PZ91, CKV93, GA94, VK96]. Sin embargo, el algoritmo que proponemos es capaz de detectar los patrones de todos los benchmarks que hemos utilizado y nos sirve para demostrar los beneficios de nuestra propuesta. Por lo tanto, una implementación más genérica del algoritmo de predicción queda fuera de los objetivos que nos hemos planteado en este trabajo.

Este algoritmo, pues, intenta detectar hasta tres strides diferentes así como el número de ejecuciones consecutivas de cada uno de ellos y las transiciones entre ellos. Para ello calcula, para cada instrucción, la separación entre dos accesos consecutivos. Si el valor coincide con el stride utilizado para el acceso previo, entonces incrementa el número de usos consecutivos del stride. Si por el contrario es un valor diferente, considera que es el siguiente nivel (cambio de iteración en el bucle exterior) y registra el nuevo valor, actualizando también la información sobre el orden de transiciones entre strides.

Además hemos querido que el algoritmo de predicción fuera capaz de tratar con aquellas instrucciones que, aún usando un único stride para cada recorrido del array implicado en un bucle interno, el valor de este stride es diferente al cambiar de iteración en alguno de los bucles externos. Para tratar con este caso, al iniciar cada grupo de aplicaciones del primer

stride se comprueba si el valor del stride ha cambiado y se actualiza en consecuencia los parámetros de la predicción.

El otro aspecto relacionado con el algoritmo de predicción es el que se refiere a las predicciones redundantes que, como comentamos en la sección 5.1.2, se pueden dar en las instrucciones cuyo patrón de accesos referencia repetitivamente páginas de un working set que cabe en memoria y se aplica sucesivamente sobre diferentes working sets. Si además del working set en uso, el sistema es capaz de mantener el siguiente working set en memoria, entonces se puede simplificar el algoritmo de selección para que el primer acceso a una página de un working set provoque la carga anticipada de la página equivalente del siguiente working set.

Para comprobar si la instrucción sigue este tipo de patrón, se debe cumplir que el uso del segundo stride sirva para repetir el acceso al mismo conjunto de páginas, situación que se puede comprobar guardando la primera dirección referenciada por la instrucción y comparándola con la accedida después de aplicar el segundo stride. Además, el tercer stride debe servir para cambiar el conjunto de páginas utilizadas, es decir, debe ser la distancia entre la última página accedida del working set actual y la primera del siguiente. Después del cambio de working set, se debe repetir el uso del nuevo subconjunto mediante los patrones determinados por los otros dos strides.

Una vez comprobado que el acceso al vector viene determinado por este tipo de patrón, se debe comprobar si es posible mantener en memoria simultáneamente dos de los subconjuntos del vector. Como simplificación, hemos supuesto que si los dos primeros working sets se pueden mantener en memoria al mismo tiempo, entonces también será posible hacerlo con el resto de working sets utilizados por la instrucción. Si esto es así, se actualiza la información que describe el patrón de accesos, registrando como único stride, la separación entre las páginas equivalentes de los dos working sets y almacenando, además, el número de accesos repetitivos que se hacen sobre un working set, para detectar cuándo la instrucción va a cambiar el working set y activar entonces la selección de páginas para el primer acceso a cada página del nuevo working set.

Hay que decir que si se captan correctamente los parámetros que caracterizan el patrón de accesos de una instrucción, entonces no es necesario calcularlos para cada ejecución de la instrucción, sino que es suficiente con aplicarlos para predecir los próximos accesos de la instrucción. Este método también permite reducir la cantidad de código de predicción que se ejecuta para cada instrucción. Por este motivo, el cálculo de cada parámetro pasa por una fase inicial en la que se calcula y se comprueba la efectividad del valor captado. Si de estas comprobaciones se deriva que el valor del parámetro es estable a través de las ejecuciones, entonces se considera validado y se pasa a la fase estabilizada, que simplemente, consiste en aplicar el valor captado sin repetir su cálculo. Si por el contrario, el código de predicción no es capaz de alcanzar la fase estabilizada, se supone que ese parámetro no tiene un comportamiento predecible y se desestima su uso para el resto de ejecuciones de la instrucción. En particular, si no es posible estabilizar el cálculo de los tres posibles strides que estamos considerando, entonces se desactiva la predicción para esa instrucción y la JVM pasa a ejecutar únicamente el tratamiento original del bytecode.

La figura 5.12 representa el grafo de estados por el que pasa el código de predicción asociado a cada instrucción. En esta figura podemos ver la evolución del código de predicción a medida que se validan los diferentes parámetros del patrón de accesos.

Hemos desglosado el algoritmo de predicción en varias funciones, dependiendo del estado en el que se encuentre la generación del patrón de acceso de la instrucción y en función del tipo de patrón obtenido. El objetivo de esta separación es conseguir reducir el código ejecutado en cada predicción al estrictamente necesario según el estado en el que se encuentre la caracterización de los accesos de la instrucción. Así, a medida que el código de predicción valida el valor de los parámetros de predicción, también cambia la función de predicción asociada a la instrucción por otra en la que ya no aparece el código de validación del patrón (ver figura 5.13). Se podría haber implementado con una única función de predicción que, mediante sentencias condicionales decidiera el fragmento de código que se debe ejecutar para el estado actual de la predicción de la instrucción. Sin embargo, hemos detectado que el coste de la ejecución de los condicionales tiene suficiente repercusión sobre el rendimiento final como para recomendar su eliminación.

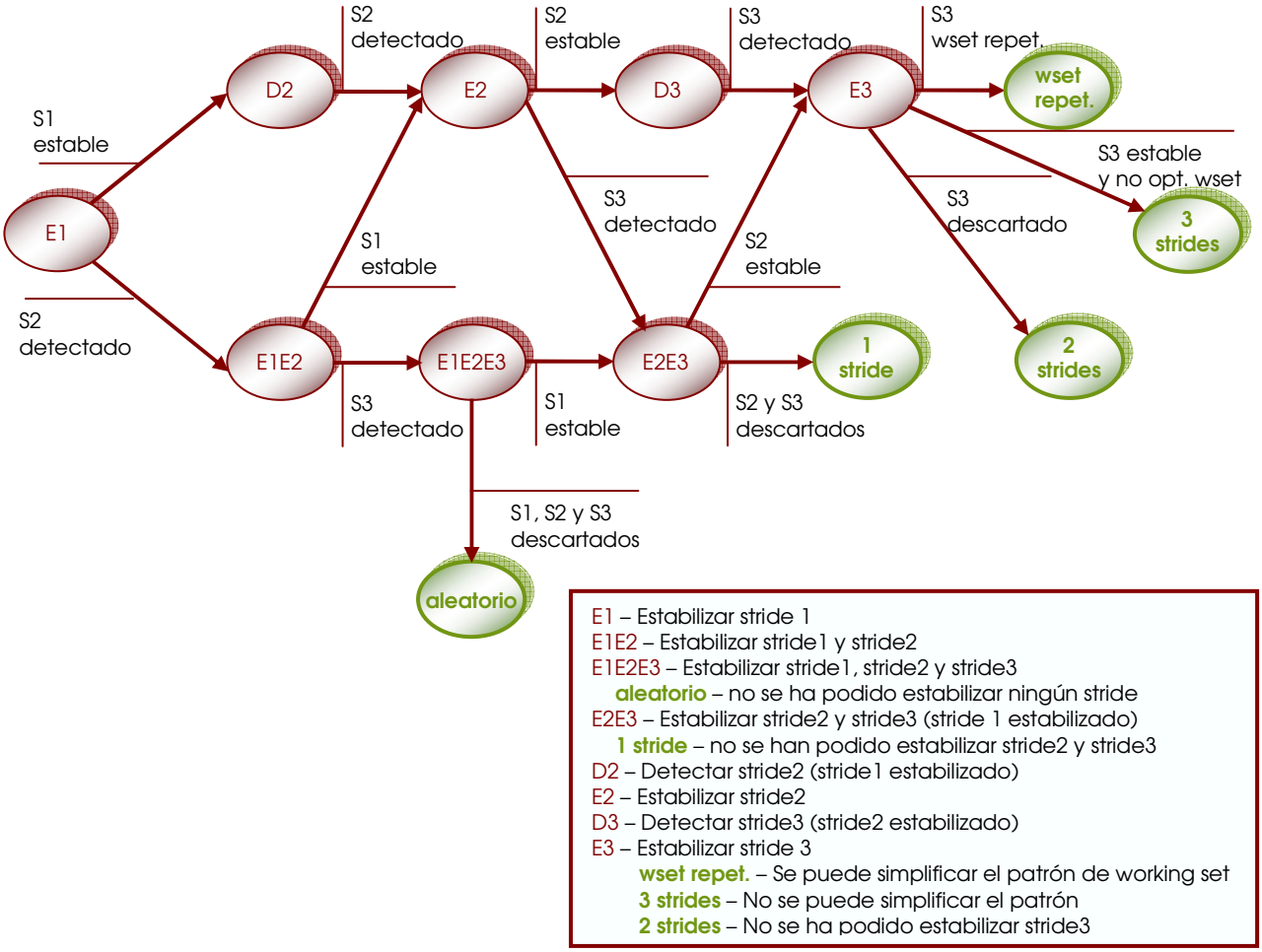

Figura 5.12 Grafo de estados del algoritmo de predicción

Heurísticas para aproximar el estado de la memoria

La tarea de selección de páginas necesita tener información sobre el estado de la memoria del programa en varias de las decisiones que debe implementar. Por ejemplo, esta información es imprescindible para evitar la petición de carga para páginas ya presentes en memoria física y puede ayudar en la decisión de la distancia de prefetch aconsejable para la solicitud de carga anticipada.

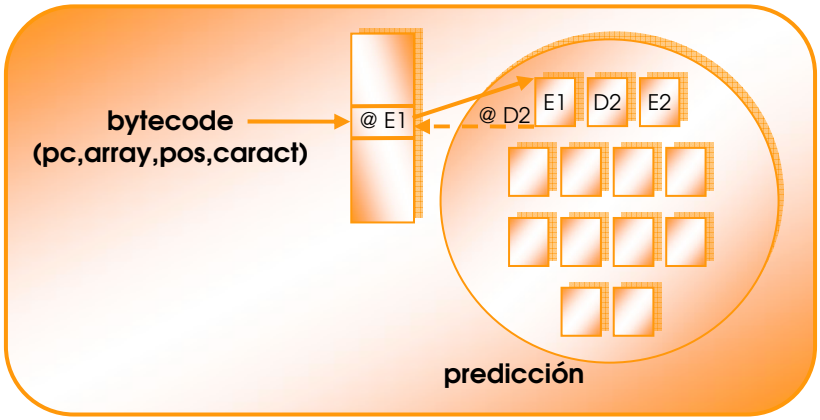

Figura 5.13 Actualización de la función de predicción asociada

Como ya hemos dicho en la subsección anterior, el código de generación de patrón de accesos es capaz de detectar algunos patrones que generan selecciones redundantes de páginas y de adaptar el patrón asociado a la instrucción para evitar estas selecciones en las siguientes predicciones.

Hay que destacar que, la solicitud de una página ya cargada en memoria física, no implica ningún error en la ejecución. Sin embargo, es importante evitarlas para optimizar la ejecución de la tarea de selección de páginas. Además también influye en el rendimiento de la tarea de carga anticipada ya que, aunque la petición de una página ya presente en memoria física no involucra un acceso a disco, la solicitud tiene un coste asociado como, por ejemplo, el debido al cambio de contexto para ceder el uso de la CPU al flujo de prefetch.

Por este último motivo, también se intenta filtrar, de las páginas seleccionadas, aquellas que ya se encuentran en memoria física para evitar el coste de una petición de carga innecesaria.

Para implementar estas dos optimizaciones, es necesario tener acceso, desde el nivel de usuario, a la información sobre el estado de la página en la memoria virtual. Los SO actuales no exportan esta información, por lo tanto, o bien se modifica el código del SO

para introducir un interfaz que permita a la JVM consultar esta información, o bien se utilizan heurísticas que permitan que la JVM aproxime esta información.

Uno de los objetivos de este capítulo es comprobar si es factible dotar al entorno de ejecución de Java de una política de prefetch efectiva y adaptada a las características de cada programa, sin necesidad de modificar el código del SO. Por este motivo, hemos aproximado la información del estado de las páginas mediante una heurística, que, aunque puede ser que no refleje el estado real de la memoria del programa, sirve para evitar suficientes peticiones redundantes como para ser beneficiosa para la ejecución de la tarea de selección.

La implementación de esta heurística consiste en mantener, dentro de la JVM, un bitmap que representa el estado en memoria virtual de todas las páginas del heap de un programa (ver figura 5.14).

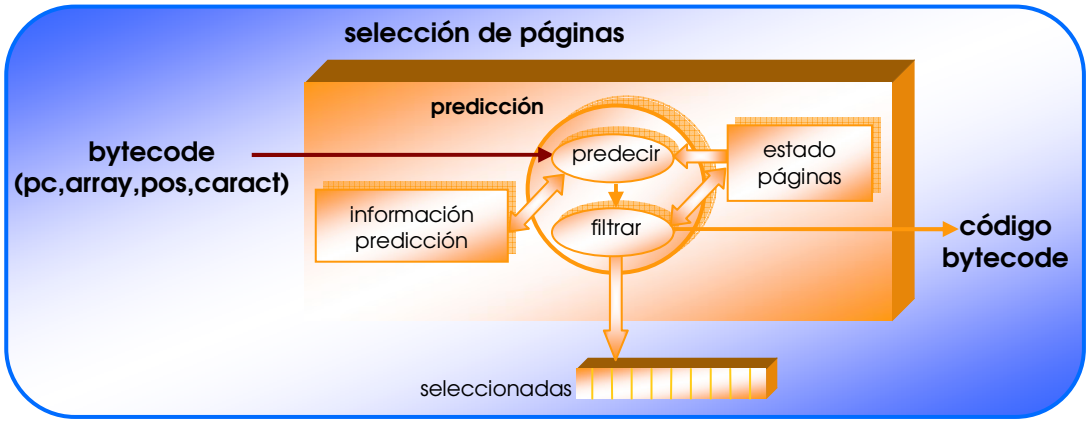

Figura 5.14 Utilización de la heurística que aproxima del estado de la memoria

La JVM marca en el bitmap todas las páginas que son cargadas en memoria, basándose en la información que ella tiene sobre los accesos del programa (i.e. cada vez que una instrucción accede a una página, la JVM actualiza el bit que representa el estado de esa

página). Esta información representa de forma bastante fiel la carga en memoria de las páginas del programa.

El otro aspecto de la memoria virtual, la expulsión al área de swap, está totalmente controlado por el algoritmo de reemplazo que implementa el SO y que decide en cada momento qué páginas son las más adecuadas para abandonar la memoria física y ser almacenadas en el área de swap. Este algoritmo se ejecuta de forma transparente a la JVM, lo que significa que ésta no conoce el instante en el que una página es expulsada y, por lo tanto, debe actualizar su estado en el bitmap. Si bien es cierto que se podría implementar un simulador del algoritmo de reemplazo en la JVM para que aproximara el momento en el que una página abandona la memoria física, el coste de ejecutar este código anularía las ventajas de tener esta información. Por este motivo, hemos optado por una implementación menos precisa, consistente en considerar que todas las páginas son expulsadas al área de swap cuando es necesario ejecutar el algoritmo de reemplazo. En la implementación que evaluamos en este capítulo, se toma la decisión de suponer la expulsión de todas las páginas en base a un nuevo parámetro que recibe la JVM y que le indica la cantidad de memoria física disponible para almacenar los arrays de grandes dimensiones.

Es decir, las páginas marcadas como presentes por el bitmap que hemos implementado son un subconjunto de las páginas que realmente están cargadas en memoria. Esto significa que el uso de este bitmap nunca será la causa de no cargar anticipadamente una página, aunque sí puede ser que no evite todas las peticiones redundantes.

Para completar la selección de páginas, es necesario considerar la distancia de prefetch adecuada para solicitar aquellas páginas para las que el prefetch tenga opciones de ser efectivo. El valor adecuado de la distancia viene condicionado por el propio código del programa y por las condiciones de ejecución. Por lo tanto, es un valor que se debería reajustar en tiempo de ejecución, para lo cual puede ser necesario utilizar la información sobre el estado de las páginas. Para la implementación que evaluamos en este capítulo, la distancia de prefetch es un valor estático (no cambia durante la ejecución) y global (se aplica el mismo valor para todas las instrucciones), que determinamos experimentalmente y que pasamos como parámetro al inicio de la ejecución. Hay que decir que hemos

adoptamos esta simplificación temporalmente, y que, sobre la versión definitiva de nuestra propuesta de prefetch, hemos realizado un estudio sobre el método apropiado para obtener la distancia y la implementación de su cálculo dinámico (ver sección 6.5.3 en el capítulo 6).

5.4.2 Solicitud de carga asíncrona: *prefetcher*

En la sección 5.2 hemos planteado las bases que va a seguir la implementación de la carga anticipada en la estrategia de prefetch transparente al SO. El mecanismo de carga propuesto, aparentemente, se puede implementar de manera muy sencilla. Sin embargo, un análisis detallado de la interacción del mecanismo con el propio código del SO, nos ha desvelado que existen detalles en la implementación del SO que pueden influir en gran manera sobre el rendimiento de la carga anticipada.

En esta sección describimos las decisiones que hemos tomado para la implementación de este código y cómo han venido influidas por la implementación del código del SO presente en nuestro entorno de trabajo.

Las modificaciones que hemos introducido en la JVM consisten en añadir el flujo de ejecución prefetcher que se encarga de hacer efectiva la carga anticipada de las páginas seleccionadas por el flujo de la JVM encargado de la ejecución de los programas.

El código que solicita la carga anticipada es un bucle muy sencillo que consiste en, mientras haya solicitudes pendientes, acceder a la dirección objetivo de la solicitud para que, mediante el mecanismo del fallo de página, se provoque la carga anticipada de la página solicitada (ver figura 5.15). Si en algún momento no hay peticiones pendientes, entonces el código de carga debe parar momentáneamente su ejecución, para evitar el consumo innecesario de CPU, y reanudarlo en cuanto haya una nueva solicitud de prefetch.

Para obtener las solicitudes de carga que debe realizar, el prefetcher accede al buffer compartido con el flujo de ejecución del programa. Extrae las peticiones del buffer siguiendo una política FIFO, es decir, primero solicita la carga de la petición más antigua que ha recibido.

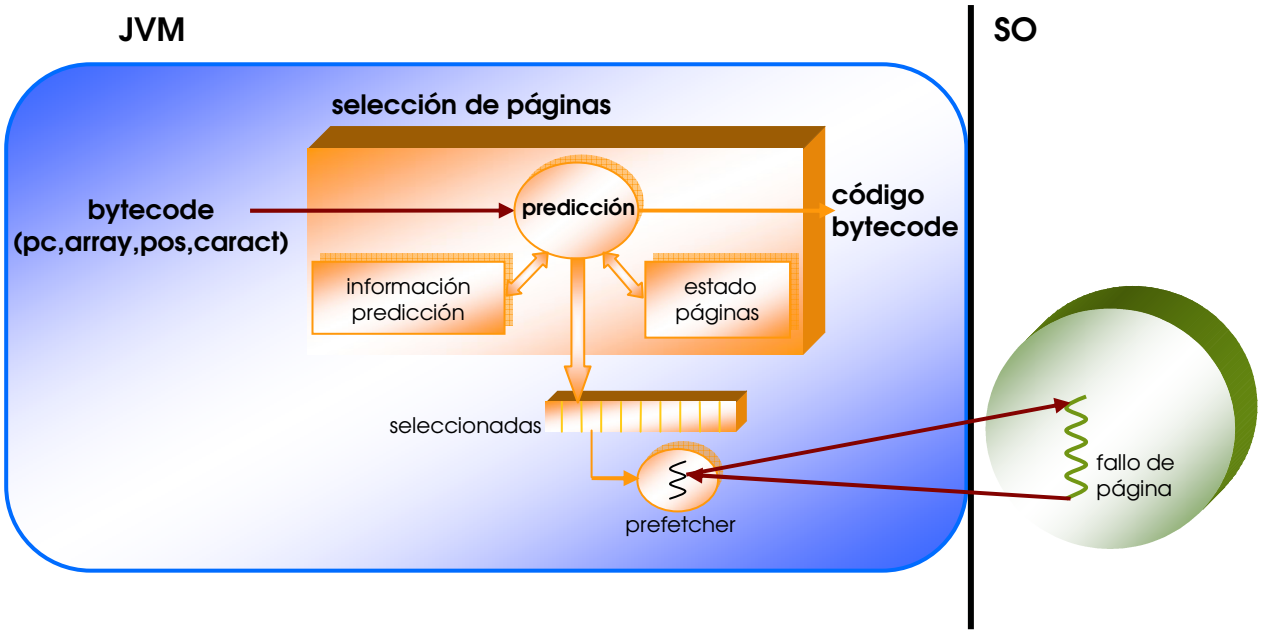

Figura 5.15 Mecanismo de carga asíncrona transparente al SO

Si en algún momento el buffer se queda vacío, es decir, no hay peticiones pendientes, el prefetcher se bloquea sobre un semáforo hasta que la JVM, al introducir una petición en el buffer y comprobar que el prefetcher está bloqueado, actúa sobre el semáforo para desbloquearlo. La necesidad de esta operación de sincronización involucra dos llamadas a sistema. Sin embargo, los experimentos que hemos realizado demuestran que esta sobrecarga es asumible por la técnica de prefetch (ver sección 5.5).

En cuanto a la situación complementaria, que se da cuando la JVM necesita depositar una nueva solicitud de carga y el buffer se encuentra lleno, lo más rápido para el flujo de ejecución del programa es, o bien descartar la petición que lleva más tiempo en el buffer esperando a ser servida, o bien descartar la nueva que no cabe en el buffer. Hemos optado por descartar la petición más antigua y hemos decidido sobreescribirla con la nueva. Esta decisión la hemos tomado asumiendo que las peticiones que hace más tiempo que se han solicitado son las que tienen menos margen para ser cargadas antes de que el programa las referencie y, por lo tanto, la probabilidad de no ser capaces de cargarlas a tiempo es más alta que para el caso de las nuevas solicitudes. De todas maneras, consideramos

necesario evaluar este último extremo para poder asegurar que esta opción es la que ofrece el mejor rendimiento posible. Además, es deseable que esta situación no se de con frecuencia, ya que implica la pérdida de oportunidades en la carga solapada de memoria. Sin embargo, se da la circunstancia de que, para los programas que hemos evaluado, hemos observado que el comportamiento habitual es tener una sola petición pendiente en el buffer. Por lo que todos los parámetros de configuración del buffer (dimensión del buffer, orden de extracción de las peticiones del buffer y tratamiento del caso del buffer lleno), no influyen en el rendimiento de los experimentos que vamos a realizar. Por lo tanto, en la implementación evaluada en este capítulo no hemos profundizado más en el análisis de esta estructura de datos.

Características del flujo de prefetch

El flujo de ejecución que hemos añadido es un nuevo proceso ligero, que hemos creado mediante el interfaz que ofrece el SO de nuestro entorno. Por lo tanto, este nuevo flujo comparte todo el espacio de direcciones con el resto de flujos de la JVM. Esto significa que el prefetcher tiene acceso a todo el heap del programa, algo necesario para que pueda provocar la carga de direcciones usando el mecanismo de fallo de página. Además, esta facilidad también permite optimizar la gestión del buffer de peticiones.

Implementar el prefetch mediante un proceso ligero también tiene repercusiones positivas en la eficiencia de la gestión de procesos implementada por el SO, ya que facilita la optimización del uso de recursos y aumenta el rendimiento de algunas de las operaciones de gestión.

La alternativa a la implementación como proceso ligero habría sido la implementación como un proceso tradicional. En este caso, se tendría que haber usado alguna técnica de memoria compartida entre procesos, para permitir que el prefetcher compartiera con el resto de flujos de la JVM tanto el heap como el buffer de peticiones de prefetch. Hay que decir que en la gestión de memoria compartida entre procesos, el SO debe garantizar la coherencia entre las visiones que ambos procesos tienen de la memoria, lo cual repercute en el tiempo de gestión de la memoria. Hemos evaluado la influencia que el uso de este tipo de memoria tiene sobre el rendimiento de los accesos a memoria de un proceso, y

hemos comprobado que el tiempo de gestión de un fallo de página sobre una zona de memoria compartida es considerablemente mayor que el tiempo de gestión involucrado cuando la zona es de memoria anónima no compartida. Es más, los accesos de escritura, aunque no impliquen un fallo de página, también tienen un coste mayor cuando se hacen sobre una zona de memoria compartida. Así, hemos observado que ejecutar un programa que escribe sobre una zona de memoria compartida tiene un rendimiento 3,5 veces peor que el mismo programa accediendo a una zona de memoria anónima.

Aunque todo parece indicar que la opción de implementar el prefetcher como un proceso tradicional se podría haber descartado sin ningún tipo de evaluación previa, hemos querido realizar esta evaluación porque durante el desarrollo de este trabajo hemos observado cómo detalles de implementación del SO pueden ser los responsables de que los resultados teóricos esperados no se correspondan con la ejecución real. En particular, en versiones anteriores del kernel de Linux, que hemos utilizado durante las fases iniciales de este trabajo (versiones 2.2.12 y 2.4.2), hemos observado un detalle de implementación de la rutina de tratamiento de fallo de página que podría perjudicar el rendimiento del prefetcher implementado como proceso ligero. En esas versiones, en el tratamiento de la excepción del fallo de página se utilizaban *locks* de granularidad muy gruesa, para garantizar la consistencia en las estructuras de datos del sistema y evitar posibles condiciones de carrera. Este método de sincronización era tan radical que no permitía el tratamiento simultáneo de fallos de página de procesos ligeros que pertenecieran a la misma tarea. Por lo tanto, si mientras se estaba resolviendo un fallo de página, el kernel recibía una excepción debido a un acceso al mismo espacio de direcciones, el flujo que hubiera provocado el segundo fallo de página se bloqueaba al inicio de la rutina de gestión de la excepción. Esto era así aunque se trataran de fallos de página totalmente independientes cuya resolución no implicara a las mismas estructuras del sistema y, por lo tanto, no fuera posible una condición de carrera que arriesgara la integridad de los datos del sistema.

Este detalle de implementación interna del SO podría perjudicar el rendimiento de la estrategia de prefetch si se implementaba el prefetcher como un proceso ligero. Ya que los fallos de página provocados por el prefetcher, con el objetivo de solapar la carga de memoria con el cálculo del programa, podrían ralentizar la resolución de otros fallos de página provocados por los flujos de la JVM y anular las ventajas de la carga anticipada.

En las siguientes versiones del kernel de Linux, el mecanismo de protección para evitar las condiciones de carrera en la gestión de los fallos de página se ha refinado, con lo que el problema que podría aparecer al usar un proceso ligero ha desaparecido haciendo que ésta sea la opción más eficiente para la implementación del prefetcher. Sin embargo, este ejemplo de la evolución del código del kernel y de su influencia sobre el rendimiento del prefetch es interesante para mostrar la sensibilidad de la estrategia de prefetch al ser implementada de forma transparente al SO.

Otras características del flujo de prefetch que hay que tener en cuenta son las relacionadas con la política de planificación de flujos del SO. Para que el prefetch sea eficaz, es importante que el SO asigne la CPU al prefetcher siempre que éste tenga una petición pendiente, para que provoque su carga con la antelación adecuada. Una vez solicitada la carga el prefetcher se bloqueará hasta que se complete el fallo de página que ha provocado, con lo que la CPU quedará libre para que el SO se la pueda asignar al resto de flujos del sistema. Para lograr este efecto, hemos utilizado una funcionalidad que ofrece Linux y que permite dar prioridad al prefetcher en el uso de la CPU. Esta funcionalidad es la que sirve para asignar a un flujo la categoría de flujo de *tiempo real*, convirtiéndolo en el candidato a ocupar la CPU siempre que está preparado para la ejecución.

Hemos comparado la influencia de este cambio en la política de planificación del prefetcher y la aplicación que hemos evaluado mejora su rendimiento hasta un 2 % con respecto a planificar al prefetcher mediante la política de planificación que usa Linux por defecto. Este bajo porcentaje de mejora se debe a que la aplicación que hemos utilizado tiene un bajo porcentaje de cálculo, lo que significa que se bloquea con mucha frecuencia, de manera que, aunque se use la política de planificación por defecto, el prefetcher tiene muchas opciones de ocupar la CPU en cuanto está preparado para ejecutarse. Por lo tanto, en una aplicación con mayor tiempo de cálculo se espera que la influencia del cambio de la política de planificación sea mayor.

Influencia de la gestión de memoria del SO

Al añadir el prefetch de memoria al entorno de ejecución, estamos introduciendo una nueva tarea en la gestión de memoria que, inevitablemente, interacciona con el resto de

tareas de gestión de memoria del entorno. Este efecto es más acusado al hacer que la estrategia de prefetch sea transparente al SO. En este caso, el SO puede tomar decisiones genéricas que perjudican el rendimiento del prefetcher y que podría evitar si, por ejemplo, fuera capaz de distinguir entre las páginas solicitadas por pura carga bajo demanda y las páginas solicitadas por el prefetcher.

En la sección anterior ya hemos visto un ejemplo de como la implementación del mecanismo de fallo de página puede influir en el rendimiento del prefetcher. Pero existen más decisiones en la gestión de memoria de Linux que pueden perjudicar el rendimiento de nuestra propuesta.

Como ya hemos dicho, Linux implementa una estrategia de prefetch muy simple que intenta favorecer a las aplicaciones que usan un patrón de accesos secuencial (ver sección 2.2 en el capítulo 2). Esta estrategia consiste en, para cada fallo de página, además de cargar la página que ha provocado la excepción, cargar también las siguientes páginas consecutivas. Por lo tanto, Linux aplicará este prefetch también para las páginas cargadas anticipadamente por nuestra estrategia de prefetch. En el mejor de los casos, si el programa que se está ejecutando sigue un patrón de accesos secuencial, la aplicación simultánea del prefetch de Linux y de nuestra estrategia de prefetch, resulta redundante. Pero en general, para las aplicaciones que siguen patrones de acceso no secuenciales, el prefetch de Linux puede perjudicar el rendimiento de nuestra estrategia de prefetch, ya que para cada página que solicitemos con antelación, Linux cargará otra serie de páginas, que aún siendo innecesarias para el programa, pueden aumentar la presión sobre el sistema de gestión de memoria.

Afortunadamente, Linux permite que su estrategia de prefetch se desactive desde el nivel de usuario, por lo que hemos adoptado esta posibilidad para las ejecuciones sobre nuestro entorno modificado con prefetch en el nivel de usuario.

El otro aspecto de la gestión de memoria que interactúa con la estrategia de prefetch es el reemplazo de memoria. Esta interacción se puede manifestar de dos maneras diferentes: primero, por la propia ejecución del código de reemplazo y, segundo, por las páginas que selecciona para ser expulsadas al área de swap. En este punto hay que comentar, una

vez más, que hemos observado grandes diferencias en el efecto que esta interacción tiene sobre rendimiento del prefetch, dependiendo de la versión de kernel de Linux presente en el entorno de ejecución. De esta manera, una evaluación de nuestra estrategia sobre una versión anterior de kernel (versión 2.4.2), nos demostró que, comparado con los resultados obtenidos sobre la versión de kernel 2.4.18, las interferencias con el reemplazo de memoria penalizaban hasta cuatro veces su rendimiento. Hay que decir que, incluso con esta mala interacción con el algoritmo de reemplazo, nuestra estrategia de prefetch también conseguía mejorar el rendimiento de los programas sobre la versión 2.4.2.

5.5 EVALUACIÓN DEL PREFETCH A NIVEL DE USUARIO

En esta sección presentamos la evaluación de la implementación final que hemos propuesto para la estrategia de prefetch transparente al SO.

Recordemos que la versión de la JVM que hemos modificado con la estrategia de prefetch es la JVM *HotSpot*, versión 1.3.1, que Sun distribuye como parte del entorno de desarrollo del paquete Java 2 SDK Standard Edition para Linux. Además el kernel de Linux sobre el que ejecutamos los experimentos es el 2.4.18, el mismo que hemos utilizado durante la implementación de esta estrategia.

La plataforma física sobre la que hemos ejecutado estos experimentos es un PC con un procesador Pentium III a 500 Mhz y 128Mb de memoria física.

5.5.1 Metodología para los experimentos

Para evaluar la efectividad de nuestra estrategia hemos evaluado los programas de prueba tanto en el entorno original de ejecución como en el entorno modificado con nuestra estrategia de prefetch. En el caso del entorno original, hemos analizado el rendimiento obtenido con el prefetch de Linux activado (comportamiento por defecto) y el obtenido al desactivar esa estrategia simple de prefetch. Para la evaluación de nuestra propuesta

de prefetch, como ya hemos comentado en la sección 5.15, hemos optado por desactivar siempre el prefetch de Linux.

La evaluación ha consistido en contar los fallos de página provocados por la ejecución de los programas, separando los fallos de página provocados por el código del programa de aquellos provocados por el prefetcher ya que, éstos últimos son los susceptibles de ser resueltos en paralelo con la ejecución del programa. Además, también contamos de forma separada aquellos fallos de página provocados por el acceso a direcciones que ya están en proceso de carga. Esta segunda clasificación también nos sirve para evaluar la eficacia del prefetch ya que estos fallos de página se deben a páginas que el prefetcher no ha sido capaz de cargar a tiempo. Además, es importante tenerlo en cuenta a la hora de analizar el tiempo de ejecución ya que, habitualmente, el tiempo involucrado en resolver uno de estos fallos de página es menor que el necesario para resolver un fallo de página completo.

También evaluamos el tiempo de ejecución de los programas, distinguiendo el porcentaje de este tiempo que ha sido necesario para resolver cada uno de los tipos de fallos de página. Esto es necesario para asegurar que, en los casos en los que el prefetcher es capaz de evitar los fallos de página de los programas, se obtiene la correspondiente reducción en el tiempo de ejecución.

Para efectuar estas medidas, utilizamos el mismo mecanismo, basado en los gestores de dispositivos de Linux, que hemos descrito en el capítulo 3, pero adaptando el código a las necesidades de la nueva clasificación. Para esta nueva clasificación, simplemente necesitamos que el kernel considere dos grupos de contadores: los asociados al prefetcher y los asociados al flujo que ejecuta el código del programa. Para ello, hemos introducido una nueva variable de configuración que la JVM inicializa con los identificadores de proceso de los flujos que interesa distinguir.

5.5.2 Programas de prueba

Para efectuar esta evaluación hemos seleccionado como programa de prueba el código de la multiplicación de matrices. El motivo de esta selección es que se trata de un kernel muy utilizado en las aplicaciones de cálculo científico y, además, se trata de un algoritmo

sencillo y muy controlado, que facilita el análisis de los resultados. Hay que decir que esta facilidad para el análisis ha sido especialmente importante durante toda la fase de desarrollo de la estrategia. Durante esta fase, el controlar perfectamente el código del programa de prueba, nos ha permitido entender aquellos puntos del código añadido a la JVM que eran críticos para poder obtener un código eficiente. Y lo que es más importante, nos ha facilitado la tarea de entender el comportamiento de las tareas de gestión de Linux y su interacción con nuestro código, para así poder llegar a una implementación que ofrece un buen rendimiento para la estrategia de prefetch.

Este programa lo hemos ejecutado con diferentes tamaños para las matrices multiplicadas. El objetivo es evaluar el comportamiento de nuestra estrategia bajo diferentes condiciones de presión para el sistema de memoria. A continuación describimos brevemente los tamaños seleccionados y la tabla 5.2 los resume.

- MATRICES PEQUEÑAS: el tamaño de estas matrices se ha seleccionado para que el sistema de memoria fuera capaz de albergar a las tres matrices durante todo el algoritmo de multiplicación. Es decir, el programa no hace uso de la memoria virtual y, por tanto, el prefetch de memoria no puede mejorar su rendimiento
- MATRICES GRANDES: el tamaño de estas matrices es tal que hace necesario el uso de la memoria virtual y, por lo tanto, la estrategia de prefetch podría beneficiarle, y además permite guardar en memoria dos de los working sets que utiliza la multiplicación, lo que significa que el algoritmo de selección de páginas puede optimizar el código de selección.
- MATRICES EXTRA-GRANDES: en este experimento se aumenta la presión sobre el sistema de memoria. El tamaño de los working sets del programa son demasiado grandes como para mantener dos simultáneamente en memoria, con lo que el algoritmo de selección de páginas no puede aplicar la optimización de simplificar el patrón de accesos.

Benchmark	Matriz A (doubles)	Matriz B (doubles)
MATRICES PEQUEÑAS	500x500 (1,9Mb)	500x500 (1,9Mb)
MATRICES GRANDES	1024x4096 (32Mb)	4096x4096 (128Mb)
MATRICES EXTRA-GRANDES	128x28672 (28Mb)	28762x2048 (448Mb)

Tabla 5.2 Tamaños de las matrices de entrada para la multiplicación (AxB)

5.5.3 Rendimiento del prefetch a nivel de usuario

En la figura 5.16 mostramos los resultados obtenidos en la evaluación del prefetch sobre el benchmark que utiliza la matriz que se mantiene por completo en memoria física durante toda la ejecución. Es decir, la multiplicación de matrices se completa sin provocar ningún fallo de página. La figura 5.16.a mostramos el tiempo de ejecución de este programa en el entorno original de ejecución, tanto con el prefetch de kernel activado (en la figura, *JVM original (prefetch kernel)*) como con esta estrategia desactivada (en la figura, *JVM original (sin prefetch kernel)*), separando el tiempo de fallo de página del resto del tiempo de ejecución del programa. En esta figura se puede observar como, efectivamente, el tiempo de cálculo ocupa el 100 % del tiempo de ejecución del programa y, además, dado que se ejecuta sin provocar fallos de página, el prefetch de kernel no tiene ningún efecto sobre su ejecución y el rendimiento del programa en los dos escenarios es idéntico. Por el mismo motivo, nuestra estrategia de prefetch tampoco puede beneficiar el rendimiento del programa. Es más, en esta implementación no hemos incluido todavía la característica de desactivar el prefetch cuando las condiciones de ejecución no lo hacen necesario, por lo que la ejecución de este programa nos sirve para evaluar la sobrecarga añadida por nuestro código de predicción. En la figura 5.16.b comparamos la ejecución en el entorno original con su comportamiento por defecto (i.e. con el prefetch de Linux activado, en la figura 5.16.b, *JVM original (prefetch kernel)*) con el comportamiento del entorno de ejecución modificado con nuestra propuesta de prefetch (en la figura 5.16.b, *JVM con prefetch*). Vemos que en la ejecución sobre el entorno que hemos modificado igualmente el tiempo de cálculo representa el 100 % del tiempo total de ejecución. Por otro lado, el incremento del tiempo total con respecto a la ejecución en el entorno original representa un 12 %. Hay que destacar que una versión definitiva de la estrategia de prefetch debe detectar si un proceso no está utilizando el mecanismo de memoria virtual y entonces desactivarse, con lo cual esta sobrecarga sólo debe tenerse en cuenta para las situaciones en las que el prefetch es beneficioso. En la implementación que presentamos en este capítulo no hemos

incorporado esta desactivación ya que su objetivo es evaluar si la estrategia de prefetch transparente al SO puede mejorar el rendimiento de los programas que requieren el uso de memoria virtual, por lo que hemos pospuesto la implementación de este detalle hasta tener el diseño definitivo de la estrategia.

Este benchmark, además de evaluar la sobrecarga del código de predicción, nos ha permitido analizar los beneficios de las optimizaciones que hemos implementado en la tarea de selección de páginas. Las instrucciones de la multiplicación de matrices siguen un patrón de accesos repetitivo sobre cada uno de sus working sets, de manera que se puede aplicar la simplificación que hemos descrito en la sección 5.1.3 sobre el patrón aplicado en la selección de páginas.

Hemos observado que el rendimiento de este benchmark, aplicando estrictamente el patrón de accesos en lugar de aplicar el patrón simplificado, es más de 6 veces peor que el rendimiento de la estrategia que utiliza la selección optimizada. Si tampoco aplicamos la optimización de evitar las predicciones consecutivas sobre la misma página, entonces el tiempo de ejecución se multiplica por otro factor de 3.

Este alto incremento el tiempo de ejecución demuestra la importancia de intentar evitar la ejecución de código de predicción que genera selecciones de páginas redundantes, y justifica las simplificaciones implementadas en el algoritmo de selección.

En las figuras 5.17 y 5.18 mostramos los resultados de la ejecución del benchmark con el siguiente conjunto de datos. Estos datos tienen la característica de no caber en memoria, y por lo tanto, una técnica de prefetch capaz de captar su patrón de accesos puede mejorar su rendimiento. Además, las dimensiones de las matrices permiten mantener en memoria al mismo tiempo varios working sets de las instrucciones, de manera que el algoritmo de selección de páginas, a medida que se referencian por primera vez las páginas de un working set, puede ir solicitando la carga de las páginas del siguiente working set.

En la figura 5.17 podemos ver el tiempo de ejecución del programa sobre el entorno original de ejecución. Podemos observar que para la ejecución con el comportamiento por defecto (*JVM original (prefetch kernel)*), el porcentaje de tiempo dedicado a resolver

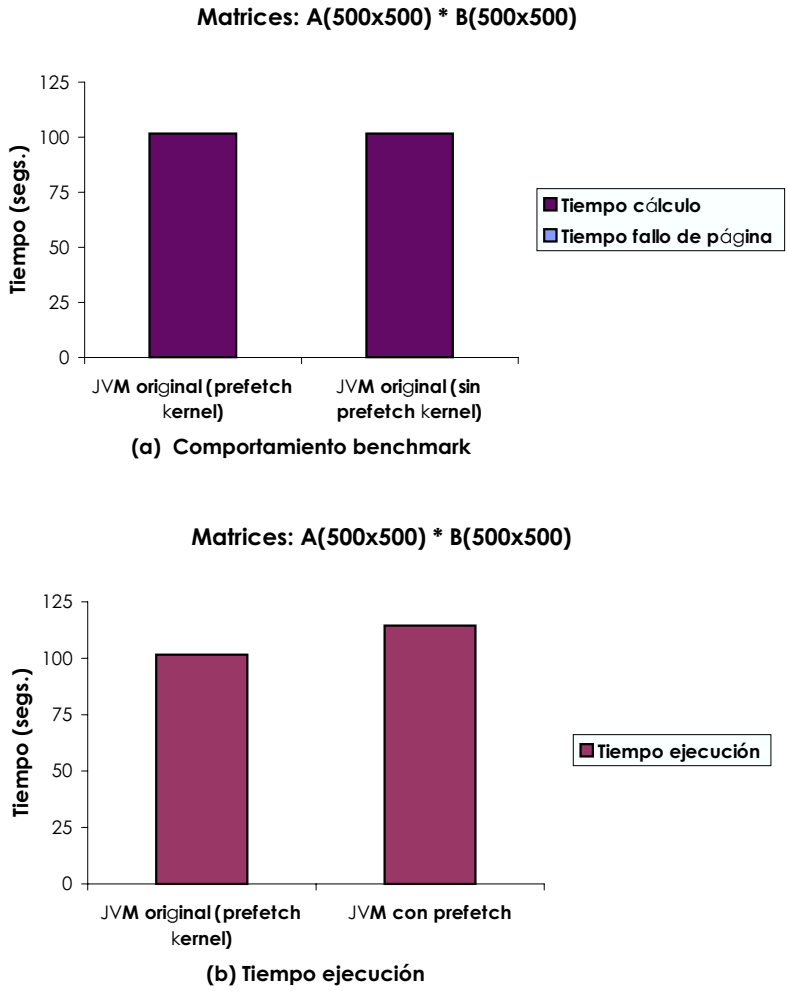

Figura 5.16 MULTIPLICACIÓN DE MATRICES PEQUEÑAS

fallos de página alcanza el 52 % del tiempo total de ejecución. En el caso de la ejecución desactivando el prefetch de kernel *JVM original (sin prefetch kernel)* este porcentaje se incrementa hasta el 84 %. Este porcentaje hay que tenerlo en cuenta a la hora de evaluar los beneficios obtenidos por nuestra técnica, ya que la eficacia del prefetch se basa en el solapamiento del tiempo de carga con el tiempo de cálculo, por lo tanto, los posibles beneficios vienen limitados por el tiempo de CPU consumido durante la ejecución de los programas.

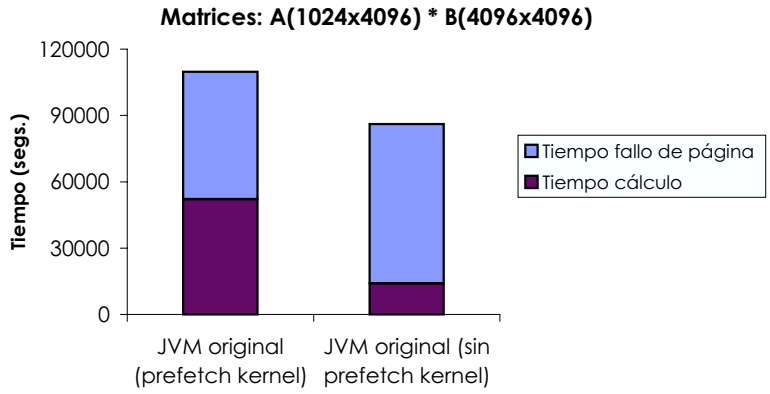

Figura 5.17 Comportamiento de la MULTIPLICACIÓN DE MATRICES GRANDES

En la figura 5.18.a comparamos el tiempo de ejecución sobre el entorno original, tanto usando la configuración por defecto (*JVM original (prefetch kernel)*) como desactivando el prefetch de kernel (*JVM original (sin prefetch kernel)*, y sobre el entorno modificado con nuestra propuesta (*JVM con prefetch*). El primer aspecto a destacar es que la ejecución del prefetch de Linux está perjudicando al rendimiento de este programa, y que, simplemente desactivando esta política ya se consigue una mejora en el rendimiento de los programas. Sin embargo, si ejecutamos el programa con nuestra estrategia de prefetch la mejora aumenta hasta un 43 %, comparado con la ejecución sobre el comportamiento por defecto del entorno original, y hasta un 30 % si se compara con la ejecución sobre el entorno original con el prefetch de kernel desactivado.

En la figura 5.18.b mostramos los fallos de página provocados por el programa en los tres entornos de ejecución que estamos considerando. Hemos separado los fallos de página en función de su tipo: fallos de página que no implican acceso a disco y que incluyen los que se provocan por el acceso a direcciones que ya están en proceso de carga (*fp soft*), o fallos de página para los que el tiempo de espera incluye la carga completa de la página (*fp hard*). Además, para la ejecución sobre el entorno modificado con nuestra propuesta de prefetch, distinguimos entre los fallos de página provocados por el código del programa (*JVM*) y los provocados por el prefetcher (*prefetcher*).

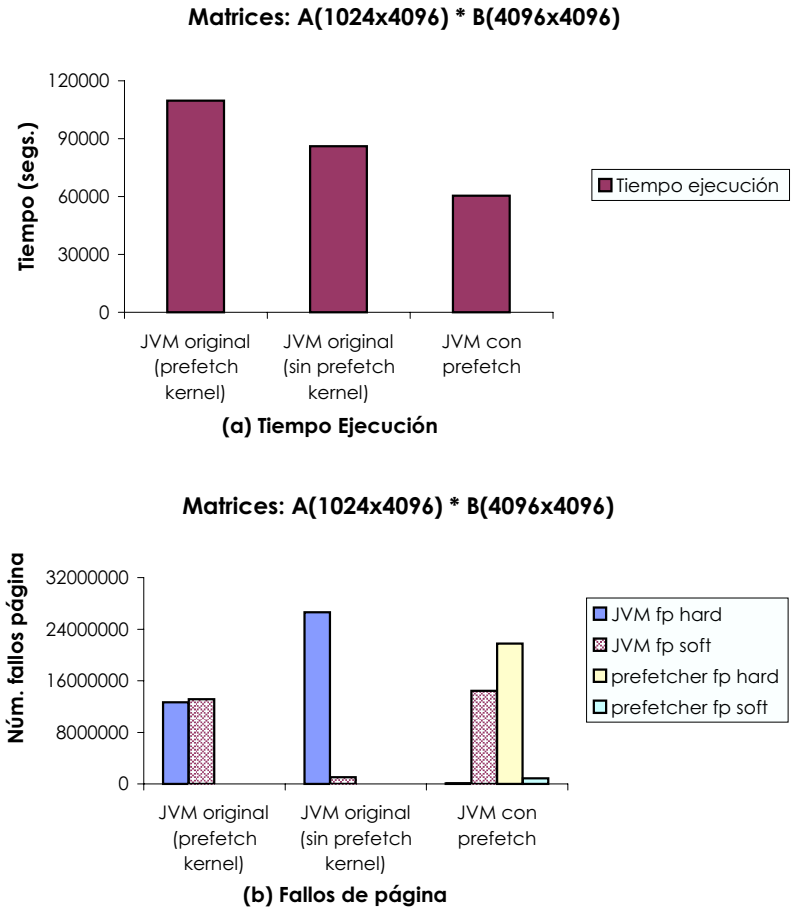

Figura 5.18 Resultados de la MULTIPLICACIÓN DE MATRICES GRANDES

Podemos ver cómo en el entorno original de Linux, el prefetch de kernel no consigue eliminar los fallos de página del programa. Esto es debido a que su algoritmo de predicción no representa los accesos a la matriz recorrida por columnas. Hay que decir que, por un efecto lateral, alguna de las páginas precargadas son utilizadas por el programa. Pero en este caso incluso el rendimiento no es bueno, ya que el programa las necesita antes de que la carga se complete, provocando un fallo de página de los que hemos categorizado como *soft*. Además, la carga de páginas no necesarias incrementa el tiempo de ejecución del programa ya que provoca que el algoritmo de reemplazo se tenga que ejecutar con mayor frecuencia para devolver al área de swap las páginas cargadas sin necesidad.

En el caso de la ejecución sobre el entorno modificado con nuestra estrategia de prefetch, podemos ver que el prefetcher no es capaz de evitar todos los fallos de página del programa. Hay que destacar que los fallos de página del programa son debidos a que el prefetcher no carga a tiempo las páginas (son fallos de página *soft*). Sin embargo, la importante reducción en el tiempo de ejecución indica que el tiempo de resolución de estos fallos es muy bajo. Además, debido al poco tiempo de cálculo consumido durante la ejecución de este programa, no es posible obtener un mayor grado de solapamiento entre las lecturas de disco y el cálculo del programa. Es decir, nuestra estrategia de prefetch se acerca a la máxima mejora teórica del rendimiento que se puede obtener utilizando prefetch de páginas.

El tercer caso que hemos considerado es aquel en el que la dimensión de los datos manipulados solo permite mantener en memoria física un working set de cada instrucción de acceso a una matriz fuente. Esto significa que no es posible aplicar la simplificación en el patrón de accesos de las instrucciones. En esta situación el prefetcher sólo puede solicitar la carga de las páginas del siguiente working set de una instrucción cuando detecta que el working set actual ya no es necesario y, por lo tanto, puede ser expulsado al área de swap para liberar el espacio en memoria física y permitir que sea ocupado por el siguiente. Sin embargo, el espacio de tiempo que transcurre desde que el working set actual deja de ser necesario hasta que se inician los accesos al siguiente working set es demasiado pequeño.

En las figura 5.19 y 5.20 mostramos los resultados de calcular sólo una fila de la matriz resultado, porque estos resultados son representativos del comportamiento de toda la ejecución. En la figura 5.19 aparece el tiempo de ejecución de este benchmark, distinguiendo entre tiempo de fallo de página y resto del tiempo de ejecución. Sólo mostramos los resultados para la ejecución sobre el entorno original con el prefetch de kernel desactivado. El motivo es porque después de 24 horas en ejecución sobre el entorno original con el prefetch de kernel activado no se había completado el cálculo de la fila resultado. Esta gran pérdida de rendimiento provocada por el prefetch de kernel se debe a que la penalización debida a la carga de páginas innecesarias se incrementa a medida que lo hace la presión sobre el sistema de memoria.

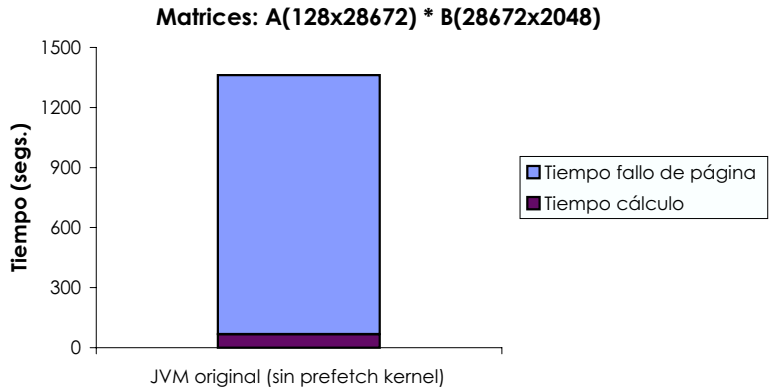

Figura 5.19 Comportamiento de la MULTIPLICACIÓN DE MATRICES EXTRA-GRANDES

En cuanto a la ejecución sobre el entorno modificado con nuestra propuesta, en la figura 5.20.b podemos ver que el prefetcher no es capaz de evitar ningún fallo de página del programa, aunque todos ellos pasan a ser fallos de página *soft*. Esto es debido al poco tiempo que pasa entre el momento en el que se puede iniciar la solicitud y el momento en el que las páginas son accedidas por el programa. Sin embargo en la figura 5.20.a podemos ver que, aunque el escenario que representa este benchmark no es favorable para que nuestra propuesta sea eficaz, la ejecución sobre nuestro entorno tiene el mismo rendimiento que la ejecución sobre el entorno sin prefetch.

5.5.4 Conclusiones de la evaluación

En esta sección hemos presentado los resultados de evaluar tres tipos de benchmarks sobre el entorno modificado con nuestra propuesta de prefetch transparente al SO.

El primer benchmark se ejecuta sin provocar fallos de página y, por lo tanto, la ejecución del código de prefetch simplemente selecciona páginas que, al comprobar que ya están cargadas en memoria, no solicita su carga. Por lo tanto nos ha servido para evaluar la sobrecarga de la ejecución de este código y cómo se consigue reducir al aplicar las optimizaciones en el algoritmo de selección que hemos descrito en 5.1.3. Esta sobrecarga,

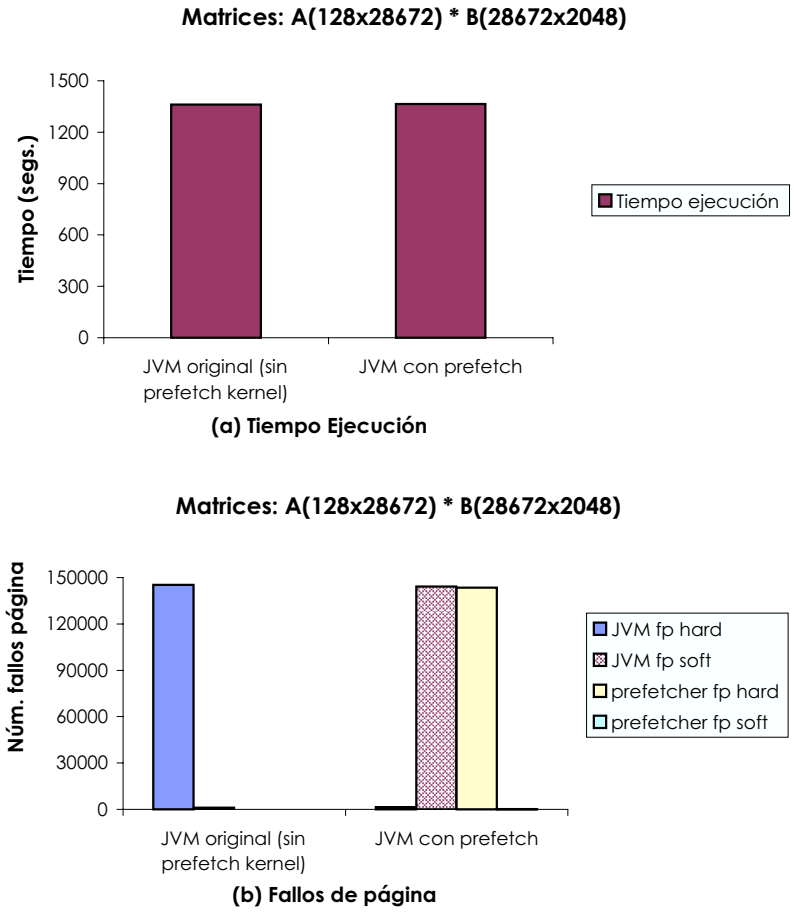

Figura 5.20 Resultados de la MULTIPLICACIÓN DE MATRICES EXTRA-GRANDES

como hemos visto en el análisis del siguiente benchmark, es completamente asumible para las aplicaciones penalizadas por el uso de la memoria virtual, ya que las ganancias de utilizar prefetch para este tipo de aplicaciones superan con creces la penalización de la ejecución del código de predicción. Además, hay que destacar que la desactivación del prefetch para aquellas aplicaciones que no usan la memoria virtual es uno de los detalles pendientes de refinar en la versión definitiva de la implementación de nuestra estrategia.

El segundo benchmark representa al conjunto de aplicaciones que hacen un uso intensivo de la memoria virtual y que son candidatas a ser mejoradas mediante un prefetch efec-

tivo de memoria. Para este benchmark hemos demostrado que el entorno de ejecución modificado con nuestro prefetch mejora un 43 % el rendimiento obtenido sobre el entorno original.

Y el tercer benchmark representa al conjunto de aplicaciones que elevan la presión sobre el sistema de memoria a tal punto que es imposible cargar con anticipación las siguientes páginas referenciadas por las instrucciones, ya que el tamaño de su working set impide ocupar la memoria física con ellas. En este caso, aunque las condiciones de ejecución desaconsejan la utilización del prefetch, nuestra técnica iguala el rendimiento que se obtiene en el entorno original de ejecución con el prefetch de Linux desactivado, y supera, con creces, el rendimiento obtenido sobre el entorno de ejecución original con la configuración por defecto, que se ve muy penalizado por el uso del prefetch que implementa Linux.

Por lo tanto, podemos afirmar que nuestra propuesta de prefetch transparente al SO consigue ofrecer a las aplicaciones el mejor rendimiento teórico que las características de sus accesos a memoria pueden esperar.

5.6 PREFETCH TRANSPARENTE AL SO: SÓLO UN PASO EN EL CAMINO

En este capítulo hemos demostrado que es posible implementar una técnica de prefetch que, utilizando el conocimiento que tiene la JVM sobre el comportamiento de los programas, se ejecute de forma transparente al SO y mejore substancialmente el rendimiento de los programas que hacen un uso intensivo de la memoria virtual.

Hemos dotado al entorno de ejecución de Java de una técnica de prefetch que cumple los requerimientos necesarios para que el prefetch sea eficaz ya que:

- Implementa una selección precisa, como parte del tratamiento de los bytecodes, aprovechando la información que tiene la JVM, en tiempo de ejecución, sobre todos los accesos a memoria de los programas y sus características.

- Respeta la portabilidad de los programas, ya que las modificaciones forman parte de la JVM, y cualquier programa ejecutado sobre ella se puede beneficiar de esta técnica.
- Es transparente al programador y al usuario, ya que no es necesario ni modificar de ninguna manera el código del programa ni recompilarlo.
- Respeta la fiabilidad de sistema, ya que la operación de carga la efectúa el SO, como respuesta a la solicitud que hace la JVM usando el interfaz de la excepción del fallo de página.

El objetivo de conseguir una técnica totalmente transparente era favorecer la portabilidad del mecanismo en sí, evitando tener que adaptarlo en función del SO presente en el sistema. Sin embargo, el desarrollo de este prototipo nos ha demostrado que esta técnica es muy sensible al resto del entorno de ejecución. Por un lado, hay que tener en cuenta que el prefetch es una tarea con importantes restricciones de tiempo: para que sea eficaz, la lectura de disco de la página seleccionada debe finalizar antes de que el programa la acceda y, además, si esto no es posible, puede ser preferible descartar esa lectura para no penalizar al resto de accesos ni de peticiones de prefetch. Por otro lado, su ejecución está interaccionando con el código del SO encargado del resto de decisiones de gestión de memoria del programa, con lo cual es necesario que las decisiones de ambos niveles de gestión de memoria no se interfieran. Pero, en esta estrategia, el SO no es consciente de las tareas ejecutadas por el prefetcher, por lo que trata a las páginas de prefetch como al resto de páginas cargadas bajo demanda y no hace ninguna comprobación especial durante la ejecución de su propio código de gestión.

Por este motivo, durante la implementación del prototipo hemos tenido que analizar, paso a paso, la interacción de cada una de nuestras decisiones de implementación con el código del SO utilizado en nuestra plataforma de desarrollo. Es más, hemos comprobado que esta interacción era diferente para diferentes versiones del kernel de Linux, y era necesario adaptar el código de prefetch de acuerdo a la nueva implementación del kernel.

Esto significa que el código de la estrategia de prefetch a nivel de usuario, no sólo depende del interfaz del SO como cualquier programa sino que, además, depende también de la implementación del SO. Por lo que, para obtener una estrategia eficiente, es necesario

implementar un código específico para la versión de SO utilizada. Es más, un cambio en esta versión implica analizar de nuevo la eficacia del mecanismo de prefetch para, en caso necesario, reajustar las decisiones a la nueva implementación del SO.

Si se asume que la eficacia del prefetch depende de la implementación del SO y que, por lo tanto, no se garantiza la portabilidad de la estrategia, el siguiente paso consiste en analizar las posibles ventajas de involucrar al SO en la estrategia de prefetch guiada por la JVM, obteniendo una estrategia de prefetch basada en la cooperación entre el SO y la JVM. En el siguiente capítulo presentamos este nuevo paso en la estrategia de prefetch.

6

PREFETCH COOPERATIVO ENTRE LA JVM Y EL SO

En este capítulo presentamos el diseño y la implementación de la estrategia de prefetch basada en la cooperación entre el SO y la JVM.

En el capítulo anterior hemos visto que es posible implementar una estrategia de prefetch transparente al SO, usando el conocimiento que tiene la JVM sobre el comportamiento del programa y substituyendo la información necesaria sobre las condiciones de ejecución por el uso de heurísticas que aproximan esta información. Sin embargo, aunque las decisiones de prefetch son transparentes al SO, su participación en la carga es inevitable si se quiere garantizar la fiabilidad del sistema. Para conseguir esta participación, manteniendo el prefetch transparente al SO, es necesario utilizar mecanismos ya existentes dentro del SO para otros propósitos, lo cual complica el código de prefetch y lo hace sumamente sensible a cambios en la versión de kernel instalada en la plataforma de ejecución. Es decir, la portabilidad del mecanismo, única ventaja de su transparencia al sistema, no se puede garantizar.

En este capítulo proponemos una estrategia en la que el SO es consciente de las operaciones de prefetch que la JVM solicita, y colabora activamente en las decisiones asociadas. El mecanismo resultante de esta colaboración es estable, eficiente y no necesita el uso de heurísticas para poder tomar las decisiones de prefetch. Además, su implementación es simple y sin los efectos laterales que sufre la estrategia transparente al SO.

La figura 6.1 refleja la participación que tienen tanto el SO como la JVM en la implementación del prefetch. En la tarea de selección de páginas, la JVM aporta su conocimiento

sobre el comportamiento del programa para predecir los próximos accesos a memoria, y el SO aporta su conocimiento sobre el estado de la máquina para que la JVM pueda optimizar la selección de páginas y decidir la distancia de prefetch adecuada. En cuanto a la carga anticipada, la JVM solicita al SO que inicie la carga asíncrona de las páginas que ha seleccionado, aunque éste tiene la decisión final sobre la conveniencia o no, de acuerdo a las condiciones de ejecución, de completar cada operación de prefetch.

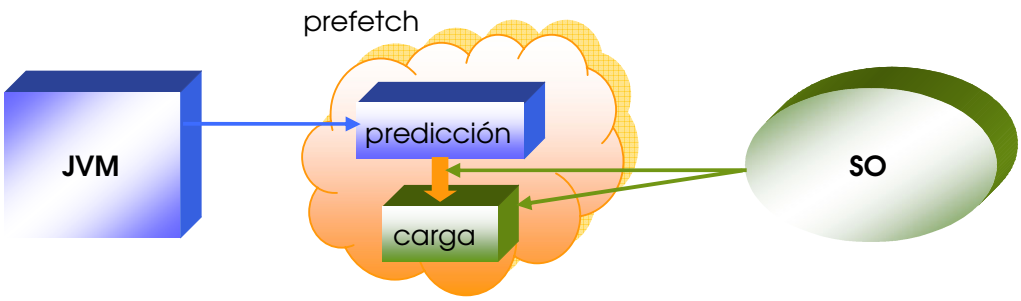

Figura 6.1 Cooperación entre JVM y SO para la estrategia de prefetch

En la sección 6.1 y en la sección 6.2 describimos en detalle estas dos tareas de la estrategia de prefetch (selección de páginas y carga asíncrona) y en la sección 6.3 describimos el interfaz que hemos añadido al SO para permitir la cooperación entre la JVM y el SO en las decisiones de prefetch. Después de estas secciones mostramos una visión general del funcionamiento de la estrategia de prefetch cooperativo (sección 6.4), y presentamos sus detalles de implementación, tanto los relacionados con código del SO como los relacionados con código de la JVM (sección 6.5). Para finalizar el capítulo, en la sección 6.6 presentamos los experimentos que demuestran la efectividad de esta estrategia y en la sección 6.7 presentamos las conclusiones que hemos extraído del desarrollo y evaluación del prefetch cooperativo.

6.1 SELECCIÓN DE PÁGINAS DE PREFETCH

En esta sección presentamos qué aspectos de la tarea de selección de páginas de prefetch se deben modificar para aprovechar la posible participación del SO dentro de la estrategia de prefetch.

Uno de los puntos importantes dentro de la tarea de selección es la predicción precisa de los accesos futuros de las instrucciones. En esta predicción no influye el estado del sistema, ya que sólo depende del comportamiento del programa y, por lo tanto, no es necesaria la participación del SO. Esto ha quedado patente en los resultados que hemos mostrado en el capítulo anterior y en la alta tasa de aciertos que hemos obtenido con la predicción transparente al SO (ver sección 5.1.1 del capítulo 5). Por lo tanto, dentro del prefetch cooperativo, hemos conservado la estrategia de predicción de páginas utilizada como parte del prefetch transparente al SO (ver sección 5.1 en el capítulo 5).

Sin embargo, hay otros aspectos relacionados con la selección de páginas de prefetch que se pueden beneficiar si el SO participa en la estrategia de prefetch.

Un aspecto de la selección de páginas que se puede mejorar consiste en eliminar las heurísticas que necesitaba la estrategia de prefetch transparente al SO para aproximar la información sobre el estado de las páginas del proceso (si estaban presentes en memoria física o si, por el contrario se encontraban en el área de swap). Hay que destacar que, aunque la heurística utilizada ha demostrado un buen comportamiento en los experimentos que hemos ejecutado, no se puede garantizar que esto sea así para otros benchmarks o entornos de ejecución.

En la estrategia de prefetch basada en la cooperación, hemos modificado el SO para que exporte esta información al nivel de usuario. Así, la JVM de una manera eficiente, puede consultar esta información y utilizarla para optimizar el algoritmo de selección y para evitar la petición de carga de páginas ya presentes en memoria física.

Otra modificación que hemos introducido en la selección de páginas se refiere a la distancia de prefetch. En el caso de la estrategia de prefetch transparente al SO, el valor de la

distancia es un parámetro de ejecución y global para todas las instrucciones que utilizaban el mecanismo de prefetch.

En la estrategia de prefetch basada en la cooperación, la distancia de prefetch pasa a ser una característica más del patrón de accesos de cada instrucción, que se calcula de forma automática y se ajusta dinámicamente para adaptarse a las condiciones de ejecución. Hemos basado este ajuste en el estado de la página que la instrucción referencia en cada momento. Si la distancia de prefetch es la adecuada, y asumiendo que la predicción de accesos es correcta, entonces los accesos realizados por la instrucción serán sobre páginas ya presentes en memoria física. Por lo tanto, hemos incorporado, como parte de la selección de páginas de prefetch, el código que comprueba si la página que referencia la ejecución actual de la instrucción está presente en memoria física y, de no ser así, corrige el valor de la distancia usado para esa instrucción, para adaptarlo a las condiciones de ejecución.

Además, la cooperación del SO en la estrategia de prefetch, permite que la JVM agrupe varias solicitudes de carga anticipada y, así, se aumente la posibilidad de que el SO pueda optimizar los accesos a disco debidos a sus peticiones de prefetch. Esto puede hacerse porque hemos añadido un nuevo interfaz al SO que permite que un flujo solicite la carga de una página de forma asíncrona, sin tan siquiera esperar a que se inicie el acceso a disco y, por lo tanto, un mismo flujo puede tener varias peticiones pendientes de carga al mismo tiempo.

6.2 CARGA ASÍNCRONA Y ANTICIPADA

En el capítulo 4 hemos visto que era necesario decidir cómo efectuar la solicitud de carga asíncrona y anticipada, porque los SO actuales no disponen de un interfaz dedicado a este objetivo, y planteamos dos posibles alternativas. La primera, se basaba en utilizar algún mecanismo ya existente dentro del SO para conseguir así una estrategia totalmente transparente al SO, y es la que hemos presentado en el capítulo anterior (ver sección 5.2 en el capítulo 5). La segunda, consiste en modificar el SO con un nuevo interfaz que permita la lectura asíncrona de páginas del área de swap, y que esté dedicado a la solicitud de

carga anticipada. Esta segunda opción implica la modificación del SO y es la solución adoptada en la estrategia de prefetch cooperativo que presentamos en este capítulo.

Añadir un interfaz dedicado para la petición de carga asíncrona tiene varias consecuencias positivas.

La primera consecuencia es que el SO diferencia las peticiones debidas al mecanismo de carga bajo demanda y las peticiones debidas al prefetch. Esto significa que puede usar esta información para tratar cada tipo de carga de acuerdo a sus requerimientos. Por ejemplo, para las solicitudes de prefetch, las condiciones de ejecución del sistema pueden aconsejar descartar la operación de prefetch para no sobrecargar el sistema de memoria. Sin embargo, ante un fallo de página no es posible descartar el acceso a disco involucrado, ya que es imprescindible para permitir que el proceso pueda continuar la ejecución.

Otra consecuencia positiva es que, la implementación del mecanismo de carga anticipada integrada dentro del SO, permite tener en cuenta la ejecución del resto de tareas de gestión de memoria que implementa el SO, lo cual favorece el objetivo de evitar las interferencias entre las decisiones de las diferentes tareas.

Por último, adoptar esta solución hace que deje de ser necesario el nuevo flujo de ejecución dentro de la JVM, que se utilizaba para dotar de asincronía al mecanismo de fallo de página. Esto representa una simplificación importante en el código introducido dentro de la JVM, y elimina la necesidad de ajustar la interacción de este nuevo flujo con el SO, con el consiguiente aumento de estabilidad y eficiencia de la estrategia de prefetch (ver las secciones 5.4.2 y 5.6 del capítulo 5).

6.3 INTERFAZ ENTRE EL SO Y LA JVM PARA PERMITIR LA COOPERACIÓN

El interfaz que hemos añadido en el SO para implementar la estrategia de prefetch cooperativo tiene dos propósitos principales. El primero es exportar al nivel de usuario la información sobre el estado de la memoria, necesaria para poder prescindir de las heurísticas

durante la tarea de selección de páginas de prefetch. El segundo propósito es permitir que desde el nivel de usuario se pueda solicitar de forma asíncrona la lectura de una página que se encuentra en el área de swap. En las siguientes secciones describimos el interfaz que hemos diseñado para cumplir ambos propósitos.

6.3.1 Exportación del estado de la memoria

Para permitir que el SO exporte la información sobre el estado de la memoria, hemos implementado una zona de memoria compartida entre el nivel de sistema y el nivel de usuario, donde el SO mantiene la información que interesa hacer visible a la JVM. Esta zona de memoria compartida, en la implementación que hemos hecho, contiene un bitmap que representa el estado de cada página de una región determinada del espacio de direcciones. Es decir, para cada página nos indica si se encuentra en memoria física o si se encuentra en el área de swap. Sin embargo, dejamos como trabajo futuro el estudio de la conveniencia de que el SO exporte otros parámetros sobre el estado de la memoria, que puedan ayudar a la JVM en la toma de decisiones de prefetch.

El SO es el encargado de mantener actualizado el bitmap ante cada solicitud de carga (debida a un fallo de página o a una solicitud de prefetch) y ante cada expulsión al área de swap realizada por el algoritmo de reemplazo. De manera que, desde el nivel de usuario, la JVM tiene una visión exacta del estado de la memoria del programa.

Por su parte, la JVM es la encargada de solicitar al SO la creación e inicialización de la zona de memoria compartida, y lo hace como parte de la inicialización del heap del programa. Para ello debe indicar al SO cuáles son los límites de la región del espacio de direcciones que interesa representar en el bitmap que, en nuestro caso, se corresponden con los límites del heap del programa.

Hemos implementado el interfaz para la configuración del bitmap utilizando el mecanismo que ofrece Linux para introducir nuevos gestores de dispositivos. Por lo tanto, hemos creado un dispositivo lógico nuevo, que representa la zona de memoria compartida, y que permite que la JVM configure el bitmap mediante el interfaz de Linux para el acceso y control de dispositivos.

Una vez configurado el bitmap, cuando la JVM necesite consultarlo, simplemente tiene que acceder a la zona de memoria compartida como a cualquier otra de sus variables. Hay que tener en cuenta que, presumiblemente, esta operación de consulta se va a efectuar en millones de ocasiones, para evitar la petición de páginas ya presentes en memoria física y la entrada innecesaria en el sistema que estas peticiones involucran. Por este motivo, es especialmente importante utilizar un método de consulta rápido y que no implique la entrada en el sistema.

En la sección 6.5.1 describimos detalladamente todos los aspectos relacionados con la implementación del bitmap que representa el estado de las páginas que albergan el heap del programa.

6.3.2 Solicitud de carga asíncrona

En esta sección vamos a describir el interfaz que hemos añadido al SO para permitir que, desde el nivel de usuario, se pueda solicitar la carga asíncrona en memoria de una página que se encuentra en el área de swap. De esta manera, la JVM podrá solicitar anticipadamente la carga de las páginas seleccionadas de prefetch y continuar con la ejecución del programa, en paralelo con la lectura de disco de las páginas.

Este interfaz consiste en una nueva llamada a sistema que recibe como parámetro la dirección de memoria que se quiere cargar de forma asíncrona (ver figura 6.2).

---

int async_page_read(int *address)
descripción: solicita la carga asíncrona de la página a la que pertenece la dirección **address**
valores de retorno:
0: solicitud de carga en curso
-1: solicitud de carga cancelada
EINTRANSIT: página ya está en tránsito
ESATDISK: gestor de disco saturado
ENOFREEMEM: memoria física libre escasa
ENOLOAD: error en la carga de página

Figura 6.2 Interfaz de la llamada a sistema de carga asíncrona

---

Ante esta petición, el SO comprueba si las condiciones de ejecución son favorables para completar la operación de carga anticipada y, si es así, reserva una nueva página en memoria física para albergar los datos que se van a cargar del área de swap, entrega la petición al gestor de disco y retorna al nivel de usuario indicando que la carga está en curso, de manera que el programa puede continuar con la ejecución en paralelo mientras se realiza la lectura de disco.

Sin embargo, si el SO determina que, dada las condiciones del sistema, efectuar esa carga anticipada puede perjudicar el rendimiento del proceso, entonces cancela la operación y retorna al nivel de usuario indicando que no ha sido posible iniciar la carga solicitada. Hemos detectado dos posibles situaciones que aconsejan la cancelación de las peticiones de prefetch: cuando la memoria física libre es muy escasa y cuando existen demasiadas peticiones pendientes de acceso a disco y, por lo tanto, el gestor de disco se encuentra saturado.

Por lo tanto, mediante este interfaz de lectura asíncrona, estamos implementando un mecanismo de carga anticipada en el que la JVM indica al SO las páginas que sería beneficioso para el proceso cargar con anticipación, pero es el SO en última instancia el que decide si esta carga es conveniente o no, en función de las condiciones de ejecución. Además, como el SO informa al nivel de usuario de que no ha sido posible completar la solicitud, se da la opción de gestionar dentro de la JVM las operaciones canceladas de la manera más adecuada para el comportamiento del programa. En la implementación que hemos hecho del prefetch cooperativo, la JVM registra las operaciones canceladas y aprovecha los momentos en los que disminuye la carga del sistema para reintentar estas operaciones.

En la sección 6.5.2 describimos en detalle la implementación de esta llamada a sistema, así como algunas alternativas de diseño que nos hemos planteado y que hemos descartado debido a su bajo rendimiento.

6.4 VISIÓN GENERAL: PREFETCH COOPERATIVO

En la figura 6.3 presentamos una visión general de la estrategia de prefetch basada en la cooperación entre el SO y la JVM, que a continuación explicamos.

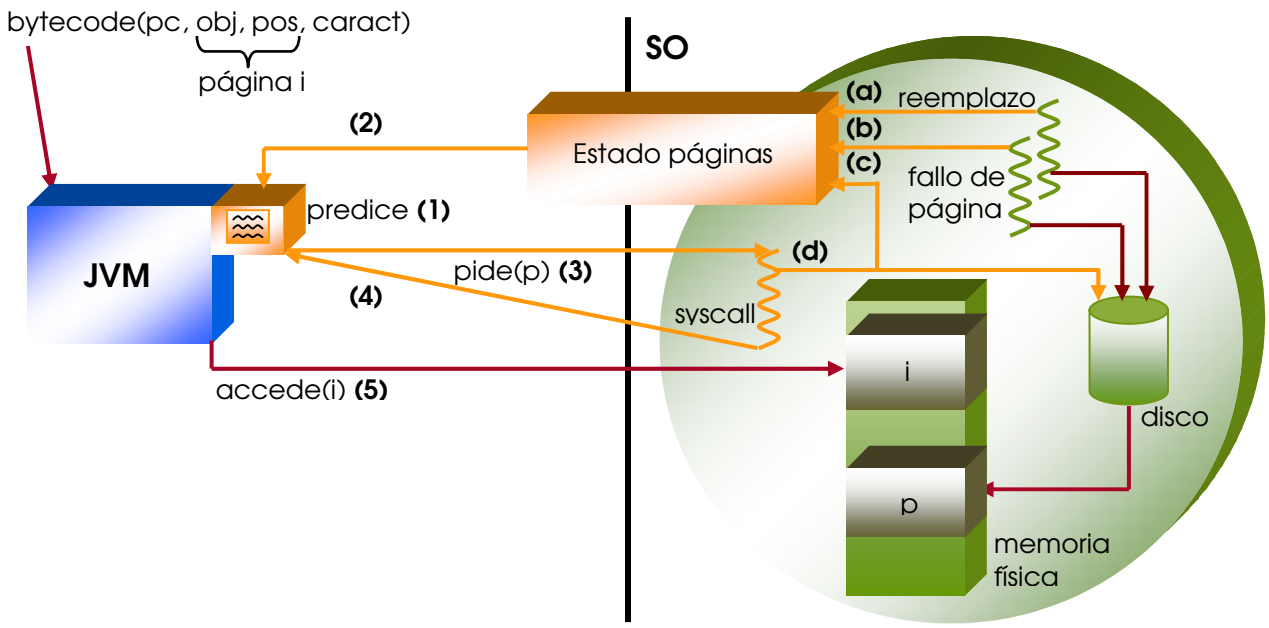

Figura 6.3 Visión general del prefetch cooperativo entre JVM y SO

Para cada bytecode de acceso a array, como en el caso de la estrategia transparente al SO, la JVM ejecuta el código de la tarea de selección de páginas (**1**). Este código actualiza, si es necesario, el valor de la distancia de prefetch y la información sobre el patrón de accesos de la instrucción y aplica ese patrón para determinar los próximos accesos de la instrucción sobre el array. Para dar la opción de que el SO agrupe las peticiones de acceso a disco, la JVM aplica varias veces el patrón para predecir un conjunto de páginas.

Además, la JVM utiliza la información sobre el estado de la memoria para filtrar, del conjunto de páginas seleccionadas, las que ya se encuentran presentes en memoria física y, por lo tanto, no es necesario solicitar su carga anticipada (**2**). El SO es el encargado de mantener actualizada la información sobre el estado de las páginas, ante cualquier

operación de intercambio con el área de swap (algoritmo de reemplazo (**a**), fallo de página (**b**) o llamada a sistema de petición de prefetch (**c**)), y de hacerla visible al nivel de usuario a través del interfaz de memoria compartida que hemos implementado.

Una vez finalizada la selección de páginas de prefetch, la JVM utiliza la llamada a sistema para pedir la lectura asíncrona de las páginas que ha seleccionado como candidatas a ser cargadas anticipadamente (**3**).

Ante esta petición, si las condiciones de ejecución son favorables, el SO inicia la carga de los datos solicitados (**d**), reservando una nueva página física para ellos, entregando al gestor del disco la solicitud de acceso e informando a la JVM de que el proceso de carga se ha iniciado de forma satisfactoria. Si, por el contrario, las condiciones de ejecución desaconsejan iniciar la operación de prefetch, el SO retorna a usuario indicando que la solicitud ha sido cancelada.

Al retornar de la llamada a sistema, la JVM comprueba cuál ha sido el resultado de la solicitud (**4**). Si el SO ha decidido cancelarla, la JVM la almacena en un buffer de peticiones pendientes para reintentar la operación si las condiciones de carga del sistema mejoran. Si, por el contrario, la carga anticipada está en curso, la JVM comprueba si tiene pendiente alguna operación cancelada previamente para reintentar su solicitud.

Finalmente, la JVM continúa con la ejecución del código del programa (**5**), de manera que, cuando éste intente acceder a a los datos que se han podido cargar con antelación, si la distancia de prefetch ha sido la adecuada, el acceso se podrá efectuar sin generar una excepción de fallo de página.

6.5 IMPLEMENTACIÓN DEL PREFETCH COOPERATIVO

En esta sección describimos las modificaciones que hemos introducido tanto en la JVM como en el SO para implementar la estrategia de prefetch cooperativo. La versión de kernel de Linux que hemos modificado es la 2.4.18-14 y la versión de la JVM que hemos

utilizado es la 1.3.1 de *HotSpot* que se distribuye con la Java 2 SDK Standard Edition de Sun para Linux

Además de las soluciones finales que hemos adoptado para la implementación, también presentamos algunas alternativas que nos hemos planteado así como el motivo que nos ha llevado a descartarlas. En la sección 6.6, como parte de la evaluación de la estrategia, presentamos los resultados de unos experimentos que sirven para validar las decisiones que hemos tomado durante la implementación del prefetch cooperativo.

En las secciones 6.5.1 y 6.5.2 describimos la implementación del interfaz que hemos incorporado en el SO, para permitir su participación en el prefetch guiado por la JVM. En cuanto a la sección 6.5.3, describe las modificaciones que hemos introducido en la JVM para utilizar este interfaz en la solicitud de carga anticipada, substituyendo el uso del prefetcher y de las heurísticas que formaban parte de la estrategia de prefetch transparente al SO.

6.5.1 Implementación para exportar el estado de la memoria

La implementación del interfaz del SO que permite exportar el estado de la memoria se puede dividir en dos partes diferentes.

La primera parte consiste en el código necesario para crear la zona de memoria compartida entre el SO y la JVM, y para configurar el bitmap que se guarda en esta zona y que representa el estado de las páginas de una región de memoria. La segunda parte es la necesaria para mantener actualizada la información que contiene el bitmap. Este código se ha introducido en el SO, como parte de la implementación de la carga de memoria y como parte de la liberación de memoria física.

Inicialización y configuración del bitmap

Como ya hemos adelantado en la sección 6.3.1, hemos basado la implementación de la zona de memoria que soporta el bitmap en el mecanismo que ofrece Linux para incorporar nuevos gestores de dispositivos. Por este motivo, hemos creado un nuevo dispositivo lógico

que representa al bitmap y hemos implementado el gestor de este nuevo dispositivo, con las funciones del interfaz de manipulación de dispositivos necesarias para que la JVM cree y configure este bitmap.

Para configurar el bitmap lo único necesario es asociarle la región que va a representar, para que así el SO pueda determinar las páginas cuyo cambio de estado debe reflejar en el bitmap. Es decir, hay que registrar los parámetros que caracterizan la región: direcciones que la limitan y espacio de direcciones al que pertenece. Para ello, hemos creado una nueva estructura de datos en el kernel que contiene las características del bitmap y la posición que ocupa en el espacio de direcciones del kernel, y que la JVM inicializa durante la fase de configuración (ver figura 6.4).

```
struct bitmap_struct {
        /* estructura de datos que describe al proceso que usa el bitmap */
    struct task_struct *bitmaptask;
        /* dirección inicial de la región representada por el bitmap */
    unsigned long initAddr;
        /* dirección final de la región */
    unsigned long lastAddr;
        /* tamaño de la región */
    unsigned long length;
        /* posición que ocupa el bitmap en el espacio del kernel */
    unsigned char *kernelBitmapAddr;
}
```

Figura 6.4 Estructura de datos que caracteriza al bitmap dentro del SO

A continuación se describe brevemente las funciones que hemos implementado como parte del gestor del bitmap.

- OPEN: recibe como parámetro el dispositivo lógico asociado al bitmap y simplemente registra al proceso que la utiliza como espacio de direcciones que se quiere representar en el bitmap.

```
bitmapFd = open(" /dev/bitmap ", O_RDONLY)
```

- IOCTL: esta es la llamada a sistema que se utiliza para configurar los dispositivos lógicos. La implementación que hemos hecho de esta función permite hacer dos operaciones de configuración diferentes que describimos a continuación.

  - Inicialización del bitmap: esta operación reserva la zona de memoria en el espacio de direcciones del kernel que soporta el bitmap y le asocia los datos que caracterizan la región que representa. Los parámetros que recibe esta llamada a sistema son la dirección inicial de la región que se quiere representar en el bitmap (es decir, la dirección base del heap) y el tamaño de esa región (es decir, el tamaño máximo del heap). Utilizando estos parámetros calcula el tamaño que debe tener el bitmap y reserva la cantidad de memoria necesaria para soportarlo, fijándola, además, de forma permanente en memoria física. Por último, inicializa todos los bits a 0, ya que se parte del estado en que ninguna página de la región está presente (se supone que la inicialización del bitmap se realiza al mismo tiempo que la inicialización de la región que va a representar). Como valor de retorno, esta función devuelve el tamaño de la zona compartida con el kernel que se ha reservado.

    ```
    bitmapSize = ioctl(bitmapFd, BITMAP_CONFIG, regionInfo)
    ```

  - Desactivación del uso del bitmap: en este caso, la llamada IOCTL no necesita ningún parámetro ya que, simplemente, desactiva el uso del bitmap, eliminando la asociación con la región que representaba y liberando la memoria que ocupaba en el kernel.

    ```
    ioctl(bitmapFd, BITMAP_DEACT, 0)
    ```

- MMAP: mediante esta función es posible mapear en el espacio de direcciones de usuario un dispositivo lógico, y permitir que el programa acceda a este dispositivo a través de la memoria donde está mapeado. La llamada MMAP consigue este efecto reservando una nueva región y asociando a esta región las funciones que, mediante el mecanismo de fallo de página, permitirán traducir los accesos a memoria en los correspondientes accesos al dispositivo. Estas funciones deben formar parte también de la implementación del gestor del dispositivo mapeado. En nuestro caso, utilizamos la llamada a MMAP dentro de la JVM para reservar una nueva región en su espacio de direcciones y mapear en ella el dispositivo del bitmap. La implementación específica de la llamada MMAP, que hemos hecho dentro del gestor del bitmap, marca a las páginas

de la nueva región para que no sean candidatas a ser expulsadas al área de swap, a continuación asocia a la región las funciones que hemos implementado para vincularla con la región del bitmap reservada en el kernel y, por último, devuelve, como valor de retorno, la dirección base de la región reservada en el espacio de usuario, que la JVM podrá utilizar para acceder al bitmap.

```
bitmapAddr = mmap(0, bitmapSize, PROT_READ, MAP_SHARED, bitmapFd, 0);
```

Respecto a la función que establece el vínculo entre la región reservada y el dispositivo mapeado, forma parte del tratamiento de los fallos de página que se producen por el primer acceso a una página lógica. Esta función es invocada por la rutina de gestión de fallo de página del SO, una vez que ha comprobado la validez del acceso, para obtener la página con la que debe actualizar la tabla de páginas del proceso y completar así la resolución de la excepción. En el caso del dispositivo del bitmap, para poder convertir los accesos a la región de la JVM en los correspondientes accesos a la región del bitmap, lo único necesario es asociar las mismas páginas físicas a ambas regiones. Por lo tanto, la función de fallo de página inicial que hemos implementado se encarga de determinar la página física correspondiente y devolverla a la rutina de fallo de página para que actualice con ella la tabla de páginas de la JVM. Hay que decir que los únicos fallos de página provocados por accesos a la región que mapea el bitmap serán los debidos al primer acceso a cada página, ya que las páginas que soportan el bitmap no son candidatas a ser expulsadas al área de swap. Por lo tanto, sólo es necesario establecer el vínculo entre ambas regiones durante el primer acceso.

- CLOSE: se libera el uso del dispositivo virtual asociado al bitmap.

```
close(bitmapFd);
```

En la figura 6.5 mostramos el código que debe ejecutar la JVM, como parte del tratamiento de inicialización del heap del programa, para inicializar y configurar el uso del bitmap, y en la figura 6.6 representamos el resultado de esta inicialización. Podemos ver que la JVM le pide al gestor del bitmap, mediante la llamada a IOCTL, que cree la zona de memoria compartida y la configure para representar el estado del heap del programa y, mediante la llamada a MMAP, le pide que mapee el bitmap en su propio espacio de direcciones.

```
. . .
bitmapFd = open(/dev/bitmap ", O_RDONLY);
/* tamaño máximo del heap */
regionInfo[0] = maxHeapAddr-minHeapAddr;
/* dirección base del heap */
regionInfo[1] = minHeapAddr ;
/* crea la región en el espacio del kernel para representar el estado del heap */
bitmapSize = ioctl(bitmapFd, BITMAP_CONFIG, regionInfo);
/* mapea el bitmap en una nueva región del espacio de la JVM */
bitmapAddr = mmap(0, bitmapSize, PROT_READ, MAP_SHARED, bitmapFd, 0);
/* completar la inicialización del heap */
. . .
```

Figura 6.5 Configuración del bitmap en la JVM

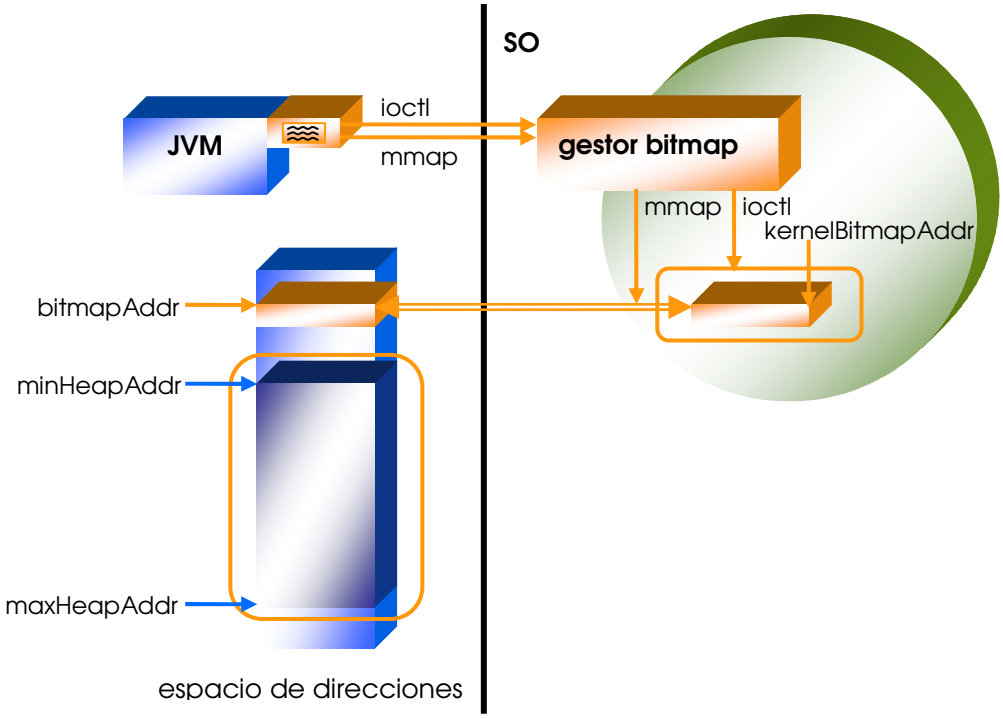

Figura 6.6 Resultado de la inicialización del bitmap

Una vez configurado el bitmap, la JVM puede comprobar el estado de cualquier página del heap accediendo a la región que la llamada MMAP ha reservado en su espacio de direcciones.

Para ello sólo debe calcular la posición del bitmap que representa el estado de la página y consultar el valor de ese bit, lo que, sabiendo cuál es la página inicial representada por ese bitmap, se reduce a una operación aritmética muy sencilla (ver figura 6.7).

```
ix = (p - minHeapPage) ≫ 3;
pos = (p - minHeapPage) & 0x7;
present = (bitmapAddr[ix] & (1 ≪ pos));
```

Figura 6.7 Acceso al estado de la página p

Código del SO para mantener el estado de las páginas

El SO es el responsable de mantener actualizada la información del bitmap, para que éste refleje en todo momento el estado de las páginas del heap del programa. Para ello, hemos introducido en el SO el código necesario para modificar el bitmap cada vez que cambia el estado de una de las páginas que representa. Es decir, hemos modificado tanto la rutina que libera páginas físicas y que, por lo tanto, provoca que una página lógica deje de estar presente en memoria física, como las rutinas que cargan en memoria física las páginas lógicas que no estaban presentes.

Lo primero que hace el código que hemos añadido es comprobar si la página cuyo estado va a cambiar forma parte de la región representada por el bitmap. Para ello basta con acceder a las variables de configuración del bitmap (ver figura 6.4), que contienen los datos que caracterizan la región, y que son inicializadas por la JVM en el momento de creación del bitmap.

La liberación de memoria, en la versión de kernel que estamos utilizando, se hace efectiva en una única rutina (TRY_TO_UNMAP_ONE) que recibe como parámetro la página que se quiere liberar y la entrada de la tabla de páginas a la que pertenece (ver figura 6.8). A través de esta entrada, es posible obtener la estructura de datos que describe el espacio de direcciones al que pertenece la página, de manera que somos capaces de comprobar si es el mismo espacio de direcciones que se está representando en el bitmap. Por otro lado, mediante la dirección lógica involucrada en la liberación de memoria, se puede comprobar si además se trata de una página de la región, y en ese caso, calcular la posición que

```
int try_to_unmap_one(struct page * page, pte_t * ptep) {
/* obtiene la dirección lógica a partir de la entrada de la TP */
unsigned long address = ptep_to_address(ptep);
/* obtiene la estructura del espacio de direcciones a partir de la entrada de la TP */
struct mm_struct * mm = ptep_to_mm(ptep);
. . .
    if (bitmap.bitmap_task->mm == mm) {
        if ((address >= bitmap.initAddr) && (address<=bitmap.lastAddr)) {
            ix = ((address≫12)≫3);
            pos = ((address≫12) & 0x7);
            bitmap.kernelBitmapAddr[ix] &= ~ (1≪pos);
        }
    }
. . .
}
```

Figura 6.8 Actualización del bitmap en la liberación de memoria

ocupa el bit que la representa para poner a cero su valor, y marcar así que la página ya no está presente en memoria física.

El otro cambio de estado de las páginas que el SO debe registrar en el bitmap es el correspondiente a su carga en memoria física (ver figura 6.9). Esto puede ocurrir como consecuencia de un fallo de página o como consecuencia de una petición de carga anticipada. En cualquiera de las dos situaciones, la rutina ejecutada tiene acceso tanto al proceso que solicita la carga como a la página que se quiere cargar en memoria física. Por lo tanto se puede comprobar si se trata de una página representada por el bitmap y, de ser así, calcular el bit que la representa para activar su valor y marcar que la página ya se encuentra presente en memoria física.

En ambas situaciones actualizamos el bitmap antes de finalizar la rutina y de retornar al nivel de usuario. Sin embargo, hay que destacar que en el caso de la llamada a sistema de carga asíncrona la rutina finaliza al entregar la solicitud de carga al gestor de disco y, por lo tanto, se está marcando como presente en memoria física una página que todavía está en proceso de ser cargada. Esta decisión se ha tomado para permitir que la JVM evite también la solicitud de páginas que ya se encuentran en tránsito.

```
. . .
/* macro current del kernel devuelve la estructura del proceso actual */
if (bitmap.bitmap_task->mm == current->mm) {
    if ((address >= bitmap.initAddr) && (address<=bitmap.lastAddr)) {
        ix = ((address≫12)≫3);
        pos = ((address≫12) & 0x7);
        bitmap.kernelBitmapAddr[ix] |= (1≪pos);
    }
}
. . .
```

Figura 6.9 Actualización del bitmap en la carga de memoria

6.5.2 Implementación de la llamada a sistema para la carga asíncrona

El interfaz que hemos implementado para permitir la petición asíncrona de carga desde el nivel de usuario consiste en una nueva llamada a sistema. Esta llamada recibe como parámetro una dirección de memoria de la página que se quiere cargar de forma asíncrona y su ejecución se completa o bien con la entrega de la petición al gestor de disco o bien con la cancelación de la solicitud, si el SO considera que llevarla a cabo puede perjudicar el rendimiento del programa.

Básicamente, el código de esta nueva llamada a sistema sigue el mismo camino de ejecución que la rutina del kernel responsable de cargar una página como respuesta a un fallo de página. La principal diferencia entre ambos códigos radica en que la carga asíncrona evita todas las situaciones que, para poder continuar con la carga de memoria, requieren bloquear al proceso que efectúa la llamada. Ante una situación de este tipo, la llamada a sistema cancela la solicitud de carga y retorna al usuario informando de que no ha sido posible completar la petición, ya que ello implicaría el retardo asociado a los bloqueos y la consiguiente pérdida de rendimiento de la estrategia de prefetch. Hay que destacar que la opción de cancelar la petición de carga no es viable para la rutina de gestión de fallo de página ya que el proceso que ha provocado esa excepción necesita que se complete la carga de la página para poder continuar la ejecución.

En la figura 6.10 mostramos el camino de ejecución tanto para el mecanismo de fallo de página (a) como para la petición de carga asíncrona (b), para poder señalar las diferencias entre ambos códigos.

En ambos casos suponemos que ya se ha validado el acceso a la dirección. Es decir es una dirección válida y el tipo de acceso está permitido por los permisos de acceso asociados a la región a la que pertenece. Por lo tanto, lo primero que hacen ambas rutinas es comprobar que la página realmente se encuentra en el área de swap. Es decir, que no está ya cargada en memoria física (1) ni se encuentra en proceso de carga (2). En cualquiera de las dos situaciones, ambas rutinas finalizan la ejecución indicando que no es necesario efectuar la lectura de disco de los datos.

Si la página está en el área de swap, el siguiente paso necesario para poder cargarla en memoria consiste en asignarle una página física libre. En la rutina de reserva de memoria libre es donde aparece la primera diferencia entre ambas situaciones de carga, si la cantidad de memoria física es inferior a un determinado umbral (cuyo valor es configurable por el administrador de la máquina) y, por lo tanto, se considera muy escasa (3).

En esta situación, la rutina de gestión de fallo de página bloquea al proceso hasta que el algoritmo de reemplazo de memoria de Linux consigue liberar memoria, enviando al área de swap las páginas que se consideran menos necesarias para los procesos en ejecución. Por el contrario, la llamada a sistema de carga asíncrona cancela la petición de carga, y retorna al usuario indicando que la cantidad de memoria libre hace imposible iniciar la carga anticipada sin bloquear al proceso.

Si la reserva de memoria finaliza con éxito, a continuación hay que entregar al gestor de disco la petición de lectura de los datos. Este paso también puede necesitar el bloqueo del proceso en situaciones de alta presión para el sistema y, por lo tanto, el código de la llamada a sistema se vuelve a diferenciar del código de resolución de fallo de página. La operación consiste en inicializar una estructura de datos, que describe el acceso que se quiere realizar, y en introducir esa estructura en una cola de peticiones pendientes asociada al gestor de disco, para que éste la atienda cuando le sea posible. El número de peticiones pendientes que puede contener esta cola está limitado por una variable del

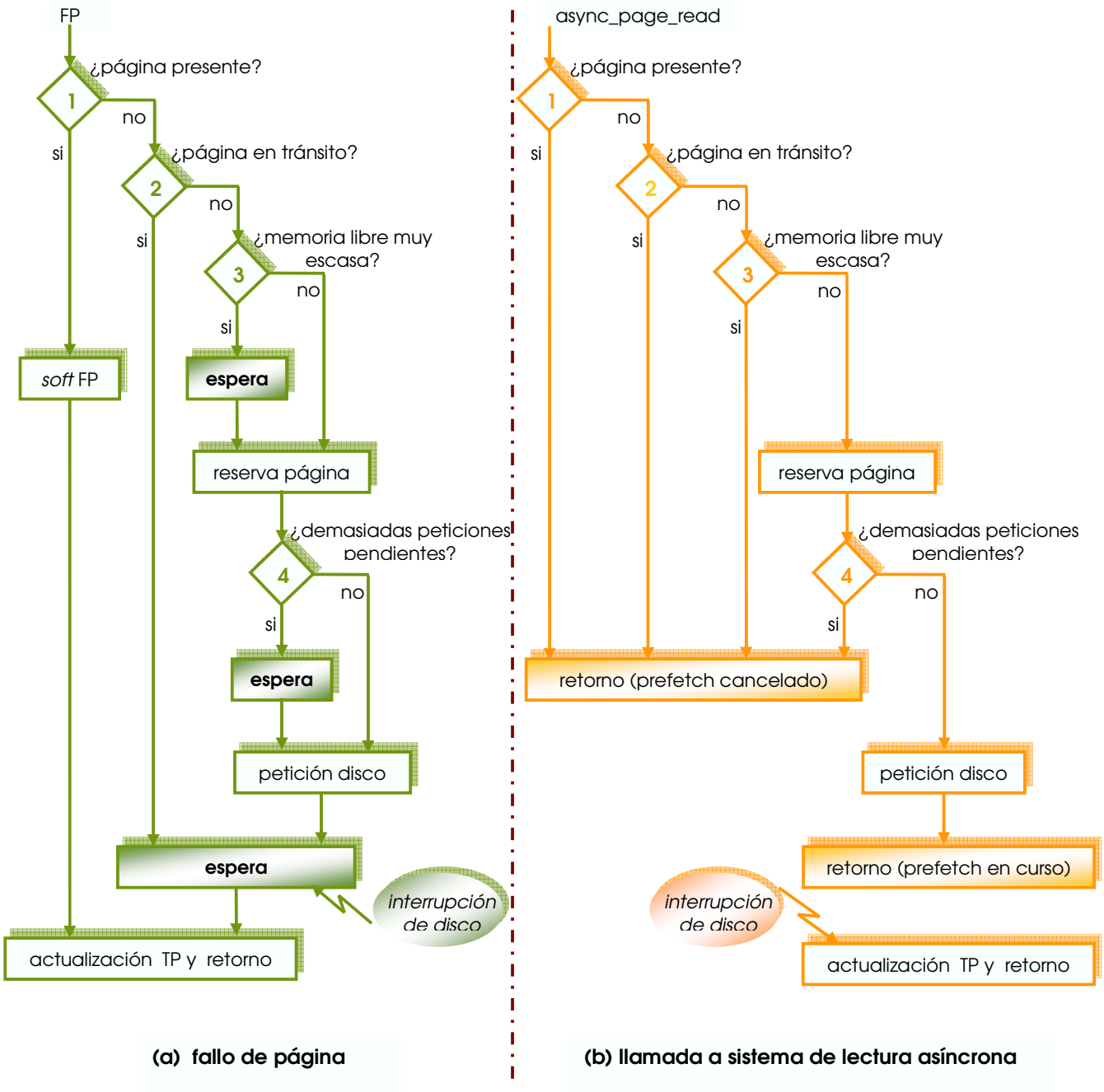

Figura 6.10 Fallo de página vs. carga asíncrona

kernel. Cuando este límite se supera, se asume que el gestor de disco está saturado y, por lo tanto, el código original de Linux bloquea al proceso que solicita el acceso a disco

hasta que alguna de las peticiones pendientes se completan (4). Hemos modificado este comportamiento sólo para las solicitudes de acceso debidas a una carga anticipada de manera que, si no es posible encolar la petición para el gestor de disco sin bloquear antes al proceso, se descarta el acceso. En esta situación, se cancela la operación de prefetch, liberando la página que se había reservado para soportar los datos y retornando al usuario el motivo de la cancelación.

Finalmente, si es posible entregar la solicitud de acceso a disco al gestor, en el caso de la rutina de fallo de página, el proceso se bloquea hasta que la lectura de la página se completa. Cuando eso ocurre, el controlador del disco genera una interrupción que el SO gestiona desbloqueando al proceso que estaba esperando este evento. El proceso, entonces, continúa la ejecución completando la rutina de gestión de fallo de página, que actualiza la tabla de páginas del proceso antes de retornar al código de usuario. En el caso de la carga asíncrona, el SO finaliza inmediatamente la llamada a sistema, informando al nivel de usuario de que la carga está en curso. Cuando el controlador de disco genere la interrupción para notificar al SO la finalización del acceso, el SO debe comprobar si se trata de una lectura asíncrona y, en ese caso, debe gestionar la interrupción completando el tratamiento de la carga anticipada. Es decir, debe ser capaz de identificar la tabla de páginas y actualizar la entrada asociada a la página cargada. Para esta actualización necesita, además de la dirección física, los permisos de acceso de la región a la que pertenece. Además este código debe respetar el tratamiento que hace Linux para evitar las condiciones de carrera que podrían dañar la integridad de sus estructuras de datos. En este sentido ha sido necesario proteger el acceso a la estructura de la página, y a la tabla de páginas. Además, ha sido necesario liberar el acceso a la entrada del área de swap que contenía la página cargada. Para ello, hemos añadido en la estructura de datos que representa una página en Linux todos los campos necesarios (ver figura 6.11). Antes de finalizar la llamada a sistema, se actualizan estos campos con los valores adecuados, ya que desde la llamada a sistema, igual que desde la rutina de fallo de página, se tiene acceso a todas las variables necesarias.

```
typedef struct page {
        . . .
    pte_t * page_table; /*puntero a la entrada de la TP involucrada */
    pgprot_t prot; /* permisos de acceso asociados a la página */
    swp_entry_t entry; /* para liberar uso de la entrada del área de swap*/
    struct mm_struct * mm; /* para liberar uso de la TP*/
}
```

Figura 6.11 Campos añadidos a la estructura que representa una página en Linux

Alternativas para la implementación de la carga asíncrona

Durante la implementación de esta llamada a sistema, nos hemos planteado dos alternativas que hemos considerado interesante evaluar.

La primera de ellas afecta al interfaz de la llamada, y se refiere a la cancelación de la solicitud de prefetch que puede hacer el SO. La alternativa es no permitir esta cancelación y dejar que la JVM decida qué operaciones de prefetch se inician, aprovechando que conoce exactamente las páginas que el proceso necesita en un futuro inmediato. De manera que, si la solicitud de carga asíncrona respeta la integridad del sistema, siempre se finaliza la llamada con el proceso de carga preparado para ser iniciado. Por lo tanto, con esta alternativa, puede ser necesario que el SO bloquee al proceso tanto durante la asignación de memoria libre como durante la introducción de la petición de lectura en la cola del gestor de disco. En la sección 6.6.2 presentamos los resultados de evaluar esta alternativa, y vemos que el tiempo de bloqueo implicado en las solicitudes de carga impiden que el programa extraiga todos los beneficios posibles de la estrategia de prefetch.

La segunda alternativa no afecta al diseño del interfaz y es sólo una decisión de implementación en la carga anticipada. Se trata de no actualizar la tabla de páginas del proceso en la rutina de atención al disco, sino esperar hasta que el proceso acceda por primera vez a la página cargada asíncronamente provocando, de esta manera una excepción de fallo de página. Hay que destacar que esta es la opción utilizada por Linux para gestionar las páginas cargadas por su estrategia de prefetch secuencial, y que estos fallos de página

se resuelven simplemente actualizando la tabla de páginas del proceso. Esta alternativa simplifica el código de gestión de la carga asíncrona y, además, evita las posibles actualizaciones innecesarias de la tabla de páginas que pueden hacerse debido a predicciones erróneas durante la fase de estabilización de los patrones de acceso. Sin embargo, hemos evaluado esta alternativa para la implementación y hemos observado que, en situaciones de alta presión para el sistema de memoria, retrasar el momento de la actualización de la tabla de páginas puede provocar un bajo rendimiento en la estrategia de prefetch (ver la sección 6.6.2).

6.5.3 Modificaciones de la JVM para usar el prefetch cooperativo

En esta sección describimos las modificaciones que se han introducido en la JVM para usar el interfaz de prefetch que hemos definido en el SO, y optimizar el rendimiento de la estrategia de prefetch explotando las posibilidades que ofrece el nuevo interfaz. Hay que destacar que, como ya hemos dicho en la sección 6.1, el mecanismo de predicción que utilizamos continúa siendo el mismo que describimos para la estrategia de prefetch transparente al SO (ver la sección 5.4.1 del capítulo 5), ya que la JVM dispone de toda la información necesaria para obtener una predicción precisa y el SO no puede aportar ningún dato que la mejore.

La primera modificación necesaria consiste en la inicialización y configuración del bitmap, utilizando el interfaz de manipulación del dispositivo lógico bitmap de la manera que se indica en la figura 6.5. Estas operaciones forman parte de la inicialización del heap y permiten que, a partir de ese momento, la JVM pueda consultar el estado de cualquier página del heap en lugar de intentar aproximarlo mediante heurísticas.

Así, en la tarea de selección de páginas, hemos substituido el uso de heurísticas por la consulta de este bitmap en todos los puntos donde la JVM necesita conocer el estado de alguna página del heap. Esto es, durante la generación del patrón de accesos, para aplicar la simplificación relacionada con el uso de working sets que describimos en la sección 5.11, y antes de solicitar al SO la carga asíncrona de una página, para evitar las peticiones redundantes. En la sección 6.6.2, mostramos los resultados de un experimento

que demuestran la relevancia que tiene para el rendimiento de prefetch el uso del bitmap, para evitar las peticiones de prefetch que se refieren a páginas ya presentes o que ya se encuentran en proceso de carga.

También hemos añadido la automatización del cálculo de la distancia de prefetch. Como ya hemos dicho en la sección 6.1, asociamos un valor de distancia de prefetch a cada instrucción como un parámetro más de su patrón de accesos, que ajustamos dinámicamente para adaptarse a las condiciones de ejecución.

Esta distancia indica la anticipación con la que se debe solicitar la carga asíncrona de una página para que cuando la instrucción acceda a esa página se haya podido completar su carga. Habitualmente, su unidad de medida es una iteración, es decir, su valor indica el número de ejecuciones de la instrucción asociada que deben ocurrir desde que se solicita la carga de la página hasta que se accede. Sin embargo, el código de selección de páginas de prefetch no se ejecuta para cada ejecución de la instrucción, sino sólo cuando esa instrucción cambia la página destino de su acceso. Por este motivo, nosotros usamos como unidad para la distancia una ejecución de código de selección. Es decir, el valor de la distancia indica el número de páginas diferentes que la instrucción accede desde que se solicita la carga de una página hasta que la instrucción la referencia.

Existe una excepción en el uso que hacemos de la distancia. Se trata de los patrones de acceso simplificados que representan el uso de working sets. En este caso, la unidad de medida de la distancia es el working set de la instrucción. Además, para este patrón de accesos, el valor de esta distancia siempre es un working set, ya que, durante los accesos a un working set predecimos las páginas que forman parte del siguiente working set que se va a referenciar.

Para el resto de patrones de acceso, ajustamos el valor de la distancia utilizando la siguiente estrategia. Antes de predecir se comprueba si la predicción anterior fue efectiva, consultando si la página que la ejecución actual referencia ya está cargada en memoria. Para ello, el código de selección de páginas se adelanta al acceso que hace el código de tratamiento del bytecode y mide el tiempo necesario para ese acceso, de manera que puede determinar si se trata de un fallo de página o no. Hay que destacar que este método no

perjudica el rendimiento de los programas, ya que lo único que hace es adelantar unas instrucciones el momento en el que se lleva a cabo el acceso que el propio programa necesita. El motivo por el que hemos utilizado este método en lugar de consultar el bitmap con el estado de las páginas, es que la información del bitmap está orientada a evitar las peticiones de carga redundante, que son las que se refieren tanto a páginas ya presentes como a páginas en proceso de ser cargadas y, por lo tanto, no distingue entre estos dos estados. Sin embargo, un acceso a una página en tránsito también indica que la distancia de prefetch no ha sido suficiente.

Si la página no está cargada, entonces se asume que la distancia de prefetch no es suficiente para las condiciones de ejecución actuales y se incrementa hasta que su valor alcanza una cota superior. Recordemos que el valor de la distancia también influye en la cantidad de páginas cargadas con antelación: cuanto mayor sea menos operaciones de prefetch se efectúan y, por lo tanto, será menor la cantidad de lecturas que se pueden solapar con el tiempo de cálculo.

Para poder implementar esta estrategia es necesario decidir el valor de tres parámetros: valor del incremento, valor inicial de la distancia y valor máximo que puede alcanzar. En la implementación que evaluamos en este trabajo, cada vez que se debe incrementar la distancia multiplicamos por dos su valor, porque esta operación es muy rápida de efectuar. Como valor máximo hemos seleccionado 1024 páginas, porque experimentalmente hemos comprobado que este valor queda lejos del que ofrece mejor rendimiento para todos los programas que hemos evaluado. Y el valor inicial viene determinado por la cantidad de páginas que se piden para cada ejecución de la selección de prefetch que, como explicamos a continuación, es 16 páginas. En la sección 6.23, se puede ver los resultados de aplicar esta heurística tan sencilla ofrece un buen rendimiento para todos los programas que hemos evaluado, sin que el tiempo de cálculo involucrado perjudique el rendimiento de los programas.

Otra modificación que hemos introducido en la tarea de selección de páginas consiste en aprovechar el uso de la llamada a sistema de lectura asíncrona para seleccionar varias páginas en cada ejecución de una instrucción y, así facilitar las optimizaciones del SO en el acceso a disco. Hay que decir que, mientras el patrón de accesos de una instrucción no

se considera estabilizado sólo se solicita la carga anticipada de una página, para evitar peticiones de prefetch erróneas (ver figura 6.12).

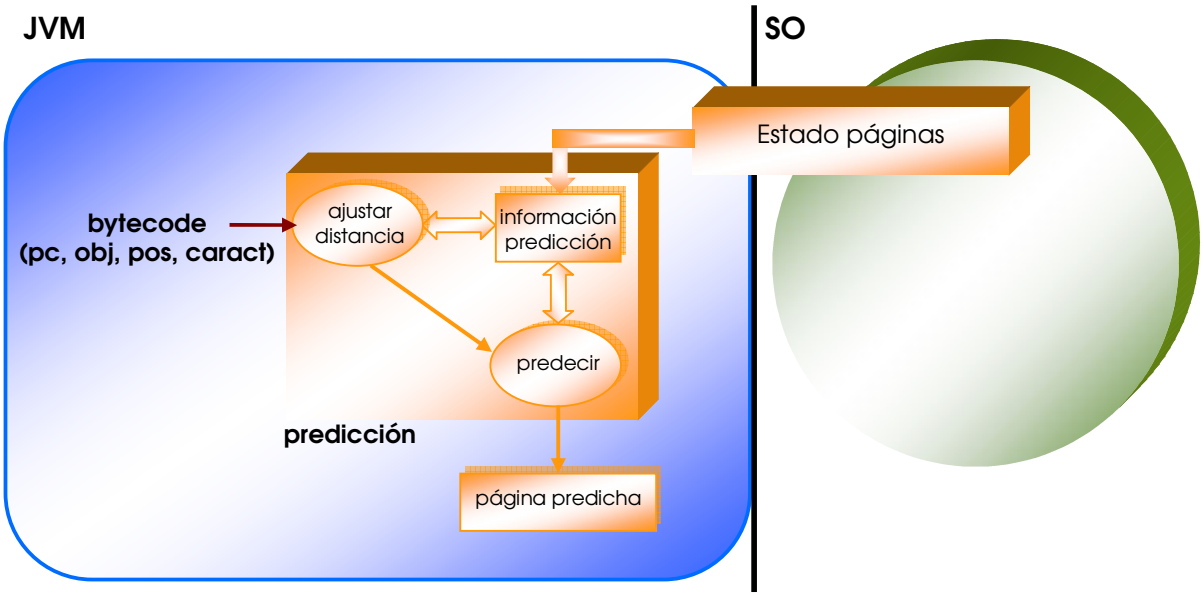

Figura 6.12 Predicción de accesos no estabilizada

El código necesario para implementar esta funcionalidad consiste simplemente en aplicar el patrón de accesos varias veces para predecir el próximo conjunto de páginas que interesa cargar anticipadamente (*cluster*). Por lo tanto, el número de ejecuciones que provocan la predicción de páginas para una instrucción se divide por el tamaño del cluster (ver figura 6.13).

En la implementación actual de nuestra estrategia hemos seleccionado como tamaño de cluster 16 páginas y éste es también el valor inicial de la distancia de prefetch. El motivo para seleccionar este valor es que es el número de páginas que se aconseja utilizar como máximo en la estrategia de prefetch secuencial implementada en Linux, ya que más allá de esta cantidad no es posible optimizar el número de accesos al área de swap.

Una vez obtenido el conjunto de páginas que interesa cargar con antelación, para cada página de ese conjunto que se encuentra en el área de swap la JVM solicita su carga asíncrona, mediante la nueva llamada a sistema. En este punto, hay que destacar que

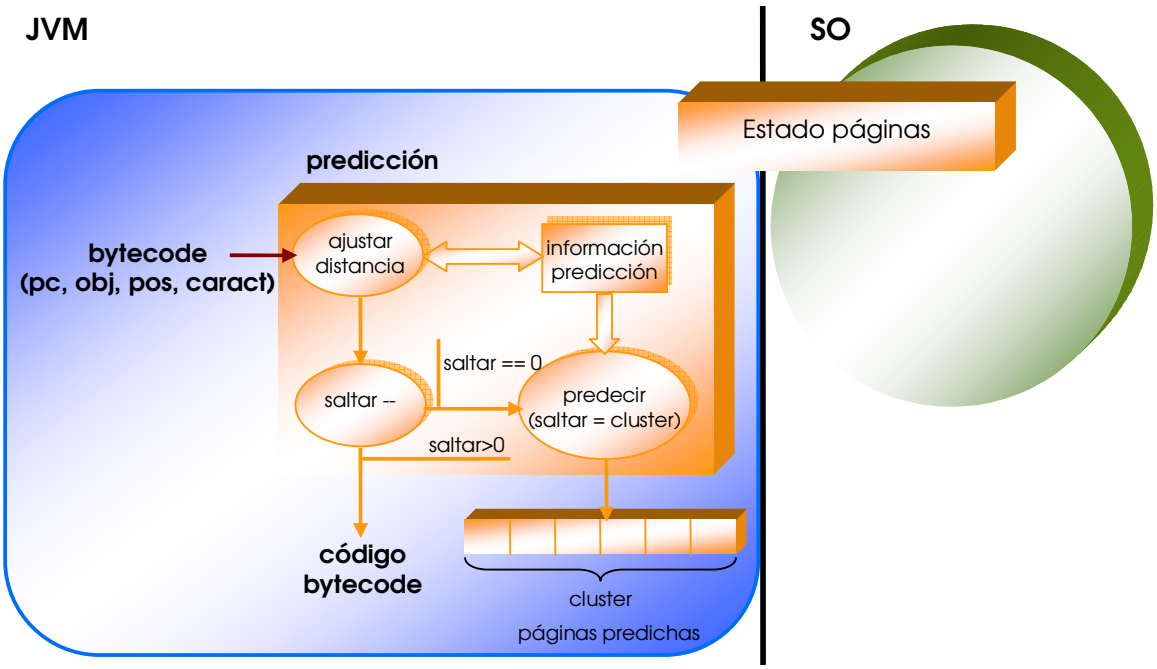

Figura 6.13 Predicción de accesos estabilizada

hemos eliminado de la JVM el flujo que se encarga del prefetch en la estrategia de prefetch transparente al SO, y todo el código necesario para gestionarlo. Esto supone una gran simplificación del código de prefetch y permite aumentar la estabilidad y eficiencia del mecanismo.

Una vez ejecutada la llamada a sistema de petición de carga, la JVM comprueba el resultado de la solicitud. En caso de que el sistema se encuentre sobrecargado y, por lo tanto, haya cancelado la operación de prefetch, la JVM almacena la petición en un buffer de páginas pendientes de prefetch (ver figura 6.14). Si, por el contrario, el sistema indica que ha entregado con éxito al gestor de disco la petición de carga, entonces la JVM inicia la gestión de las peticiones que tenga pendientes de prefetch, aprovechando que las condiciones del sistema son más favorables para la carga en memoria (ver figura 6.15).

La gestión que hemos implementado para las operaciones de prefetch canceladas por el SO es muy sencilla. El orden de tratamiento es FIFO y, como ya hemos dicho, se inicia si las solicitudes de carga correspondientes a la ejecución de la instrucción actual se completan

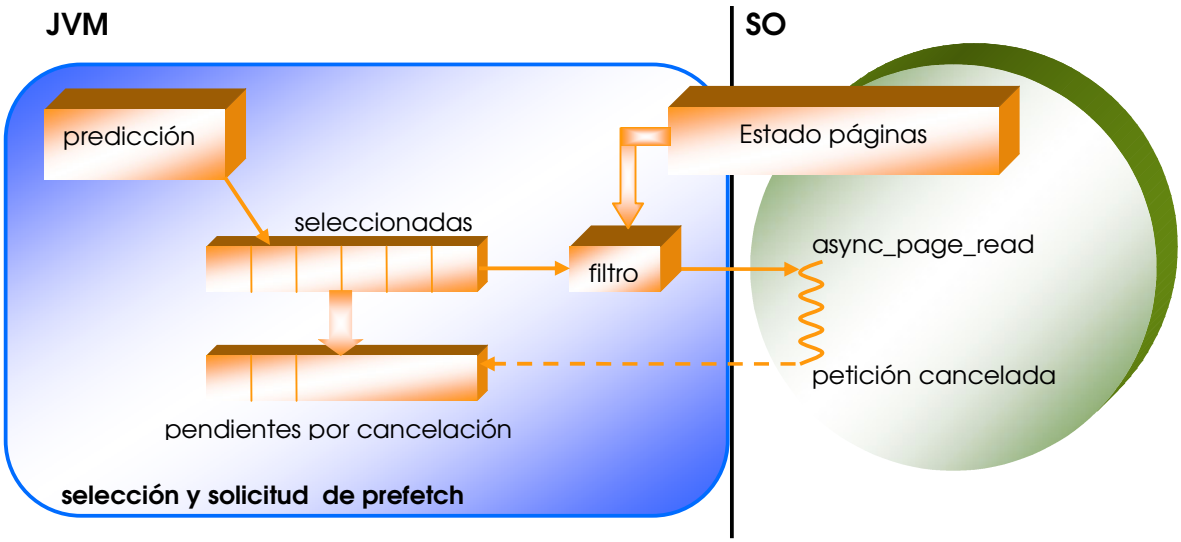

Figura 6.14 Condiciones no favorables para el prefetch: operaciones canceladas

con éxito. Para tratar cada petición, se extrae del buffer de peticiones pendientes y se comprueba el estado de la página involucrada y, si todavía se encuentra en el área de swap, se solicita de nuevo su carga asíncrona. El tratamiento de las peticiones pendientes se suspende cuando ya no es posible completar con éxito las solicitudes de carga, es decir, cuando el SO cancela una de las operaciones, y se reanuda cuando el SO sea capaz de volver a aceptar solicitudes de carga asíncrona.

Hay que decir que esta política tan sencilla para la gestión de las operaciones ha demostrado un buen rendimiento para la estrategia de prefetch en los experimentos que hemos ejecutado (ver sección 6.6.2).

Alternativa para el tratamiento de las operaciones de prefetch canceladas

Una alternativa más simple para el tratamiento de las operaciones de prefetch canceladas por el SO consiste en descartarlas. Hay que destacar que esta es la opción adoptada por la estrategia de prefetch secuencial que ofrece Linux, en la versión de kernel que hemos usado en esta implementación, ya que, cuando la carga de la máquina es alta, simplemente

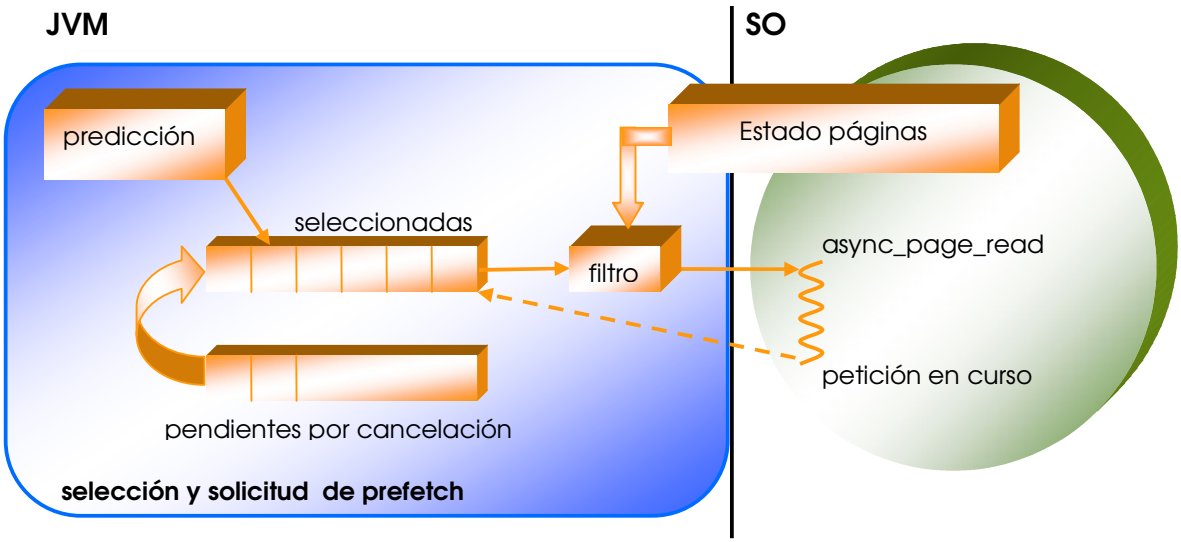

Figura 6.15 Condiciones favorables para el prefetch: prefetch en curso

desactiva su estrategia de prefetch, para dedicarse sólo a las cargas que realmente son imprescindibles para que los procesos continúen.

Sin embargo, hemos evaluado el uso de esta opción en el prefetch cooperativo y hemos visto que provoca que baje su rendimiento (ver sección 6.6.2). El motivo es que, cuando el sistema de memoria se encuentra bajo una alta presión, las probabilidades de solicitar una operación de prefetch en un momento de saturación del sistema son muy altas y, por lo tanto, son muchas las operaciones que, en primera instancia, se tienen que cancelar. Como esta alternativa no da la opción de reintentar estas operaciones, todas las cancelaciones se convierten en oportunidades perdidas para solapar los accesos a disco con el tiempo de cálculo del proceso.

6.6 EVALUACIÓN DEL PREFETCH COOPERATIVO

En esta sección presentamos los resultados de los experimentos que hemos realizado para evaluar la estrategia de prefetch cooperativo. Estos experimentos se pueden separar en dos grandes grupos. El primer grupo de experimentos, que se describen en la sección 6.6.2, son los que nos han servido para validar las decisiones que hemos tomado durante la imple-

mentación de la estrategia. El segundo grupo de experimentos compara el rendimiento de un conjunto de aplicaciones ejecutadas sobre el entorno modificado con nuestra propuesta de prefetch cooperativo y ejecutadas sobre el entorno original (ver sección 6.6.3).

Hemos ejecutado todos estos experimentos sobre un PC con un procesador Pentium IV a 1,8 GHz y con 256Mb de memoria. El objetivo de las técnicas de prefetch es mejorar el rendimiento de los programas que hacen un uso intensivo de la memoria virtual. Por este motivo, para cada programa, hemos seleccionado la cantidad de memoria física que hace necesario que el programa utilice el mecanismo de memoria virtual durante su ejecución, y hemos arrancado la máquina limitando a esa cantidad la memoria disponible en el sistema.

6.6.1 Metodología para los experimentos

Para evaluar el rendimiento del prefetch cooperativo hemos utilizado el mismo mecanismo de contadores utilizado en el capítulo 3 y en el capítulo 5, adaptándolo a las necesidades de esta evaluación.

En estos experimentos nos interesa desglosar el tiempo de ejecución separando el tiempo dedicado a resolver fallos de página del resto del tiempo de ejecución. A su vez, clasificamos el tiempo de fallo de página en función del tipo de fallo de página involucrado: fallos de página que no requieren un acceso a disco para ser completados (fallos soft), como, por ejemplo, los que se deben al acceso a páginas que se encuentran en proceso de intercambio con el área de swap, y fallos de página que sí necesitan del acceso a disco para poder ser resueltos (fallos hard). También, hemos evaluado el tiempo de bloqueo que sufre el proceso durante las operaciones de prefetch, para lo cual hemos introducido un nuevo contador en el kernel. Del resto de tiempo de ejecución, la única distinción añadida que nos interesa es saber la cantidad de tiempo dedicada a ejecutar la tarea de selección de páginas. Este tiempo sólo puede ser medido por la JVM, por lo que, utilizando los contadores hardware de tiempo que ofrece nuestro procesador, hacemos que la JVM acumule en una variable el tiempo invertido en esa tarea.

En cuanto a las peticiones de lectura del área de swap, queremos distinguir entre las que provienen de la llamada a sistema de prefetch y las que solicitan los fallos de página del

programa, separando también los fallos de página que realmente implican un acceso al área de swap, y los que se pueden resolver sin ese acceso.

Por último, hay que tener en cuenta la influencia que puede tener en el rendimiento de los programas la estrategia de prefetch secuencial implementada por Linux ya que, en el mejor de los casos, sus decisiones son redundantes con respecto a las decisiones del prefetch cooperativo. En la versión de kernel que hemos utilizado para desarrollar y evaluar el prefetch cooperativo (versión 2.4.18-14), si la cantidad de memoria física libre es muy baja, Linux descarta la utilización de su estrategia de prefetch. Este comportamiento, que no se daba en las versiones previas de Linux que hemos utilizado durante el desarrollo de este trabajo, hace previsible que Linux no aplique sus propias decisiones de prefetch durante la ejecución de nuestros experimentos, ya que éstos involucran una alta carga para el sistema de memoria. De todas maneras, hemos optado por desactivar explícitamente el prefetch original de Linux durante la ejecución de los benchmarks sobre el prefetch cooperativo.

6.6.2 Evaluación de las decisiones de diseño

En esta sección presentamos el grupo de experimentos que hemos utilizado para validar las decisiones de diseño y de implementación que hemos tomado en el desarrollo de esta estrategia.

En estos experimentos era importante utilizar un benchmark que facilitara el análisis de los resultados, por lo que hemos seleccionado el kernel de multiplicación de matrices. El tamaño de las matrices de entrada que hemos seleccionado hace necesario el uso de la memoria virtual, y por lo tanto tiene sentido aplicar la estrategia de prefetch para mejorar su rendimiento (ver tabla 6.1). Además, con estas matrices de entrada, es posible mantener varios working sets en memoria, ya que, como se puede ver en la tabla 6.1, se necesita alrededor de 16Mb para almacenar una fila de la matriz recorrida por filas (matriz A) y varias columnas consecutivas de la matriz recorrida por columnas (matriz B). Por lo tanto, se puede simplificar el patrón de accesos asociado a las instrucciones tal y como se describió en la sección 5.1.3 del capítulo 5. Es decir, una vez captado el patrón de accesos, se toma una decisión de prefetch para el primer acceso a cada página de un

working set y, como consecuencia, se solicita la carga anticipada de la página equivalente del siguiente working set. En este tipo de simplificación, pues, medimos la distancia de prefetch en unidades de working set del programa y siempre vale una unidad.

Matriz	Tamaño	Páginas fila	Páginas columna
Matriz A (ints)	1024x4096 (16Mb)	4 (0,015Mb)	1024 (4Mb)
Matriz B (ints)	4096x16384 (256Mb)	16 (0,062Mb)	4096 (16Mb)

Tabla 6.1 Características de las matrices de entrada para la multiplicación (AxB)

Los resultados que mostramos en esta sección son los obtenidos tras el cálculo de cinco filas de la matriz resultado, ya que esto es suficiente para analizar el comportamiento de las alternativas para la implementación del prefetch cooperativo.

Eficiencia del bitmap

En este primer experimento medimos la necesidad de utilizar el bitmap que representa el estado de las páginas del heap. Para ello hemos querido estimar el coste que implicaría hacer todas las solicitudes de carga, sin filtrar las que se refieren a páginas ya presentes en memoria física.

El coste de este tipo de solicitudes redundantes se limita prácticamente al coste de entrar en el sistema, ya que el código de la llamada a sistema lo primero que hace es comprobar en la tabla de páginas del proceso si la página ya está presente en memoria y, de ser así, retorna a usuario inmediatamente. Por lo tanto, medir el tiempo utilizado para ejecutar los traps involucrados es una buena estimación de la sobrecarga que implica no filtrar las peticiones redundantes. Para hacer esta medida, hemos substituido el uso de la llamada a sistema de prefetch por una llamada a sistema vacía, y hemos medido el tiempo asociado a las llamadas tanto si se consulta el bitmap para eliminar las peticiones redundantes como si no se hace.

Bitmap	Num. peticiones pref.	Tiempo de trap	Tiempo de fp hard
No usado	369.604.819	241,18 segs	279,51 segs
Usado	421.952	0,29 segs	267,63 segs

Tabla 6.2 Influencia en el rendimiento del bitmap del estado de las páginas

En la tabla 6.2 mostramos el resultado de este experimento. Hemos contado para cada caso, el número de peticiones de prefetch que se ejecutan, el tiempo de trap implicado en esas peticiones y el tiempo que el SO ha dedicado a resolver fallos de página del proceso.

Podemos observar que, si no se utiliza el bitmap para filtrar peticiones redundantes, el número de entradas en el sistema es casi 370 millones y el tiempo necesario para ejecutarlas es alrededor de 241 segundos. Teniendo en cuenta que el tiempo dedicado a resolver fallos de página es de alrededor de 268 segundos, aunque la estrategia de prefetch sea capaz de evitar todos los fallos de página, la sobrecarga involucrada en el mecanismo para cargar anticipadamente las páginas hace imposible mejorar el rendimiento final del programa.

Sin embargo, si se consulta el bitmap y se entra en el sistema sólo para aquellas páginas que se encuentran en el área de swap, entonces el número de traps se reduce a unos 422 mil con un tiempo asociado de tan solo 0,29 segundos.

Es decir, la consulta del bitmap permite dividir por un factor mayor de 1000 la sobrecarga que añaden los traps al sistema sobre el mecanismo de prefetch. Por lo tanto, se incrementa considerablemente el margen para los posibles beneficios de la estrategia de prefetch.

Alternativas en el comportamiento de la solicitud de carga anticipada

Durante el diseño e implementación de la estrategia de prefetch, hemos considerado tres posibles comportamientos de las solicitudes de carga anticipada. En esta sección presentamos los resultados de los experimentos que hemos realizado para validar la decisión final que hemos adoptado en nuestra estrategia de prefetch.

Lo primero que hay que decir es que las tres opciones coinciden en dar a la JVM el papel de guía del prefetch, ya que es ella la que, en base al conocimiento que tiene sobre el comportamiento del programa, selecciona las páginas que conviene cargar con antelación para evitar fallos de página.

Partiendo de esta base, las dos primeras alternativas afectan a la participación del sistema en las decisiones de prefetch. La primera opción es hacer que una llamada a sistema de prefetch, siempre que respete la fiabilidad del sistema, concluya con la petición de lectura entregada al gestor de disco. Es decir, las peticiones de prefetch que hace la JVM son de ejecución *obligatoria* y, por lo tanto, el SO se limita a efectuar la operación, bloqueando en caso necesario al proceso que la solicita (ver sección 6.5.2). La segunda opción es la que hemos adoptado como parte de nuestra propuesta. Consiste en dar al SO la facultad de cancelar las solicitudes de prefetch cuando las condiciones de ejecución hacen necesario bloquear al proceso para poder completar la solicitud. Con esta alternativa cada petición de prefetch de la JVM es una *pista* para mejorar el rendimiento del programa, pero la decisión final queda en manos del SO.

La alternativa de implementar la petición de prefetch como una *pista* tiene a su vez dos posibles comportamientos. La primera posibilidad es que el SO implemente la cancelación de forma transparente a la JVM. En este caso, la JVM no es consciente de qué solicitudes se han cancelado y, por lo tanto, no tiene opción de reaccionar adecuadamente (*pista sin reacción*). La segunda posibilidad consiste en que el SO informe a la JVM, mediante el valor de retorno de la llamada a sistema, sobre aquellas operaciones que ha debido cancelar para no perjudicar el rendimiento del proceso. De esta manera, la JVM puede gestionar estas operaciones canceladas reintentando su solicitud cuando las condiciones de ejecución sean más favorables (*pista con reacción*). Este último es el comportamiento que sigue nuestra propuesta de prefetch cooperativo.

En la figura 6.16 comparamos las tres alternativas de comportamiento de una petición de prefetch: *obligatoria*, *pista sin reacción* y *pista con reacción*. La figura 6.16.a muestra el tiempo de ejecución desglosado del benchmark: tiempo de fallo de página hard (*tfp hard*), tiempo de fallo de página soft (*tfp soft*), tiempo de bloqueo de la petición de prefetch (*bloqueo petición*), tiempo de selección de páginas de prefetch (*selección*) y resto de tiempo de cálculo (*cálculo*). Por otro lado, la figura 6.16.b muestra el número de fallos de página (*fp hard* y *fp soft*) y el número de páginas cargadas por prefetch (*carga anticipada*).

Lo primero que podemos observar es que la opción de solicitud *obligatoria* consigue evitar prácticamente todos los fallos de página del programa y, como consecuencia, se elimina

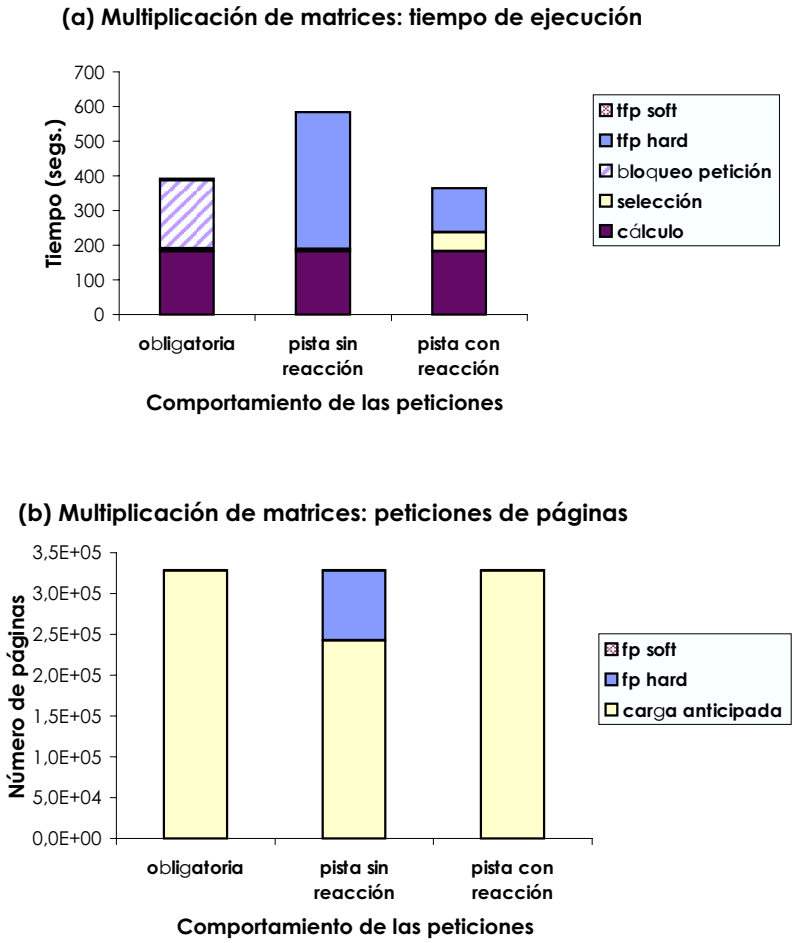

Figura 6.16 Posibles comportamientos de las peticiones de carga anticipada

el tiempo dedicado a resolver fallos de página. Sin embargo, el tiempo de bloqueo que aparece durante la atención a las peticiones de prefetch es significante, lo cual impide un total solapamiento entre los accesos a disco y el cálculo del proceso y limita los beneficios que se pueden obtener del prefetch.

Hemos analizado detalladamente las causas que provocan el tiempo de bloqueo, para determinar en qué situaciones era aconsejable cancelar la petición de prefetch. Los motivos de bloqueo que hemos encontrado son dos: la saturación de la cola de peticiones del gestor de disco y la escasez de memoria física libre durante la reserva de la página para

soportar los datos cargados. De hecho, la mayor parte del tiempo de bloqueo es debido a la saturación de la cola de disco. Por este motivo, decidimos evaluar el comportamiento de la estrategia de prefetch al cancelar la operación sólo cuando la cola del gestor de disco se saturaba. Los resultados de esa evaluación nos demostraron que esas cancelaciones no eran suficientes para eliminar el tiempo de bloqueo ya que, entonces, aumentaba la presión sobre la tarea de reserva de memoria y, por lo tanto, aumentaba considerablemente el tiempo de bloqueo implicado en esa reserva.

Por lo tanto, para eliminar el tiempo de bloqueo asociado a las peticiones de prefetch es necesario cancelar las peticiones tanto en el caso de saturación de disco como en el caso de saturación del sistema de reserva de memoria.

Respecto a la alternativa de cancelar las peticiones de forma transparente a la JVM (*pista sin reacción*), hay que decir que es la alternativa que peor rendimiento ofrece a la estrategia de prefetch (ver figura 6.16.a). El motivo es que la ejecución del benchmark está ejerciendo mucha presión sobre el sistema de memoria, con lo cual es necesario cancelar muchas peticiones de prefetch. Como consecuencia, más del 25 % de las páginas cargadas con antelación en la opción de petición *obligatoria* pasan a ser fallos de página en la ejecución sobre esta alternativa (ver figura 6.16.b). Hay que destacar que la estrategia de prefetch secuencial implementada por Linux tiene un comportamiento similar a esta opción: si el sistema se encuentra sobrecargado Linux decide descartar la ejecución de la carga anticipada sin considerar posibles reintentos.

Por último, la opción de *pista con reacción* es la alternativa que hemos seleccionado como parte de la estrategia de prefetch cooperativo ya que es la que ofrece un mejor rendimiento. Esta alternativa coincide con el caso de *pista sin reacción* en que ambas consiguen que las peticiones de prefetch se ejecuten sin ningún tiempo de bloqueo asociado, lo que permite que las lecturas asíncronas de disco se solapen con el tiempo de cálculo del proceso (ver figura 6.16.a). La diferencia radica en la cantidad de cargas asíncronas que se completan con éxito. La opción de petición *pista con reacción* es capaz de aprovechar las bajadas momentáneas en la carga del sistema para reintentar y completar las operaciones previamente canceladas, lo cual permite que evite la mayoría de fallos de página del proceso a diferencia de la opción *pista sin reacción*. Hay que decir que los reintentos de

las operaciones canceladas llevan asociado un aumento en el tiempo dedicado a la tarea de selección de páginas, que es ignorable para las otras dos alternativas. Sin embargo, este incremento de tiempo de cálculo no impide el buen rendimiento de la estrategia de prefetch al adoptar esta alternativa. Además, hay que destacar que, como hemos dicho en la sección 6.5.3, la implementación de la gestión del buffer de peticiones pendientes se basa en una política FIFO muy simple, por lo que cabe la opción de que una estrategia más sofisticada pueda optimizar este tratamiento.

Alternativas en la actualización de la TP

Como hemos explicado en la sección 6.5.2, la carga asíncrona de una página no se puede dar por finalizada hasta que se actualiza la tabla de páginas del proceso, estableciendo la asociación entre la página lógica y la página física y, consiguiendo de esta manera, que los accesos del proceso a esas direcciones se puedan resolver sin provocar fallo de página.

Hemos evaluado las dos alternativas que se pueden utilizar para completar este tratamiento y mostramos los resultados de esta evaluación en la figura 6.17. La primera es imitar el tratamiento que hace Linux para las páginas cargadas mediante su estrategia de prefetch secuencial. Este tratamiento consiste en esperar al primer acceso del proceso a una página de prefetch para actualizar su entrada en la tabla de páginas (*act. fp* en la figura 6.17). La segunda opción, que es la que hemos adoptado en el prefetch cooperativo, consiste en modificar la rutina de atención al disco para que se encargue de actualizar la entrada de la tabla de páginas en cuanto la lectura de la página concluya (*act. int. disco* en la figura).

La figura 6.17.a muestra el tiempo de ejecución desglosado del benchmark y la figura 6.17.b muestra los fallos de página y el número de páginas cargadas con antelación. Podemos ver, que para ambas opciones, el número de páginas cargadas de forma asíncrona es el mismo (*carga anticipada* en la figura 6.17.b). Sin embargo, el efecto de estas cargas anticipadas sobre el rendimiento del proceso depende de la opción utilizada para actualizar la tabla de páginas. Podemos observar que, con la opción de actualizar la tabla de páginas durante el primer acceso a cada página, las operaciones de prefetch no consiguen evitar que el proceso se ejecute provocando un alto número de fallos de página de acceso a disco (*fp hard* en la figura 6.17.b). De tal manera que, como se ve en la figura 6.17.a, el tiempo

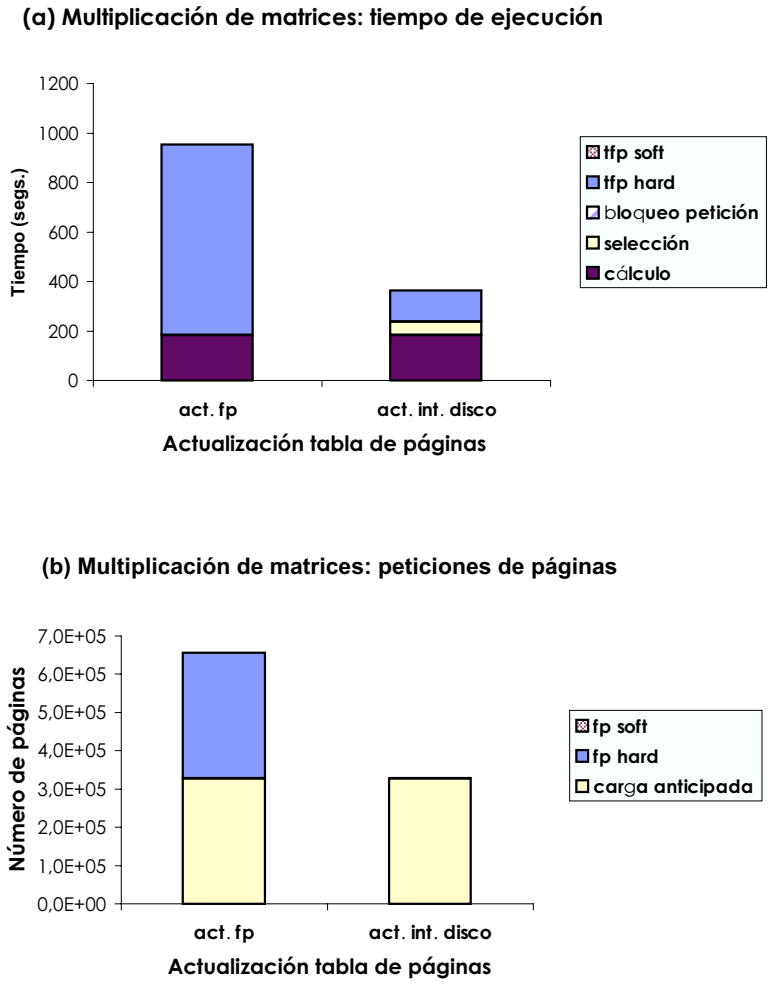

Figura 6.17 Alternativas para la actualización de la tabla de páginas

dedicado a resolver fallos de página (*tfp hard*) es seis veces mayor que para la ejecución utilizando la alternativa de actualizar la tabla de páginas al gestionar las interrupciones de disco.

La explicación a este comportamiento la encontramos en la interacción con el algoritmo de reemplazo de memoria que implementa Linux. Este algoritmo se ejecuta cuando es necesario liberar memoria física y selecciona, para enviar al área de swap, las páginas que los procesos no están usando activamente. En la implementación que hace la versión de

Linux sobre la que hemos desarrollado el prefetch, se asume que las páginas que no se encuentran ligadas a ninguna tabla de páginas son las candidatas más adecuadas para ser expulsadas al área de swap. Por lo tanto, si retrasamos el momento de asociar a la tabla de páginas correspondiente las páginas cargadas por prefetch, estamos aumentando la probabilidad de que sean expulsadas de nuevo al área de swap antes de que el proceso las referencie. Este efecto, unido a la alta frecuencia con la que es necesario ejecutar el algoritmo de reemplazo cuando la presión sobre el sistema de memoria es alta, explica el bajo rendimiento de la estrategia de prefetch cuando la actualización de la tabla de páginas se efectúa durante el acceso del proceso. Sin embargo, si en el momento de finalizar la lectura de disco actualizamos la tabla de páginas correspondiente, entonces esas páginas pasan a tener las mismas oportunidades de permanecer en memoria física que las páginas cargadas bajo demanda, consiguiendo así que permanezcan en memoria hasta que el proceso las referencia.

6.6.3 Evaluación de los beneficios del prefetch cooperativo

En esta sección presentamos la evaluación de la estrategia de prefetch cooperativo. Para ello, hemos ejecutado varios núcleos de cálculo y aplicaciones sobre el entorno de ejecución que ofrece Linux por defecto y sobre le entorno modificado con nuestra estrategia. Recordemos que hemos desactivado el prefetch de Linux para las ejecuciones de los benchmarks sobre el entorno con nuestra estrategia. Por otro lado, para evaluar la influencia que tiene la estrategia de prefetch de Linux sobre el rendimiento del entorno original de ejecución, hemos ejecutado los benchmarks también sobre el entorno original desactivando su estrategia de prefetch.

Durante la evaluación de la estrategia de prefetch hemos querido aislar la influencia que puede tener sobre su rendimiento la heurística que utilizamos para automatizar el cálculo de la distancia de prefetch. Por este motivo, hemos hecho un primer grupo de experimentos utilizando como distancia de prefetch un valor determinado experimentalmente. En el segundo grupo de experimentos utilizamos la distancia obtenida de forma automática en tiempo de ejecución, de manera que podemos evaluar la eficacia de la heurística que hemos implementado.

Benchmarks

Además del kernel de multiplicación de matrices, que hemos utilizado para validar las decisiones de diseño, utilizamos algunos benchmarks de los suministrados como parte del conjunto de programas de prueba JavaGrande [BSW+00] y como parte del conjunto de programas Java NAS [FSJY03].

En la sección 6.6.2 hemos descrito las características del kernel de multiplicación de matrices. Por otra parte, en la sección 3.1.1 del capítulo 3 se puede encontrar una descripción de los benchmarks de JavaGrande que vamos a utilizar, de los que seleccionamos únicamente aquellos que se pueden ver afectados por nuestra estrategia de prefetch, es decir, los que utilizan arrays de grandes dimensiones.

Respecto al conjunto de programas Java NAS hemos seleccionado el kernel RHS. Se trata de una rutina usada por los programas SP y BT, que forman parte de los programas suministrados como parte de los Java NAS. Esta rutina calcula el vector del lado derecho de un sistema de ecuaciones. Hemos seleccionado la versión *clase B*, que requiere 128Mb para albergar todo el conjunto de datos. Todos estos datos están organizados en una serie de arrays que la rutina RHS utiliza con un patrón de accesos strided. Para ejecutar este programa, arrancamos la máquina limitando su tamaño de memoria física a 128Mb.

La tabla 6.3 resume las principales características de los experimentos que realizamos.

Programas	**Memoria Boot**	**Memoria arrays grandes**	**Tipos de acceso**
MULTIPLICACIÓN DE MATRICES	128Mb	336Mb	Strided (3 niveles)
RHS CLASE B	128Mb	185Mb	Strided (3 niveles)
CRYPT SIZEC	64Mb	143,05Mb	Secuencial
SPARSE SIZEC	32Mb	45,75Mb	Aleatorio y secuencial
FFT SIZEA	32Mb	32Mb	Strided dinámico (3 niveles)
HEAPSORT SIZEC	64Mb	95,37Mb	Strided y aleatorio

Tabla 6.3 Resumen de las características de los experimentos

Resultados de los experimentos con distancia de prefetch manual

En esta sección comparamos los resultados que hemos obtenido al ejecutar los benchmarks sobre el prefetch cooperativo (*Prefetch cooperativo*) y sobre el entorno original de ejecución tanto con el comportamiento por defecto que ofrece Linux (*Entorno original (prefetch kernel)*) como desactivando el prefetch de Linux (*Entorno original (sin prefetch kernel)*). En la ejecución sobre el prefetch cooperativo, la distancia de prefetch que utilizamos la hemos decidido de forma experimental y se aplica el mismo valor para todas las instrucciones.

Los resultados de los experimentos que hemos realizado se muestran desde la figura 6.18 hasta la figura 6.23. En estas figuras presentamos tanto el tiempo de ejecución de los programas como las peticiones de acceso al área de swap. Hemos desglosado el tiempo de ejecución distinguiendo el tiempo dedicado a resolver fallos de página de acceso a disco (*tfp hard*), el tiempo dedicado a resolver fallos de página que no requieren de ese acceso (*tfp soft*) y el resto de tiempo del programa (*cálculo*) que incluye también el dedicado a la tarea de selección de páginas. Hay que destacar que hemos obviado el tiempo de bloqueo debido a las peticiones de prefetch, porque es nulo para la implementación de estas peticiones que hemos seleccionado. En cuanto a las peticiones de acceso al área de swap, distinguimos entre las debidas a peticiones de carga asíncrona (*carga anticipada*), las debidas a fallos de página que realmente requieren esa lectura (*fp hard*), y las debidas a fallos de página que se pueden resolver sin ese acceso (*fp soft*).

El primer aspecto a destacar es que todos estos benchmarks sobrecargan el sistema de memoria virtual. Por este motivo, en cuanto Linux detecta esta sobrecarga, decide desactivar su estrategia de prefetch independientemente de la configuración decidida por el administrador de la máquina. Esto explica que el comportamiento de los benchmarks en el entorno original, tanto con la configuración por defecto como con el prefetch de Linux desactivado, sea muy similar.

En la figura 6.18 mostramos los resultados de la ejecución de la MULTIPLICACIÓN DE MATRICES, ejecutada sobre 128Mb de memoria y con una distancia de un working set.

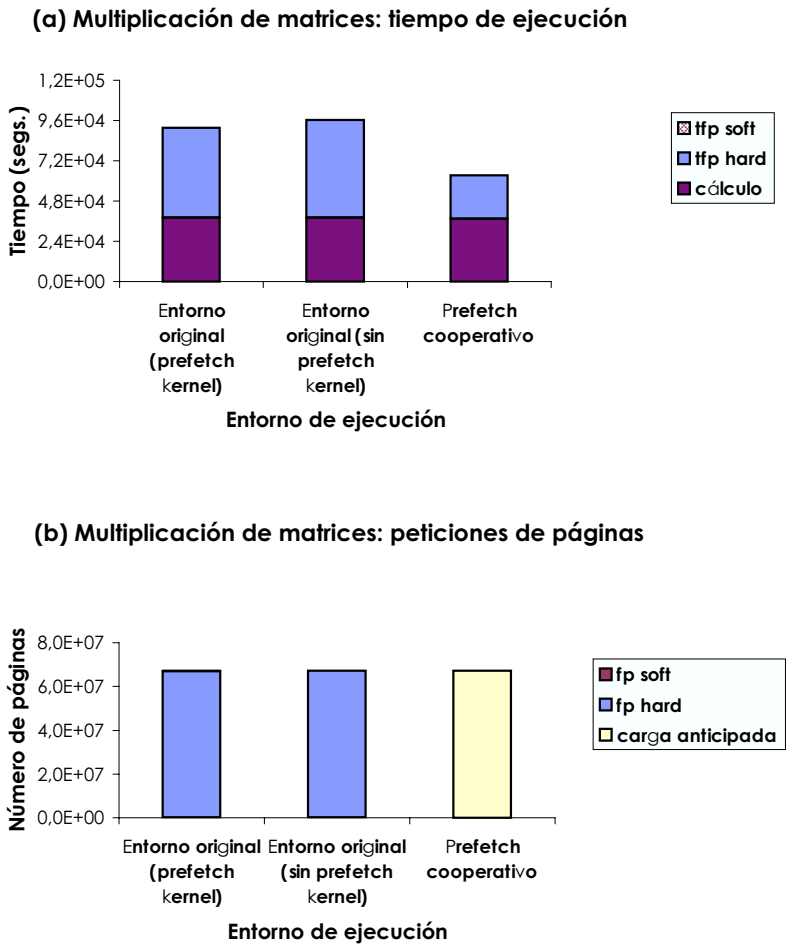

Figura 6.18 Resultados de la MULTIPLICACIÓN DE MATRICES

La figura 6.18.a muestra el tiempo de ejecución y la figura 6.18.b muestra las peticiones de acceso al área de swap. Podemos observar que las páginas cargadas con antelación con el prefetch cooperativo evitan la mayoría de fallos de página de la aplicación (ver figura 6.18.b). Como consecuencia, en la figura 6.18.a se puede ver que el rendimiento del programa ejecutado sobre nuestro entorno mejora alrededor de un 22 % comparado con el obtenido sobre el entorno original de Linux. Hay que destacar que usando prefetch cooperativo, aunque el número de fallos de página hard se puede ignorar, el tiempo necesario para resolverlos aún representa alrededor de un 35 % del tiempo total de ejecución del benchmark. Esto significa que el tiempo medio de fallo de página aumenta considerable-

mente. Este incremento se debe a que la ejecución con el prefetch cooperativo aumenta el número de peticiones acumuladas en la cola del gestor de disco, con lo que también aumenta el tiempo de respuesta del gestor a esas peticiones y, por lo tanto, el tiempo que pasa desde que el proceso provoca el fallo de página hasta que el gestor es capaz de completar la lectura de los datos solicitados. Por este motivo, para conseguir que la estrategia de prefetch sea eficaz es muy importante conseguir eliminar la mayoría de los fallos de página, ya que la penalización sobre el rendimiento de los que no se evitan es muy alta.

En las figuras 6.19.a y 6.19.b mostramos los resultados de la ejecución de RHS. La cantidad de memoria física utilizada es de 128Mb y, para el prefetch cooperativo se ha utilizado un valor de distancia de prefetch de 32 páginas. En la figura 6.19.b se observa que el prefetch cooperativo es capaz de eliminar la mayoría de los fallos de página de acceso a disco que el programa genera durante su ejecución en el entorno original. Sin embargo, aparece un incremento en la cantidad de fallos de página soft, debido a las páginas solicitadas con antelación cuya carga no es posible completar antes de que el proceso las referencie. A pesar de este incremento, hay que destacar que el tiempo dedicado a resolver fallos de página, considerando tanto los fallos de página hard como los soft, es menos de la mitad que el necesario para resolver los fallos de página en el entorno original de Linux (ver figura 6.19.a). Esta importante reducción en el tiempo de resolución de fallo de página se refleja también en una mejora del rendimiento global del programa alrededor del 40 % con respecto al rendimiento obtenido en el entorno original de ejecución (ver figura 6.19.a).

Los resultados de la ejecución de Crypt se muestran en las figuras 6.20.a y 6.20.b. Este experimento se ha realizado utilizando 64Mb de memoria física y con una distancia de prefetch para el prefetch cooperativo es de 128 páginas.

En la figura 6.20.a podemos comparar los resultados de la ejecución sobre el entorno original de ejecución y sobre el entorno modificado con el prefetch cooperativo. Podemos observar que la ejecución con el prefetch cooperativo reduce en un 73,2 % el tiempo dedicado a resolver fallos de página, lo cual involucra una mejora en el rendimiento global de un 13,11 % con respecto a la ejecución en el entorno original de Linux. Hay que tener en cuenta que este programa, ejecutado sobre el entorno original, dedica sólo un 18 % de

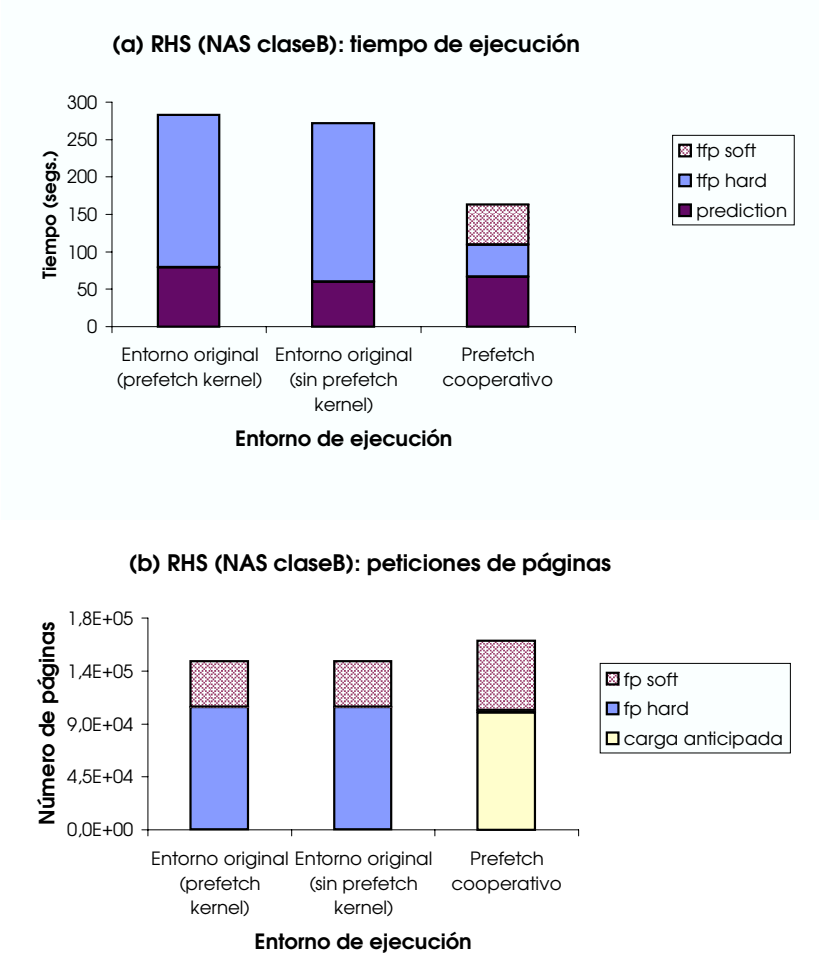

Figura 6.19 Resultados de RHS

su tiempo total de ejecución a la resolución de fallos de página. Por este motivo, aunque nuestra estrategia de prefetch consigue una reducción muy importante en el tiempo de fallo de página, el impacto de esta reducción sobre el tiempo total de ejecución no es tan significativo. En la figura 6.20.b podemos comprobar que el prefetch cooperativo elimina más del 97 % de los fallos de página provocados por el programa en el entorno original de ejecución. También podemos observar, que tal y como ocurría en el caso de la ejecución de RHS, se incrementan el número de fallos de página soft, aunque esto no impide que la reducción del tiempo de fallo de página sea importante.

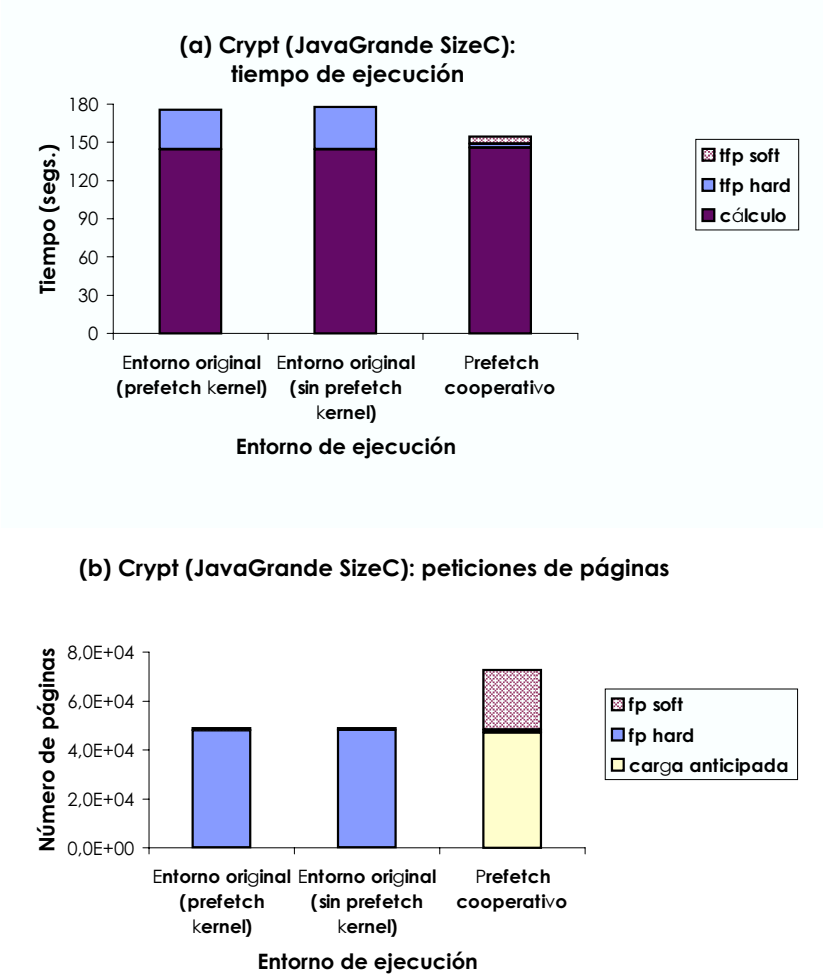

Figura 6.20 Resultados de CRYPT

En la figura 6.21 mostramos los resultados de la ejecución del benchmark SPARSE, ejecutado sobre 32Mb de memoria física y utilizando una distancia de prefetch de 64 páginas en el prefetch cooperativo.

En la figura 6.21.a podemos ver el tiempo de ejecución del benchmark, y se observa que el prefetch cooperativo consigue una mejora del 24,4 % sobre el tiempo de ejecución obtenido en el entorno original de ejecución. Hay que decir que, a pesar de esta mejora del rendimiento, el tiempo de fallo de página sigue siendo considerable. Este tiempo de fallo de página se debe, en su mayor parte, a que el sistema no es capaz de cargar con la su-

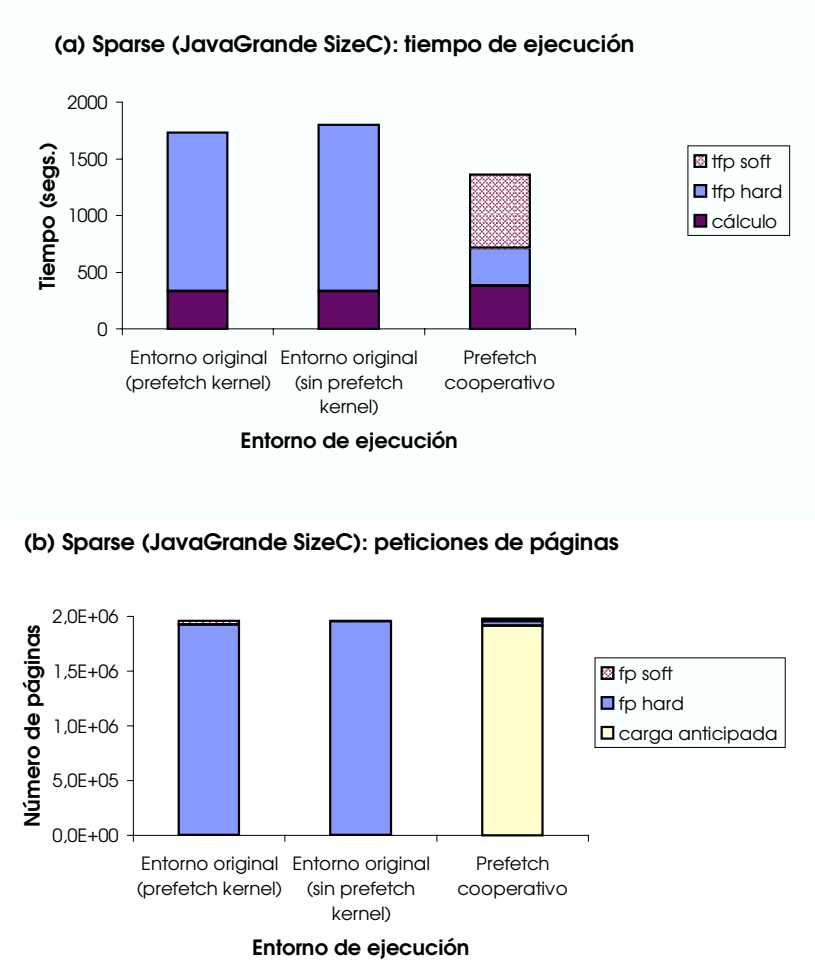

Figura 6.21 Resultados de SPARSE

ficiente antelación las páginas solicitadas asíncronamente, con lo que sólo se consigue un solapamiento parcial entre tiempo de acceso a disco y tiempo de cálculo. La explicación a este comportamiento la encontramos en el bajo porcentaje de tiempo de cálculo del programa, que representa sólo un 18,4 % del tiempo total de ejecución, y que limita el grado de solapamiento alcanzable por el prefetch de memoria. Por otro lado, en la figura 6.21.b podemos ver que la mayoría de accesos que involucran fallos de página en el entorno original de ejecución son solicitados anticipadamente por el prefetch cooperativo, de manera que se eliminan el 97,7 % de los fallos de página hard. Los fallos de página hard que no se pueden evitar son los debidos a los accesos aleatorios que ejecuta este programa

y que, por lo tanto, son impredecibles. Tenemos que remarcar que la estrategia de prefetch cooperativo es capaz de detectar este comportamiento y se desactiva para aquellas instrucciones impredecibles, evitando de esta manera la sobrecarga de ejecutar un código que no puede beneficiar su rendimiento.

En la figura 6.22 mostramos el resultado de ejecutar el benchmark FFT. La cantidad de memoria física sobre la que se ha ejecutado este experimento es 32Mb y la distancia usada por el prefetch cooperativo es 8 páginas. La figura 6.22.a muestra el tiempo de ejecución. La primera cosa a destacar es que el tiempo de cálculo de este benchmark representa sólo un 0,35 % del tiempo total de ejecución. Esto significa que esta aplicación ofrece muy pocas posibilidades para solapar tiempo de carga de disco y tiempo de cálculo, y, por lo tanto, es difícil mejorar su rendimiento usando la técnica de prefetch. Por este motivo, el prefetch cooperativo se comporta de forma muy parecida al comportamiento por defecto del entorno original. En este punto hay que destacar que el tiempo de cálculo debido a la tarea de selección no es significativo, como se puede observar comparando el tiempo de ejecución en el entorno original y el tiempo de ejecución sobre el entorno modificado con el prefetch cooperativo. Sin embargo, en la figura 6.22.b se puede comprobar que el prefetch cooperativo es capaz de evitar alrededor de un 55 % de los fallos de página que provoca el programa en su ejecución sobre el entorno original. Esta alta tasa de aciertos en las predicciones, sin un aumento significativo del tiempo de cálculo, es especialmente destacable porque este programa utiliza un patrón de accesos complejo que, aunque se basa en el uso de strides, utiliza varios valores de stride que cambian a lo largo de la ejecución. Por tanto, este resultado anima a pensar que otras aplicaciones con patrones de acceso complejos pero con un relación mejor balanceada entre tiempo de cálculo y tiempo dedicado a accesos a disco, también pueden mejorar su rendimiento si se ejecutan usando el prefetch cooperativo

Por último, en la figura 6.23 mostramos los resultados de ejecutar el benchmark HeapSort. La cantidad de memoria física sobre la que se ha ejecutado este experimento es 64Mb. Recordemos que el comportamiento de este benchmark es prácticamente por completo aleatorio, ya que el número de ejecuciones de las instrucciones con acceso strided es muy bajo comparado con el número total de instrucciones ejecutadas. Por lo tanto,

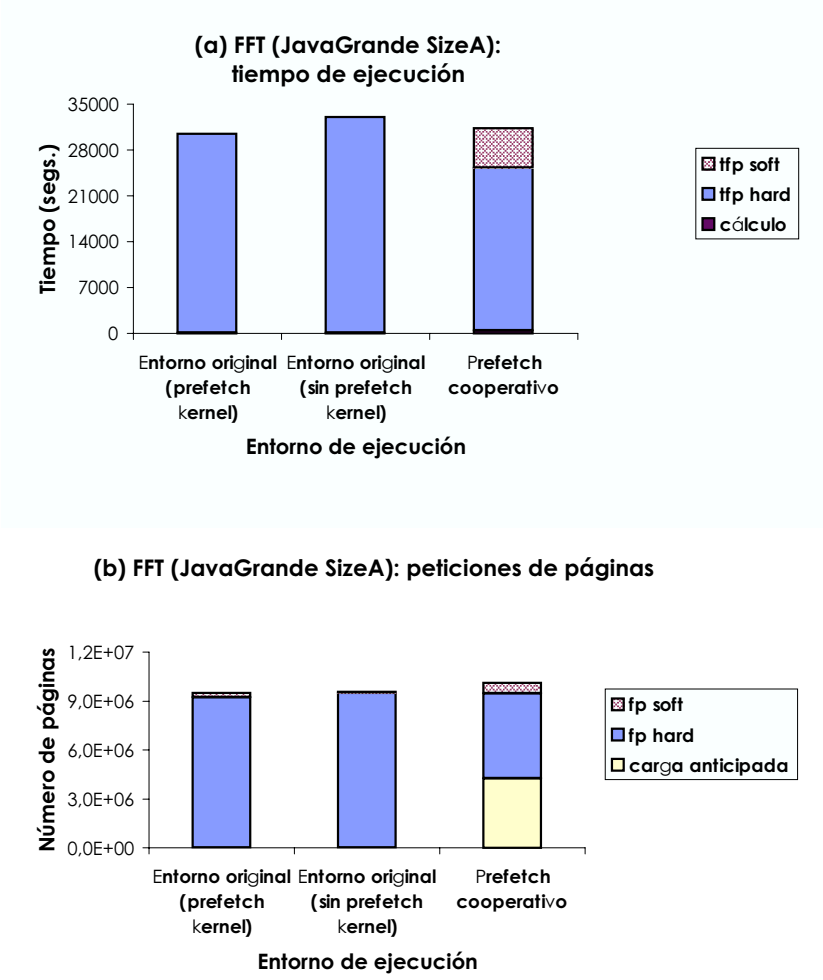

Figura 6.22 Resultados de FFT

como nuestra estrategia de prefetch detecta esta situación y se desactiva, el resultado de la ejecución es equivalente para ambos entornos de ejecución (ver figura 6.23).

Efectos de la automatización de la distancia

En la sección anterior hemos presentado la evaluación de la estrategia de prefetch, utilizando como distancia de prefetch para cada benchmark el mejor valor que hemos obtenido experimentalmente. En esta sección mostramos qué influencia tiene sobre el rendimiento

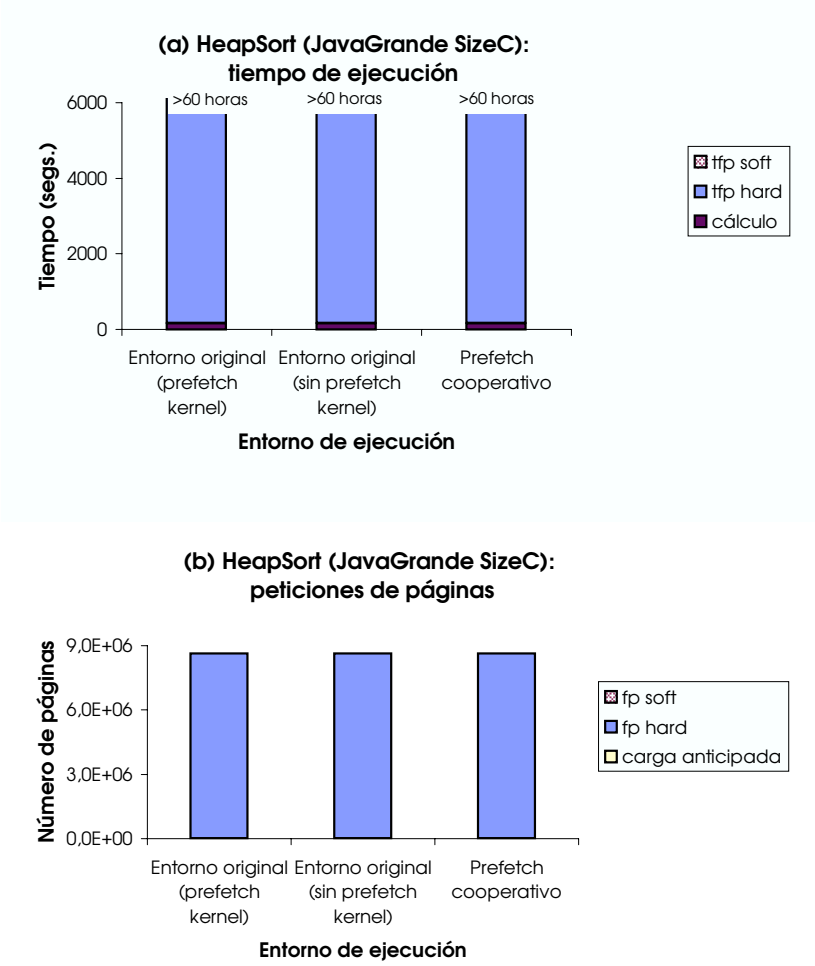

Figura 6.23 Resultados de HEAPSORT

de los programas la incorporación en la estrategia del cálculo automático de la distancia de prefetch.

Antes de iniciar el análisis es conveniente aclarar que para la implementación de la estrategia que utiliza la distancia de prefetch recibida como parámetro, este valor se aplica por igual a todas las instrucciones para las que se utiliza prefetch, sin tener en cuenta las posibles particularidades de cada una de ellas. Por lo tanto, aunque el valor que hemos seleccionado para cada programa es el que ofrecía mejores resultados no se garantiza

que este resultado no se pueda mejorar si se permite utilizar una distancia para cada instrucción, como ocurre en la estrategia con distancia automatizada.

En las figuras 6.24, 6.25 y 6.26 comparamos el rendimiento de los programas ejecutados sobre el entorno original, tanto con la configuración por defecto (*Entorno original (prefetch kernel)*) como con el prefetch de Linux desactivado (*Entorno original (sin prefetch kernel)*), y sobre la estrategia de prefetch cooperativo, cuando la distancia de prefetch es un parámetro de la ejecución (*Prefetch cooperativo (dist. manual)*)y cuando se calcula la distancia adecuada para cada instrucción (*Prefetch cooperativo (dist. aut.)*). Para cada uno de los benchmarks mostramos el tiempo de ejecución distinguiendo entre el tiempo dedicado a resolver fallos de página hard (*tfp hard*), fallos de página soft (*tfp soft*) y el resto del tiempo del programa (*cálculo*).

Mostramos los resultados de aquellos benchmarks que se pueden ver afectados por esta automatización del cálculo. Es decir, todos los programas excepto la MULTIPLICACIÓN DE MATRICES, que como utiliza el patrón simplificado basado en working sets las predicciones asociadas a sus instrucciones se refieren siempre al siguiente working set, HEAPSORT, que como utiliza un patrón de accesos aleatorio desactiva la estrategia de prefetch, y FFT, cuyo porcentaje de tiempo de cálculo no es suficiente para poder solapar la carga de sus páginas.

En la figura 6.24 mostramos los resultados de la ejecución de RHS. Podemos observar que la automatización del cálculo de la distancia aumenta el rendimiento de la estrategia de prefetch ya que es capaz de disminuir aún más el tiempo dedicado a resolver fallos de página hard. Esto se debe a la flexibilidad que da poder tener diferentes distancias para cada instrucción, ya que es posible adaptar ese valor a las necesidades del prefetch asociado a cada una. Así, el cálculo experimental para las distancias globales nos había llevado a seleccionar como parámetro una distancia de 32 páginas. Mientras que el cálculo automatizado selecciona diferentes distancias para diferentes instrucciones en un rango que va desde 16 páginas hasta 256. Como consecuencia, la estrategia de prefetch cooperativo con distancias calculadas automáticamente mejora el rendimiento del benchmark alrededor de un 43,03 % si se compara con su ejecución en el entorno original.

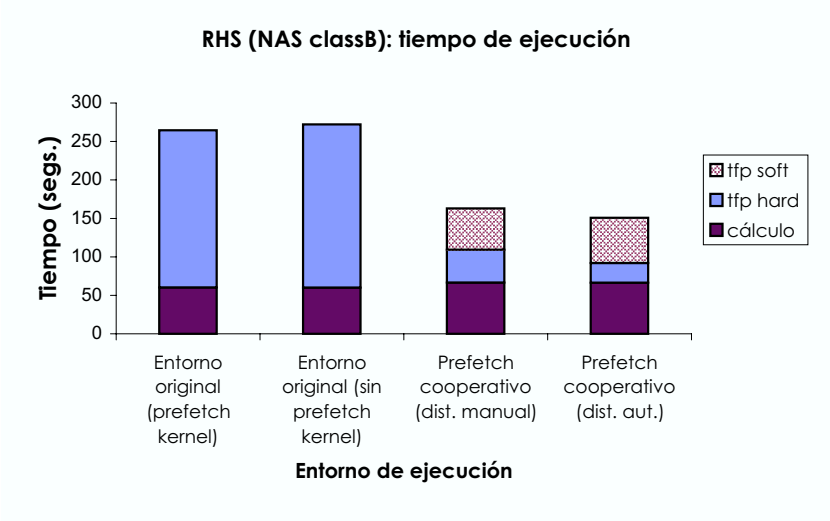

Figura 6.24 Influencia de la automatización de la distancia para RHS

Las figuras 6.25 y 6.26 muestran, respectivamente, los resultados para la ejecución de CRYPT y de SPARSE. Podemos comprobar que el efecto de incluir la automatización de la distancia es similar para los dos benchmarks. En ambos casos, la estrategia de prefetch sigue mejorando el rendimiento del entorno original de Linux, tanto si se utiliza la configuración por defecto como si se desactiva el prefetch de Linux. Así, la estrategia con distancia automática mejora el rendimiento obtenido en el entorno original alrededor de un 10,29 %, para el caso de CRYPT y alrededor de un 20,12 % para el caso de SPARSE. Sin embargo esta mejora es ligeramente inferior a la que obtenemos seleccionando la distancia experimentalmente. Si comparamos el resultado de las dos implementaciones de la estrategia de prefetch podemos ver que, para estos dos benchmarks, el uso de la distancia automática y a nivel de instrucción es capaz de reducir el tiempo dedicado a resolver fallos de página hard. Sin embargo, aumenta el tiempo necesario para resolver fallos de página soft, debidos a accesos a páginas en tránsito, ya que, las características de estos benchmarks, impiden que el sistema logre cargar a tiempo las páginas solicitadas.

Hay que remarcar que esta ligera disminución del rendimiento al automatizar la distancia no impide que la estrategia de prefetch cooperativo siga mejorando, de manera significativa, el rendimiento ofrecido por el entorno original de Linux. Además, esta mejora sigue

estando muy cerca del máximo beneficio teórico que una estrategia de prefetch puede ofrecer a unos programas con el comportamiento de CRYPT y SPARSE.

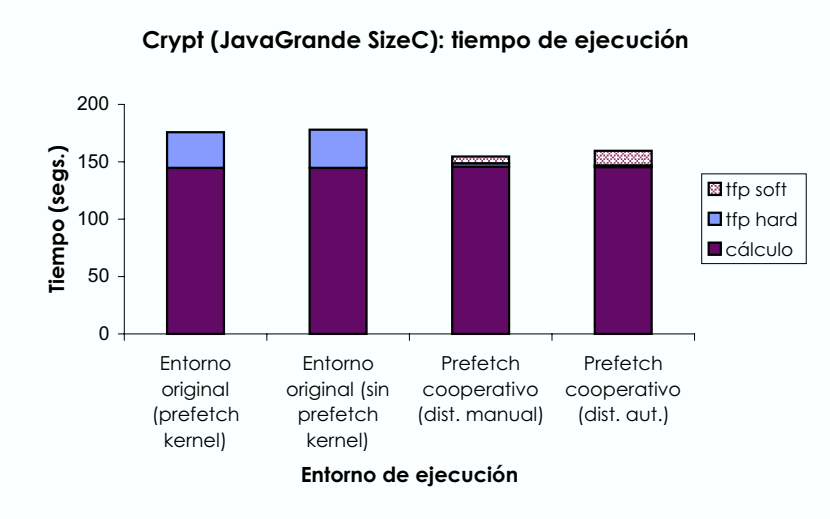

Figura 6.25 Influencia de la automatización de la distancia para CRYPT

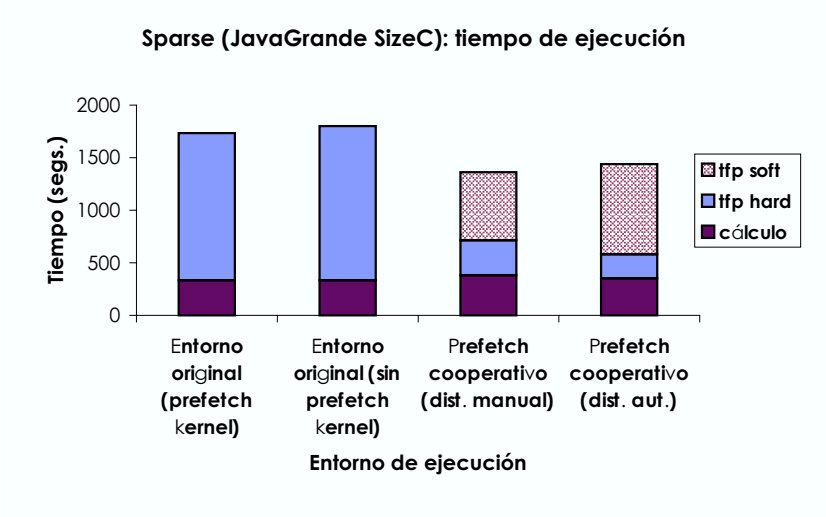

Figura 6.26 Influencia de la automatización de la distancia para SPARSE

Por lo tanto, estos experimentos demuestran que es posible que la JVM calcule, en tiempo de ejecución, la distancia de prefetch adecuada para las peticiones de cada instrucción, adaptando este valor a las condiciones de ejecución. Hay que destacar que, aunque en

algunos casos el rendimiento no alcance el obtenido por el valor calculado de forma experimental, la disminución del rendimiento es muy baja y está ampliamente compensada por los beneficios que supone para el usuario el cálculo automático de ese valor.

6.6.4 Conclusiones de la evaluación

En esta sección hemos presentado los resultados de la evaluación del prefetch cooperativo. En primer lugar, describimos los experimentos en los que hemos apoyado nuestras decisiones de diseño e implementación. En segundo lugar, comparamos la ejecución de algunas aplicaciones sobre el entorno original de ejecución y sobre el entorno modificado con nuestra estrategia de prefetch cooperativo.

El prefetch cooperativo ha demostrado ser una estrategia eficaz, capaz de mejorar el rendimiento que obtiene la ejecución de los benchmarks sobre el entorno original de ejecución, cuando las características de los programas los hacen susceptibles de ser mejorados a través del prefetch. Esta mejora se consigue por varios motivos:

- La JVM es capaz de captar los patrones de acceso de las instrucciones que tienen un comportamiento regular y usarlo para aplicar una predicción específica para cada una de ellas. De hecho, es capaz de captar patrones complejos, como lo demuestra la alta tasa de aciertos que se obtiene en la ejecución del programa FFT. Con esta predicción precisa se puede implementar un prefetch más eficaz que el resultante de la predicción simple que aplica el kernel de Linux en el entorno original, que sólo puede beneficiar a las aplicaciones que acceden de forma secuencial.
- La JVM también es capaz de detectar qué instrucciones no tienen un comportamiento predecible, y desactivar el prefetch únicamente para esas instrucciones. De esta manera se elimina la sobrecarga que se tendría al cargar en memoria páginas seleccionadas de forma equivocada, pero se permite que el resto de instrucciones del programa con patrón predecible se beneficien del uso del prefetch. Se puede comprobar los beneficios de este comportamiento en el rendimiento que obtiene el programa Sparse.
- El prefetch cooperativo es capaz de sacar más rendimiento del sistema en situaciones de alta presión. En estas situaciones la estrategia de prefetch del entorno original

simplemente descarta las peticiones de prefetch. En el caso del prefetch cooperativo también se cancelan para no perjudicar el rendimiento del programa, pero se aprovechan los momentos en los que esta carga baja para reintentar de nuevo las solicitudes de prefetch, consiguiendo completar con éxito un alto porcentaje de ellas. Por este motivo, incluso para aquellos programas que siguen el patrón de accesos secuencial como CRYPT, el prefetch cooperativo supera el rendimiento del prefetch secuencial que se intenta aplicar en el entorno original de Linux.

- Además, el incremento de tiempo de cálculo debido a la ejecución del prefetch cooperativo es asumible por las aplicaciones, dado el alto porcentaje de mejora que se obtiene con la carga anticipada de las páginas accedidas.

Hay que destacar que el porcentaje de la mejora obtenida para cada programa depende de la relación que haya entre tiempo de cálculo y tiempo dedicado a resolver fallos de página.

Si el tiempo de cálculo del proceso es muy bajo comparado con el tiempo de fallo de página, entonces se reducen las posibilidades de poder solapar el tiempo de carga anticipada con el tiempo de cálculo del programa. Por ejemplo, en el caso del benchmark SPARSE, aunque el prefetch cooperativo elimina casi el 97 % de sus fallos de página hard, la mejora obtenida en el rendimiento no pasa del 24,4 %, ya que el programa no dispone de más tiempo de cálculo para solapar las cargas anticipadas. El caso extremo de esta situación lo tenemos en la ejecución del programa FFT, que sólo dedica un 0,35 % de su tiempo a cálculo y por lo tanto no da opción a solapar el tiempo de acceso a disco. A pesar de ello, hay que decir que con este comportamiento del programa tan poco favorable para usar prefetch, la ejecución del programa sobre el entorno con prefetch cooperativo tiene un rendimiento equivalente al que se obtiene sobre el entorno original de ejecución.

Por otro lado, si el porcentaje de tiempo de fallo de página es mucho menor que el porcentaje de tiempo de cálculo, entonces, aunque la estrategia de prefetch consiga reducir gran parte del fallo de página la repercusión sobre el rendimiento total no tendrá un impacto equivalente. Es lo que sucede en el caso de la aplicación CRYPT que dedica sólo un 18 % de su tiempo a resolver fallos de página, por lo que la reducción del 73 % que la

estrategia de prefetch consigue sobre su tiempo de fallo de página se corresponde con un 13,11 % de mejora sobre el rendimiento global del programa.

De esta manera, en función de las características del programa, el uso de la estrategia de prefetch cooperativo consigue mejorar el rendimiento hasta un 40 %, para el caso de RHS, pasando por un 24,4 % para SPARSE, un 22 % para la MULTIPLICACIÓN DE MATRICES y un 13,11 % para CRYPT, y consigue no empeorarlo para aquellos programas cuyo rendimiento no es mejorable utilizando una técnica de prefetch de páginas como ocurre con FFT o HEAPSORT.

En cuanto a la automatización del cálculo de la distancia, hemos demostrado que es posible que la JVM realice este cálculo de forma automática, ajustándolo de acuerdo a los cambios en las condiciones de ejecución, con un coste asociado totalmente ignorable, si se compara con las ventajas que tiene para el usuario la automatización del cálculo de ese parámetro.

6.7 COOPERACIÓN ENTRE JVM Y EL SO: LA BASE PARA UNA ESTRATEGIA DE PREFETCH ESTABLE Y EFICAZ

En este capítulo hemos presentado el diseño y la implementación de la estrategia de prefetch cooperativo que hemos añadido al entorno de ejecución de Java.

Esta estrategia se basa en utilizar tanto el conocimiento de la JVM sobre el comportamiento del programa como el conocimiento del SO sobre las condiciones de ejecución del sistema, para tomar las decisiones de prefetch más adecuadas para mejorar el rendimiento del programa. La JVM selecciona las páginas que se deben cargar con antelación, usando la información que ella tiene sobre el programa y la que exporta el SO sobre el estado de la máquina, y solicita al SO la carga asíncrona esas páginas. El SO por su parte, efectúa las operaciones de carga, sólo si las condiciones de ejecución son favorables.

De esta manera, se consigue una técnica de prefetch que iguala las ventajas que ofrecía la estrategia transparente al SO (que describimos en el capítulo 5). Pero, además, consigue superar sus limitaciones involucrando al SO en la toma de decisiones, ya que constituye un mecanismo más estable y es capaz de explotar mejor los recursos del sistema en situaciones de alta presión.

Es decir, conseguimos una estrategia que cumple todos los requerimientos necesarios para que el prefetch sea eficaz:

- Predice de forma precisa las próximas referencias a memoria de las instrucciones.
- Respeta la portabilidad de los programas Java.
- Es transparente al programador y al usuario.
- Respeta la fiabilidad del sistema.

Pero además, con la participación del SO en las decisiones de prefetch:

- No es necesario utilizar heurísticas para aproximar el estado de la máquina, ya que el SO exporta al nivel de usuario la información sobre ese estado que permite optimizar la selección de páginas de prefetch.
- Se puede utilizar un interfaz dedicado para la solicitud de carga por prefetch, de manera que:
  - El SO puede gestionar estas solicitudes de forma diferente al resto de accesos a disco. Esto le permite, por ejemplo, descartar las operaciones de prefetch si el sistema de memoria está sobrecargado y, por lo tanto, completar esas operaciones puede empeorar el rendimiento del programa.
  - El SO puede controlar la interacción de las operaciones de carga anticipada con el resto de tareas de gestión de memoria implementadas por el SO.
  - La JVM puede agrupar varias solicitudes de prefetch, dando opción al SO a optimizar los accesos a disco involucrados.

  - la JVM puede reintentar las operaciones de prefetch canceladas, al detectar que las condiciones de ejecución son más favorables si todavía es posible mejorar el rendimiento de la instrucción mediante su carga anticipada.

- Se simplifica el código de prefetch ejecutado en el nivel de usuario, lo cual reduce el impacto que las decisiones de gestión del SO puede tener sobre el rendimiento de ese código. Recordemos que la estrategia de prefetch transparente al SO requiere un nuevo flujo de ejecución que se encarga de solicitar la carga anticipada. Por lo tanto, el rendimiento de la estrategia depende de las decisiones de gestión que el SO tome sobre la ejecución de ese flujo, que, además pueden depender de detalles de implementación de la versión de kernel instalada en la máquina.

Por lo tanto, la cooperación entre la JVM y el SO permite implementar un prefetch eficaz, mediante un mecanismo estable, capaz de adaptarse tanto a las características del programa como a las condiciones de ejecución para mejorar de una forma considerable el rendimiento de los programas.

7

TRABAJO RELACIONADO

En este capítulo presentamos el trabajo relacionado con los aspectos desarrollados en nuestro trabajo. Para ello distinguimos tres grandes grupos. El primer grupo es el relacionado con la evaluación del uso de la memoria que hacen los programas Java, primer paso necesario para hacer las propuestas de mejora de su gestión. El segundo grupo se refiere a los trabajos que intentan mejorar el rendimiento de la gestión de memoria de la plataforma de ejecución de Java. Por último, el tercer grupo engloba las estrategias orientadas a ofrecer a los programas una gestión de recursos específica para su comportamiento.

7.1 EVALUACIÓN DEL USO DE LA MEMORIA

Existen muchos trabajos que estudian cada una de las tareas de gestión de memoria por separado: memoria virtual, reserva de memoria lógica y liberación de memoria lógica. Sin embargo, la mayor parte de estos trabajos se han hecho para otros entornos de ejecución y no para Java. Hay que tener en cuenta que el rendimiento de la gestión de memoria depende en gran medida de los patrones de acceso de los programas, y estos patrones están muy relacionados con las características del lenguaje. Por este motivo, era necesario evaluar el rendimiento de estas tareas en el entorno de ejecución de Java.

A continuación describimos algunos de los trabajos que han analizado el comportamiento de los programas Java y las diferencias que existen entre estos trabajos y la evaluación que hemos presentado en este trabajo.

Kim y Hsu han estudiado la interacción que hay entre las memoria cache y los programas Java, cuando se ejecutan utilizando compilación en tiempo de ejecución (*Just In Time Compiler*) [KH00]. Además en este trabajo estudian la influencia que puede tener el tamaño inicial del heap sobre el rendimiento de la memoria cache, y mencionan que este tamaño puede influir también sobre el rendimiento de la memoria virtual.

Li et. al. también analizan la ejecución sobre compiladores al vuelo [LJNS01]. Su trabajo se centra en entender la actividad del SO durante la ejecución de programas Java. En esta actividad se incluye la gestión de la memoria virtual y, por lo tanto, también cuantifican el tiempo invertido en esta gestión. Sin embargo, no distinguen entre el tiempo debido al código del programa y el tiempo debido al código de gestión de la JVM.

El estudio de Dieckmann y Hölzle [DH99] estaba orientado a ayudar a los implementadores de garbage collectors, y mostraba el tipo de objetos que los programas Java reservan así como la distribución en el tiempo de esas reservas.

La evaluación que presentamos en este trabajo consiste en un análisis completo del rendimiento de la memoria virtual, inexistente hasta el momento. En este análisis aislamos la zona del espacio de direcciones más afectada por el uso de la memoria virtual y separamos la sobrecarga debida al código del programa de la debida al código de gestión de la JVM.

Además, los estudios previos sobre el comportamiento de los programas Java se han hecho sobre los programas del conjunto SPECjvm98 [SPE98]. Este conjunto de programas de pruebas, aunque realizan numerosos accesos a memoria y, por lo tanto, hacen un uso intensivo de memoria, tienen un working set pequeño, que requiere muy poca memoria física. Dieckmann y Hölzle mostraron que estos programas se pueden ejecutar sobre un heap de tamaño menor que 8Mb. Esto significa que durante su ejecución no necesitan la intervención de la gestión de memoria virtual y que el porcentaje de tiempo dedicado a resolver fallos de página es ignorable comparado con el tiempo de cálculo de los programas.

Sin embargo, los programas de prueba que hemos utilizado en la evaluación que presentamos son parte del conjunto de programas de prueba suministrado por el grupo de trabajo JavaGrande (ver sección 3.1.1 del capítulo 3). Estos programas tienen working

sets mayores que requieren de una mayor cantidad de memoria física y, por lo tanto, se ven afectados por la ejecución de la gestión de memoria virtual.

7.2 GESTIÓN DE MEMORIA EN JAVA

A medida que la popularidad de Java ha ido creciendo y se han diversificado los programas que utilizan su plataforma de ejecución, ha aumentado la preocupación por el rendimiento ofrecido por su gestión de memoria. Por este motivo, han aparecido numerosos estudios dedicados a mejorar este rendimiento.

Así, podemos encontrar propuestas para que la gestión del espacio lógico que implementa la JVM respete la localidad de referencia de los programas. Por ejemplo, hay propuestas de algoritmos de garbage collection que al gestionar los objetos supervivientes no alteran la localidad de los datos [Boe00, SGBS02]; también hay propuestas que durante la asignación de memoria a los objetos creados intentan tanto favorecer la localidad de referencia como disminuir el tiempo dedicado a cada ejecución del garbage collector [SGBS02, SGF+02, SZ98].

Fuera del ámbito de la gestión del espacio direcciones, también encontramos propuestas para mejorar el rendimiento de los accesos a objetos pequeños. Por ejemplo, Cahoon y McKinley [CM01, CM02] proponen añadir prefetch de cache al entorno de ejecución de Java. Este trabajo se centra en mejorar los accesos a pequeños arrays y a objetos enlazados como parte de una lista. Para ello proponen modificar el compilador de Java para insertar en el código generado las operaciones de prefetch de cache adecuadas. Por lo tanto, el análisis que proponen es estático, realizado en tiempo de compilación, y no pueden utilizar la información sobre el comportamiento dinámico de los programas. Hay que decir que también proponen simplificar las técnicas de análisis que se utilizan en compilación tradicional, para incorporarlas como parte del compilador al vuelo que puede utilizar Java en tiempo de ejecución. De esta manera, el análisis para insertar las operaciones de prefetch podría usar también la información sobre el comportamiento dinámico de los programas. Sin embargo, dejan para una futura evaluación la viabilidad de esta propuesta.

Hay que destacar que ninguno de estos trabajos han considerado las necesidades de los programas que trabajan sobre objetos de grandes dimensiones en uso durante toda la ejecución, para los que no influye la ejecución de las tareas de gestión del espacio lógico. Al contrario, todos los trabajos previos se centran en mejorar el rendimiento de programas que durante su ejecución realizan numerosas creaciones de objetos pequeños con un tiempo de vida corto.

Además, en ningún caso aprovechan el potencial que ofrece el conocimiento que tiene la JVM sobre el comportamiento dinámico de los programas para adaptar las decisiones de gestión a sus necesidades.

7.3 GESTIÓN ESPECÍFICA DE RECURSOS

Durante los últimos años han aparecido muchas propuestas que intentan conseguir una gestión de recursos específica para los programas. En esta gestión podemos distinguir dos tareas principales: la toma de decisiones y la realización de esas decisiones. Teniendo en cuenta estas dos tareas, podemos clasificar las estrategias propuestas en tres grupos diferentes.

En el primer grupo de propuestas, el sistema operativo se encarga tanto de las toma de decisiones como de su realización. En estos trabajos los sistemas operativos utilizan la información que tienen sobre el comportamiento de los programas, para intentar hacer predicciones sobre el comportamiento futuro, y, en función de esas predicciones gestionar los recursos. Debido a la poca información que tienen sobre el comportamiento de los programas, las predicciones que pueden hacer son, en general, poco precisas. Por ejemplo, Vitter y Krishnan han trabajado sobre algoritmos para *pure prefetching* [VK96, KV94], un modelo teórico de carga anticipada y asíncrona, que ignora la ejecución del algoritmo de reemplazo y que asume que no hay latencia de prefetch; y Vilayannur et al. [VSK05] han trabajado para predecir cuál es el momento adecuado para reemplazar una página determinada, para favorecer el rendimiento de las decisiones de reemplazo.

Todas estas estrategias se basan en determinar el patrón de accesos de los programas usando como datos de entrada la única información que tiene el sistema operativo sobre los accesos de los programas, esto es, los fallos de página. Por lo tanto, esta poca información limita la precisión de los algoritmos de predicción.

Fraser y Chang [FC03] proponen utilizar ejecución especulativa para anticipar los accesos a disco realizados por las aplicaciones. Esta estrategia implica una profunda y complicada modificación del sistema operativo y sólo puede ser efectiva en entornos en los que la CPU está ociosa durante un tiempo suficiente. La propuesta se basa en tener un flujo de ejecución *especulativo* que ejecuta por adelantado el código de la aplicación, aprovechando los instantes en los que la CPU pasaría a estar ociosa. El sistema operativo tiene que garantizar que el flujo especulativo no modifica los datos utilizados por el flujo de ejecución real, lo que implica el uso de la técnica de *copy-on-write*, aumentando la cantidad de memoria necesaria para la ejecución del programa. Este incremento puede penalizar el rendimiento de las aplicaciones que tengan un alto consumo de memoria, aunque las decisiones de prefetch sean precisas. Por este motivo, como parte de la estrategia, han implementado un módulo que evalúa los beneficios obtenidos de la ejecución especulativa, para poder desactivarla si estos beneficios son bajos.

En el otro extremo, tenemos el grupo de propuestas que permiten que el nivel de usuario suministre el código de gestión que contendrá la toma de decisiones y su realización. Esta alternativa ha sido estudiada profundamente a lo largo de los años [SS96]. Así, encontramos propuestas que van desde los *microkernels* iniciales [AR89, Jr,87], los sistemas operativos extensibles (como Vino [SESS96] o Spin [BSP+95]) y los exokernels [EKJ95]. Este grupo de propuestas se enfrentan al compromiso entre dos características necesarias y muchas veces opuestas: fiabilidad y eficiencia. Esto es así porque tienen que garantizar que el código que suministra el usuario, y que ejecuta las decisiones de gestión, no arriesga la integridad del sistema. Además tienen otra gran desventaja que es la necesidad de que el programador aporte el código de gestión que conviene utilizar para su programa. Esto implica que el programador debe, en el mejor de los casos, analizar el tipo de gestión que es conveniente para optimizar el rendimiento del programa lo cual no siempre es una tarea sencilla asumible por cualquier programador, y, en el peor de los casos, debe implementar también el código de esa gestión.

El tercer grupo de propuestas, se basan en la cooperación entre el nivel de usuario y el nivel de sistema para gestionar los recursos. En estas propuestas el nivel usuario se puede encargar de la toma de decisiones mientras que el sistema operativo es el encargado de realizarlas. La opción de pedirle a un usuario que modifique el código del programa para insertar las operaciones de gestión no es siempre una alternativa viable. Por este motivo se ha trabajado en la inserción automática de estas peticiones.

Hay muchos trabajos sobre compiladores para que este software, durante la generación del ejecutable, se encargue de analizar el código de los programas y de insertar las operaciones de gestión [BMK96, BM00]. Sin embargo, esta opción tiene varias limitaciones. La primera limitación es que la necesidad de que el usuario posea el código fuente de los programas, lo cual no siempre es posible. Otra limitación importante es que los compiladores sólo disponen de la información estática. Por lo tanto, no pueden tener en cuenta todas aquellas características de los programas que dependen de condiciones de ejecución. Además, la eficacia de algunas decisiones de gestión depende no sólo del programa, sino también de las características de la máquina y de sus condiciones en el momento de ejecutar el programa. Así, para poder tener en cuenta el estado de la máquina sería necesario recompilar el programa cada vez que se quiera ejecutar.

Algunos trabajos han intentado complementar el trabajo del compilador para que se pueda considerar al menos el estado de la máquina en tiempo de ejecución [CG99]. Estos trabajos añaden al entorno de ejecución una capa de software que, en tiempo de ejecución, filtran aquéllas operaciones insertadas por el compilador pero no adecuadas a las condiciones actuales de la máquina. Sin embargo esta capa no tiene acceso a las características dinámicas de los programas. En cualquier caso, para que el usuario pueda ejecutar el programa usando esta estrategia, es necesario poseer el código fuente del programa lo que, en general, no se puede suponer que suceda.

En este grupo de trabajos también se puede incluir el propuesto por Guitart et. al. [GMTA03]. Este trabajo se centra en mejorar el rendimiento de aplicaciones Java paralelas ejecutadas sobre multiprocesadores de memoria compartida. Para ello propone permitir la cooperación entre el nivel de usuario y el nivel de sistema para decidir el grado de paralelismo que debe explotar el programa para obtener un buen rendimiento. Sin em-

bargo, es responsabilidad del programador insertar en el código las operaciones adecuadas para informar al sistema de sus necesidades.

Nuestra propuesta también se basa en la cooperación entre el nivel usuario y el sistema operativo, sin embargo supera todas las limitaciones de la opción de los compiladores y es totalmente automática y transparente al programador y al usuario. Además, ni tan solo es necesario disponer del código fuente de los programas. En nuestra propuesta la JVM es la encargada de guiar las decisiones de gestión utilizando la información que tiene sobre el comportamiento dinámico de los programas, lo que permite adaptar automáticamente las decisiones de gestión a las necesidades del programa. El SO es el encargado de llevar a cabo estas decisiones sólo si las condiciones de ejecución son favorables, lo que garantiza la fiabilidad del sistema y facilita tener en cuenta el estado del sistema a la hora de tomar las decisiones de gestión.

8

CONCLUSIONES Y TRABAJO FUTURO

8.1 RESUMEN DEL TRABAJO

El objetivo principal de este trabajo ha sido demostrar que es posible mejorar el rendimiento de los entornos con ejecución basada en máquinas virtuales como, por ejemplo, el entorno de ejecución de Java o de C#, explotando las características propias del entorno para ofrecer una gestión de recursos más adecuada al comportamiento de los programas.

Una de las características más relevantes de este tipo de entornos de ejecución es la portabilidad de los programas. Sin embargo, el precio de esta portabilidad es una disminución del rendimiento con respecto al que se puede obtener en un entorno tradicional basado en la compilación. Esto es así ya que los programas se ejecutan sobre una máquina virtual que, en tiempo de ejecución, convierte el código independiente de la plataforma física en el correspondiente a la máquina real sobre la que se intenta ejecutar el programa. En este trabajo hemos propuesto aprovechar, precisamente, la ejecución basada en el uso de una máquina virtual para ofrecer a los programas una gestión de recursos específica para su comportamiento, mejorando de esta manera el rendimiento de los programas. Para demostrar los posibles beneficios de esta estrategia, nos hemos centrado en la gestión del recurso memoria y en el entorno de ejecución de Java.

De esta manera, hemos analizado el uso de la memoria que hacen los programas Java cuyo rendimiento viene limitado por la gestión de memoria. Los objetivos de este análisis, novedoso en el ámbito de ejecución de Java, han sido determinar las posibles vías para mejorar el rendimiento de la gestión de memoria del entorno de Java [BCGN03]. Para

ello hemos comprobado la influencia que tiene el uso de la memoria virtual, teniendo en cuenta tanto la zona del espacio de direcciones más afectada como la interacción con las tareas de gestión del espacio de direcciones que implementa la JVM.

Las conclusiones de esta evaluación del uso de la memoria, nos han llevado a determinar que una estrategia de prefetch de páginas eficaz puede mejorar el rendimiento de estos programas. Por este motivo, hemos analizado los requerimientos de las diferentes tareas de prefetch para poder asignar cada tarea al componente del entorno de ejecución más adecuado para realizarla. Las conclusiones de este análisis nos han llevado al siguiente reparto de tareas: la selección de páginas de prefetch debe ser guiada, en tiempo de ejecución, por la JVM, y la carga en memoria debe ser efectuada por el SO. De esta manera es posible dotar a la estrategia de prefetch de:

- Un algoritmo de predicción preciso implementado a nivel de instrucción, que tenga en cuenta la información que posee la JVM en tiempo de ejecución, sobre todos los accesos que realiza la instrucción y sobre las características específicas de la instrucción y del objeto que accede. Esta cantidad de información sobre el comportamiento dinámico de los programas supera con creces a la que tienen disponible tanto el SO como el compilador y, por lo tanto, la precisión de la predicción implementada por la JVM es mucho mayor que la puede ofrecer cualquier otro componente de los entornos tradicionales basados en compilación.
- Una selección de páginas capaz de adaptarse a las condiciones de ejecución presentes en cada momento, necesario, por ejemplo, para determinar la distancia de prefetch adecuada o para filtrar peticiones de prefetch redundantes. Esta capacidad de adaptación no existe si se implementa la selección de páginas con un análisis estático en tiempo de compilación.
- Una carga en memoria que respete la fiabilidad del sistema, ya que garantiza que el SO, único componente del entorno fiable, realiza los accesos al hardware involucrados en la operación de carga.
- Un mecanismo totalmente transparente al programador y al usuario, que respeta el paradigma de portabilidad de Java y que ni tan sólo requiere del usuario el código

fuente del programa, a diferencia de las estrategias basada en modificar el código generado por el compilador.

A partir de este primer análisis hemos contemplado dos posibles alternativas para la estrategia de prefetch, que hemos implementado y evaluado.

Una primera opción consiste en dotar al entorno de ejecución de Java de un mecanismo de prefetch totalmente transparente al SO [BGCN06b, BGCN06c]. Esta opción se basa en utilizar heurísticas para aproximar las condiciones de ejecución, necesarias para adaptar las decisiones de prefetch al estado en el que se encuentra en cada momento el sistema, y en solicitar la carga anticipada de las páginas mediante un mecanismo ya existente en el SO para otros propósitos (la excepción del fallo de página). La ventaja de esta opción es que favorece la portabilidad de la estrategia de prefetch ya que, al ser transparente al SO, no requiere modificar el sistema de las máquinas donde se quiera incorporar. La evaluación de esta alternativa ha ofrecido una mejora, con respecto al entorno original de ejecución, de hasta el 43 % para los benchmarks utilizados. Sin embargo, un análisis detallado nos ha llevado a descartarla por varios motivos. El principal motivo es que el grado de portabilidad de la estrategia no es tan alto como el que se esperaba ya que, aunque este método no requiere la modificación del SO, sí que es necesario analizar cuidadosamente la interacción entre el código de prefetch que hemos introducido y el resto de decisiones de gestión que sigue tomando el SO, ajeno a este nuevo nivel de gestión introducido. Además, el uso de heurísticas no ofrece las mismas garantías de éxito como poder consultar la información real, que se puede obtener con una mayor involucración del SO en las decisiones de prefetch.

La segunda opción, que es la que hemos adoptado como propuesta final para la estrategia de prefetch, consiste en implementar una estrategia cooperativa en la que tanto la JVM como el SO participen en las decisiones de prefetch [BGCN06a]. En esta opción, se substituye el uso de heurísticas por la consulta de la información real sobre el estado de la máquina, lo cual dota al sistema de una mayor fiabilidad. Además, la mayor participación del SO también permite obtener un mecanismo más estable, ya que el SO es capaz de controlar la interacción de las decisiones de prefetch con el resto de decisiones que toma en la gestión de los recursos de la máquina.

Para implementar esta estrategia, ha sido necesario modificar el SO con el interfaz necesario para implementar la cooperación entre la JVM y el SO que, básicamente, consiste en una zona de memoria compartida en la que el SO exporta al nivel de usuario la información sobre el estado de la memoria, y una nueva llamada a sistema que permite solicitar la carga asíncrona de una página que se encuentra en el área de swap.

Así pues, el papel de ambos componentes en la estrategia final de prefetch es el siguiente:

- La JVM utiliza su información sobre el comportamiento dinámico del programa para implementar la predicción a nivel de instrucción. Además utiliza la información que exporta el SO sobre el estado de la memoria para determinar la distancia de prefetch, filtrar las peticiones redundantes de prefetch, y determinar si las condiciones del sistema son favorables para solicitar nuevas cargas anticipadas, e incluso reintentar peticiones que fueron solicitadas en momentos poco adecuados, o si es más apropiado posponer la solicitud hasta que la carga del sistema sea menor.
- El SO efectúa los accesos a disco y a memoria física necesarios para realizar las cargas solicitadas por la JVM, cancela aquellas solicitudes que pueden perjudicar el rendimiento del programa dada la carga del sistema y exporta al nivel de usuario la información sobre el estado de la memoria para que la JVM la utilice durante la selección y solicitud de páginas de prefetch.

La evaluación de esta estrategia nos ha permitido comprobar que las mejoras en el rendimiento de los programas, con respecto a la ejecución en el entorno original, alcanzan hasta un 40 %. Hay que decir que hay programas cuyas características limitan el posible porcentaje de mejora del rendimiento que pueden obtener mediante una técnica de prefetch. Por ejemplo, si el programa no tiene un comportamiento predecible o si no tiene suficiente tiempo de cálculo para poder solapar la carga con el consumo de CPU. Sin embargo, en el peor de los casos, para los programas que no pueden ser mejorados mediante una estrategia de prefetch, la ejecución sobre el entorno modificado con nuestra estrategia ofrece un rendimiento similar a la ejecución sobre el entorno original.

Por lo tanto, la estrategia de prefetch cooperativo que hemos desarrollado en este trabajo constituye un ejemplo que cómo mejorar el rendimiento de los programas Java, mediante una política de gestión totalmente adaptada al comportamiento de cada programa, capaz de utilizar tanto el conocimiento que tiene la JVM sobre los programas, como el conocimiento que tiene el SO sobre las condiciones de ejecución.

8.2 CONTRIBUCIONES DE ESTE TRABAJO

Así pues, las principales contribuciones de esta tesis son:

- **Hemos demostrado que es posible mejorar el rendimiento de los programas con ejecución basada en la interpretación, sin renunciar a las características que hacen de estos entornos una opción atractiva para la ejecución de programas.**
- **Hemos demostrado que los entornos de ejecución basados en máquinas virtuales constituyen el escenario ideal para implementar una gestión de recursos adaptada al comportamiento de los programas, que supera las limitaciones presentes en los entornos tradicionales basados en compilación. Así, en estos entornos es posible ofrecer a los programas políticas de gestión de recursos específicas, que se adapten a su comportamiento dinámico y respeten la fiabilidad del sistema, de una forma totalmente transparente al programador y al usuario.**

Para llegar hasta estas aportaciones, hemos acotado nuestro caso de estudio a la gestión de memoria en el entorno de ejecución de Java, y hemos aportado las siguientes contribuciones aplicadas:

- Hemos evaluado de forma detallada el comportamiento de los programas Java cuya ejecución necesita el mecanismo de memoria virtual, que nos ha llevado a determinar que una política de prefetch eficaz puede mejorar su rendimiento, ya que:

  - El uso de la memoria virtual influye de forma significativa sobre los programas Java cuando la memoria física disponible no es suficiente para soportar el conjunto de datos del programa
  - La zona del espacio de direcciones que se ve más afectada es la zona donde se almacenan los objetos, de manera que es posible obviar la influencia que tiene el uso de la memoria virtual sobre los accesos al resto de zonas del espacio de direcciones.
  - Existen programas que apenas requieren la participación de las tareas de gestión del espacio de direcciones que implementa la JVM, por lo que mejorar la interacción entre la ejecución de estas tareas y la gestión de memoria implementada por el SO no es un camino factible para mejorar su rendimiento.
  - El tipo de objetos que provocan la necesidad del uso de la memoria virtual son arrays de grandes dimensiones, que ocupan la mayor parte del espacio de direcciones del programa. Además, estos objetos son accedidos siguiendo un patrón predecible, por lo que es posible anticipar cuáles serán los próximos accesos del programa e implementar su carga anticipada.
  - El rendimiento del sistema de memoria está muy lejos del rendimiento ofrecido por una gestión óptima de memoria, por lo que el margen de mejora es muy amplio.

- Hemos analizado la manera más adecuada para repartir las tareas de prefetch entre los diferentes componentes del entorno de ejecución para obtener una política eficiente, adaptada a las características de cada programa, fiable, que respete la portabilidad de los programas Java y que sea transparente al usuario y al programador.
- Hemos demostrado que es posible obtener una política de prefetch eficaz, totalmente dirigida por la JVM y sin involucrar al SO en la decisiones de gestión asociadas. Sin embargo, es necesario utilizar heurísticas para aproximar las condiciones de ejecución del sistema. Además, el mecanismo resultante es muy sensible a la implementación del resto de tareas de gestión del SO. Esto hace necesario un análisis detallado de la interacción del mecanismo de prefetch con la versión del SO instalada en cada plataforma donde se quiera implantar comprometiendo, de esta manera, la potencial portabilidad que puede tener una estrategia implementada por completo en el nivel de usuario.

- Hemos diseñado, implementado y evaluado una estrategia de prefetch basada en la cooperación entre la JVM y el SO. Esta política de gestión adapta sus decisiones al comportamiento de cada programa y a las condiciones de ejecución del sistema, y supera las limitaciones de las estrategias de prefetch propuestas en los entornos tradicionales basados en compilación.

8.3 TRABAJO FUTURO

Como trabajo futuro nos planteamos los siguientes puntos:

- Refinamiento de la estrategia de prefetch cooperativo, explorando los siguientes aspectos:
  - Cantidad de páginas solicitadas para cada predicción: en este trabajo hemos fijado el número máximo de peticiones en base a las posibles optimizaciones que puede hacer Linux en el acceso al área de swap. Como trabajo futuro nos planteamos analizar en profundidad la influencia que este parámetro puede tener sobre la estrategia de prefetch.
  - Gestión de las operaciones canceladas: en la implementación presentada en este trabajo, la gestión de las solicitudes de prefetch canceladas se hace mediante una política FIFO muy sencilla, que ha dado buen resultado para los programas que hemos evaluado. En versiones futuras de nuestra estrategia, queremos investigar la influencia que puede tener sobre el rendimiento una política más elaborada.
  - Automatización del cálculo de la distancia: en este trabajo hemos demostrado que la JVM es capaz de adaptar el valor de la distancia de prefetch a las condiciones de ejecución del sistema. Esta adaptación la hemos implementado mediante una heurística muy sencilla que incrementa el valor de la distancia si el actual no es suficiente para completar la carga anticipada. Como parte de nuestro trabajo futuro pretendemos evaluar posibles refinamientos sobre esta heurística como, por ejemplo, permitir que el valor de la distancia también pueda disminuir si la presión sobre el sistema de memoria disminuye durante la ejecución.

  - Desactivación del prefetch: en este trabajo hemos implementado la desactivación del prefetch cuando el comportamiento de las instrucciones no es predecible. También hemos implementado la cancelación de las operaciones de prefetch cuando las condiciones de ejecución desaconsejan llevarlas acabo. Sin embargo, queda pendiente desactivar la estrategia para los programas que no utilizan el mecanismo de memoria virtual y, que por lo tanto, verían ralentizada su ejecución por la estrategia de predicción y solicitud de páginas ya presentes en memoria física. La implementación de esta funcionalidad es muy sencilla, teniendo en cuenta que el SO puede informar a la JVM sobre el estado del sistema, usando la zona de memoria que comparten.
  - Nuevas funciones de predicción: en este trabajo nos hemos centrado en mejorar los accesos a objetos de tipo array de grandes dimensiones, ya que este tipo de objetos era el que provocaba el uso de la memoria virtual en los benchmarks con los que hemos trabajado. Como parte de nuestro trabajo futuro queremos incorporar nuevos algoritmos de predicción, capaces de gestionar el acceso a objetos de otro tipo.

- Evaluación de la estrategia de prefetch cooperativo en aplicaciones pertenecientes a otros ámbitos, ya que todos los benchmarks utilizados en este trabajo son de cálculo científico.

- Evaluación de la influencia que puede tener el uso de compiladores en tiempo de ejecución sobre el rendimiento del prefetch. Hay que decir que la estrategia propuesta es totalmente aplicable si se utiliza un compilador al vuelo, ya que es suficiente con modificar el compilador, para que incorpore en el código máquina que genera las instrucciones encargadas de seleccionar y solicitar las páginas de prefetch. Sin embargo, es necesario evaluar los posibles efectos laterales que esta compilación al vuelo pueda tener sobre el rendimiento de la estrategia de prefetch.

- Implementar la cooperación entre la JVM y el SO para ofrecer también una gestión específica para el uso de otros recursos como, por ejemplo, el procesador.

REFERENCIAS

[AES97] Anurag Acharya, Guy Edjlali, y Joel Saltz. The utility of exploiting idle workstations for parallel computation. En *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurements and Modeling of Computer Systems*, páginas 225–236, Seattle, WA, Junio 1997.

[AGH00] Ken Arnold, James Gosling, y David Holmes. *Java Programming Language, The 3rd Edition*. Addison Wesley Professional, 2000.

[AR89] Vadim Abroossimov y Marc Rozier. Generic virtual memory management for operating system kernels. *ACM, SIGOPS Operating Systems Review*, 23(5):123–136, Diciembre 1989.

[AW02] Tom Archer y Andrew Whitechapel. *Inside C#, 2nd edition*. Microsoft Press, 2002.

[BCGN03] Yolanda Becerra, Toni Cortes, Jordi Garcia, y Nacho Navarro. Evaluating the importance of virtual memory for java. En *Proceedings of the IEEE 2003 International Symposium on Performance Evaluation of Systems and Applications*, Austin, TX, Marzo 2003.

[BFH03] Fran Berman, Geoffrey Fox, y Tony Hey. *The Grid: past, present, future*, capítulo 1, páginas 9–50. Wiley and Sons, Ldt, Marzo 2003.

[BGCN06a] Yolanda Becerra, Jordi Garcia, Toni Cortes, y Nacho Navarro. Cooperative page prefetching: an example of os and vm cooperation. *Pendiente de revisión*, 2006.

[BGCN06b] Yolanda Becerra, Jordi Garcia, Toni Cortes, y Nacho Navarro. Java virtual machine: the key for accurated memory prefetching. En *Proceedings of the 2006 International Conference on Programming Languages and Compilers (pendiente de publicación)*, Las Vegas, NE, Junio 2006.

[BGCN06c] Yolanda Becerra, Jordi Garcia, Toni Cortes, y Nacho Navarro. Memory prefetching managed by the java virtual machine. Technical Report UPC-DAC-RR-2006-2, Computer Architecture Dept., UPC, Barcelona, España, 2006.

[BM00] Angela Demke Brown y Todd C. Mowry. Taming the memory hogs: Using compiler-inserted releases to manage physical memory intelligently. En *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, páginas 31–44, San Diego, CA, Octubre 2000.

[BMK96] Angela Demke Brown, Todd C. Mowry, y Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. En *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, páginas 3–17, Seattle, WA, Octubre 1996.

[Boe00] Hans-J. Boehm. Reducing garbage collector cache misses. Technical Report HPL-2000-99, Hewlett-Packard Laboratories, Palo Alto, CA, Julio 2000.

[BSP+95] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, y Susan Eggers. Extensibility, safety and performance in the spin operating system. En *Proceedings of 15th Symposium on Operating Systems Principles*, páginas 267–283, Saint-Malô, France, Diciembre 1995.

[BSW+00] J.M. Bull, L.A. Smith, M.D. Westhead, D.S. Henty, y R.A. Davey. A benchmark suite for high performance java. *Concurrency, Practice and Experience*, (12):21–56, 2000.

[CG99] Fay Chang y Garth A. Gibson. Automatic I/O hint generation through speculative execution. En *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, páginas 1–14, New Orleans, LA, Febrero 1999.

[CHL+00] Ralph Christ, Steven L. Halter, Kenton Lynne, Stephanie Meizer, Steven J. Munroe, y Mark Pasch. Sanfrancisco performance: A case study in performance of large-scale Java applications. *IBM Systems Journal*, 39(1):4–20, Febrero 2000.

[CKV93] Kenneth M. Curewitz, P. Krishnan, y Jeffrey S. Vitter. Practical prefetching via data compression. En *Proceedings 1993 ACM-SIGMOD Conference on Management of Data*, páginas 257–266, Washington, D.C., Mayo 1993.

[CM92] Henry Clark y Bruce McMillin. DAWGS - a distributed compute server utilizing idel workstations. *Journal of Parallel and Distributed Computing*, 14(2):175–186, Febrero 1992.

[CM01] Brendon Cahoon y Kathryn S. McKinley. Data flow analysis for software prefetching linked data structures in java. En *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, páginas 280–291, Barcelona, España, Septiembre 2001.

[CM02] Brendon Cahoon y Kathryn S. McKinley. Simple and effective array prefetching in java. En *Proceedings of the ACM Java Grande Conference*, páginas 86–95, Seattle, WA, Noviembre 2002.

[DH99] Sylvia Dieckmann y Urs Hölzle. A study of the allocation behavior of the SPECjvm98 java benchmarks. En *Proceedings of the 1999 European Conference on Object-Oriented Programming*, Lisboa, Portugal, Junio 1999.

[Ecm05] Ecma International. Common Language Infrastructure (CLI), common generics. Technical Report TR/89, Ecma International, Junio 2005.

[EKJ95] Dawson R. Engler, M. Frans Kaashoek, y James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. En *Proceedings of 15th Symposium on Operating Systems Principles*, páginas 251–266, Saint-Malô, France, Diciembre 1995.

[FC03] Keir Fraser y Fay Chang. Operating system I/O speculation: How two invocations are faster than one. En *Proceedings of the 2003 Usenix Annual Technical Conference*, páginas 325–328, San Antonio, TX, Junio 2003.

[FGT02] Geoffrey Fox, Dennnis Gannon, y Mary Thomas. Editorial: A summary of grid computing environments. *Concurrency and computation: practice and experience*, 14(13-15):1035–1044, Noviembre-Diciembre 2002.

[FSJY03] Michael Frumkin, Matthew Schultz, Haoquiang Jin, y Jerry Yan. Performance and scalability of the NAS parallel benchmarks in Java. En *Proceedings*

of the International Parallel and Distributed Processing Symposium, páginas 139–145, Niza, Francia, Abril 2003.

[GA94] James Griffioen y Randy Appleton. Reducing file system latency using a predictive approach. En *Proceedings of the 1994 Summer USENIX Conference*, páginas 197–207, Boston, MA, Junio 1994.

[GG97] José González y Antonio González. Speculative execution via address prediction and data prefetching. En *Proceedings of the ACM 1997 International Conference on Supercomputing*, páginas 196–203, Viena, Austria, Julio 1997.

[GM96] James Gosling y Henry McGilton. The Java language environment. A white paper. Technical report, Sun Microsystems, Inc., Mayo 1996.

[GMTA03] Jordi Guitart, Xavier Martorell, Jordi Torres, y Eduard Ayguadé. Application/kernel cooperation towards the efficient execution of shared-memory parallel java codes. En *Proceedings of the 17th IEEE International Parallel and Distributed Processing Symposium*, Niza, Francia, Abril 2003.

[JGF01] Java Grande Forum. `http://www.javagrande.org`, 2001.

[JL96a] Richard Jones y Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley and Sons Ldt., Septiembre 1996.

[JL96b] Richard Jones y Rafael Lins. *Mark-Sweep Garbage Collection*, capítulo 4, páginas 75–96. Wiley and Sons Ldt., Septiembre 1996.

[Jr.87] Avadis Tevanian Jr. Arquitecture-independent virtual memory management for parallel and distributed environments: The mach approach. Technical Report CMU-CS-88-106, PhD. Thesis, Computer Science Dept., Carnegie Mellon University, Pittsburg, PA, Diciembre 1987.

[KaMR02] Paul H.J. Kelly y Susanna Pelagatti annd Mark Rossiter. Instant-access cycle-stealing for parallel applications requiring interactive response. En *Proceedings of the 8th International Euro-Par Conference*, páginas 863–872, Paderborn, Germany, Agosto 2002.

[KFH+06] Gopi Kandaswamy, Liang Fang, Yi Huang, Satoshi Shirasuna, Suresh Marru, y Dennis Gannon. Building web services for scientific grid applications. *IBM journal of research and development*, 50(2-3):249–260, Marzo-Mayo 2006.

[KH00] Jin-Soo Kim y Yarsun Hsu. Memory system behavior of Java programs: Methodology and analysis. En *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*, páginas 264–274, Santa Clara, CA, Julio 2000.

[Kra96] Douglas Kramer. The Java platform. A white paper. Technical report, Sun Microsystems, Inc., Mayo 1996.

[KV94] P. Krishnan y Jeffrey S. Vitter. Optimal prediction for prefetching in the worst case. En *Proceedings fifth Annual SIAM/ACM Symposium on Discrete Algorithms*, páginas 392–401, Arlington, Virginia, Enero 1994.

[LJNS01] Tao Li, Lizy K. John, Vijaykrishnan Narayanan, y Anand Sivasubramaniam. *Characterizing Operating System Activity in SPECjvm98 Benchmarks*, capítulo 3, páginas 53–82. Kluwer Academic Publishers, 2001.

[LY99] Tim Lindholm y Frank Yellin. *The Java Virtual Machine Specification, second edition.* Addison Wesley, Abril 1999.

[MMG+00] José E. Moreira, Samuel P. Midkiff, Manish Gupta, Pedro V. Artigas, Marc Snir, y Richard D. Lawrence. Java programming for high-performance numerical computing. *IBM Systems Journal*, 39(1):21–56, Febrero 2000.

[MMGL99] José E. Moreira, Samuel P. Midkiff, Manish Gupta, y Richard D. Lawrence. High performance computing with the array package for Java: A case study using data mining. En *Proceedings of the Supercomputing '99 Conference*, Portland, OR, Noviembre 1999.

[PZ91] Mark Palmer y Stanley B. Zdonik. Fido: A cache that learns to fetch. En *Proceedings of the 17th International Conference on Very Large Data Base*, páginas 255–264, Barcelona, España, Septiembre 1991.

[Ric00] Jeffrey Richter. Microsoft .NET framework delivers the platform for an integrated, service-oriented web. *MSDN Magazine*, 15(9), Septiembre 2000.

[SESS96] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, y Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. En *Proceedings of the 1996 Symposium on Operating System Design and Implementation*, páginas 213–227, Seatle, WA, Octubre 1996.

[SGBS02] Yefim Shuf, Manish Gupta, Rajesh Bordawekar, y Jaswinder Pal Singh. Exploiting profilic types for memory management and optimizations. En *Proceedings of the ACM Symposium on Principles of Programming Languages 2002*, páginas 194–205, Portland, OR, Enero 2002.

[SGF+02] Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Apple, y Jaswinder Pal Singh. Creating and preserving locality of java applications at allocation and garbage collection times. En *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2002*, páginas 13–25, Seattle, WA, Noviembre 2002.

[SMM99] Darko Stefanović, Kathryn S. McKinley, y J. Eliot B. Moss. Age-based garbage collection. En *Proceedings of ACM 1999 SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, páginas 370–381, Denver, CO, Noviembre 1999.

[SPE98] SPEC JVM98. `http://www.spec.org/osg/jvm98`, 1998.

[SS96] Christopher Small y Margo I. Seltzer. A comparison of OS extension technologies. En *Proceedings of the 1996 Usenix Conference*, páginas 41–54, San Diego, CA, Enero 1996.

[Sun01] Sun Microsystems, Inc. The Java HotSpot Virtual Machine. Technical white paper, Abril 2001.

[SZ98] Matthew L. Seidl y Benjamin .G. Zorn. Segregating heap objects by reference behavior and lifetime. En *Proceedings of the 8th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, páginas 12–23, San Jose, CA, Octubre 1998.

[VK96] Jeffrey S. Vitter y P. Krishnan. Optimal prefetching via data compression. *Journal of the ACM*, 43(5):771–793, Septiembre 1996.

[VSK05] Murali Vilayannur, Anand Sivasubramaniam, y Mahmut Kandemir. Proactive page replacement for scientific applications: A characterization. En *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software*, páginas 248–257, Austin, TX, Marzo 2005.

[WLM92] Paul R. Wilson, Michael S. Lam, y Thomas G. Moher. Caching considerations for generational garbage collection. En *Proceedings of the ACM Conference on Lisp and Functional Programming*, páginas 32–42, San Francisco, CA, Junio 1992.