Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
(Mención Computación)

Gestión de caché SDRAM en una jerarquía no volátil RRAM

Autor:
D. Adrián Lamela Pérez

Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
(Mención Computación)

Gestión de caché SDRAM en una jerarquía no volátil RRAM

Autor:
D. Adrián Lamela Pérez

Tutor:
D. Benjamín Sahelices Fernández

Índice general

Índice de tablas

Índice de figuras

Agradecimientos

Me gustaría dedicar unas palabras de agradecimiento a aquellas personas y entidades que me han brindado su apoyo a lo largo de la realización de este Trabajo de Fin de Grado.

En primer lugar, al Grupo de Investigación Reconocido "Caracterización de Materiales y Dispositivos Electrónicos" de la Universidad de Valladolid, y especialmente a Benjamín Sahelices Fernañdez y Helena Castán Lanaspa, por abrirme las puertas del GIR y permitir una colaboración en una tarea investigadora que ha sido (casi) siempre de lo más placentera. Gracias también a Helena por su enorme preocupación e interés por mi carrera académica, pues he podido aprovechar oportunidades importantes gracias a la información que me ha proporcionado desde el principio.

A mi familia, con especial dedicatoria a mi madre, mi tía, y mi hermana. Mi madre, Beatriz, por apoyarme en esta etapa académica que llega a su fin y por reforzar mi libertad tanto en mis elecciones académicas como personales. A mi hermana, Patricia, por llenar de alegría los días que no tenían tanta; me esforzaré por seguir siendo el referente por el que me tomas. A mi tía, Virginia, por ofrecerse a arreglar conmigo cualquier problema que se presentara, por enseñarme la mejor manera de buscar soluciones, y por ayudarme a comprender que mis límites estaban mucho más allá de lo que podía imaginar. Gracias a todos por ese toque de cariño que todo el mundo debería tener.

A mis compañeros de clase y grandes amigos, Raúl e Irene. Habéis conseguido que el paso por la Universidad haya sido realmente divertido. Su cercanía, sincera amistad e inolvidables momentos han hecho que tanto el desarrollo de este TFG como toda la carrera haya sido de lo más llevadera, incluso en las épocas de exámenes. A Irene, por estar siempre dispuesta a prestar ayuda y compartir todo el material. A Raúl, por hacerme reír hasta en las clases más duras y enseñarme ese lado despreocupado tan necesario hoy en día. Os agradezco de corazón vuestro apoyo, y sabed que yo también estaré cuando me necesitéis.

Sobre todo, a Manuel Jiménez. Gracias por tu paciencia, tu comprensión, tu apoyo incondicional y tu ayuda sea cual sea el momento en el que la pida. Te agradezco que hayas sido una fuente de inspiración tan valiosa y toda la energía que me has transmitido. No olvidaré todas aquellas ocasiones en las que te has preocupado más por mi bienestar que por el tuyo propio.

También a Cristina Rueda Sabater, catedrática en el departamento de Estadística e Investigación Operativa. Sus ideas y correcciones sobre la parte más estadística de este trabajo han sido muy valiosas para mí. Gracias por dedicar una parte de tu tiempo en ello y en otros

tantos consejos y opiniones.

Por último, me gustaría dar las gracias a Benjamín Sahelices Fernández por acogerme bajo su tutela en este TFG, por el trato tan personal y amigable desde el comienzo, por la transparencia con la que habla y expone su punto de vista, por la ayuda y el tiempo dedicado cada vez que surgían dudas, por la alegría y pasión desprendida de esas reuniones semanales, y en definitva por todas las oportunidades que me han sido ofrecidas.

Abstract

The resistive switching phenomenon has been studied for many years, and it is a field of greatest interest in technology and science. Memories based on this phenomenon, known as Resistive RAM or RRAM, are quite promising to replace *Flash* memories and classical SRAM and DRAM. Some of their advantages are their density, speed, power consumption, and durability.

RRAM memories are non-volatile, which means no energy is needed to keep the information in a consistent state. Their density allows us to design one-terabyte capacity memories, and even more. The growing number of applications that demand large amounts of memory (for example, those related to *Big Data*) serves as motivation in this invetigation. The main drawback is that RRAM technology is ten times slower than current DRAMs. If we consider a traditional DRAM memory, faster but smaller, as a cache of an RRAM, we could provide faster access without renouncing the advantages of these resistive memories.

The main goal of this work is to provide an architecture where an SDRAM is configured as a cache over an RRAM. The system we shall design will also include a prefetching technique, that analyzes past memory access and tries to predict which lines will be used in the future, in order to shorten the global access time. Specifically, the prefetching technique is divided into two phases. The first one tries to pull apart memory zones attending their behavior, while the second one predicts the near-future of all these zones.

In Chapter 2 we discuss how a cache works, as well as the basic organization and operations of conventional DRAMs. In Chapter 3 we present the data that we are going to use in the rest of the work and the software we use to collect this information. In Chapter 4 we will use clustering algorithms to group applications attending their behavior, measured in terms of temporal locality and algorithmic locality. Finally, in Chapter 5, the prefetching system will be presented, which is based on hidden Markov and linear models.

Resumen

Aunque el fenómeno de la conmutación resistiva es conocido desde hace bastante tiempo, es de gran interés en la actualidad, tanto en el campo científico como tecnológico. Unas memorias basadas en este fenómeno, conocidas como Resistive RAM o RRAM, son unas de las más prometedoras para sustituir tanto a las memorias no volátiles *Flash* como a las volátiles SRAM y DRAM. En gran parte se debe a las buenas características que poseen en términos de densidad, velocidad, consumo energético y durabilidad.

Las memorias RRAM son no volátiles, por lo que no necesitan energía continua para mantener la información almacenada en un estado consistente, y pueden llegar a tener capacidad de terabytes. Poco a poco se están desarrollando aplicaciones más y más exigentes, especialmente con el auge del *Big Data*, que requerirán capacidades muy superiores a las que podemos encontrar hoy en día. El principal inconveniente en comparación con las memorias DRAM comercializadas actualmente es que son del orden de 10 veces más lentas. Parece buena idea considerar una memoria tradicional, más rápida pero más pequeña, como una caché sobre una RRAM más grande. Esto permitiría un aumento significativo de la velocidad de acceso sin renunciar a las ventajas de las memorias resistivas.

Lo mencionado establece el objetivo principal de este Trabajo Fin de Grado: determinar una pequeña arquitectura que permita configurar una memoria SDRAM sobre una memoria RRAM. El sistema que se pretende construir no sólo describirá la memoria SDRAM como caché de una RRAM, sino que el grueso del trabajo pasa por desarrollar una técnica de reconocimiento de patrones para predecir los accesos futuros y reducir el tiempo global del sistema. Concretamente, este sistema de prebúsqueda, aunque se detallará más adelante, pasa por distinguir patrones de comportamiento en los accesos, asociados con las zonas de memoria donde se producen, y examinar cada uno de ellos por separado.

En el Capítulo 2 se abordan temas relacionados con la tecnología de las memorias, necesario para comprender correctamente el resto de la memoria. En el Capítulo 3 se encuentra una descripción sobre cómo se obtienen los datos necesarios para el diseño de esta prebúsqueda. En el Capítulo 4 se emplean técnicas de clustering para agrupar diferentes aplicaciones de las que se tienen datos en función de la localidad temporal y algorítmica. Finalmente, en el Capítulo 5 se desarrolla un procedimiento de reconocimiento de patrones y un sistema de prebúsqueda basado en un modelo oculto de Markov y diversos modelos lineales.

Capítulo 1

Introducción y objetivos

El presente Trabajo Fin de Grado es un trabajo de Investigación de la rama de Arquitectura de Computadores, realizado en colaboración con el Grupo de Investigación Reconocido "Caracterización de Materiales y Dispositivos Electrónicos". Concretamente, se encuadra en una de las líneas de investigación más importantes de dicho GIR, dedicada a la caracterización de materiales para dispositivos de conmutación resistiva.

Aunque el fenómeno de la conmutación resistiva es conocido desde hace bastante tiempo, es de gran interés en la actualidad, tanto en el campo científico como tecnológico. Unas memorias basadas en este fenómeno, conocidas como Resistive RAM o RRAM, son unas de las más prometedoras para sustituir tanto a las memorias no volátiles *Flash* como a las volátiles SRAM y DRAM. En gran parte se debe a las buenas características que poseen en términos de densidad, velocidad, consumo energético y durabilidad [1].

Los fenómenos de conmutación resistiva han sido principalmente observados en diversas estructuras metal-óxido-metal (MIM) y metal-óxido-semiconductor (MIS). La causa de este fenómeno es la creación de un nano-filamento conductor que conecta dos electrodos. Estos filamentos pueden romperse y formarse de nuevo mediante la aplicación de un potencial externo. Existen entonces dos diferentes estados resistivos (baja y alta resistencia), y el dispositivo puede permanecer en el mismo estado durante un largo período de tiempo. Esto motiva una de las características principales de las RRAM: la no volatilidad.

Se plantean muchos interrogantes acerca del origen de los mecanismos de conducción y de la conmutación resistiva. Por consiguiente, antes de abordar el uso comercial de las RRAM, es necesario resolver algunos aspectos fundamentales, como la existencia de fluctuaciones en los diferentes estados resistivos, responsables de la variabilidad entre ciclos y entre dispositivos. Existen en la literatura numerosos artículos sobre memorias resistivas que intentan dar respuesta a estos y otros muchos interrogantes. Desde hace algún tiempo se ha venido estudiando el incremento y decremento gradual de la resistencia en los dispositivos RRAM cuando se aplican señales adecuadas [3]. Las investigaciones más importantes pasan por la demostración de que estos dispositivos se pueden comportar exactamente como elementos de una memoria analógica y

estudios exhaustivos sobre el fenómeno de conmutación resistiva en diferentes materiales.

Pocos investigadores han prestado atención a los cambios en la capacitancia durante la conmutación entre estados. Éstos y las variaciones en la admitancia proporcionan información importante para diferenciar los materiales más utilizados en la construcción de estos dispositivos. En [3] se presenta un análisis completo de los estados intermedios que pueden sufrir las memorias resistivas, en función de la admitancia. En [4] se estudian las diferentes formas de controlar la conductancia en estados intermedios de las RRAM. En [2] se presenta un estudio de las estructuras MIM y los cambios que sufren entre los dos estados en términos de la conductancia y susceptancia.

Además de estos problemas que necesitan ser resueltos antes de producirse el salto a las memorias no volátiles, existe otro pequeño inconveniente. Aunque las RRAM tienen una mayor densidad de almacenamiento y puedan alcanzar tamaños mucho mayores que las actuales DRAM del mercado, son del orden de diez veces más lentas. Por ello, este trabajo de investigación se centra en cómo podemos combinar las memorias resistivas con otras memorias tradicionales, más pequeñas pero más rápidas, para acelerar el sistema de memoria en general.

Las memorias RRAM pueden llegar a tener capacidad de terabytes. Poco a poco se están desarrollando aplicaciones más y más exigentes, especialmente con el auge del *Big Data*, que requerirán capacidades muy superiores a las que podemos encontrar hoy en día. Parece buena idea considerar una memoria tradicional, más rápida pero más pequeña, como una caché sobre una RRAM más grande. Esto permitiría conservar las grandes ventajas de las RRAM, como su gran capacidad y la no volatilidad, y simultáneamente aumentar las velocidades de acceso. Dada la capacidad de las RRAM, una memoria SDRAM de unos 16GB (un tamaño de lo más popular en los ordenadores personales de hoy en día) puede configurarse sobre una RRAM, e ir almacenando los accesos más frecuentes, de forma que no sea necesario esperar a que la RRAM proporcione la información. De la misma forma que los procesadores disponen de un sistema de caché sobre las memorias RAM, ésto permitiría un aumento significativo de la velocidad de acceso sin renunciar a las ventajas de las memorias resistivas.

La configuración de las memorias caché se aprovecha de dos fenómenos muy conocidos: la localidad temporal y la localidad espacial. Por un lado, la primera nos dice que, una vez accedida una porción de memoria, es muy probable acceder de nuevo a ella en un corto período de tiempo. Por ello, el almacenamiento de esta información en una caché más rápida permite el acceso a información repetida de forma eficiente. Además, también existe la localidad espacial, que afirma que la probabilidad de acceder a posiciones de memoria cercanas a corto plazo es muy alta. Ésta es bastante más complicada de analizar y medir que la anterior, puesto que no hay que prestar atención a lo que ocurre en la misma porción, sino a las zonas cercanas anteriores y posteriores a ella.

El aprovechamiento de la localidad temporal pasa simplemente por el almacenamiento de la información en una caché de forma temporal, hasta que haya pasado suficiente tiempo o abandone su posición en favor de otras porciones más recientes. La localidad espacial se puede aprovechar de varias formas. Por un lado, muchos sistemas no sólo traen una porción de memoria a la caché

cuando es requerida, sino que también actúa sobre las adyacentes. Otros sistemas más sofisticados pasan por la implementación de un pequeño sistema de prebúsqueda que analiza los accesos a memoria e intenta predecir cuáles serán los siguientes.

La gran mayoría de los sistemas de prebúsqueda recientes son algo básicos, priorizando la facilidad de implementación y cálculos sencillos [5]. Las cachés más utilizadas están en un nivel muy cercano al procesador, la demanda de información es extremadamente alta, y no pueden permitirse cálculos y técnicas complejas, puesto que el tiempo entre dos accesos es muy limitado. Sin embargo, esta situación no se asemeja a la jerarquía que se propone en este trabajo. Al trabajar al nivel de memorias RAM, muchos accesos quedan enmascarados por los aciertos en las cachés superiores, por lo que el examen del patrón de accesos es más complicado al disponer sólo de información parcial. Como ventaja se tiene que el tiempo disponible para procesar la información y llevar a cabo la prebúsqueda es mucho mayor. Por ello, podemos considerar técnicas más sofisticadas que las actuales, con más cálculos y operaciones, a cambio de más precisión.

Lo mencionado establece el **objetivo principal** de este Trabajo Fin de Grado: determinar una pequeña arquitectura que permita configurar una memoria SDRAM sobre una memoria RRAM. El sistema que se pretende construir no sólo describirá la memoria SDRAM como caché de una RRAM, sino que pasa por el diseño de un mecanismo de prebúsqueda para predecir los accesos futuros y reducir el tiempo global del sistema. Concretamente, este sistema de prebúsqueda, aunque se detallará más adelante, pasa por distinguir patrones de comportamiento en los accesos, asociados con las zonas de memoria donde se producen, y examinar cada uno de ellos por separado.

En el Capítulo 2 se abordan temas relacionados con la tecnología de las memorias, necesaria para comprender correctamente el resto del trabajo. Existe una sección dedicada al estudio sobre la organización de las memorias caché. Aunque generalmente se piensa en una caché como aquella memoria muy rápida y cercana al procesador, la realidad es que una memoria caché es cualquier memoria que actúa sobre otra de mayor tamaño, almacenando la información más relevante, para evitar los accesos en la situada en el nivel inferior. Examinaremos, además de la organización lógica y los tipos de cachés que hay, cuáles son las políticas de reemplazo que determinan las porciones de memoria desechadas cuando ésta se encuentra llena. A continuación se proporciona una visión detallada de las memorias DRAM (*Dynamic Random Access Memory*), pasando por su organización lógica, el funcionamiento de las peticiones de acceso, los diferentes módulos de memorias DRAM existentes, la topología habitual, el protocolo de acceso y el controlador de memoria.

En el Capítulo 3 se aborda la descripción de la obtención de los datos. En nuestro caso, utilizaremos una herramienta de Intel® para monitorizar el comportamiento real de varias aplicaciones atendiendo a los accesos a memoria, llamada Intel® Pin. Esto nos permitirá obtener un fichero con información sobre las peticiones al sistema de memoria RAM, que contiene los datos sobre el momento en el que se ha producido dicha petición, la línea de memoria a la que hace referencia y el tipo de petición (lectura o escritura). Con ello podremos realizar diferentes simulaciones y probar el mecanismo de prebúsqueda sobre la arquitectura de memoria deseada.

Nótese que la información de los accesos a memoria fuera de un chip comercial cualquiera (que contiene el procesador y el sistema de caché) es suficiente para nuestro propósito.

Una vez obtenidos los datos, y de forma previa al estudio del mecanismo de prebúsqueda y la jerarquía SDRAM-RRAM, se presenta una clasificación exhaustiva de las aplicaciones involucradas que formarán parte del análisis. En el Capítulo 4 se realiza un estudio sobre la localidad temporal y un nuevo tipo de localidad no introducida hasta ahora, la localidad algorítmica. Ésta última hace referencia a la existencia de patrones en los accesos a memoria debido a la naturaleza propia del algoritmo o procedimiento implementado. Utilizando técnicas de clustering, agruparemos a las diferentes aplicaciones en función de su comportamiento. Por un lado, interesa determinar qué aplicaciones tienen una localidad temporal muy marcada y cuáles no suelen repetir accesos, pues puede ser relevante de cara al diseño del mecanismo de prebúsqueda. Aquellas aplicaciones con una fuerte localidad temporal pueden no necesitar tanto una prebúsqueda, siempre que dicha localidad sea suficiente para obtener una tasa de éxitos aceptable. Además, el estudio de la utilización de memoria y de la localidad algorítmica nos permite conocer qué aplicaciones tienen una utilización alta de la memoria y cuáles no. En éste último caso dispondremos de más tiempo para realizar los cálculos derivados de la prebúsqueda. El estudio completo de este Capítulo requiere adaptar el fichero de datos, para lo que han utilizado varios scripts escritos en Python. Para el agrupamiento y clustering se ha utilizado R.

Finalmente, en el Capítulo 5 se proponen tres formas para el reconocimiento de patrones de los accesos a memoria. El objetivo es diferenciar zonas en la memoria virtual, de forma que los accesos cercanos que compartan una cierta tendencia se consideren dentro de un determinado grupo. Una vez que se consiga esta separación en el comportamiento, será más sencillo diseñar la prebúsqueda, pues ésta trabajará sobre zonas concretas que tengan ciertas cosas en común en su comportamiento. De entre las tres formas que se exponen, finalmente se considerará un modelo oculto de Markov para explicar esta diferencia. Con ello, se diseñará el mecanismo de prebúsqueda y se probará sobre las aplicaciones reales consideradas, a partir de los datos obtenidos en el Capítulo 3, mediante un script de R.

1.1. Planificación del trabajo

En esta breve sección se llevará a cabo una planificación sencilla de las tareas involucradas en el desarrollo del trabajo. La planificación no es exhaustiva ni detallada debido a la naturaleza de la propia investigación. En las reuniones semanales con el tutor se discuten los avances y resultados producidos a lo largo de la semana anterior, y se detallan las nuevas tareas para la próxima semana.

El tiempo invertido en este TFG se estima en 300 horas, correspondientes a los 12 ECTS que deben ser dedicados. El ritmo de trabajo es de 15 horas semanales durante 20 semanas. La siguiente lista de tareas (desarrollada a posteriori) indica cómo se ha repartido la carga de trabajo a lo largo del tiempo. Todas las tareas han sido realizadas de forma secuencial.

1. Estado del arte: estudio y lectura de los diferentes libros y artículos involucrados en la investigación, así como un resumen de los aspectos fundamentales, que formará el Capítulo 2 de esta memoria. Dedicación: 2 ECTS (50 horas).

2. Obtención de datos de aplicaciones reales sobre accesos a memoria. Muchos de los datos ya habían sido obtenidos con anterioridad por parte del GIR, por lo que en su mayor parte, el tiempo dedicado a esta sección ha sido a la adecuación de los ficheros de datos para los propósitos que nos interesan. En menor medida, también se ha invertido algo de tiempo en la obtención de algunos ficheros de datos nuevos. Dedicación: 0,5 ECTS (12,5 horas).

3. Caracterización de aplicaciones: estudio y agrupamiento de diferentes aplicaciones en función de su comportamiento. En esta sección distinguimos dos tipos de agrupaciones:

   a) Agrupación atendiendo a la localidad temporal:

   - Extracción de características medibles para realizar el agrupamiento. Dedicación: 0,5 ECTS (12,5 horas).
   - Desarrollo de la técnica para el agrupamiento. Dedicación: 1 ECTS (25 horas).
   - Obtención de resultados y escritura de los mismos en la memoria. Dedicación: 0,5 ECTS (12,5 horas).

   b) Agrupación atendiendo a la localidad algorítmica:

   - Extracción de características medibles para realizar el agrupamiento. Dedicación: 0,5 ECTS (12,5 horas).
   - Desarrollo de la técnica para el agrupamiento. Dedicación: 1 ECTS (25 horas).
   - Obtención de resultados y escritura de los mismos en la memoria. Dedicación: 0,5 ECTS (12,5 horas).

   Dedicación total: 4 ECTS (100 horas).

4. Estudio de la localidad espacial y desarrollo del sistema de prebúsqueda. Esta parte incluye:

   a) Estudio del modelo binomial para el cacheo de bloques de memoria. Dedicación: 0,5 ECTS (12,5 horas).

   b) Estudio de la información inmediata anterior para decidir el cacheo de bloques de memoria. Dedicación: 0,5 ECTS (12,5 horas).

   c) Desarrollo del modelo Oculto de Markov e implementación del mismo. Dedicación: 1 ECTS (25 horas).

   d) Desarrollo de un sistema de prebúsqueda sobre el sistema de reconocimiento de Markov. Dedicación: 1 ECTS (25 horas).

   e) Desarrollo e implementación de un simulador de caché para probar el sistema de prebúsqueda. Dedicación: 0,5 ECTS (12,5 horas).

f) Obtención de resultados y escritura en el informe. Dedicación: 1 ECTS (25 horas).

Dedicación total: 4,5 ECTS (112,5 horas).

5. Tareas finales de escritura y depuración de la memoria. Dedicación: 1 ECTS (25 horas).

Capítulo 2

Contexto tecnológico

Los sistemas de memoria actuales requieren de diseños de jerarquías de memoria eficientes para poder funcionar respetando unas especificaciones mínimas en cuanto a precio y velocidad, además de proporcionar un sistema de almacenamiento no volátil. Todos los procesadores necesitan un sistema de memoria de acceso aleatorio para poder soportar el modelo de Von Neumann, donde tanto los datos como las instrucciones residen en memoria, y aquellas porciones que contienen datos relacionados no tienen por qué estar juntas [6].

Un supuesto ideal donde se tuviera una sola memoria para satisfacer todos estos requisitos sería enormemente caro y prácticamente imposible de construir, pues muchas de las características requeridas son contrarias. No es posible tener una única memoria no volátil y suficientemente grande como para satisfacer las necesidades de computación, esperando que a su vez sea tan rápida como para que el procesador actúe directamente sobre ella sin ser un cuello de botella.

Para conseguir un sistema equilibrado, es necesario agrupar varios tipos de memorias en una jerarquía, que pueda proporcionar todos los requerimientos a un precio asequible, donde cada nivel esté optimizado para una cierta tarea. La mayoría de sistemas tienen más de tres niveles, aunque a grandes rasgos se pueden distinguir los siguientes.

- En el nivel más cercano al procesador (excluyendo sus propios registros) se tiene una memoria **caché**, y proporciona un acceso muy rápido a los datos (incluso menor que un nanosegundo), además de tener un consumo energético muy bajo. Por contra, son muy caras, por lo que no pueden ser muy grandes (del orden de KiloBytes) [7].

- Después se situaría una **memoria DRAM** (Dynamic Random Access Memory), que proporciona un almacenamiento relativamente grande y barato al compararlo con una caché, y relativamente rápido con respecto a un disco.

- Como último nivel se necesita un **almacenamiento no volátil** para almacenar todos los ficheros del usuario, así como el Sistema Operativo, la configuración de arranque, y demás archivos necesarios para el funcionamiento del sistema. Es muy grande comparado con una

caché, pues actualmente se barajan almacenamientos de TeraBytes para uso personal, pero también increíblemente lento, pues el acceso es del orden de milisegundos.

Generalmente los sistemas tienen varios niveles de memorias caché, por lo que suele haber más de tres niveles en una máquina real. Aún así, independientemente del número de componentes de la jerarquía, cada uno se puede encuadrar en alguno de los tres tipos especificados arriba en función de sus características de velocidad, consumo energético y capacidad.

El acceso se realiza por niveles, de forma que cuando el procesador lanza una petición para leer unos datos, en primer lugar se comprueba si están en la caché. En caso afirmativo se tiene un acierto de caché, y se pueden elevar los datos inmediatamente al procesador. Si por el contrario no están, hay que pedir los datos al nivel inferior (fallo de caché). La petición se va encadenando sucesivamente hasta que finalmente se encuentran los datos, momento en el cual se elevan esos mismos (y generalmente también los que se encuentran *alrededor*) hacia el nivel más alto donde el procesador puede leerlo directamente. En la figura 2.1 se encuentra un diagrama que resume los niveles de una jerarquía de memoria típica.

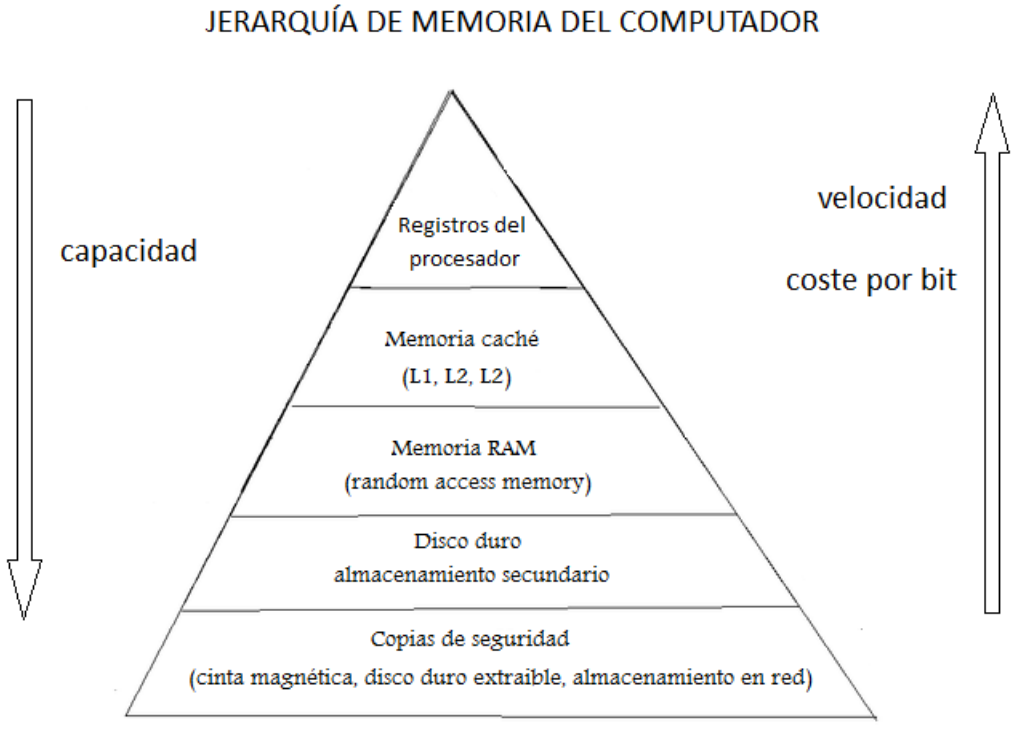


Figura 2.1: Diagrama piramidal de una jerarquía de memoria.

Este sistema funciona debido a los principios de *localidad*. Aunque volveremos sobre ello más adelante, a grandes rasgos se ha comprobado experimentalmente que:

- Es muy probable volver a utilizar una *misma* porción de memoria en un corto espacio de tiempo. A esto se le llama **localidad temporal**.

- Es muy probable utilizar porciones *cercanas* de memoria en un corto espacio de tiempo, lo que se denomina **localidad espacial**.

Gracias a ello, se pueden diseñar sistemas donde no sea necesario acceder al último nivel de memoria para responder a una petición de lectura; los datos pueden encontrarse ya en la memoria caché si se han utilizado hace poco o si se ha utilizado una porción cercana (de ahí lo de elevar también los datos que se encuentran alrededor) y el tiempo de acceso disminuye significativamente. Esto complica un poco el esquema, y se hace necesario definir políticas de reemplazo de bloques de memoria, así como el mapeo de direcciones de memoria más grandes en un dispositivo más pequeño.

Una jerarquía de memoria bien implementada es barata, proporciona accesos rápidos al procesador, y no consume demasiada energía, además del requisito indispensable de tener una parte no volátil. Desde el punto de vista del procesador, se puede acceder a grandes cantidades de datos, pero el acceso es en término medio mucho más rápido que leer siempre de un disco duro no volátil.

2.1. Memorias caché

Una memoria caché no incluye en su definición cómo debe ser implementada. Se puede utilizar una pequeña memoria como caché de la memoria principal DRAM, pero una DRAM más rápida puede actuar como caché de una DRAM más lenta. De hecho, la propia motivación de este trabajo es determinar las mejores formas de utilizar una SDRAM (Synchronous Dynamic Random Access Memory) como caché de una RRAM (Resistive Random Access Memory). Ambas serán detalladas más adelante. El objetivo principal de una caché es predecir el comportamiento futuro de la aplicación, para reducir la carga de las operaciones sobre niveles inferiores en la jerarquía de memoria.

Las cachés se pueden clasificar en tres tipos:

- Una caché *transparente* tiene definidas ciertas heurísticas para decidir qué datos retienen, cómo los sustituyen (políticas de reemplazo), y cuál es la asociatividad que determina las porciones de memoria donde se mapean los datos. Opera independientemente de las aplicaciones sobre las que se ejecuta. Nótese que si se quiere reemplazar una porción de memoria que ha cambiado no basta con eliminarla, sino que es necesario hacer permanentes esas modificaciones en el nivel inferior.

- Una caché *manejada por la propia aplicación* no define estas heurísticas, pues el propio cliente decide sobre las operaciones mencionadas anteriormente.

- Por supuesto, existen esquemas híbridos entre los dos.

Para cualquier caché, es necesario determinar las características en cuanto a tres dimensiones ortogonales:

- La **organización** de la caché: la estructura lógica que define cómo se almacenan los datos.
- Las **heurísticas de almacenamiento** para decidir si se retiene o no un elemento en un determinado momento.
- Las **heurísticas de consistencia** que garanticen que los datos que se utilizan son los válidos en cada momento.

Además, como se ha indicado anteriormente, lo normal es que haya varios niveles de memorias caché. En la figura 2.2 se muestra la jerarquía de memorias caché para la microarquitectura de Intel Nehalem, utilizada en la primera generación de los procesadores Intel Core i5 e i7 [8]. Puede comprobarse que se utiliza hasta una memoria caché de nivel 3 antes de llegar al nivel de la memoria principal.

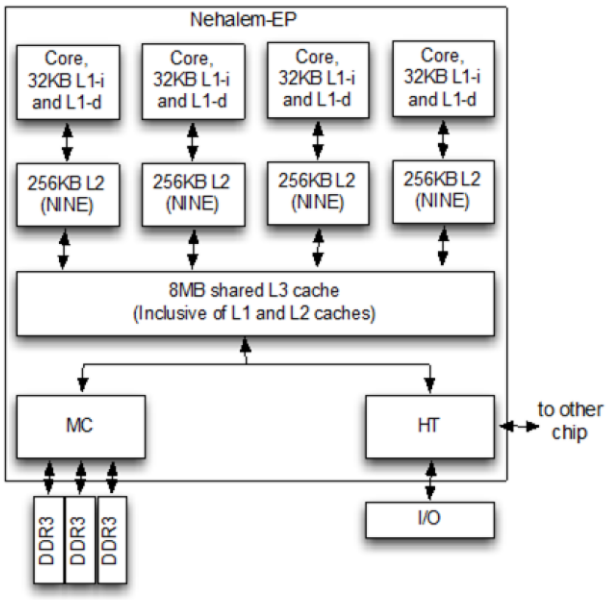


Figura 2.2: Organización de la caché de la microarquitectura de Intel Nehalem.

2.1.1. Principios de localidad

Las memorias caché funcionan debido a los principios de **localidad de referencia**. Los procesos tienden a exhibir un comportamiento predecible en cuanto a los accesos a memoria, de lo cual podemos aprovecharnos para el diseño de las heurísticas de almacenamiento de información en una caché más pequeña que la memoria a la que sirve. Es conveniente destacar que los fenómenos de localidad no están garantizados; tan sólo son comportamientos que tienden a tener la mayoría de los procesos. Existen fundamentalmente tres principios de localidad:

- **Localidad temporal**: es la tendencia de los programas a utilizar los mismos datos en períodos próximos de tiempo, lo que puede proporcionar una evidente heurística para la gestión de los datos de una memoria caché: almacenarlo por si es requerido en un futuro próximo. La única limitación es el tamaño de la memoria.

- **Localidad espacial**: es la tendencia de los programadores y compiladores a agrupar objetos relacionados en espacios de memoria consecutivos. Como consecuencia, parece conveniente agrupar los datos en *bloques de caché*, mayores que un único elemento, de manera que cuando se solicita un dato concreto, se eleva todo el bloque que lo contiene, consiguiendo que los elementos cercanos se consideren también.

- **Localidad algorítmica**: en muchos casos se tiene un comportamiento predecible que no se puede explicar con ninguno de los principios anteriores. Supongamos un programa que accede repetidamente a varias estructuras de datos de forma intercalada que se encuentran consecutivamente en memoria (por ejemplo, puede pensarse en dos vectores ordenados que se combinan en uno solo). El comportamiento del programa es predecible, pero no puede ser capturado con la localidad temporal (no se vuelve a acceder a la misma posición de memoria) ni tampoco enteramente con la espacial (pues se accede a los dos vectores intercaladamente). Esta localidad es típica en aplicaciones que realizan operaciones repetidas sobre grandes conjuntos de datos que se almacenan en estructuras dinámicas y no cambian en períodos cortos de tiempo (por ejemplo, el algoritmo *Z-buffer*).

2.1.2. Organización lógica

La organización lógica define cómo se almacenan los datos en una caché. Una caché almacena cadenas de datos que se llaman **bloques de caché**. Cada elemento es referenciado con una dirección de memoria que se puede dividir en dos zonas: el ID del bloque, y el desplazamiento dentro del bloque (figura 2.3). Como la caché sólo maneja bloques, los datos se mueven a este nivel.

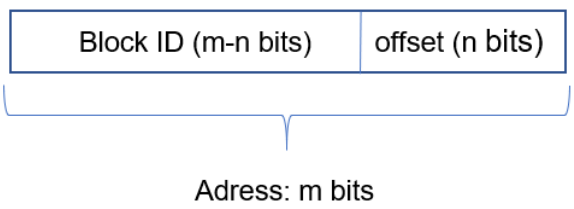


Figura 2.3: Dirección de memoria que se divide en ID de bloque y desplazamiento dentro del bloque.

La caché está compuesta por varias entradas, donde cada una de ellas almacena la siguiente información:

- La **etiqueta** de la caché. Puesto que una caché es mucho más pequeña que la memoria a la que sirve, se necesita algún mecanismo que permita identificar si un cierto elemento está o no almacenado en la caché. Para esto sirven las etiquetas: hacen referencia a un grupo de bloques e indica qué elemento se almacena en la entrada correspondiente.
- Bits de **estado** que, entre otras cosas, indican si la entrada está vacía o contiene datos inválidos o que deben sobreescribirse.
- Los **datos**.

Dado que la caché tiene capacidad para almacenar más de un bloque, es necesario determinar en qué lugar se pueden encontrar los datos correspondientes a un ID de bloque. Para ello, se define un **conjunto** como una lista de posibles entradas de caché, de manera que cada bloque está asociado a un único conjunto. Para asegurar esta condición, el número de conjunto se elige a partir de ciertos bits del ID de bloque, como se indica en la figura 2.4. Una vez que se ha determinado el número de conjunto para un cierto ID de bloque, los datos se almacenarán únicamente en la lista de entradas que se corresponden con ese conjunto. Nótese que sólo es necesario almacenar el campo etiqueta en la entrada de la caché para hacer referencia a un bloque.

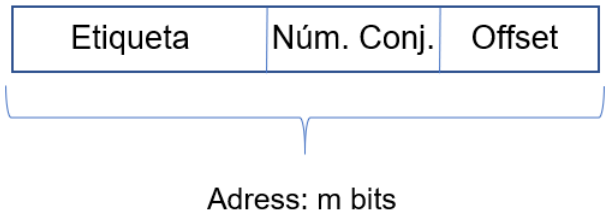


Figura 2.4: División de una dirección de memoria considerando las partes del ID del bloque

En este contexto, surgen varias maneras de organizar una caché:

- Una memoria de mapeo directo está compuesta por tantos conjuntos como posibles bloques, de manera que cada bloque tiene un único conjunto que no comparte con otros bloques.
- Una memoria totalmente asociativa tiene un único conjunto, por lo que todos los bloques se encuadran en el mismo y no es posible conocer a priori si un bloque se encuentra en la caché a menos que se recorran todas las entradas.
- Una memoria asociativa por conjuntos tiene más de un conjunto, donde cada uno incorpora más de un bloque. De esta manera se tiene una idea de dónde se puede encontrar un bloque, sin recorrer todas las entradas posibles. En la figura 2.5 se puede encontrar una memoria caché de 512 KB asociativa por conjuntos de nivel 4: se pueden almacenar hasta 4 bloques de datos que pertenecen al mismo conjunto.

Además es especialmente importante el problema de mantenimiento de la consistencia. Cuando se modifica un dato cualquiera en la memoria caché hay dos formas de hacer permanente dicho cambio, lo que da lugar a las dos políticas de escritura que se describen a continuación.

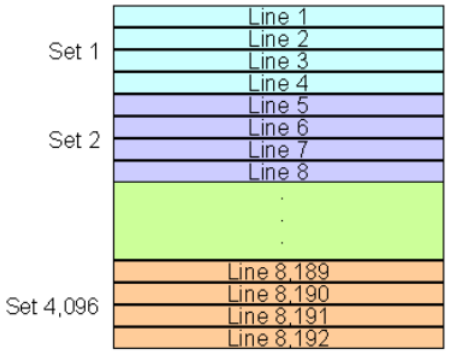


Figura 2.5: Memoria caché L2 de 512 KB configurada como asociativa por conjuntos de nivel 4.

- La política *write-through* escribe inmediatamente en el nivel inferior la modificación realizada, para que esté disponible la versión actualizada para todos los procesos que lo soliciten. Esta solución pasa por un aumento del tiempo y coste (ancho de banda, consumo energético), al ser obligatorio realizar todas las escrituras en el nivel inferior.

- La política *write-back* almacena información en la entrada correspondiente de la caché (a través del *dirty bit*) que indica que ese dato debe hacerse permanente en el nivel inferior, pero aún no se ha realizado el reemplazo. Esta política se aprovecha de los principios de localidad explicados anteriormente, puesto que si una aplicación ha escrito un cierto dato, es muy probable que vuelva a usarlo en un futuro cercano. En este caso, si se produce una reescritura posterior, no tendría sentido hacer dos escrituras en el nivel inferior cuando se puede realizar sólo una. La desventaja principal es que, si estamos hablando de un sistema multiprocesador, es posible que si cada uno mantiene una caché diferente, se pueda acceder a alguna versión "obsoleta" de la información al no hacerse persistente inmediatamente.

2.1.3. Políticas de reemplazo

Las políticas de reemplazo de una caché proporcionan una heurística para desechar bloques en la caché en pos de otros nuevos. En una caché las entradas se van llenando progresivamente y, cuándo la memoria está llena, es necesario desechar una entrada para poder guardar otra nueva.

Al desechar una entrada se puede hacer de dos formas: *silenciosamente* o con reemplazo. Si el bloque de caché ha sido modificado (esto es, el *dirty bit* está activado) y ese mismo bloque es el objetivo del algoritmo de reemplazo, es necesario escribir la nueva información en memoria principal. Si, contrariamente, no se ha modificado la información, basta con desechar la entrada sin más (es "silenciosa").

Una memoria caché con mapeo directo no necesita de un algoritmo para la selección de víctima. Cada bloque sólo puede ser ubicado en una posición, por lo que si la entrada está ocupada es necesario desechar lo que contiene. En una memoria totalmente asociativa las entradas se van llenando hasta que no se permitan más entradas, y es en este caso cuando hay que seleccionar una víctima de entre todas las entradas de la caché para ser sustituida por el nuevo bloque.

En una memoria asociativa por conjuntos, cada bloque de datos tiene una serie de entradas donde se puede ubicar. Un nuevo bloque se ubica en una entrada cualquiera del conjunto asociado, siempre que haya entradas libres. En caso de que todas estén ocupadas, el algoritmo para la política de reemplazo debe entrar en juego de nuevo.

Hay muchos criterios para definir diferentes políticas de reemplazo. El más eficiente sería aquel que descarta la información que no se va a necesitar más, o que más va a tardar en utilizarse. A este simple pero inalcanzable algoritmo se le conoce como el **algoritmo óptimo de Bélády**. Dado que es imposible conocer el futuro, es necesario buscar otras heurísticas basadas en alguna regla que funcionen bien en la práctica. Las más conocidas y utilizadas se listan a continuación [15].

- **First In First Out (FIFO)**. Al utilizar este algoritmo la caché se comporta como una cola FIFO. Sin importar cuántas veces se ha utilizado, se desechará el bloque que lleve más tiempo en el conjunto correspondiente de la caché.

- **Last In First Out (LIFO)**. El comportamiento de la caché aquí es el contrario a la cola FIFO: se desechará el último bloque que ha sido traído en lugar del primero.

- **Least recently used (LRU)**. Se descarta la entrada que lleva más tiempo sin usarse. Para poder aplicar LRU se necesita mantener información sobre la última vez que se utilizó un bloque, y hacer una búsqueda en el momento del reemplazo para encontrar la más inactiva. Es más caro de implementar, aunque funciona muy bien en la práctica.

- **Time aware least recently used (TLRU)**. Es una variante del LRU para cachés que almacenan bloques con un tiempo de vida limitado, y que añade un concepto nuevo a cada entrada de la caché: TTU (Time To Use). El TTU para cada entrada define el tiempo de vida que resta a esa entrada antes de ser desalojada obligatoriamente. El TLRU da prioridad para desalojar aquellas entradas con un TTU pequeño.

- **Most recently used (MRU)**. A diferencia del LRU, el algoritmo de reemplazo MRU desecha el último bloque usado. Aunque puede sonar contradictorio, Chou & DeWitt [19] mostraron que, cuando una estructura de datos es accedida repetidamente con un patrón de bucle secuencial, MRU es el mejor algoritmo de reemplazo. Funciona bien si la probabilidad de acceder a un elemento es mayor cuanto más antiguo sea.

- **Random replacement (RR)**. Como su nombre indica, la víctima se elige al azar de entre todas las posibles.

- **Least frequently used (LFU)**. Funciona de forma similar a LRU, aunque ahora, en lugar de almacenar el momento del último acceso, contamos cuántas veces ha sido accedido cada bloque. El elemento desechado es el que menos veces ha sido utilizado.

2.1.4. Inclusión y exclusión

Ya hemos visto cómo la jerarquía de memoria establece una partición vertical sobre las diferentes memorias que existen en un sistema. Sin embargo, dentro de un mismo nivel, es posible hacer particiones horizontales. En el caso de tener una partición horizontal en un nivel de caché se denomina **cache multi-lateral**. Un ejemplo de partición horizontal es la organización de la memoria en bancos, de lo que hablaremos más en detalle en secciones posteriores. Otra posibilidad es incluir varias memorias complementarias en un mismo nivel, cada una con un propósito. Mientras que la partición vertical se utiliza para conseguir accesos más rápidos a memoria mediante los procedimientos descritos anteriormente, las particiones horizontales sirven para controlar la energía que se consume en un determinado nivel mediante la utilización de la partición más adecuada en cada caso.

Los principios de *inclusión* y *exclusión* definen las relaciones existentes entre dos particiones cualesquiera de la jerarquía de memoria, independientemente de su posición vertical u horizontal. Por supuesto, son posibles esquemas híbridos a medio camino entre los dos.

Una relación de inclusión garantiza que cualquier elemento en una cierta partición tiene una copia en otra. Puede que el lector piense que es una pérdida de tiempo y de recursos almacenar dos veces la misma información, pero recordar que acceder a un elemento en el primer nivel de la jerarquía es millones de veces más rápido que en el último debería hacerle cambiar de opinión. Una relación de este tipo además, garantiza que si se quiere sustituir una porción de memoria que se sabe se encuentra en niveles inferiores y que no ha sido modificada, basta con descartarla inmediatamente.

Por otro lado, una relación de exclusión permite conocer con certeza que si un elemento se encuentra en una partición, no es posible que se encuentre en otra. El conocimiento de esta relación permite agilizar los procesos de control de la consistencia que subyacen en todos los sistemas.

2.2. Visión general de las memorias DRAM

La memoria DRAM, por sus siglas en inglés (*Dynamic Random Access Memory*) es lo que se conoce como memoria principal. Todo programa, para ejecutarse, debe cargarse en memoria principal. La DRAM se conecta al procesador a través de un intermediario que se conoce como el **controlador de memoria**, de manera que el procesador no necesita conocer cómo se maneja el módulo de memoria concreto. Aunque hablaremos del mismo más adelante, sus funciones principales son las relacionadas con el manejo de la DRAM a bajo nivel y la más que probable compatibilidad con varios procesadores, como ocurre en prácticamente todas las arquitecturas modernas.

La figura 2.6 muestra un ejemplo de las conexiones y responsabilidades del controlador de memoria. En este dibujo se muestra la CPU como un único chip en el que ya se incluyen las memorias caché correspondientes. La comunicación del procesador con la memoria se hace a través

de este controlador, también llamado *puente norte* (*North-Bridge*). Más abajo se encuentran los dispositivos de entrada y salida. En el dibujo, las conexiones SATA, USB y PCI se pueden utilizar para conectar módulos de memoria no volátiles que sirvan en el nivel más bajo de la jerarquía. Al controlador de entrada y salida se le conoce a veces como *puente sur* (*South-Bridge*).

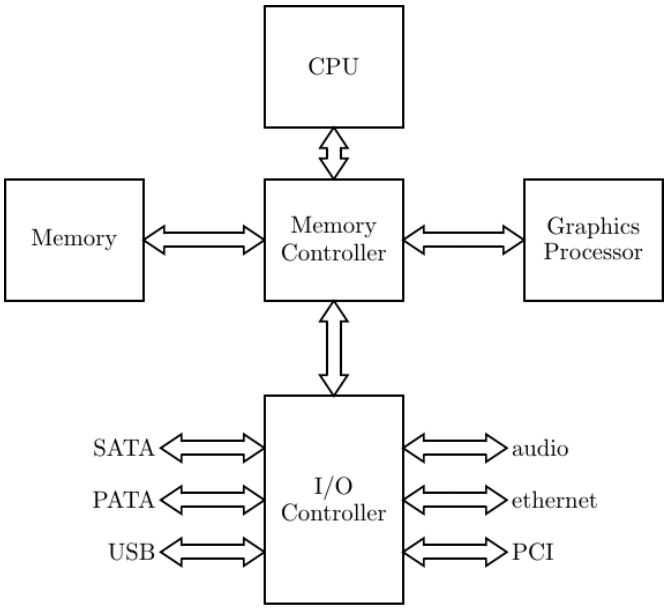


Figura 2.6: Esquema de organización del controlador de memoria y sus interacciones con distintos componentes [9].

2.2.1. Celdas, arrays de memoria, bancos y ranks

La DRAM es un tipo de memoria RAM que utiliza un par transistor-condensador para cada celda de almacenamiento que guarda un bit de información. La palabra *dinámica* hace referencia a la necesidad de refrescar continuamente la información del condensador (es decir, leer y volver a escribir el mismo bit), pues de no hacerse la información se perdería al cabo de un tiempo.

Cada módulo DRAM contiene uno o varios **arrays de memoria**, que no son más que la organización en una matriz bidimensional de las celdas de memoria. Puesto que se organizan en filas y columnas, cada celda de memoria se especifica en un array de memoria mediante estos dos parámetros. El controlador de memoria es el que conoce como leer o escribir información en una posición dada. En la figura 2.7 se encuentra un ejemplo de un array de memoria de una DRAM con capacidad para 128KB de datos. Cada palabra (fila) está compuesta por 512 bits (64 Bytes de datos). Además, cada celda de memoria está representada con un transistor y un condensador. Aunque en la figura no se encuentra explícitamente representado, es necesario una amplificación de la señal una vez leída la posición correspondiente.

Esta estructura para las celdas de memoria ha sido la más utilizada y conocida (1T1C). Existen otras estructuras menos usadas, como la variante con tres transistores (3T1C), que tiene velocidades de lectura más altas. Son bastante más grandes que las celdas 1T1C, por lo que la

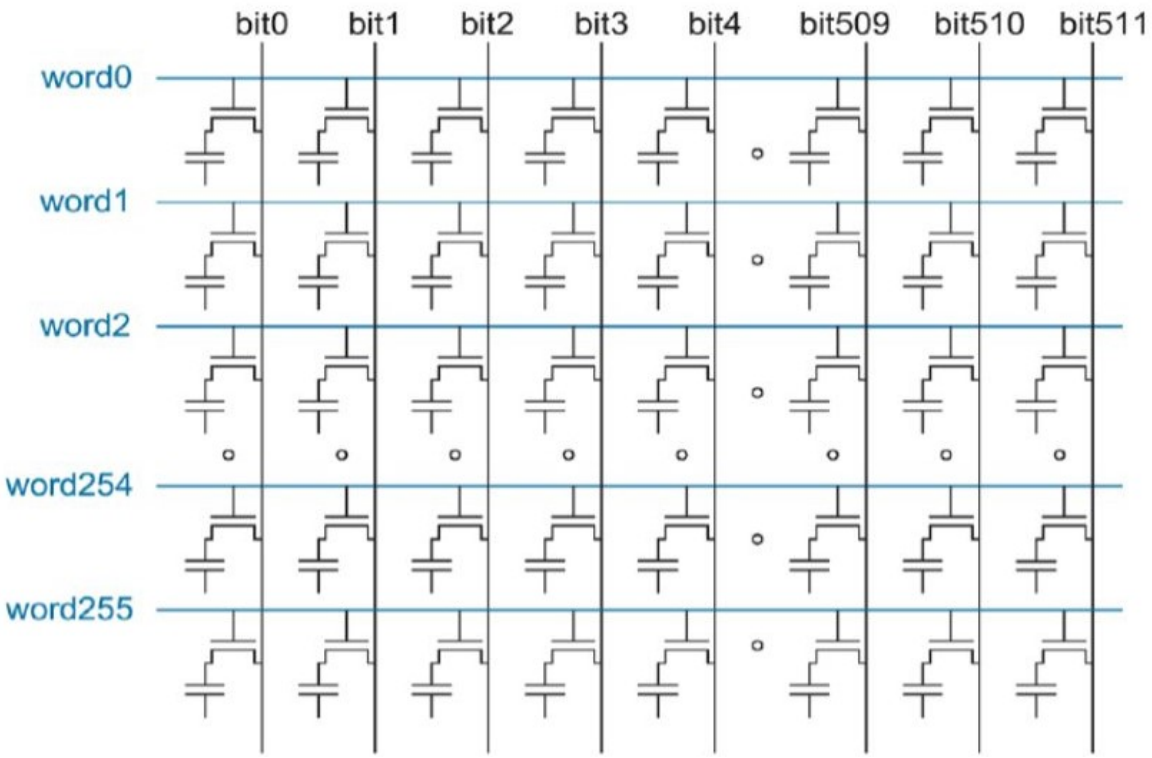


Figura 2.7: Organización bidimensional de un array de memoria en una DRAM [10].

mayoría de los dispositivos DRAM actuales están basados en 1T1C, priorizando la densidad sobre la velocidad. La investigación para el desarrollo de nuevas estructuras y tecnologías continúa en nuestros días.

El acceso a una celda de memoria se produce mediante la activación del transistor correspondiente aplicando un determinado voltaje en la puerta de acceso. La aplicación de otro voltaje que procede, por ejemplo, de un bit que se quiere escribir en la celda, carga el condensador con la información. Sin embargo, la carga almacenada en el condensador se va perdiendo con el paso del tiempo (de ahí la necesidad de refrescarlo). Para asegurar que se mantienen los datos correctos, cada celda de memoria se debe leer y volver a escribir con la misma información a intervalos regulares de tiempo. Una celda normal puede almacenar la información correcta hasta algunos segundos. Sin embargo, es necesario garantizar que el bit representa información válida en todo momento, por lo que el tiempo habitual entre refrescos es de unos 32 o 64 milisegundos.

Una forma de caracterizar un módulo DRAM es mediante el número de arrays de memoria que contiene, y cómo interaccionan entre sí: pueden actuar de forma totalmente independiente unos de otros, de la misma manera, o presentar un comportamiento por grupos. En todos los siguientes ejemplos, conviene imaginar la memoria DRAM con una tercera dimensión a partir de la figura 2.7, donde se superponen los distintos arrays de memoria.

- Si todos los arrays de memoria actúan de la misma forma, operan como una única unidad de memoria. Especificada una posición, se realiza la operación sobre todos los arrays de memoria existentes. Por ejemplo, si se tiene una DRAM con 8 arrays, realizar una operación

de lectura sobre una fila y una columna lleva a la lectura de 8 bits.

- Si todos los arrays actúan independientemente, no basta con indicar la fila y la columna, sino que también es necesario indicar cuál de todos los arrays se quiere tener en cuenta. Sobre el ejemplo anterior, equivaldría a especificar la fila, columna y un número del 1 al 8 que haga referencia al array. Antes se tenía un ancho de banda de 8 bits por operación, pero ahora el ancho de banda es 1.

- Es posible que los arrays de memoria se agrupen por conjuntos, donde todos los arrays de un mismo conjunto operen como una única entidad, pero sean completamente independientes de los arrays de otro grupo. Si consideramos dos grupos sobre el ejemplo, tendríamos 4 arrays de memoria en cada grupo. La especificación de una operación necesita de la fila, columna y el número de grupo (1 o 2), y tiene un ancho de banda de 4 bits. A cada uno de estos grupos se le conoce como **banco**.

Cada banco de memoria es independiente y, salvo ciertas restricciones estructurales, se puede activar, precargar, leer o escribir al mismo tiempo que otros bancos. La utilización de varios bancos de memoria permite lograr anchos de banda mayores a partir de dispositivos más lentos, como hemos comprobado en el ejemplo.

Los módulos de memorias DRAM se venden normalmente como DIMM (*dual in-line memory module*, módulo de memoria con contactos duales). Como en general incluyen varias DRAM, cada una de ellas con una configuración de bancos potencialmente diferente, puede pensarse en "bancos" de DRAMs a nivel de DIMM. A esto se le conoce como **rank**, y cada uno de ellos se diseña para operar en exclusión mutua.

Finalmente, un sistema se puede componer de distintos DIMMs, cada uno de ellos con uno o varios ranks. Cada rank es un conjunto de dispositivos DRAM que operan al unísono, aunque internamente cada DRAM puede tener una configuración de bancos de memoria. Cada uno de estos bancos está compuesto por un cierto número de arrays de memoria, y el ancho de banda de dicha DRAM es equivalente al número de arrays de cada banco. El mecanismo de ranks, bancos y arrays permite ampliar el ancho de banda con accesos paralelos a distintas celdas de memoria. En la figura 2.8 puede encontrarse un ejemplo de división de dos DIMMs, cada uno de ellos en dos ranks, donde cada rank tiene cuatro bancos.

2.2.2. Buses para comunicarse

La comunicación entre el controlador de memoria y los DIMMs de un sistema tiene lugar a partir de buses que llevan la información, tanto para leer y escribir datos, como para indicar las operaciones que se quieren realizar. Los buses en un estilo de organización JEDEC se clasifican por su función en cuatro grupos:

- Los **buses de datos**.

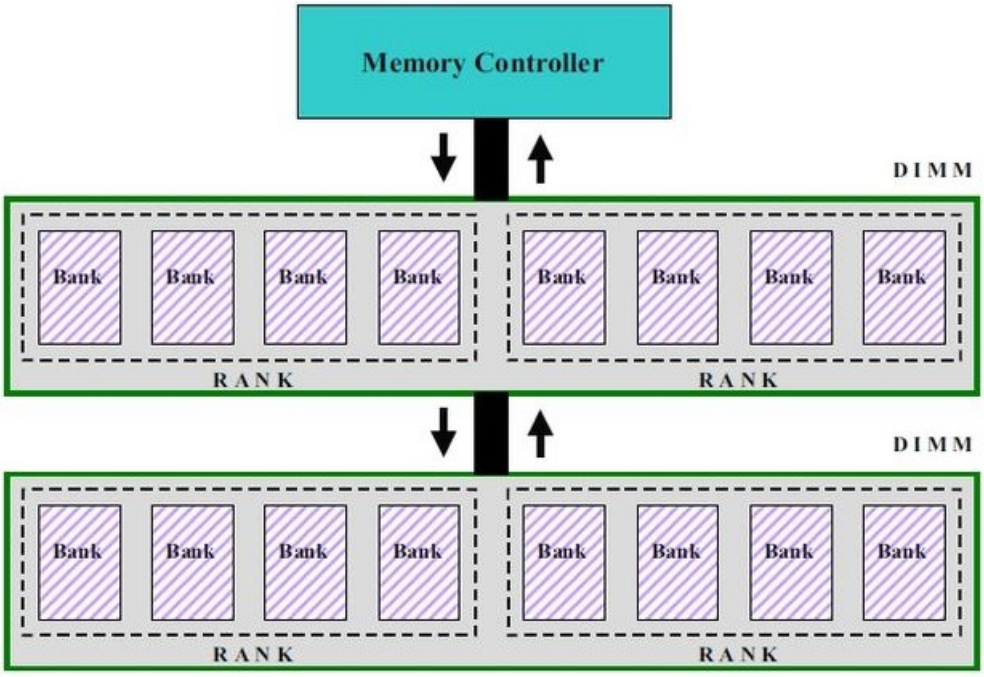


Figura 2.8: Organización de un par de DIMMs en ranks y bancos. Fuente: *m5sim.org* [11]

- Los **buses de dirección**.
- Los **buses de control**.
- Los **buses de selección de chip**.

El bus de datos sirve para llevar la información que se quiere leer o escribir, y suele tener un ancho de banda suficiente para llevar a cabo estas operaciones en un tiempo razonable (por ejemplo, 64 bits). El bus de dirección indica la posición como número de fila y columna para una DRAM determinada, y es más ancho cuanto mayor sea el tamaño de la DRAM, lo suficiente como para indexar la información. El bus de control tiene ciertos bits de información necesarios, como la señal de reloj.

Por último, el bus de selección de chip es el que se encarga de seleccionar la DRAM correspondiente dentro de un rank determinado. Este bus debe seleccionar únicamente la o las DRAMs sobre la que se va a realizar la operación, por lo que debe ser tan ancho como el número de DRAMs del sistema. Aunque todas las DRAMs están conectadas al mismo bus de control y de direcciones, el bus de selección de chip evita que todas pongan una respuesta en el bus de datos (en caso de una petición de lectura, por ejemplo) activando sólamente el módulo deseado.

2.2.3. Pasos en una petición típica

Consideremos una petición de lectura sobre un grupo de DIMMs en un sistema (por ejemplo, el mostrado en la figura 2.8). Los pasos que se siguen son los siguientes:

1. El microprocesador envía al controlador de memoria una petición de lectura a una dirección determinada. Esta dirección es la que maneja el procesador atendiendo al total de memoria que hay en el sistema, pero necesita ser traducida por el controlador para poder leer los datos.

2. La petición llega al controlador de memoria, que la almacena en una cola de peticiones y las va despachando siguiendo algún tipo de algoritmo de planificación.

3. Cuando la petición se atiende, el controlador de memoria descompone la dirección proporcionada por el procesador en los componentes adecuados que permitan identificar el rank y el banco, así como la fila y la columna dentro del banco. A estas dos últimas las llamaremos *dirección de fila* y *dirección de columna*. El identificador de banco es normalmente un grupo de bits, mientras que la identificación de cada rank se lleva a cabo con el bus de selección de chip.

4. Las líneas de información del banco seleccionado deben ser precargadas antes de poderse utilizar, y se colocan en un nivel lógico a medio camino entre el 0 y el 1.

5. Se activa la fila seleccionada dentro del rank y banco elegido mediante la señal de selección de chip en el bus correspondiente y la dirección de banco y de fila sobre el bus de direcciones. Esto hace que la DRAM prepare una fila entera de datos, aunque no se vayan a usar todos, y suele llevar unos pocos nanosegundos. Una fila también se conoce como una **página**, puesto que la activación de una fila generalmente provoca elevar toda ella como un bloque a niveles superiores para aprovechar la localidad. Un problema derivado de este esquema es que el tamaño de las páginas depende de la estructura de la DRAM en lugar de ser elegido por el procesador para adecuarlo mejor al tamaño de la caché.

6. Se envía la dirección de columna sobre el bus de direcciones que provocan la lectura únicamente de la porción deseada de memoria. Una columna es la dirección de memoria más pequeña a la que se puede acceder.

7. Se colocan los bits de información en el bus de datos. A la transición de datos en este bus se le llama “pulso”.

8. El controlador de memoria recibe los datos y se los pasa al procesador que los solicitó.

Nótese que la dirección de fila y la dirección de columna se transmiten por el mismo bus pero a dos tiempos, de manera que la información sobre la columna sólo se transmite cuando la fila entera está disponible para ser leída.

Además, todo este proceso se realiza gobernado por una señal de reloj, que indica cúando se deben realizar las operaciones en cada momento. La señal de reloj es una señal periódica con flancos de subida y de bajada e intervalos regulares en los que la señal está en 1 y en 0. Cuando la señal de reloj pasa de 0 a 1 se dice que es un flanco de subida, mientras que si pasa de 1 a 0 estamos

considerando un flanco de bajada. El procesador tiene su propia señal de reloj que gobierna la ejecución de las instrucciones. Atendiendo al reloj que utiliza la DRAM podemos clasificarlas en dos grupos:

- Memorias **DRAM síncronas** (SDRAM): existe una señal de reloj que controla la DRAM. Los pasos internos de una DRAM ocurren con los flancos de subida o bajada de la señal de reloj global. Por cada ciclo de reloj, se transmite un pulso de datos.
- Memorias **DRAM asíncronas**: las acciones internas de la DRAM ocurren bajo el mandato del controlador de memoria, que dice cuándo se deben realizar las operaciones. Se pueden hacer a intervalos de tiempo diferentes que la dada por la frecuencia de la señal de reloj.

2.2.4. Evolución de la arquitectura DRAM

En los 80 y los 90, las DRAM comenzaron a ser el cuello de botella que lastraba el rendimiento del sistema completo. Como consecuencia, la interfaz de la DRAM comenzó a evolucionar para satisfacer las necesidades del momento.

La arquitectura DRAM básica estaba gobernada por una señal de reloj, pero fue descartada en poco tiempo en favor de otras arquitecturas asíncronas. La evolución de estas memorias pasó por la DRAM asíncrona, *Fast Page Mode DRAM (FPM)*, *Extended Data-Out (EDO)*, *Burst-mode EDO (BEDO)*, y finalmente una vuelta a la sincronía con las SDRAM. Este camino seguido está centrado en la **mejora de rendimiento** a partir de cambios estructurales. Desde las SDRAM la industria ha seguido diferentes caminos modificando las interfaces, que se dividen en aquellas mejoras centradas en **reducir la latencia** y en las que buscan aumentar más el rendimiento.

2.2.4.a. Modificaciones estructurales para mejorar el rendimiento

En comparación con las clásicas DRAM, las FPM permitían a una fila seguir seguir activa para múltiples peticiones de diferentes columnas. Si en la DRAM original se quiere acceder a otra columna de la misma fila que la petición anterior, es necesario reiniciar el proceso descrito en la sección 2.2.3, con la correspondiente desactivación y activación de dicha fila. En las FPM DRAM esto no era necesario, pues se permitía a la fila permanecer abierta a lo largo de múltiples peticiones de acceso a columnas con muy poca circuitería adicional.

Las EDO DRAM modificaban las salidas de las DRAM para añadir unos pequeños registros que mantenían los datos leídos un poco más de tiempo, lo que a su vez hacía que se mantuvieran más en el bus de datos. Las BEDO DRAM añaden a esto un contador que, en combinación con una dirección de columna, hacen que no sea necesario que el controlador de memoria proporcione una nueva dirección si lo que se quiere es leer la siguiente columna.

En las BEDO, la circuitería de selección de columna actúa en conjunto con una señal interna generada en lugar de con una señal externa que provenga desde el controlador. Al estar más cerca

de los circuitos correspondientes, los tiempos de activación son menores y menos variables. La SDRAM modifica este esquema y dirige toda la circuitería interna (selección de fila, selección de columna, etc.) con una señal de reloj.

2.2.4.b. SDRAM

Las FPM, EDO y BEDO son manejadas directamente por el controlador de memoria. Son asíncronas porque las señales que indican que las entradas correspondientes son una dirección de fila (RAS, *Row Adress Strobe*) o de columna (CAS, *Column Adress Strobe*) pueden llegar en cualquier momento. Cuando se recibe esta señal, el dispositivo interpreta que las señales que está recibiendo son, respectivamente, una dirección de fila o columna.

La otra opción es hacer que estas señales sólo puedan llegar a intervalos regulares de tiempo, por lo que la DRAM debe estar gobernada por una señal de reloj. La ventaja principal es que asociar las transferencias de datos y control con una señal de reloj hace que el momento en el que ocurren exactamente los diferentes eventos es mucho más predecible y menos sesgado. La consecuencia de la reducción del sesgo es que el sistema es capaz de manejar mejor las transiciones entre peticiones, provocando un aumento en el rendimiento.

Las SDRAM, al igual que las BEDO, soportan el concepto de *ráfaga*. Poseen un registro programable que contiene información sobre el tamaño de la ráfaga. La DRAM lo utiliza para saber cuántas columnas de datos debe colocar en el bus de datos a lo largo de los sucesivos ciclos de reloj, por lo que potencialmente devolverán varios bytes en ciclos de reloj "próximos". Gracias a esto se pueden eliminar algunas señales de control, como la asociada a la dirección de columna (CAS), reduciendo el ancho de banda utilizado.

Aunque las SDRAM tienen un coste mayor que las BEDO y no ofrecen ninguna mejora en el rendimiento general a una velocidad de reloj determinada, la presencia de la señal de reloj en la SDRAM permite que la misma no se vea tan afectada en cambios de velocidad. Esto no ocurre en las DRAM asíncronas, y en concreto, tampoco en las BEDO.

Desde el desarrollo de las SDRAM, las mejoras que se han seguido buscando tienen que ver con las interfaces y en sentidos diferentes. Las enfocadas en el rendimiento y mejoras del ancho de banda pasan por el desarrollo de las memorias DDR. Una DDR SDRAM (*Double Data Rate SDRAM*), o DDR a secas, es un tipo de memoria RAM de la familia de las SDRAM que otorga a ciertos módulos SDRAM, disponibles en encapsulado DIMM, la capacidad de transferir simultáneamente datos por dos canales distintos en un mismo ciclo de reloj [12].

Por otro lado, las mejoras encaminadas hacia una reducción de la latencia se han considerado menos relevantes, y por tanto, menos estudiadas. Desde las BEDO, no se ha propuesto nignuna arquitectura que provoque una mejora significativa a un coste razonable. Las mejoras en este campo pasan por aumentar la velocidad de los circuitos o mejorar la latencia global mediante memorias caché. Algunas menciones destacables son las *Enhanced SDRAM (ESDRAM)* y las *Reduced Latency DRAM (RLDRAM)*.

2.2.4.c. Resistive Random Acess Memory

Las memorias RRAM (*Resistive Random Access Memory*) son un tipo de memoria RAM no volátil, es decir, la información no se pierde si se deja de suministrar corriente eléctrica al dispositivo. Esto provoca, por ejemplo, que no sea necesario refrescar la información como en las DRAM, además de permitir una mayor densidad de información por unidad de superficie y tamaños mucho mayores. No obstante, tiene como principal desventaja su lentitud en comparación con las SDRAM, además de su desgaste con el tiempo.

El funcionamiento está basado en el cambio de la resistencia de un material dieléctrico sólido. Fundamentalmente se han estudiado los fenómenos de la conmutación resistiva en diversas estructuras metal-óxido-metal (MIM) y metal-óxido-semiconductor (MIS). La aplicación de una señal adecuada en estos materiales provoca la creación de un filamento conductor que conecta dos electrodos, y permite o impide el paso de la corriente, diferenciando así dos estados que sirven para almacenar un bit de 1 o 0. Este filamento puede permanecer en su estado durante largos períodos de tiempo, sin necesidad de refresco de ningún tipo. Sin embargo, aún quedan muchos interrogantes por resolver antes de abordar un uso comercial de las RRAM, como la existencia de fluctuaciones entre los diferentes estados resistivos [1], [2], [3], [4].

Aunque tiene un gran potencial como sustituto de las memorias flash, la industria aún no ha considerado que los beneficios que aporta una RRAM compensen los cambios necesarios que se deben realizar en la jerarquía de memoria. A principios de los 2000 se comienza a gestar la idea del desarrollo de las RRAM y se empiezan a comercializar en pequeñas muestras de poca capacidad. Aunque grandes empresas han dedicado muchos recursos a investigaciones en este campo, aún no se ha encontrado la mejor forma de explotar esta tecnología.

En comparación con otras memorias no volátiles (como las PRAM, *Phase-change RAM*), las RRAM son capaces de operar a mayor velocidad. Tienen una estructura de celda más simple que las MRAM (*Magnetic RAM*), pues se pueden construir con un diodo y una resistencia. Comparadas con las memorias flash, se requiere un menor voltaje para funcionar.

2.3. Organización de la memoria DRAM

A lo largo de esta sección emplearemos la terminología de rank, banco, fila y columna definido anteriormente, así como el concepto de **canal** que se introduce a continuación.

Supongamos un sistema con un único controlador de memoria y varias DRAMs. Muchas veces se conoce a este controlador por sus siglas en inglés, DMC (*DRAM memory controller*). El DMC controla un canal de comunicación con los distintos módulos de memoria que se instalan en un sistema. En la mayoría de sistemas modernos, los buses de datos están estandarizados y delimitan el ancho de este canal de comunicación. Sin embargo, es posible conectar el DMC con varios módulos de memorias que tienen un ancho diferente. Aquí se introducen las diferencias entre **canal lógico** y **canal físico**, así como la configuración de *dual channel*.

En la figura 2.9 puede encontrarse una configuración típica de *dual channel*, donde se acoplan dos modulos DDR a un controlador DMC. En este caso se tiene un canal lógico de 128 bits, conocido por el DMC, pero que realmente está compuesto por dos canales físicos de 64 bits, cada uno conectado a su respectivo módulo DDR.

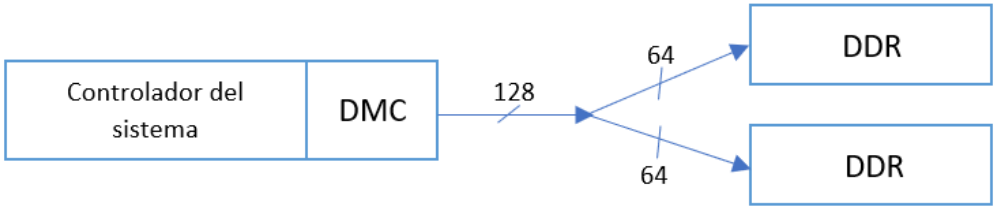


Figura 2.9: Esquema simple de una configuración en Dual Channel de un sistema de memoria.

En la configuración *dual channel* el DMC conoce únicamente un canal, y cada uno de los dos módulos DDR opera por separado con su canal físico. Sin embargo, también son comunes los sistemas donde en lugar de un único DMC existen varios, cada uno de ellos controlando distintos buses. En la figura 2.10 se muestra la configuración teniendo en cuenta que ahora hay dos DMCs, y cada uno de ellos está conectado mediante un canal físico a un módulo DDR. El uso de diferentes controladores tiene varias ventajas, entre las que fundamentalmente se encuentra un aumento del ancho de banda y de las prestaciones del sistema.

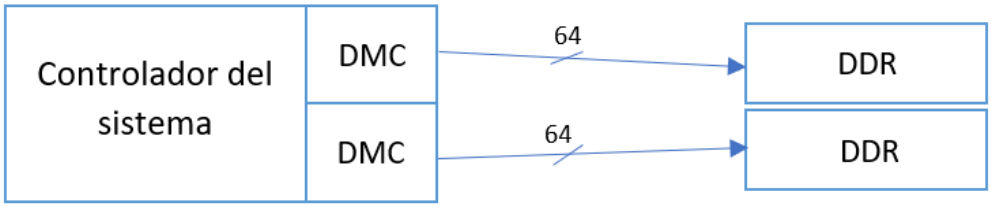


Figura 2.10: Esquema simple de una configuración con dos DMCs y dos canales.

La versión más encontrada en la mayoría de los sistemas actuales es la que sólo tiene un DMC pero varios canales físicos, aunque muchas veces se diseñan para utilizar estos canales en exclusión mutua. Las dos variaciones más comunes en este escenario son las que se describen a continuación.

En la primera se utilizan módulos de memoria potencialmente diferentes, y se hace corresponder cada uno con un canal físico. El controlador del sistema opera en modo "asimétrico" y controla los diferentes canales físicos por separado. Como sólo hay un DMC, no se puede acceder a varios canales físicos simultáneamente. El máximo ancho de banda del sistema completo es el ancho de banda más alto del conjunto de los módulos DRAM instalados. Fundamentalmente se utiliza para añadir más cantidad de memoria, pero no ampliar las prestaciones.

La segunda se encuentra en los sistemas FPM DRAM de alto rendimiento (*Fast Page Mode DRAM*). En este caso, es posible especificar una dirección de columna y acceder a los datos que ahí se encuentran, pero el sistema continúa devolviendo información de las siguientes columnas hasta que se desactiva el bit correspondiente que lo controla. Sin embargo, no permite el acceso a

varias columnas simultáneamente. Una solución a esta limitación es la utilización de varios canales FPM DRAM que operan intercaladamente, para lo cual se necesita un controlador de memoria más sofisticado que pueda enviar peticiones de acceso a diferentes columnas, cada una de ellas por un canal diferente.

En los sistemas DDRx SDRAM, se pueden transmitir dos pulsos de datos en un mismo ciclo de reloj. Cada acceso a una columna implica una ráfaga de accesos que depende de la programación concreta del módulo. En un sistema DDR2 DRAM, cada lectura devuelve como mínimo 4 columnas de datos. Los sistemas DDR3, usados desde principios de 2010, tienen como ventaja respecto de los DDR2 la posibilidad de hacer transferencias de datos más rápidamente y mejoras significativas en el rendimiento en niveles de bajo voltaje, lo que provoca un menor consumo de energía. Como desventaja, la latencia de acceso es proporcionalmente más alta. Las memorias DDR4 (2013) ahorran aún más energía que sus predecesoras operando a un voltaje menor y tienen tasas más altas de frecuencias de reloj y transferencia de datos, aunque de nuevo a costa de una mayor latencia.

2.3.1. Módulos de memoria

Desde las primeras generaciones de ordenadores se permitía la actualización de la memoria DRAM. Al principio, los computadores traían unas ranuras extras donde se podían insertar dispositivos DRAM y actualizar unos independientemente de otros. Sin embargo, el proceso de actualización era muy laborioso y difícil, por lo que poco después se sustituyeron los chips individuales por módulos de memoria.

Los módulos de memoria son una mejor solución para la actualización de la memoria principal en un ordenador. Permiten una abstracción a un nivel mayor, donde los fabricantes elaboran chips que ya están compuestos por varios dispositivos DRAM y las conexiones necesarias, de manera que se prepara lo más posible para que los módulos sean *plug and play*.

La complejidad del proceso de actualización se reduce en gran medida a nivel de DRAM independientes, aunque poco a poco los módulos de memoria son cada vez más sofisticados. A continuación se presentan algunos de los módulos más conocidos.

2.3.1.a. Single In-Line Memory Module (SIMM)

Los módulos SIMM corresponden al primer estándar de la industria acordado sobre los años 90. SIMM hace referencia a aquellos módulos en los que los contactos de ambos lados situados en la parte de abajo tenían idénticas características eléctricas. En primer lugar se utilizaban las memorias SIMM de 30 pines, para migrar posteriormente a las de 72 pines.

Las SIMM de 30 pines permitían enviar 8 o 9 señales en el bus de datos, además de las señales necesarias para su funcionamiento (direcciones, selección de chip, etc.). A finales de los 80 se solían utilizar cuatro módulos de 30 pines para proporcionar una interfaz de memoria con una

anchura de 36 bits que soportara comprobaciones de paridad. Fue a principios de los 90 cuando se sustituyeron por los módulos de 72 pines.

Las SIMM de 72 pines, por contra, permitían una anchura de entre 32 y 36 bits en el bus de datos, además de las señales de control necesarias. El cambio fue motivado por las necesidades cada vez mayores de los ordenadores en cuanto a ancho de banda.

2.3.1.b. Dual In-Line Memory Module (DIMM)

A finales de los 90 se produce una transición de las FPM DRAM a las SDRAM, lo que provoca que las SIMM de 72 pines cedan a favor de los módulos DIMM. Los DIMM son físicamente más grandes y proporcionan una interfaz para el bus de datos con anchura de 64 o 72 bits. A diferencia de las SIMM, los contactos de cada uno de los lados tienen diferentes características eléctricas.

Las DIMM para los ordenadores personales suelen ser no registradas (*Unregistered DIMM* o *Unbuffered DIMM*), y no tienen ningún registro entre la DRAM y el controlador de memoria del sistema. Esto hace que haya más carga eléctrica en el Controlador de Memoria y permite sistemas con menos módulos, en contraposición con los módulos que sí tienen este buffer, más típicos de grandes servidores.

2.3.1.c. Registered Memory Module (RDIMM)

A diferencia de los módulos UDIMM, para satisfacer las necesidades de memoria de servidores y estaciones de trabajo más grandes, se necesita un módulo que incluya un pequeño buffer entre el DMC y los dispositivos DRAM. Tener un gran número de DRAMs tiende a sobrecargar los buses de datos y a dar problemas en la carga de la información. Los módulos que incluyen un registro que actúa de intermediario pueden solucionar este problema de sobrecarga, además de permitir una mayor flexibilidad en la combinación de memorias de características o fabricantes diferentes.

Un módulo RDIMM alivia la sobrecarga eléctrica provocada por un grán número de DRAMs a través de registros que almacenan temporalmente las direcciones y señales de control a nivel de interfaz del módulo. Concretamente, pueden reducir el número de cargas eléctricas que el controlador de memoria debe manejar directamente. La señal que se conecta con el sistema de memoria se divide en la parte que va desde el DMC a los bufferes intermedios, y la que conecta estos registros con los dispositivos DRAM.

Como punto negativo, el almacenamiento intermedio de las señales de control y direcciones de memoria en un buffer introduce un paso más en la espera para servir la petición, por lo que inevitablemente la latencia de acceso se ve incrementada en todas las transacciones.

Existe otro tipo de módulo, llamado *Small Outline Dual In-line Memory Module (SO-DIMM)*, que está diseñado y estandarizado explícitamente para ocupar poco espacio, siendo especialmente utilizado en pequeños *notebooks* y portátiles.

2.3.1.d. Serial Presence Detect (SPD)

Desde que los módulos de memoria comenzaron a evolucionar, es inevitable que cada uno tenga sus características en cuanto a latencia de acceso a columna o a fila, tiempo de precarga, o de activación de una fila. Esta variabilidad en cuanto a los módulos DRAM incrementa la complejidad de diseño cuando se quiere conseguir la máxima compatibilidad posible entre diferentes módulos. No obstante, esto no siempre es posible, ya que las nuevas generaciones de dispositivos DRAM pueden tener ciertas características de diseño físicas que las hagan incompatibles con los *slots* actuales, o ciertas características que el DMC no está preparado para soportar.

Para reducir la confusión que provoca la sustitución de módulos de memoria, cada uno de ellos viene con información sobre su configuración y parámetros de latencia que se almacena en una memoria de sólo lectura. El contenido es leído por el DMC en el proceso de inicialización para optimizar los accesos todo lo posible. Esta memoria de sólo lectura se conoce como *Serial Presence Detect*.

2.3.2. Topología habitual

En la figura 2.11 [13] se muestra un ejemplo de sistema de memoria donde 16 dispositivos DRAM están conectados a un mismo DMC. Estas DRAM se organizan en 4 ranks. Es posible apreciar cómo las conexiones unidireccionales para pasar direcciones, el bus de comandos, el bus bidireccional de datos y el bus de selección de chip no muestran un patrón "regular", y cada uno tiene sus particularidades.

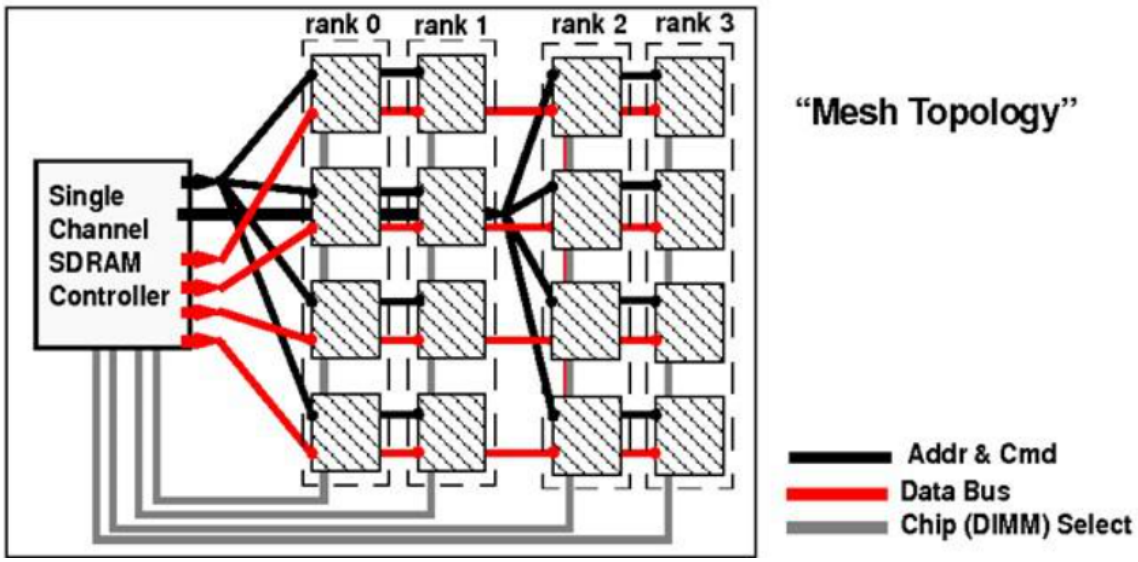


Figura 2.11: Topología de un sistema de memoria DRAM.

Cuando se quiere realizar una operación en este ejemplo, las señales de direccionamiento y selección de comando se propagan a todas las DRAM, pero la señal de selección de chip hace que

sólo se active un rank determinado. Cada DRAM dentro de un rank está conectada a una parte del bus de datos en el que servir los bits leídos, por ejemplo, en una petición de lectura.

La topología del sistema de memoria determina la longitud que deben recorrer las señales eléctricas y, por tanto, influyen en gran medida en el rendimiento. La topología típica, como la presentada en el ejemplo anterior, se ha visto inalterada en el paso de las DRAM originales a las FPM, SDRAM y DDR. Un ejemplo de topología radicalmente diferente es la de un sistema *Direct RDRAM*, aunque no será presentada aquí.

2.4. Protocolo básico de acceso a memoria DRAM

El protocolo de acceso a memoria define todos los posibles comandos y restricciones de tiempo que el DMC maneja para el flujo de datos con las distintas DRAM. El protocolo y los comandos que se explican en esta sección están descritos bajo un marco general, puesto que cada arquitectura diferente los adapta para sacar el máximo partido a los dispositivos.

Examinar completamente y con detalles el protocolo completo de acceso a memoria es complicado debido a la existencia de incontables combinaciones y ordenaciones de los comandos. Por una parte, es necesario hablar de los comandos básicos a partir de los cuáles se forman otros más complejos, así como de las interacciones entre los comandos de una DRAM básica. Ésto último permite conocer la latencia del sistema completo y otras características relevantes, como el ancho de banda.

2.4.1. Comandos básicos

El protocolo de acceso a memoria DRAM asume que dos comandos se llevan a cabo simultáneamente siempre que no se requiera el uso de un recurso compartido. Por este motivo, muchas de las operaciones se superponen para reducir el tiempo necesario en un aceso de lectura o escritura. Concretamente, en un acceso a memoria se distinguen cuatro fases que se pueden superponer parcialmente en el tiempo.

1. En la fase 1, el comando concreto se transporta desde el DMC a través de los buses correspondientes, y la DRAM lo decodifica cuando lo recibe.

2. En la fase 2 se tiene un flujo de datos de una fila en el banco elegido dentro de la DRAM. En primer lugar, los bits almacenados en las celdas de memoria pasan al array de amplificación (los bits necesitan amplificarse para poder trabajar con ellos); posteriormente se vuelven a llevar a las celdas individuales.

3. En la fase 3 hay un flujo de datos entre la interfaz de entrada y salida (interfaz E/S) y los registros de lectura o escritura, dependiendo del tipo de operación.

4. En la fase 4, si estamos manejando una petición de lectura, se colocan los datos de una columna en el bus de datos para llevarlos hasta el DMC. Si es una pertición de escritura, los datos van desde el DMC hasta la columna elegida.

El protocolo de acceso a DRAM también define una serie de restricciones temporales entre comandos consecutivos. Abstrayendo las características temporales, los comandos pueden ser descritos a más alto nivel y ser comprendidos mejor.

2.4.1.a. Comando de acceso a fila

El comando de acceso a fila, o comando de activación de fila, tiene como objetivo llevar los datos de las celdas de una determinada fila hasta el array de amplificación, y después volverlos a colocar en las celdas correspondientes. Como parte de este comando, hay dos tiempos que deben tenerse en cuenta.

- t_{RDC} (*Row to Column Command Delay*) es el tiempo necesario para llevar los datos desde las celdas individuales al array de amplificación. Después del tiempo t_{RCD} es posible hacer una lectura o escritura de una columna mediante la conexión que existe entre el array de amplificación y el DMC a través del bus.
- Aunque tras t_{RCD} los datos ya están disponibles, es necesario volver a cargar la fila. Esta operación se puede realizar en paralelo con el movimiento de los datos a través del bus. Al tiempo invertido en el paso de los datos al array de amplificación y su retorno se le conoce como t_{RAS} (*Row Access Strobe latency*). Tras este tiempo, se ha completado el acceso y restauración de la fila, y el sistema está preparado para otro comando de acceso a fila.

2.4.1.b. Comando de lectura de columna

Tras el tiempo t_{RCD} en el comando de acceso a fila, es posible realizar un comando de lectura de columna. En este caso se copian los datos de los arrays de amplificación en el bus de datos para que el DMC los pueda recibir. Hay tres tiempos importantes en este comando:

- El tiempo t_{CAS} (*Column Access Strobe latency*) es el tiempo que le lleva a la DRAM colocar los datos en el bus tras la recepción del comando. Como normalmente se transmiten ráfagas de datos desde la aparición de las DRAM modernas, es necesario tener en cuenta los dos tiempos siguientes.
- El tiempo t_{CCD} (*Column to Column Delay*) es el tiempo invertido en el procesamiento de cada una de las pequeñas ráfagas de datos que se tienen que considerar internamente.
- El tiempo t_{BURST} es el tiempo que cada ráfaga de datos permanece en el bus para que el DMC lo pueda recibir correctamente. Suele ser mayor que t_{CCD}.

2.4.1.c. Comando de escritura de columna

En este comando hay un flujo de datos desde el DMC hasta el array de amplificación de la columna sobre la que se va a escribir. Las fases son similares a las del comando de lectura, pero con la dirección del movimiento de los datos invertida.

Los tiempos que se deben tener en cuenta son:

- El tiempo t_{CWD} (*Column Write Delay*) es el tiempo que ocurre entre la colocación de la petición para un comando de escritura en el bus de comandos y la colocación de los datos concretos a escribir en el bus de datos. Si los dos se envían a la vez, entonces $t_{CWD} = 0$.
- Inmediatamente tras t_{CWD} hay que volver a tener en cuenta t_{BURST}, el tiempo necesario que los datos deben estar en el bus hasta llegar correctamente al array de amplificación.
- En el caso de que después se vaya a realizar un comando de precarga debe respetarse el tiempo t_{WR} (*Write Recovery Time*). Este es el tiempo necesario que transcurre entre la llegada de los datos al array de amplificación, y su paso a las celdas de memoria.
- En el caso de que después se vaya a realizar un comando de lectura debe respetarse el tiempo t_{WTR} (*Write-to-Read Turnaround Time*). Este es el tiempo que transcurre hasta que los recursos derivados de la E/S son liberados.

2.4.1.d. Comando de precarga

El acceso a datos en una DRAM es un proceso en dos pasos. En primer lugar tiene el acceso a la fila que lleva los datos al array de amplificación, a lo que le sigue uno o varios comandos de acceso a columna. La segunda fase es el comando de precarga, que reestablece los sensores y arrays de amplificación y los prepara para otro acceso a fila. El tiempo asociado a este *reset* es t_{RP} (Row Precharge Time). Un comando de acceso a fila debe hacerse tras un comando de precarga.

Los dos tiempos implicados en los accesos a fila, t_{RAS} y t_{RP}, en ocasiones se combinan mediante una suma simple para formar t_{RC} (*Row Cycle Time*). Este es el mínimo tiempo que debe transcurrir entre dos accesos a filas diferentes en un mismo banco, y constituye la mayor limitación en cuanto a las velocidades que pueden alcanzar las DRAM.

2.4.1.e. Comando de refresco

Como hemos mencionado anteriormente, debido a la pérdida gradual de la carga en un condensador, los datos deben ser leídos y reescritos cada cierto tiempo para asegurar que no se pierden. Esta tarea está asociada al comando de refresco. Sin embargo, refrescar la información consume evidentemente ciertos recursos (energía y ancho de banda).

El tiempo entre dos comandos de refresco consecutivos para una misma fila debe ser menor que el tiempo que tarda la carga eléctrica en degradarse. El correcto uso de este comando garantiza

que los datos en una determinada fila son íntegros y válidos. Aunque depende del tipo de DRAM y de su arquitectura interna, típicamente se envían unos 8192 comandos de refresco cada 64 milisegundos.

Habitualmente existe un registro en la DRAM que contiene la última fila que se ha refrescado. El DMC envía un comando de refresco a la DRAM sin especificar una fila concreta, y el propio dispositivo incrementa el contador de fila en una unidad para saber a cuál toca aplicarlo a continuación. Además, el comando de refresco se realiza concurrentemente sobre todos los bancos. El tiempo invertido hasta que todos ellos llevan a cabo un refreso se conoce como t_{RFC} (*Refresh Cycle Time*).

2.4.1.f. Ciclo de lectura

Una vez que conocemos los comandos básicos en una DRAM, podemos preguntarnos cuál es el proceso completo de lectura. Un comando de acceso a fila lleva muchos bits de información a los arrays de amplificación en un banco determinado. Desde ahí, un comando de lectura de columna hace que algunos de esos bits pasen al bus de datos y el DMC los pueda recibir.

Tras el tiempo t_{RCD} en el que los datos son llevados al array de amplificación, el DMC envía el comando de lectura de columna, y concurrentemente, los datos se vuelven a restaurar en las celdas de memoria correspondientes. Tras t_{RAS}, el DMC envía un comando de precarga para preparar los arrays de amplificación para otro acceso.

En ciertas aplicaciones, como en *streaming* a través de una DRAM, es probable que a un cierto acceso de columna le siga un acceso a la siguiente (localidad espacial). En estos casos, hacer que los bits permanezcan en los arrays de amplificación evita una operación innecesaria de acceso a fila. Los sistemas de memoria con este comportamiento se conocen como *open-page*. Por otro lado, en aplicaciones donde estas situaciones no se suelen dar, se prefiere utilizar sistemas de memoria *close-page* que llevan a cabo un comando de precarga inmediatamente después del acceso a fila, para preparar el sistema para otro acceso.

2.4.1.g. Ciclo de escritura

Un ciclo de escritura es similar al de lectura, pero con la salvedad de que los datos que se vuelven a colocar en las celdas de memoria en el proceso de recuperación son distintos. Los datos son colocados en el bus por el DMC y pasan a través de la interfaz de E/S hasta llegar al array de amplificación y, después, a las celdas de memoria.

La consecuencia directa de esto es que el tiempo de un ciclo de fila está restringido por el tiempo del ciclo de escritura. Un ciclo de fila se define como la mínima cantidad de tiempo que tiene que transcurrir para que una DRAM pueda proporcionar el acceso a cualquier fila de cualquier banco. Todas las acciones relacionadas con la escritura deben ser llevadas a cabo antes de que el comando de acceso a fila termine, para poder hacer persistentes los datos, y antes de que el comando de

precarga se lleve a cabo. Formalmente, t_{RAS} debe ser suficientemente amplio como para verificar $t_{RAS} \geq t_{RCD} + t_{CWD} + t_{CCD} + t_{WR}$.

2.4.1.h. Comandos compuestos

A lo largo de la evolución de las DRAM, poco a poco se han ido diseñando comandos más complejos a partir de ciertas combinaciones de los básicos.

Un ejemplo es el comando de lectura de columna y precarga en los sistemas *close-page*. La precarga inmediata permite que el DMC pueda colocar otro comando de acceso a fila en el bus de comandos inmediatamente después, ya que de lo contrario habría que enviar una señal de precarga igualmente.

Otro ejemplo de comando complejo es el comando de acceso retardado a columna. No es más que un comando de acceso a columna que se pospone un cierto número de ciclos en el dispositivo DRAM. La ventaja es que el DMC puede colocar el comando de acceso a columna inmediatamente después del de acceso a fila (en caso contrario debería esperar un cierto tiempo), lo que simplifica el diseño del controlador de memoria.

2.4.2. Interacciones entre comandos

Hasta ahora sólo hemos tenido en cuenta la utilización de los recursos disponibles para determinar si dos comandos se pueden solapar en el tiempo o no. Sin embargo, no es la única limitación, pues incluso es posible que no esté permitido ejecutar un comando después de otro concreto en ciertas circunstancias. Las interacciones posibles entre diferentes comandos son necesarias en los sistemas DRAM modernos, especialmente en los *open-page*. La solicitud de un comando en estos sistemas depende del estado actual de la DRAM, y las posibilidades de interacción son también mayores.

Como ya se ha descrito anteriormente, los comandos de **lectura de columnas consecutivas** en un mismo rank, banco y canal se pueden combinar con ráfagas. Para ello, t_{BURST} debe ser mayor que t_{CCD}, para asegurar que la ráfaga se transmite correctamente antes de pasar a la siguiente. De igual forma, también se pueden programar comandos de escritura consecutivos.

Otro ejemplo de interacción delicada es el **comando de precarga inmediatamente después** del de lectura de una columna. Si no se respeta el tiempo suficiente como para colocar los datos en el bus de datos, y se resetea el array de amplificación antes de tiempo, los datos leídos por el DMC no serán correctos. El tiempo t_{RTP} es lo mínimo que hay que esperar entre un comando de lectura y de precarga.

Por otro lado, **las lecturas de diferentes filas** en un mismo banco necesitan varios comandos de acceso a fila, con el correspondiente incremento de coste en comparación con accesos a columnas de una misma fila. Para cada nueva fila, los datos de toda ella deben ser llevados a los arrays de amplificación, además del comando de precarga. En el mejor caso se puede realizar el comando

de precarga inmediatamente después del anterior acceso a fila. Sin embargo, estamos suponiendo que ha transcurrido un tiempo t_{RAS} desde el último acceso, necesario para que los datos se hayan devuelto a sus correspondientes celdas. En caso contrario, hay que esperar hasta que t_{RAS} se complete antes de hacer la precarga.

Es especialmente interesante el caso de múltiples **comandos de lectura o escritura sobre diferentes ranks**. En una lectura es posible que las diferentes peticiones no se puedan paralelizar, dependiendo del mecanismo de sincronización del sistema de memoria. En una escritura, por contra, se puede paralelizar dependiendo de las conexiones del bus. Si se producen simultáneamente varios comandos de lectura de columna a varios ranks, cada uno de ellos debe tomar el control del bus compartido, colocar los datos, y devolver el control para que otro rank lo tome. Las escrituras tienen más potencial de ser concurrentes puesto que es el DMC el único que mantiene bajo control el bus y no necesita cedérselo a ningún otro componente.

Otras situaciones que deben ser consideradas en el diseño de un sistema de memoria, aunque no serán explicadas en este texto con más detalle, son los **comandos de escritura consecutivos en un banco**, **escritura inmediatamente tras una lectura** y viceversa.

Además de las restricciones de tiempo que existen, de las cuáles sólo se han mencionado unas pocas, existen otros factores que limitan la capacidad de actuación de las DRAM. Un ejemplo es que, a medida que avanzan las investigaciones sobre nuevas memorias con mejores prestaciones, también lo hace la cantidad de energía que consumen y cada vez en mayor medida. Operar a mayor velocidad cuesta más, y en muchas ocasiones los propios ingenieros limitan el consumo de las DRAM y, con ello, sus capacidades reales.

2.5. Controlador de memoria DRAM

El controlador de una memoria DRAM, o DMC, tiene la responsabilidad de manejar internamente los dispositivos DRAM conociendo sus especificaciones concretas acerca de sus tiempos de respuesta, señales eléctricas, etc. El diseño de los DMC determina algunas de las características más relevantes de un sistema de memoria, como la latencia o el ancho de banda.

2.5.1. Arquitectura del controlador de memoria DRAM

La principal función del DMC es servir de interfaz para el acceso y escritura de los datos. Su diseño es complejo, debido a la complejidad intrínseca del protocolo de acceso a DRAM.

Un DMC se puede diseñar para minimizar el tamaño de las DRAM, su consumo eléctrico, maximizar el rendimiento, o alcanzar un equilibrio. Concretamente, en el diseño e implementación del DMC, son particularmente importantes los siguientes aspectos:

- Políticas de manejo de buffer de fila.

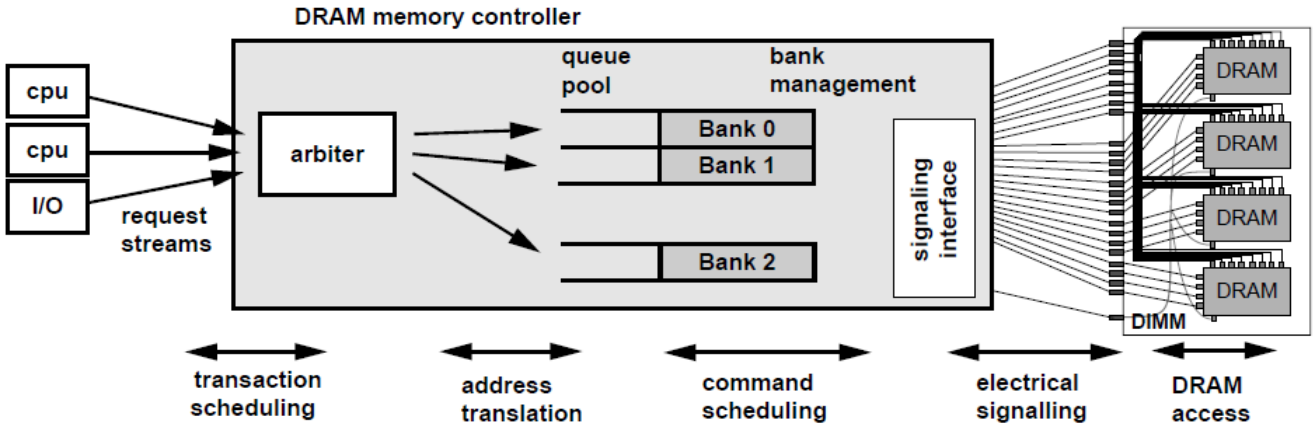


Figura 2.12: Arquitectura de un controlador de memoria DRAM. Fuente: [14].

- Esquema de traducción de direcciones de memoria.
- Ordenación de los comandos DRAM y transacciones de memoria.

Serán explicados posteriormente en esta sección.

La figura 2.12 muestra los componentes básicos de un DMC. Tanto la CPU como los dispositivos de E/S pueden realizar peticiones al DMC. Estas peticiones pasan primero por un árbitro que decide sobre la planificación y coloca las peticiones en una cola. La planificación puede ser de lo más variada. Por ejemplo, una petición de baja prioridad puede ser seleccionada sobre una de más prioridad si se realiza sobre la fila actualmente activa en un sistema *open-page*.

Cuando una petición gana el arbitraje y entra al DMC, se traduce a los comandos básicos de DRAM vistos en la sección anterior, que se colocan en una cola que maneja el DMC. Esta cola puede ser genérica para todo el sistema de memoria, o puede haber una cola para cada rank o cada banco. Cuando un comando está listo para ejecutarse, se envía la señal a cada DRAM.

2.5.2. Políticas de manejo de buffer de fila

Los arrays de amplificación en una DRAM pueden actuar como buffers que almacenan los datos de una fila. Existen dos políticas que ya han sido mencionadas anteriormente: *open-page* y *close-page*.

Los sistemas *open-page* se benefician de la localidad de los programas y no llevan a cabo una precarga tras un acceso a fila. Sólamente se resetean los arrays de amplificación en un acceso a otra fila diferente. En contraposición, los sistemas *close-page* se benefician de procesos con baja localidad, localidad poco predecible, o con pocos accesos a memoria.

No obstante, es conveniente mencionar algunas políticas híbridas que no adoptan un enfoque tan estricto como los sistemas *open-page* o *close-page*. Algunos procesos pueden exhibir cambios en la localidad dependiendo del momento de ejecución, y la utilización de la historia de ejecución

del programa parece relevante para decidir el mejor modo de acceso. Dos modos de implementar una política híbrica, aunque existen más, son los siguientes.

- Llevar un contador del número aciertos y fallos de fila consecutivos. Un acierto de fila es cuando se accede a la misma fila que el acceso anterior, mientras que un fallo es evidentemente la situación contraria. Si en un momento dado el sistema funciona como *open-page*, pero hay un número muy elevado de fallos de fila, entonces se podría pasar a un sistema *close-page*. Por otro lado, si estamos en un *close-page* pero se está accediendo repetidamente a una misma fila, pasaríamos a un *open-page*.

- Utilizar un temporizador que se restablece a un valor predefinido cada vez que se produce un acceso a una fila distinta. Con cada ciclo de reloj, el temporizador se va decrementando en una unidad. Si llega a cero, se lleva a cabo un comando de precarga para resetear el array de amplificación.

La elección del tipo de política determina en gran medida el diseño del esquema de traducción de direcciones, la ordenación de los comandos, y el mecanismo de reordenación de transacciones. Además, también provoca cambios en el rendimiento y consumo eléctrico de la DRAM. Por ejemplo, un sistema *close-page* puede ser adecuado si no se realizan demasiadas peticiones al dispositivo. En caso de usar un *open-page*, el coste de mantener los datos en el array de amplificación demasiado tiempo sin utilizarlos no compensa el coste de precarga y acceso a fila.

2.5.3. Esquema de traducción de direcciones de memoria

Además de las políticas de manejo de fila, la forma en la que se traduce una dirección de memoria para determinar su ubicación física afecta directamente al rendimiento del sistema completo. Cuando el DMC recibe una petición sobre una dirección concreta, tiene que convertir esa información en términos de cuál es el canal, rank, banco, fila y columna donde están los datos. A este proceso también se le conoce como mapeo de direcciones.

Consideremos el caso de un sistema de memoria donde no se da demasiada importancia al mapeo o traducción de direcciones. Es posible que una cierta aplicación solicite accesos a memoria con direcciones físicas muy parecidas, pero que resulten en distintas filas de un mismo banco debido al mecanismo de traducción. Esto hace que haya muchos **conflictos** en un mismo banco, que se producen cuando se quiere acceder a una fila distinta y se necesita un comando de precarga y de acceso a fila. Evidentemente, todo este proceso conlleva una disminución en el rendimiento si lo comparamos, por ejemplo, con un sistema donde el mapeo coloca datos "adyacentes" en distintas filas de diferentes bancos. De esta última forma, el acceso se puede llevar a cabo con cierto grado de paralelismo a la vez que se minimiza la probabilidad de conflictos en un banco.

Al contrario que las políticas de manejo de buffer de filas, un mapeo de direcciones no puede ir cambiando sobre la marcha y se necesita decidir a priori. En los subsiguientes

apartados, se considera un sistema de memoria convencional donde el procesador opera en un espacio de direcciones virtuales y un pequeño TLB[1] permite convertiras a direcciones físicas, independientemente del esquema de traducción de bajo nivel dentro de las DRAM. El objetivo será determinar a partir de la dirección física, los índices que permitan localizar unívocamente la posición de los bytes de información.

2.5.3.a. Posibles formas de paralelización

La posible paralelización que existe en un sistema de memoria determina los esquemas que se utilizan para el mapeo o traducción de direcciones.

A nivel de *canal* se tiene el mayor grado de paralelización, pues no existen restricciones en diferentes canales lógicos gobernados por diferentes DMC. Dos controladores de memoria independientes pueden operar de forma totalmente concurrente si cada uno de ellos tiene habilitado un canal.

Los accesos paralelos a nivel de *rank* pueden ocurrir en un mismo canal, pero siempre sujeto a la disponibilidad de los recursos adecuados: buses de datos, comandos, etc. Sin embargo, existen penalizaciones al llamar de forma consecutiva a comandos sobre diferentes ranks por el coste de cambio en las activaciones.

De forma similar, los accesos consecutivos a memoria en distintos *bancos* están sujetos a la disponibilidad de los recursos. Planificar comandos consecutivos entre bancos de un mismo rank no tiene el coste que sí tiene el cambio de rank, puesto que no es necesaria una transferencia de control del bus entre ranks. Cambios entre bancos son preferibles antes que cambios entre ranks.

A nivel de *fila*, no es posible realizar accesos simultáneos en un mismo banco. Los accesos a *columnas* deben ser secuenciales, aunque en varias ocasiones se ha discutido la ventaja de realizar accesos a columnas consecutivas en una misma fila en forma de ráfagas.

2.5.3.b. Esquemas de traducción

Supongamos un sistema de memoria que tiene K canales independientes que pueden ser gobernados por uno o varios DMC. Cada uno de estos canales está conectado a uno o más módulos de memoria que están formados por L ranks, y cada rank a su vez contiene B bancos. En cada uno de los bancos se suponen R filas y C columnas, y se almacenan V bytes en cada par fila-columna. El tamaño total de la memoria sería, bajo estas circunstancias, $K \cdot L \cdot B \cdot R \cdot C \cdot V$.

En una jerarquía de memoria, el acceso a una cierta parte del mapa de una DRAM conlleva elevar una porción de datos más amplia para aprovechar la localidad. Supondremos que la granularidad de los accesos sobre este sistema de memoria es a nivel de bloque de caché. El

[1] Un TLB, o *Translation Lookaside Buffer*, es una memoria caché que contiene partes de la tabla de paginación, la cual relaciona las direcciones virtuales con las físicas. Se utiliza para poder traducir rápidamente direcciones, aunque en muchos casos es necesario revisar la tabla de paginación porque la entrada no se encuentre en el TLB.

número de bloques de caché disponibles en una fila se denotará por N, y el número de bytes de cada uno de los bloques será Z. Supondremos que la relación $N = CV/Z$, que permite determinar el número de bloques de caché de cada fila, produce un número entero. En sistemas sencillos se suele dar el caso de $N = 1$, aunque no tiene por qué ser así.

Por simplicidad, supondremos que todos los parámetros definidos son potencias de dos. Se utilizan las letras minúsculas para referirse al logaritmo en base dos de la correspondiente mayúscula. Por ejemplo, si consideramos B bancos por cada rank, $b = \log_2 B$ es el número de bits necesarios para representar la dirección del banco. En general, si el tamaño de la memoria es $K \cdot L \cdot B \cdot R \cdot C \cdot V$, se necesitarán $k + l + b + r + c + v$ bits para direccionar unívocamente cada byte de la memoria.

Para concluir con la descripción de la notación utilizada, se utilizarán los dos puntos ":" para separar los bits en una dirección de memoria. De esta forma, $[k : l : b : r : c : v]$ no sirve sólo para informar sobre el tamaño de la memoria, sino para indicar qué porciones sirven para identificar cada elemento en el esquema de traducción. De nuevo, por simplicidad, y aprovechando la igualdad $CV = NZ$, se separará la dirección en los campos $[k : l : b : r : n : z]$ en lugar de tener en cuenta cada columna exacta.

En primer lugar trataremos la **traducción en un esquema open-page**. Para maximizar el rendimiento, los bloques de caché consecutivos se dividen en diferentes canales para que el acceso a datos consecutivos de memoria se pueda realizar con el máximo nivel de paralelización. Además, se prefiere que los distintos bloques de caché consecutivos se mapeen en la misma fila del mismo banco y del mismo rank. El esquema de mapeo que permite esto es $[r : l : b : n : k : z]$. Los primeros bits de una dirección de memoria indican la fila, mientras que los siguientes el rank y el banco. Para permitir que los bloques de caché consecutivos se paralelicen, los bits que indican el canal se colocan después del bloque de caché. Los últimos z bits de dirección sólamente sirven para determinar el desplazamiento dentro del bloque de caché. Para un bloque de caché cualquiera, el siguiente se corresponderá con incrementar en z unidades la dirección $[r : l : b : n : k : 0]$. El campo modificado será el referido al canal, de forma que el acceso a otro canal se pueda realizar de forma inmediata.

Por otro lado hablaremos de la **traducción en un esquema close-page**. De forma similar al anterior, interesa que direcciones consecutivas se mapeen en canales diferentes para maximizar la concurrencia. Sin embargo, en un sistema *close-page*, mapear bloques de caché consecutivos en el mismo rank, banco y fila llevaría, en caso de localidad espacial, a múltiples conflictos en un mismo banco, siendo necesario esperar a que se lleve a cabo el comando de precarga. Si el acceso fuera a otra fila, otro banco u otro rank, no sería necesario esperar esta cantidad de tiempo, puesto que la precarga ya estaría hecha de antemano. Para minimizar las probabilidades de un conflicto, los bloques adyacentes se mapean primero en distintos canales, después en distintos bancos, y después en distintos ranks. Recordemos que los cambios entre bancos son mucho más rápidos que entre ranks, y de ahí el orden de preferencia. El resultado sería un esquema de direcciones $[r : n : l : b : k : z]$.

2.5.3.c. Paralelismo vs extensión

Una característica fundamental de los sistemas modernos es su extensibilidad. Cualquier usuario puede modificar el sistema de memoria principal simplemente quitando o añadiendo módulos de memoria. En términos del esquema de traducción de direcciones, esto quiere decir que los rangos de direcciones de canal, fila, columna, rank y banco deben ser lo suficientemente flexibles como para no perjudicar la extensibilidad.

Supongamos un sistema en el que se pueden insertar módulos con uno o varios ranks. Interesa que los bits de direcciones que indican el rank sean los más altos posibles, para minimizar el impacto de su cambio. Mover los bits de los ranks a posiciones altas hacen que, puesto que típicamente se utiliza una porción secuencial del espacio de direcciones, sólo se terminen usando unos pocos ranks de todos los que hay disponibles. Este sistema pierde algo de concurrencia a nivel de rank en los accesos a memoria.

De forma similar, supongamos otro sistema donde los canales se puedan configurar independientemente de los módulos de memoria. En este caso las direcciones más altas estarían reservadas para la selección de canal, y el posible paralelismo a nivel de canal (y, desgraciadamente, el más fácil de explotar) no estaría disponible para aplicaciones individuales.

En los esquemas de traducción propuestos en la sección 2.2.3, la información sobre el canal estaba reservada en bits de órdenes bajos y medios. Sin embargo, para facilitar la extensibilidad, habría que modificar esta prioridad para colocarnos en posiciones más altas. Una posible modificación para sistemas *open-page* sería $[k : l : r : b : n : z]$, correspondiente a priorizar el canal y el rank respectivamente, y mantener el mismo orden de las demás porciones. En un sistema *close-page* se podría utilizar $[k : l : r : n : b : z]$.

Debido a la pérdida de paralelismo en favor de la extensibilidad, es necesario llegar a un consenso en los sistemas que son finalmente distribuidos. Por ejemplo, si se va a diseñar un pequeño computador para controlar un cierto sistema y no se prevee que deba ser modificado, es interesante priorizar el rendimiento.

2.5.4. Optimización del rendimiento

El rendimiento que ofrece un sistema de memoria también se ve afectado por las políticas de ordenación de transacciones de memoria y de comandos. Una transacción de memoria es una petición que un dispositivo realiza al DMC a alto nivel, sin conocer la organización exacta de las DRAM que componen el sistema de memoria. Cada transacción de memoria resulta en varios comandos al traducirse en un DMC.

Es posible diseñar un sistema para priorizar el rendimiento a pesar de que las políticas de reordenación sean muy complejas, o por otro lado limitar la complejidad de las mismas a costa de un menor desempeño. No existe un algoritmo ideal para todas las situaciones, pues aunque la elección depende fuertemente de la organización física del sistema de memoria, también depende

de los requerimientos de cada aplicación. En algunos casos se permite que el procesador pueda cambiar directamente la política de planificación para mejorar el rendimiento.

En esta sección examinaremos diferentes formas de mejorar el rendimiento de una DRAM, entre las que se encuentran el *write caching*, la organización en colas de las peticiones, los mecanismos de refresco, y políticas de acceso compartido.

2.5.4.a. Write caching

Las operaciones de escritura son no críticas en términos de latencia, al contrario de las de lectura. Es preferible aplazar las operaciones de escritura en favor de las de lectura, siempre que el modelo de ordenación soporte esta optimización y se pueda asegurar un funcionamiento consistente del programa. Esto implica que, si se accede a una posición de memoria que debería haber sido modificada pero que aún no se ha hecho persistente, la lectura debería garantizar la utilización del nuevo dato de algún modo.

Las DRAM incurren en una penalización al realizar una petición de lectura tras una de escritura debido al "cambio de sentido" de los buses para transferir información. En consecuencia, un comando de lectura de columna tras uno de escritura de columna tiene una penalización de tiempo hasta que los recursos estén disponibles. Las técnicas de *write caching* dan prioridad a todas las peticiones de lectura sobre las de escritura. Cuando se ha acumulado una cierta cantidad de escrituras pendientes en un pequeño buffer, se escriben todas, aprovechando los mecanismos ya descritos de transferencias de ráfagas de datos.

No obstante, a medida que se acumulan peticiones de escritura, las de lectura tienen cada vez una mayor latencia. Para asegurar que los datos utilizados sean correctos, es necesario comparar las direcciones de lectura de cada petición con todas las escrituras pendientes, incrementando el tiempo de acceso.

2.5.4.b. Colas de peticiones

Las transacciones o peticiones de memoria se traducen en secuencias de comandos DRAM, y algunos de estos comandos se pueden ejecutar simultáneamente. Para facilitar la planificación, los comandos se colocan en una o varias colas donde se organizan teniendo en cuenta las restricciones que haya, siempre garantizando el correcto funcionamiento de la transacción. El controlador puede priorizar algunos comandos sobre otros, por ejemplo, en base a los siguientes criterios.

- Prioridad de la transacción de memoria.
- Disponibilidad de recursos.
- Dirección de banco de la petición.
- Tiempo de espera de la petición (para evitar la inanición).

- Historial de acceso del agente que realizó la petición.

Una posible organización por colas es la que mantiene una cola de comandos para cada banco del sistema de memoria. La figura 2.13 muestra este esquema. En esta organización se supondrá que todas las peticiones tienen la misma prioridad.

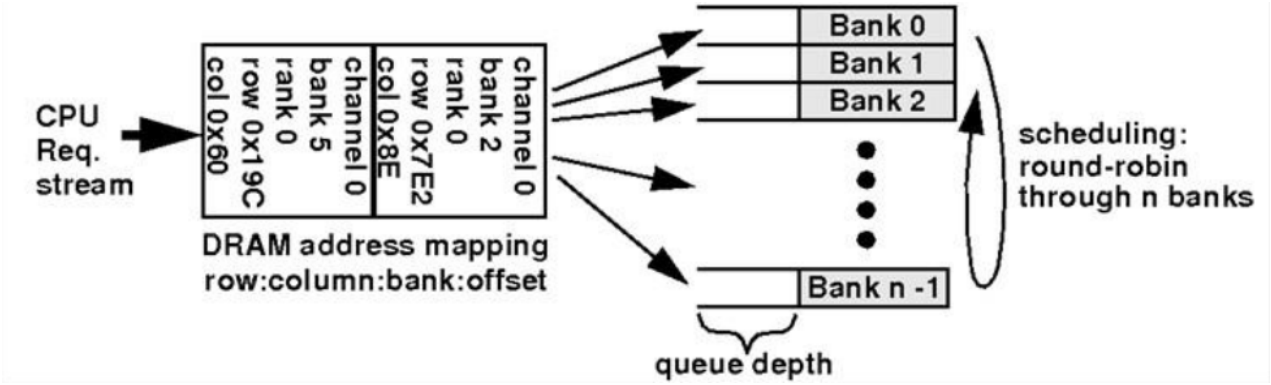


Figura 2.13: Organización de colas de peticiones por bancos.

Una transacción de memoria que llega al controlador se debe traducir primero a una serie de comandos, y después traducir la dirección virtual de memoria a la dirección física. En función del banco, cada comando DRAM se coloca en una de las colas que administra el DMC. Si se trata de un sistema *open-page*, por ejemplo, se pueden dirigir todas las peticiones de acceso a columna de una misma fila para ahorrar tiempo. Cuando se hayan satisfecho estas peticiones y sea necesario cambiar de fila, se puede hacer una precarga de los arrays de amplificación y, en lo que dura esta operación, pasar a las operaciones pendientes de otro banco.

Una forma de planificar el tiempo que se está con cada cola de peticiones (es decir, con cada banco), es mediante una adaptación del conocido Round-Robin. Se puede establecer un número de ciclos o de comandos que se atienden en cada cola, y después pasar a la siguiente. Se ha comprobado experimentalmente que se reduce significativamente el número de conflictos de banco siempre que haya suficientes operaciones en las demás colas para llevar a cabo una precarga completa antes de volver al mismo banco. En un sistema *close-page* se maximiza el tiempo entre accesos a un mismo banco, por ejemplo, si limitamos a uno el número de comandos por cada cola activa. En un sistema *open-page* no siempre se producen buenos resultados, puesto que normalmente se esperan varios accesos a columnas por cada fila activa y la política Round-Robin puede cortar esta secuencia por la mitad. En sistemas *open-page* son mejores políticas basadas en prioridades que en Round-Robin.

2.5.4.c. Mecanismos de refresco

Para asegurar la integridad de los datos almacenados, se necesitan comandos de refresco cada cierto tiempo. En función de quién encarga los comandos de refresco podemos distinguir dos situaciones. En la primera y menos común, cada DRAM es responsable de refrescar sus propias

celdas. Esto proporciona un controlador de memoria más simple y la apariencia de que la memoria DRAM no es dinámica a más alto nivel, ya que el DMC no sabe nada sobre refrescos.

En la segunda, es el controlador de memoria envía los comandos de refresco. Una DRAM entera debe ser completamente refrescada cada **período de refresco**, que típicamente son 32 o 64 milisegundos. Supongamos que se necesitan 8192 comandos de refresco por cada período de refresco, como ocurre en muchas DRAM. El DMC estaría programado para enviar un comando de refresco a todos los bancos que forman el sistema (concurrentemente) cada $7.8\mu s$. Cada banco lleva un contador que indica qué fila debe refrescarse, por lo que el DMC no debe preocuparse por eso.

Los comandos de refresco se pueden aplazar ligeramente en favor de otras peticiones de lectura o escritura que tengan ciertos requisitos sobre latencia máxima. El DMC podría mantener tres grupos de colas: una para lecturas, otra para escrituras, y otra para refrescos. Las peticiones de las dos primeras provienen de fuera, mientras que la cola de comandos de refresco se genera internamente. Cada comando de refresco tiene un contador que indica el número de ciclos que ha sido aplazado. En condiciones normales, si existen peticiones de lectura o escritura en el sistema, éstas se priorizan sobre el refresco, a no ser que el contador de retardo de refresco supere un valor umbral. En este caso, se asigna una máxima prioridad al comando de refresco.

2.5.4.d. Acceso compartido

En un sistema donde existan varios agentes que actúen sobre el mismo sistema de memoria, se necesita garantizar que el mecanismo de ordenación de comandos no prolonga indefinidamente alguna petición. Los agentes pueden referirse a algunos procesadores que actúan sobre el mismo sistema de memoria, o a un procesador ejecutando muchos procesos.

En ciertos sistemas es mejor priorizar un acceso igualitario de todos los agentes al sistema de memoria, que priorizar el rendimiento global de la DRAM. Esto último puede lograr una máxima utilización de la memoria, pero puede que el reparto del uso de la misma no sea equitativo entre los agentes.

La aproximación más simple es utilizar ideas parecidas a los mecanismos e refresco. Si consideramos que existen varios agentes que insertan comandos DRAM a intervalos regulares de tiempo, podemos añadir un contador a estas peticiones y asignarles más prioridad cuando el contador pasa de un umbral. La similitud que existe con el refresco es evidente, pues se podría considerar al propio DMC como un agente más que inserta comandos de refresco en el sistema.

Mecanismos más soficsticados tienen en cuenta que no todas las peticiones tienen la misma prioridad, ni todos los agentes las mismas necesidades de latencia o ancho de banda, pero estas extensiones no serán consideradas en este trabajo.

Capítulo 3

Accesos a memoria fuera del chip

Una vez detalladas las características más importantes de las memorias caché y de las memorias DRAM, vamos a describir las herramientas software necesarias para la obtención de datos sobre accesos a memoria. En el proyecto que nos atañe, vamos a utilizar la herramienta Intel® Pin para simular un sistema que esté compuesto por tres niveles de memoria caché (véase la figura 3.1). Supondremos un sistema multiprocesador donde cada una de las CPU tiene una memoria caché L1 y una L2. Una caché de último nivel (LLC) sirve a todos los procesadores.

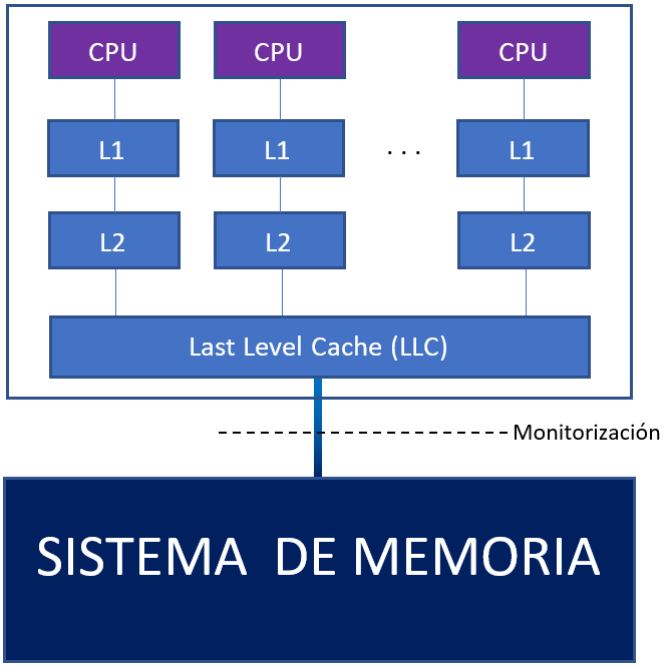


Figura 3.1: Esquema de la simulación y nivel de monitorización de Pin

Durante la ejecución normal de un programa cualquiera, si se necesita un dato que no está presente en la LLC, es necesario acceder al nivel inferior de la jerarquía de memoria donde se encuentra la RAM. Ahora mismo no es relevante qué tipo de memoria RAM constituye el sistema de memoria ni cómo esté organizado.

Pin se utilizará para monitorizar las peticiones que se hagan al sistema de memoria debidos a fallos de caché. El objetivo es disponer de un fichero de datos con el histórico de las operaciones enviadas al DMC. Con esta información es posible realizar simulaciones de diferentes esquemas de memoria principal, y concretamente la que nos interesa en este trabajo: una SDRAM sobre una RRAM.

Las operaciones que se pueden llevar a cabo sobre el sistema de memoria principal son las siguientes:

- Operación de lectura. Si en un cierto momento se necesita un dato que no está en ninguno de los niveles de caché, se necesita realizar una operación de lectura sobre la memoria principal.
- Operación de escritura. Si se necesita escribir un cierto dato que no se encuentra presente en la memoria caché, hay que realizar una petición de escritura al sistema de memoria.
- Una operación de reemplazo tiene lugar cuando un elemento de la caché LLC marcado con el *dirty bit* (esto es, contiene información modificada que aún no se ha hecho persistente en memoria principal) es desalojado de la misma. Es necesario realizar una operación de escritura sobre el sistema de memoria para garantizar la persistencia de la información modificada.

Las operaciones de lectura y de escritura serán las más interesantes, pues reflejan el verdadero comportamiento de la aplicación. En una operación de lectura se elevan los datos y porciones cercanas a la memoria caché, desalojando otras entradas si es necesario. En una operación de escritura, aunque pueda sonar contradictorio, también se produce una lectura de los datos en la memoria caché. La modificación solicitada (la escritura) se hace sobre los datos que almacena la caché, y muy raramente sobre memoria principal. Cuando esta entrada sea desalojada, dicha modificación sí se hará en memoria principal.

Una razón por la que resultaría conveniente que la escritura se hiciera directamente sobre el sistema de memoria es el caso en el que se tengan múltiples sistemas trabajando sobre una memoria compartida. Para asegurar la integridad de los datos en cualquier momento, es necesario que las modificaciones se hagan sobre memoria principal. Salvo que se indique lo contrario, no supondremos este tipo de comportamiento en nuestro trabajo.

Por otro lado, las operaciones de reemplazo no conllevan una lectura desde el sistema de memoria. No dependen de los accesos de la aplicación, sino de la estructura de la memoria caché y los desalojos producidos. En muchos casos interesará tratarlas aparte para estudiar por separado las secuencias de lecturas y escrituras, sin que éstas resulten empañadas por los reemplazos que, por otra parte, son mucho más impredecibles.

El objetivo de esta sección es convertir los fallos de la caché en un fichero de datos tratable para la posterior agrupación de aplicaciones en función de su comportamiento, así como el diseño de un mecanismo de prebúsqueda. Primeramente, se describirá la herramienta Intel® Pin, utilizada para obtener este fichero. Seguidamente, hablaremos del formato de los datos obtenidos.

3.1. Intel® Pin

Intel® Pin es una herramienta bajo la licencia de Intel® centrada en el análisis de la ejecución de cualquier proceso. Tiene soporte en sistemas Linux, OS X y Windows, sobre ejecutables creados para la arquitectura IA-32, Intel® 64 y ciertas arquitecturas Intel® integradas. Para una mayor información sobre Pin, consúltese el manual de usuario disponible desde la página oficial de Intel® [16].

Pin permite insertar código (cualquier tipo de código) escrito en C o C++ en cualquier lugar durante la ejecución de un programa. El código se añade dinámicamente mientras el proceso principal está en ejecución. Pin proporciona un conjunto de APIs que permiten abstraerse de las particularidades de un juego de instrucciones para realizar ciertas operaciones. Por ejemplo, es posible pasar el contenido de los registros en un determinado momento como parámetros al nuevo código insertado.

Pin automáticamente guarda el estado de ejecución de un programa (contenido de los registros, contador de programa, etc.) para la ejecución del código inyectado, y después recupera el estado original para continuar con la ejecución del proceso principal. Esto permite un análisis de la mayoría de los aspectos de un proceso que se ejecuta sobre un sistema, como por ejemplo generar una traza minuciosa de la ejecución del programa, o realizar simulaciones de comportamiento sobre cachés virtuales. Algunos ejemplos de preguntas que se pueden resolver utilizando Pin podrían ser determinar el número de instrucciones ejecutadas en un programa, o cuántas ramificaciones (saltos de programa) se llevan a cabo.

3.1.1. Funcionamiento básico

La manera más sencilla de pensar en Pin es como un compilador "Just-In-Time", que recibe como entrada un ejecutable binario. Pin intercepta la ejecución de la primera instrucción y genera ("compila") un nuevo código compuesto por la instrucción original y las secuencias de control inyectadas que se deseen. Entonces transfiere el control a la secuencia generada. Una vez que esta secuencia ha terminado, el control es devuelto a Pin. Pin examina el siguiente código que será ejecutado e inserta el que sea conveniente en cada caso, repitiéndose así el proceso.

En el modo JIT (Just-In-Time) sólo se generan secuencias para el código que es ejecutado. Esto significa que, si hay ciertas partes que el programa nunca ejecutará, Pin nunca las examinará ni generará secuencias que lo contengan. El código original sólo se puede usar de referencia. A medida que Pin va generando código, se le brinda al usuario la oportunidad de inyectar su propio código entre instrucciones.

Pin sólo es capaz de examinar las instrucciones que realmente se ejecutan, independientemente de dónde se encuentren o cuándo se ejecuten realmente, aunque hay ciertas excepciones en saltos entre partes del programa.

3.1.2. Pintools

La instrumentación o control de un proceso tiene en cuenta dos aspectos:

- El mecanismo que decide dónde y qué código se debe ejecutar.
- El código que se ejecuta en el punto de inserción.

Al código relevante sobre la primera parte se le conoce como **código de instrumentación**, mientras que el del segundo aspecto es referido como **código de análisis**. Los dos componentes se guardan en un único ejecutable final, llamado **Pintool**. Un Pintool es como un plug-in que regula la generación de código de Pin.

El Pintool registra las funciones y rutinas que se deben llamar en cualquier punto donde se deba inyectar código. A este aspecto es a lo que hemos llamado el componente de instrumentación. Se encarga de inspeccionar el código que será generado, sus propiedades, y decidir si inyectar llamadas a rutinas de análisis.

Por otro lado, el componente de análisis es el que contiene el código propiamente inyectado. Puede tener cualquier propósito: desde simular el comportamiento de una caché para averiguar la tasa de fallos de página (un fallo se produce cuando los datos no se encuentran en la caché y deben ser traídos de memoria principal) hasta simplemente guardar en un fichero la traza completa del programa. Esta parte está bajo el control de Pin, que se asegura de guardar el estado de ejecución del programa antes y después del código inyectado para no alterar su ejecución normal.

Un Pintool también puede registrar rutinas en función de cuando se producen ciertos eventos, como la creación de varios hilos en un programa o tras una llamada a *fork*.

3.1.3. Observaciones sobre Pin

Algunas consideraciones importantes que se deben tener en cuenta sobre Pin y un Pintool son las siguientes.

- Un Pintool funciona como un plug-in del ejecutable original, por lo que utiliza el mismo espacio de direcciones que Pin y que el ejecutable a ser monitorizado. Por lo tanto, el Pintool tiene acceso a todos los datos que maneja el ejecutable.
- Pin y los Pintools monitorizan el comportamiento del ejecutable desde la primera instrucción. Para ejecutables que se compilan con librerías externas, esto implica que dichas librerías también serán visibles por el Pintool, ya que la carga dinámica de los recursos suele ser lo primero que hace un proceso.

- Al crear un Pintool es más importante centrarse en el código de análisis que en el de instrumentación. El de instrumentación simplemente redirige las llamadas a las funciones de análisis en el momento oportuno, y estas funciones serán llamadas (previsiblemente) muchas veces. Es mejor centrarse en mejorar este código, que se repetirá en varias ocasiones.

La instrumentación de Pin es Just-In-Time, lo que quiere decir que Pin toma el control cada cierto tiempo para insertar código de análisis en las partes que se consideren convenientes. En función de dónde Pin toma el control de la aplicación (o mejor dicho, la parte de instrumentación de Pin), podemos distinguir diferentes modos de funcionamiento.

La **instrumentación de instrucción** hace que Pin tome el control antes de que se ejecute cada instrucción. Pin decide en ese momento si se realizará una llamada al código de análisis o, por el contrario, se continua la ejecución normal del programa. Para la ejecución de cada instrucción es necesario que Pin tome el control, guarde el estado actual del proceso, examine la instrucción, decida qué tipo de llamada de análisis se debe realizar, recupere el estado del proceso, devuelva el control al proceso original, y se ejecute la instrucción. Esto conlleva un evidente aumento del tiempo de ejecución del programa.

La **instrumentación de traza** no tiene por qué detenerse en cada instrucción. La instrumentación ahora tiene lugar a nivel de BBL (*Basic Block*). Un BBL es una secuencia de instrucciones que está perfectamente delimitada por un punto de entrada y un punto de salida. Un punto de entrada será típicamente la primera instrucción a ejecutar tras un *jump*, y una salida que será un *jump*, *call* o *return* generalmente. Pin toma el control en cada ramificación que hay y examina el siguiente BBL a ejecutar. Pin puede decidir insertar código entre instrucciones (en cuyo caso estaríamos en un modelo parecido a la instrumentación de instrucción), o simplemente inyectar código de análisis al principio del BBL.

La **instrumentación de rutina** hace que Pin tome el control cada vez que una nueva rutina es cargada en memoria. Puede inyectarse código al principio de la rutina que ha sido llamada, pero también se puede realizar un análisis a nivel de instrucción. No obstante, Pin no puede detectar BBLs trabajando en este modo.

El último modo de funcionamiento se conoce como **instrumentación de imagen**, y Pin sólo toma el control al inicio de la ejecución, cuando se crea la imagen del proceso en memoria, y puede insertar código al principio. También puede programarse para llamar a una función de análisis en cada llamada a una rutina o, igual que en todos los casos anteriores, llamar a una misma función antes de cada instrucción. A este nivel tampoco puede discriminarse entre BBLs.

3.1.4. Propósito para utilizar Pin

La herramienta Intel® Pin se utilizará en un contexto de simulación de los niveles superiores de una jerarquía de memoria. Se desea monitorizar el número y tipo de accesos a memoria de una aplicación cualquiera, para generar un fichero de traza donde el comportamiento del proceso esté

descrito por las solicitudes al sistema de memoria principal. Con estos datos, es posible simular el comportamiento de un sistema de memoria con parámetros arbitrarios, puesto que se tiene la secuencia de operaciones solicitadas, así como el instante temporal en que se realiza.

Concretamente, la infraestructura de Pin nos permitirá crear un script para simular un sistema de caché de tres niveles, donde la última caché sea compartida por todos los procesadores. Las operaciones a monitorizar serán, sobre todo, las de lectura y escritura, aunque también son de especial interés aquellos casos en los que el desalojo de una entrada de caché provoca una operación de escritura si el *dirty bit* está activo. A este último tipo se le conocerá como **reemplazo**.

3.2. Fichero de traza y aplicaciones monitorizadas

La monitorización que se hace de Pin es a nivel de instrucción, lo que significa que se examina individualmente cada una de las ejecutadas por la aplicación. Si la instrucción resulta ser una petición al sistema de memoria, se recogen los datos relevantes de dicha instrucción.

El programa Pin se puede ejecutar en distintas condiciones. Entre ellas, se examinarán las siguientes configuraciones:

- Puede elegirse el tamaño de la LLC entre dos posibles: 16MB o 32MB.
- Puede elegirse el tipo de asociatividad de la LLC. Una posibilidad es que sea asociativa por conjuntos de nivel 1, es decir, cada bloque de caché sólo puede ser localizado en una única entrada. La otra posibilidad es que la asociatividad sea por conjuntos de tamaño 8, donde cada bloque de caché, una vez determinado el conjunto al que pertenece, puede ser colocado en cualquiera de las 8 entradas.

El programa no está preparado para realizar modificaciones sobre los tamaños o asociatividad de las cachés L1 (32KB) y L2 (2MB).

La salida del programa Pin será un archivo de traza *.nvt* que contendrá información sobre todos los accesos a memoria principal. Será una secuencia de entradas, donde cada entrada esté formada por:

- Número de ciclo donde se ha llevado a cabo la petición.
- Tipo de la petición (*R*, Read; *W*, Write).
- Dirección de memoria donde se dirige la petición.
- Datos involucrados en la transacción.

Para el agrupamiento y descubrimiento de patrones de accesos a memoria se han elegido ciertas aplicaciones para ser monitorizadas. La lista completa puede ser consultada en la tabla 3.1. Todas ellas pertenecen al repertorio estándar SPEC2006 excepto Firefox, Openoffice, gcc y g++.

Astar	Bzip2	Firefox	g++
gcc	gobmk	h264ref	hmmer
lbm	libquantum	mcf	milc
namd	omnetpp	Openoffice	povray
sjeng	specrand	sphinx3	Xalan

Tabla 3.1: Listado de aplicaciones a monitorizar.

Bajo el script de Pin preparado para simular una jerarquía y monitorizar los accesos a memoria, las 20 aplicaciones han sido ejecutadas. Para cada una de ellas, se han tenido en cuenta cuatro posibles escenarios:

- LLC de 16MB y asociatividad de nivel 1.
- LLC de 16MB y asociatividad de nivel 8.
- LLC de 32MB y asociatividad de nivel 1.
- LLC de 32MB y asociatividad de nivel 8.

De esta forma, en total se generan 80 **ficheros de traza** con extensión *.nvt*. Para cada acceso a memoria interesa guardar el número de ciclo en el que se produce, de qué tipo es, y a qué dirección de memoria se dirige la petición. Un ejemplo de comienzo de un fichero de traza se puede encontrar en la tabla 3.2. En ella se pueden comprobar los cuatro primeros accesos a memoria que realiza Firefox cuando la LLC tiene 16MB y la asociatividad es de nivel 1.

Ciclo	**Tipo**	**Dirección**
1	R	0x7f487cb82c00
2	W	0x7ffecf8e0240
3	R	0x7f487cb83980
16	R	0x7f487cda7e40

Tabla 3.2: Comienzo del fichero de traza .nvt para Firefox en el caso de 16MB de LLC y asociatividad de nivel 1.

Dado que una aplicación puede generar muchísimos accesos a memoria, el fichero de traza puede ser sorprendentemente grande. En el caso del fichero de traza de Firefox, ocupa algo más de 1GB, ya que ha realizado más de 7 millones de accesos a memoria. Otras aplicaciones pueden generar menos accesos, aunque hay algunas cuyo tamaño de fichero de traza sobrepasa los 20GB. En general, cuanto mayor es el tamaño del fichero de traza, es porque la aplicación es más exigente, ha estado en ejecución durante más tiempo, y por tanto ha realizado más accesos a memoria.

Debido al enorme tamaño del fichero de traza, es necesario resumir la información de alguna manera para aprovechar no sólo mejor el espacio, sino también mejorar la legibilidad e

interpretabilidad de la información. En este contexto se propone una transformación del fichero *.nvt* en otro con extensión *.outA* a partir de un script de Python.

En el fichero *.outA* se quiere almacenar la información sobre las peticiones de acceso a memoria, agrupando todas aquellas que se refieran a una misma dirección. Este fichero será referido a partir de ahora como **fichero de accesos a memoria**, o simplemente **fichero de accesos**. El objetivo es disponer de la información de la siguiente forma:

Dirección de memoria - (número de acceso, tipo) [(número de acceso, tipo)] ... (3.1)

Para cada dirección de memoria se almacena el instante temporal y el tipo de la petición solicitada sobre esa dirección de memoria. El número de elementos almacenados dependerá del número de peticiones realizadas sobre una misma dirección. A diferencia del fichero de traza, en este no mediremos el instante temporal como el ciclo en el que se produce la solicitud. En su lugar se toma el número de solicitud realizada, empezando siempre por el uno. El segundo acceso a memoria se tomará con el valor 2, independientemente del número de ciclos que hayan transcurrido entre ambos. Esta aproximación permite abstraerse del nivel de procesamiento de la aplicación y utilizar únicamente la información sobre los accesos a memoria. Si durante un largo periodo de tiempo no se lleva a cabo ningun acceso, el contenido de la memoria caché no cambia y esta situación no es de interés. Así también evitamos la distorsión introducida por otras tareas ajenas a la propia aplicación monitorizada, que puedan afectar al tiempo de ejecución.

Continuando con el fichero de traza de Firefox, si lo convertimos a un fichero de accesos, las cuatro primeras líneas son las siguientes:

0x7f487cb82c00 1 R

0x7ffecf8e0240 2 W

0x7f487cb83980 3 R

0x7f487cda7e40 4 R 329010 R 998785 R 1053536 R 1321342 R 1422107 R 1431184 R 1572377 R 2357256 R 2728021 R 3595809 R 3939819 R 4008074 R 4145967 R 4275977 R 4399616 R 4765392 R

En este pequeño ejemplo podemos ver como la línea a la que se accede por cuarta vez vuelve a ser consultada otras 16 veces, y todas las peticiones son de lectura. Las tres primeras líneas sólo se acceden una vez y no vuelven a ser consultadas.

Un punto importante es que en el fichero de accesos se han eliminado todos los accesos a memoria de tipo **reemplazo**. De hecho, no cuentan como número de acceso, por lo que aunque haya varios accesos de reemplazo entre dos peticiones de lectura o escritura, la diferencia temporal será de 1 acceso. Esto se hace para evitar la distorsión introducida por los reemplazos, que no son propiamente una operación lanzada de forma directa para acceder a un dato, sino desencadenada de forma indirecta por un desalojo de la memoria caché.

En vista del fichero de accesos, es posible darse cuenta de ciertos patrones recurrentes en las peticiones a una misma dirección de memoria:

- En muchísimas líneas sólo se realiza un único acceso, que puede ser o bien de lectura, o bien de escritura. Aunque pueda parecer que hacer un acceso de escritura para luego no leer el dato es un sinsentido, hay que recordar que es posible que se esté utilizando a nivel de caché y que no haya sido desalojado nunca, por lo que no se necesita una operación de lectura para consultarlo.
- Otro patrón recurrente es el de múltiples accesos, pero únicamente de lectura. En el ejemplo de Firefox puede observarse este comportamiento.
- El último patrón que parece repetirse también está relacionado con varios accesos, aunque está vez se encadenan solicitudes de lectura y escritura.

Además del fichero de accesos, se considera también un **fichero de ciclos** con extensión *.out*. Contiene la misma información que el primero en cuanto a secuencias de accesos, pero indica el instante temporal exacto (en número de ciclo) en que se realiza una petición en lugar del número de acceso. Siempre trabajaremos con el fichero de accesos, salvo que se indique explícitamente lo contrario y se trabaje con ciclos.

A partir del fichero de accesos es posible obtener información de todo tipo. Una posibilidad es el estudio de la localidad temporal de la aplicación, tal y como se hace en el Capítulo siguiente. También se puede extraer información sobre la localidad espacial, la distribución del número de accesos a una misma línea, e información sobre la estabilidad e irregularidad en los accesos a memoria, como veremos en los Capítulos posteriores.

Capítulo 4

Agrupamiento de aplicaciones en función de los accesos a memoria

Gracias a la utilización de Pin, disponemos de un fichero de traza con información detallada sobre las peticiones de accesos a memoria. En muchas ocasiones será más conveniente trabajar con el fichero de accesos, al agrupar la información por línea de memoria. En este Capítulo se realiza un agrupamiento de las 20 aplicaciones monitorizadas atendiendo a la localidad temporal y algorítmica, utilizando los ficheros de accesos procedentes de Pin.

En primer lugar analizaremos el comportamiento de las aplicaciones en función de su localidad temporal, es decir, el tiempo que tardan en volver a utilizar una línea de memoria. Veremos que podemos distinguir varios grupos de aplicaciones, en función de si tienden a repetir accesos a memoria o no. Veremos también que la localidad temporal medida entre la caché y la memoria principal depende fuertemente de la configuración de la LLC, modificando el comportamiento de algunas aplicaciones de forma drástica.

También consideraremos la localidad algorítmica al final de este Capítulo, aunque en menor medida. La forma de llevar a cabo este análisis es mediante la densidad de operaciones, medido en número de operaciones lanzadas cada 1.000 ciclos. Comprobaremos cómo hay ciertos patrones que se repiten a lo largo del tiempo para algunas aplicaciones, y de nuevo observaremos un cambio importante al mejorar la caché LLC.

4.1. Agrupamiento en base a la localidad temporal

La localidad temporal puede ser fácilmente examinada a partir del fichero de accesos. Una aplicación con una alta localidad temporal tarda poco en volver a acceder a la misma porción de memoria. Si el diseño de la caché es correcto y ésta no está sometida a demasiado estrés, entonces no debería haber una petición próxima en el tiempo a una misma dirección. Si todo ha ido bien, entonces las siguientes veces que se acceda al dato concreto, con una probabilidad muy alta estará en la memoria caché y no será necesario realizar un acceso a memoria principal.

Sin embargo, puede haber ciertos motivos que provoquen un acceso temprano a una dirección de memoria principal después de otro. El primero de ellos tiene que ver con una mala elección de la víctima en el algoritmo de reemplazo de la caché. Si se elige un determinado dato para desecharlo, pero se accede poco después, significa que el algoritmo de reemplazo no ha acertado. En estos casos se produce un **fallo de conflicto**. La segunda es más difícil de remediar, pues tiene que ver con el tamaño del conjunto de trabajo que necesita la aplicación. Si la caché es demasiado pequeña, pero se están accediendo continuamente a muchos datos, tantos que no caben en la caché, entonces se estarán produciendo múltiples fallos que no pueden ser evitados. En este caso, se dice que la caché está sometida a un alto nivel de estrés, y de los fallos se dice que son **fallos de capacidad**.

En general, es preferible que los accesos a una misma dirección estén repartidos o que sean tardíos. Si se suceden demasiado pronto muestran un comportamiento patológico en el que hay demasiados fallos de caché.

Una forma de medir la localidad temporal a partir del fichero de accesos es tener en cuenta aquellas líneas para las que ha habido dos accesos o más, y tomar como distancia temporal el tiempo que ha transcurrido entre un acceso y el siguiente. El tiempo se mide, de nuevo, en número de accesos intermedios a otras posiciones de memoria. Cada línea que ha sido accedida p veces, con $p > 1$, proporciona $p - 1$ valores de distancia temporal para medir la localidad. La creación de este **fichero de localidad temporal**, con extensión *.loctmp* se realiza a partir de un script en Python que toma como entrada el fichero de accesos.

En la tabla 4.1 se encuentran las tres primeras entradas del fichero de localidad temporal para el ejemplo de Firefox que estamos manejando. En el fichero de accesos vimos que la línea que se accedía por cuarta vez era utilizada varias veces más. En el fichero de localidad temporal puede comprobarse más fácilmente cuánto se tarda en volver a acceder. La primera columna hace referencia a la dirección de memoria involucrada; la segunda y tercera columna al número de acceso la primera vez y la siguiente; la columna de distancia temporal es la resta de las anteriores e indica cuánto tarda se en volver a acceder a la dirección involucrada; las últimas columnas indican los tipos de las operaciones. Nótese que la columna realmente interesante y la que aporta luz sobre la localidad temporal es la de distancia temporal.

Dirección	Nº acceso 1	Nº acceso 2	Distancia temporal	Tipo 1	Tipo 2
0x7f487cda7e40	4	329010	329006	R	R
0x7f487cda7e40	329010	998785	669775	R	R
0x7f487cda7e40	998785	1053536	54751	R	R

Tabla 4.1: Ejemplo de las primeras líneas del fichero de localidad temporal para el caso de Firefox, con una LLC de tamaño 16MB y asociatividad de nivel 1.

El objetivo de esta sección es agrupar distintas aplicaciones mediante algún tipo de clasificación no supervisada. Sin embargo, con los datos de la localidad temporal presentados en la tabla 4.1, existen algunos problemas.

1. El agrupamiento se desea realizar para cada valor del par tamaño y asociatividad. Esto quiere decir que se pretenden agrupar 20 aplicaciones en base a semejanzas en el comportamiento de la localidad temporal. En el caso de Firefox que estamos manejando como ejemplo, se tienen casi 3 millones de valores para explicar el comportamiento de la localidad. Es evidente la necesidad de resumir la información de alguna manera antes de poder realizar el clustering.

2. El agrupamiento debe hacerse de forma que todas las aplicaciones estén descritas por el mismo número de variables, pero el número de datos de la localidad temporal no tienen por qué ser el mismo en los diferentes casos. Depende únicamente del número de accesos a memoria y, por tanto, del tipo de la aplicación.

3. Dada una aplicación, tamaño, y asociatividad de caché, se puede considerar la información de la distancia temporal como realizaciones una misma distribución aleatoria. En concreto, pueden considerarse las variables aleatorias $Y_1, Y_2, \ldots, Y_n$, cada una de ellas como el tiempo transcurrido entre dos accesos consecutivos a una misma dirección de memoria. No parece acertado aplicar un algoritmo de clustering directamente sobre las realizaciones de variables aleatorias.

En base a estas dificultades, se proponen dos formas de abordar la situación. Cada una de ellas se expone en la subsección correspondiente.

4.1.1. Agrupamiento de la distribución completa

Como primera solución, se propone resumir la localidad temporal en un histograma. Sea $b-1$ el número de puntos intermedios que separan los intervalos en el histograma, sin contar con el mínimo y el máximo. Para generar el histograma, se dividen los datos en b intervalos de igual longitud y se sustituye el valor numérico por el intervalo en el que está contenido. Seguidamente, contamos cuántos elementos hay dentro de cada uno de los b intervalos. Nótese que, según el procedimiento descrito, no es realmente un histograma, sino un gráfico de barras tras discretizar los datos, aunque seguiremos llamándolo histograma por comodidad.

No obstante, para mejorar la interpretabilidad de los resultados, se va a realizar una ligera modificación en los datos originales. Es posible que el tiempo que se tarda en volver a acceder a una posición de memoria sea relativamente bajo, pero haya un número muy pequeño de ocasiones en los que se tarda mucho. Esto puede ser debido, por ejemplo, a la repetición de una operación tiempo después, que requiera volver a cargar datos que no están en la caché. En estos casos, el rango de valores en los que se dividen los intervalos será exageradamente alto únicamente por la acción de unas pocas perturbaciones.

Supongamos, por ejemplo, que el número medio de accesos entre dos accesos consecutivos a una posición de memoria es de 100. Consideremos una situación en la que, normalmente, el orden de tiempo oscila en torno a 100, pero que, por algún motivo desconocido, se vuelve a acceder a ciertas posiciones de memoria mucho más tarde; por ejemplo, tras 10000 accesos. Al hacer un histograma

con estos datos, la altura de las primeras barras será muy elevada en comparación con las demás, pues la mayoría de accesos son cercanos. La existencia de unos pocos accesos tardíos modifica la estructura del histograma y resulta poco informativo. En previsión de estas situaciones donde haya unos accesos raros, tardíos e impredecibles, se propone eliminar el 5 % de las observaciones más altas. Esto supone representar el histograma únicamente con el 95 % de los datos, eliminando valores atípicos, y proporcionando una mayor interpretabilidad al mismo al poner el foco donde se produce la mayoría de observaciones.

Un ejemplo de un histograma construido según este procedimiento se encuentra en la figura 4.1, con el supuesto de Firefox que hemos venido manejando hasta ahora. En primer lugar, se eliminan del análisis el 5 % de las observaciones más alejadas, es decir, aquellos casos donde el tiempo hasta el siguiente acceso es tan elevado como para considerarse extraño. Puede apreciarse una fuerte curva decreciente, lo que indica que la mayoría de accesos a una misma posición de memoria ocurren, generalmente, en una fase temprana. Sea R la máxima distancia temporal entre dos accesos consecutivos para una aplicación dada tras la transformación propuesta ($R = 1.864.468$ para Firefox). En el eje x del histograma (la figura de la izquierda) se representan $b = 100$ intervalos de la misma longitud, partiendo desde 0 hasta R, junto con las frecuencias relativas. Quizá sea más fácil de interpretar el polígono de frecuencias que está representado en la parte derecha. A la vista del gráfico se puede concluir que aproximadamente el 60 % de los accesos consecutivos a memoria conllevan entre 1 y $0.2R = 372.893$ accesos intermedios. De hecho, el 20 % de los accesos consecutivos tienen entre 1 y $0.01R = 18644$ accesos intermedios. La fuerte asimetría en esta distribución sugiere que es muy común acceder tempranamente a los datos, y la probabilidad decrece con el tiempo. Nótese que hemos eliminado el 5 % de las observaciones más raras, por lo que los porcentajes no se pueden tomar como una referencia absoluta.

Tras la obtención del histograma para todas las aplicaciones, podemos resumir la distribución de la distancia temporal en base a las b alturas de cada barra en el histograma. Si b es lo suficientemente pequeño como para permitir el clustering, ahora todas las aplicaciones están descritas por el mismo número de variables y los problemas mencionados anteriormente desaparecen. El clustering nos va a permitir agrupar aplicaciones cuyos histogramas sean semejantes y, con ello, la distribución de la distancia temporal.

La elección de b es importante. Si se toma un valor demasiado elevado, el clustering será más costoso y existe un mayor riesgo de sobreajuste. En este escenario es más difícil realizar un agrupamiento, puesto que es más complicado detectar semejanzas. Si por el contrario, b es demasiado pequeño, el ajuste será demasiado grosero y muy sesgado. Este es un claro ejemplo del problema sesgo-varianza en el ajuste de modelos. Tras varios ensayos de prueba y error, se ha comprobado que al utilizar $b = 10$ el ajuste que se produce es razonable. Todos los histogramas para las aplicaciones que serán representados a continuación utilizan 100 intervalos para permitir una visualización más fina de la distribución, pero hay que tener siempre presente que el agrupamiento se realiza con $b = 10$.

Por otro lado, se ha comprobado que existen ciertos casos donde el número de accesos a memoria

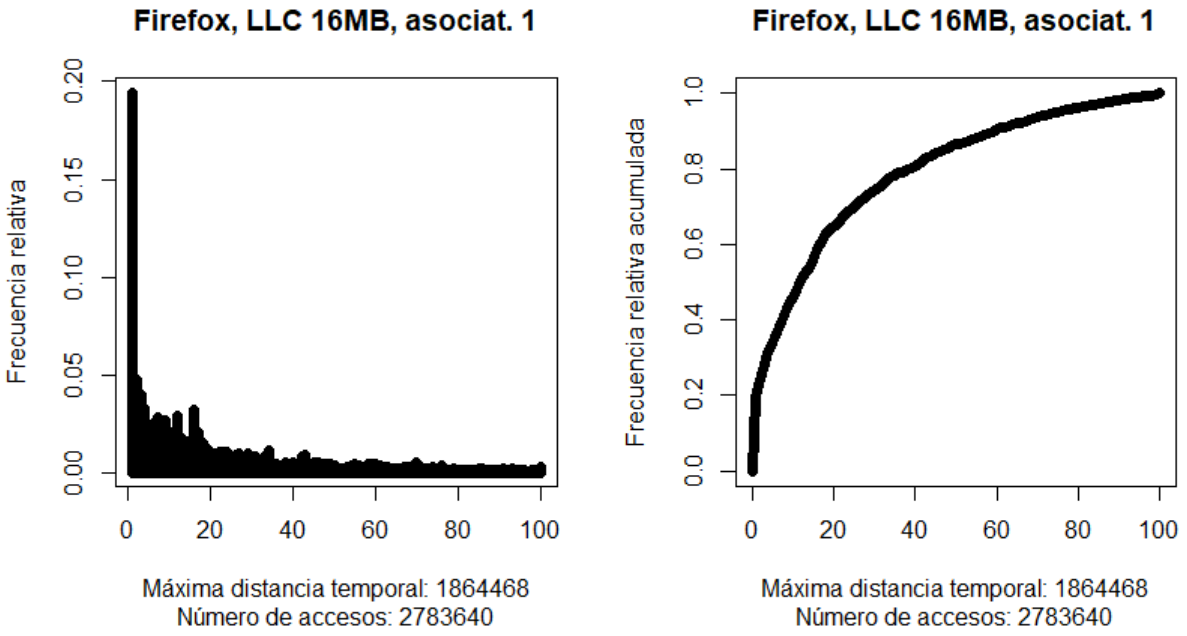


Figura 4.1: Histograma de la distribución de la localidad temporal para Firefox y polígono de frecuencias relativas acumuladas, con LLC de tamaño 16MB y asociatividad de nivel 1.

es pequeño (para *specrand* con una la LLC de 16MB y asociatividad de nivel 1 se producen únicamente 4 accesos a memoria). No parece razonable resumir 4 valores de una distribución temporal en un histograma con $b = 10$ intervalos para realizar el clustering. Por este motivo, todos aquellos casos donde se disponga de menos de 50 valores para la distribución temporal se han eliminado del análisis. Nótese que un número bajo de accesos implica que el conjunto de trabajo de la aplicación cabe en la LLC y la memoria principal está siendo muy poco utilizada.

La última consideración necesaria antes de realizar el agrupamiento es la **medida de distancia** que se va a utilizar para valorar semejanzas y diferencias entre los histogramas. Cualquier algoritmo de clustering, y en particular un método ascendente jerárquico como el que utilizaremos aquí, se basa en una medida de distancia entre las observaciones para agruparlas. De hecho, la entrada real a los algoritmos de este tipo es una matriz de distancias entre todos los elementos.

El problema de definir una medida de distancia para datos agrupados e histogramas ha sido cada vez más discutido debido al creciente volumen de datos con el que se necesita trabajar. En particular, L. Billard & Jaejik Kim [17] proponen una metodología completa para el agrupamiento de histogramas. En nuestro caso utilizaremos un método más sencillo, por haber proporcionado buenos resultados en la práctica. En la teoría de probabilidad y la estadística, la medida de divergencia de **Jensen-Shannon** (JSD) [18] es un método para medir semejanzas o distancias entre dos distribuciones de probabilidad. Está basada en la divergencia de Kullback-Leibler, con una extensión para añadir la propiedad simétrica. Puesto que un histograma es una aproximación razonable de la función de probabilidad, especialmente si el número de datos de los que se dispone es elevado, parece una buena candidata para ser utilizada en el propósito que nos atañe.

En resumen, el preprocesamiento que se ha realizado a las observaciones de la distancia temporal para cada caso ha sido:

1. Se elimina el 5 % de los valores más elevados para paliar el efecto de medidas atípicas.
2. No se consideran aquellos casos donde haya menos de 50 mediciones.
3. Se toman $b = 10$ alturas para el agrupamiento, y $b = 100$ en las representaciones gráficas.
4. La medida de distancia para valorar las diferencias es la basada en JSD.

A partir de la matriz de distancias, se propone un clustering jerárquico ascendente basado en el método de Ward. Teniendo en cuenta que la LLC puede tener 16MB o 32MB, y que la asociatividad puede ser de nivel 1 o de nivel 8, se tienen 4 escenarios para probar las aplicaciones. El agrupamiento discutido será únicamente para el escenario de 16MB y asociatividad de nivel 1, pues es donde se espera una tasa más elevada de fallos de caché, y por tanto más accesos a memoria. El objetivo del clustering es dividir a las aplicaciones en función de sus accesos a memoria para poder diseñar optimizaciones por separado.

El dendograma resultante de aplicar el método descrito se puede consultar en la figura 4.2. Es fácil ver que fundamentalmente hay cuatro grupos que permiten separar a las aplicaciones atendiendo al criterio JSD. Por un lado, todas las aplicaciones del grupo de la izquierda tienen un comportamiento muy similar, además de ser el grupo más grande. Por otro lado, los grupos de la derecha están formados por menos elementos.

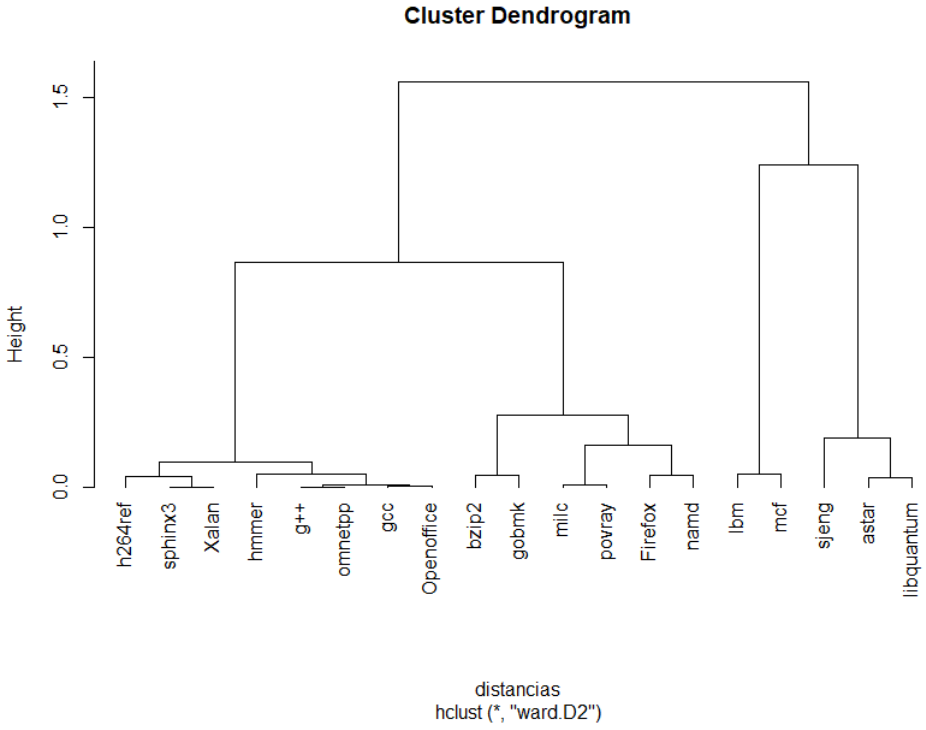


Figura 4.2: Dendograma para las aplicaciones con LLC de 16MB y asociatividad de nivel 1

Las figuras 4.3, 4.4, 4.5, 4.6 contienen los histogramas de todas las aplicaciones que han participado en el análisis, separados por los cuatro grupos que sugiere el dendograma. Debajo de cada uno se incluyen unos pequeños resúmenes de la distribución: la máxima distancia temporal que determina la escala del eje x, el número de accesos repetidos que se han producido en total, y el valor medio.

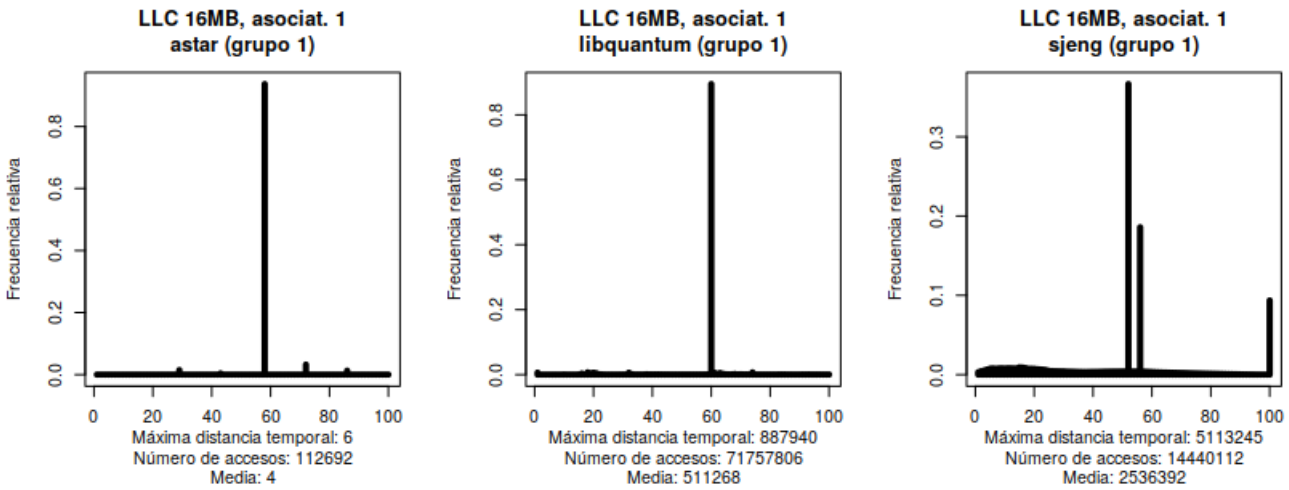


Figura 4.3: Aplicaciones del grupo 1 para el clustering cuando la LLC es de 16MB y la asociatividad por conjuntos de nivel 1

Es necesario llamar la atención sobre algo importante antes de continuar. Dada la forma en la que se ha construido el histograma, no se pueden comparar las aplicaciones entre sí en la escala de tiempo (eje x) representada. Los datos de cada aplicación se han discretizado en 100 intervalos de igual longitud, por lo que la amplitud y rango de los mismos no son directamente comparables. Si dos aplicaciones tienen unos histogramas similares donde el mayor peso se sitúa a la izquierda, no es correcto decir "las dos aplicaciones tienden a volver a acceder tempranamente a los datos una vez utilizados". Si la máxima distancia temporal en la primera aplicación es de, por ejemplo, 1.000 accesos, pero en la segunda es de 10.000, entonces el primer intervalo de accesos considerado si tomamos $b = 100$ es de $[1, 10]$ accesos para la primera aplicación y $[1, 100]$ para la segunda. Si la primera barra tiene alturas similares en las dos aplicaciones, se estaría indicando que aproximadamente hay el mismo número de casos que caen entre 1 y 10 accesos en la aplicación 1 y entre 1 y 100 accesos en la aplicación 2.

Sin embargo, puesto que el rango de la distancia temporal puede variar de manera drástica entre aplicaciones, esta escala relativa parece más adecuada para comparar la distancia temporal de manera global. Así, se mide el comportamiento de la localidad temporal tomando como referencia el rango en el que se suele mover la distribución de la distancia temporal. La afirmación correcta sería más bien que "las dos aplicaciones tienden a volver a acceder tempranamente a los datos una vez utilizados, cada una en relación con el comportamiento habitual y la escala apropiada para cada aplicación". No obstante, no se incluirá esta aclaración en todos los casos, por ser redundante y demasiado tediosa. Se recomienda tener siempre presente esta diferencia en las escalas para la

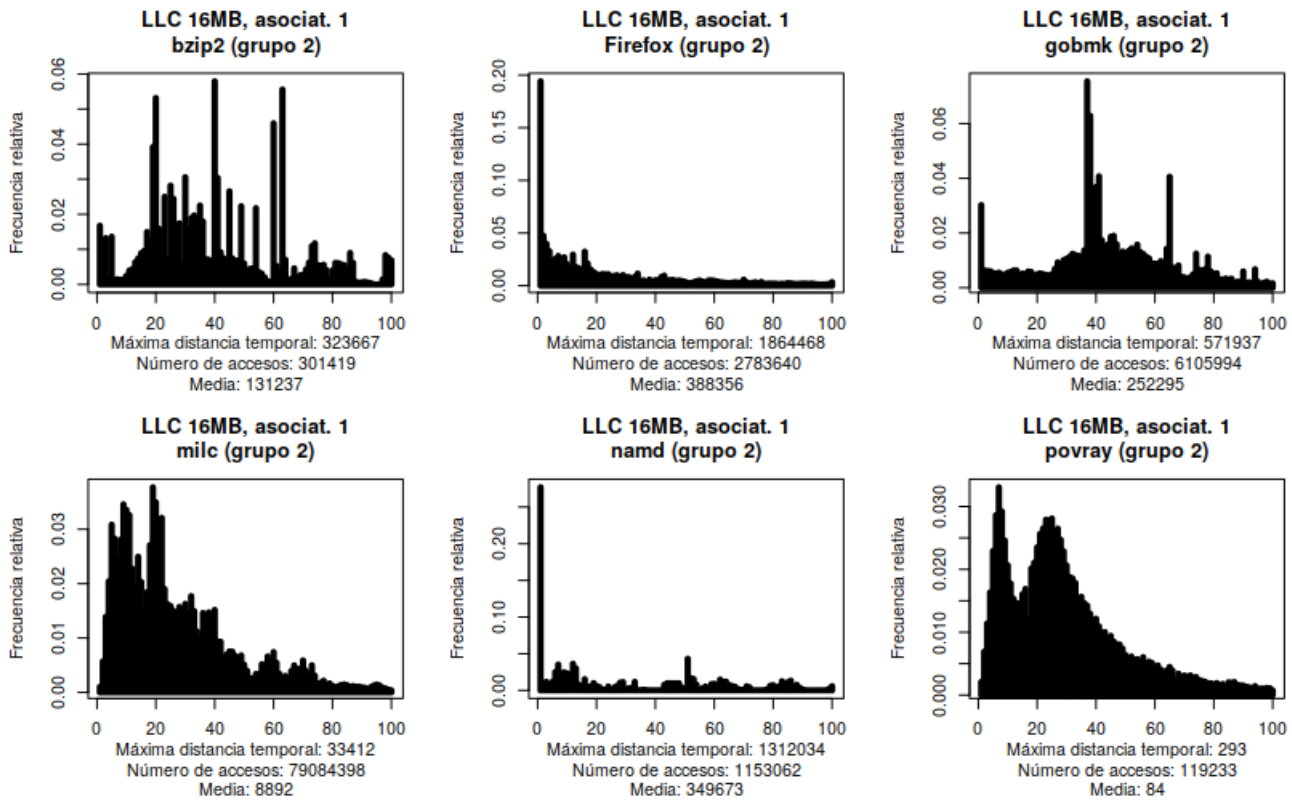


Figura 4.4: Aplicaciones del grupo 2 para el clustering cuando la LLC es de 16MB y la asociatividad por conjuntos de nivel 1

interpretación de los resultados. Como indicativo del rango de la distancia temporal se incluye, en la parte inferior del histograma el valor máximo de la distancia temporal.

A la vista de los histogramas y la agrupación de las aplicaciones, podemos deducir que:

- El grupo 1 está formado por aplicaciones que no acceden tempranamente de nuevo a la misma información, sino que hay un número de accesos no despreciable entre medias. Si R_i denota la máxima distancia temporal para la aplicación i del grupo 1, entonces con una alta probabilidad se volverá a acceder a una misma posición de memoria aproximadamente dentro de $0.6R_i$ accesos. Nótese que para "astar", la máxima distancia temporal es 6, y que entonces la mayoría de accesos se repite cada 4 accesos intermedios. Además, ocurre un número no despreciable de veces, pues se tienen 112.692 valores de la distancia temporal. Esto distingue a "astar" de sus dos compañeras, donde el rango es mucho más elevado en comparación.

- El grupo 2 se corresponde con aplicaciones cuyo tiempo entre accesos a una misma dirección está distribuido a lo largo del rango, es decir, hay accesos repetidos en casi cualquier intervalo de tiempo. No obstante, es cierto que la forma de los histogramas es diferente. En el caso de 'firefox" y 'namd" la forma es parecida: hay varios accesos repetidos tempranos, y van decreciendo progresivamente, aunque siempre hay unos pocos. El decrecimiento para 'milc" es más lento, igual que para 'povray". Esta última merece una mención aparte, pues la

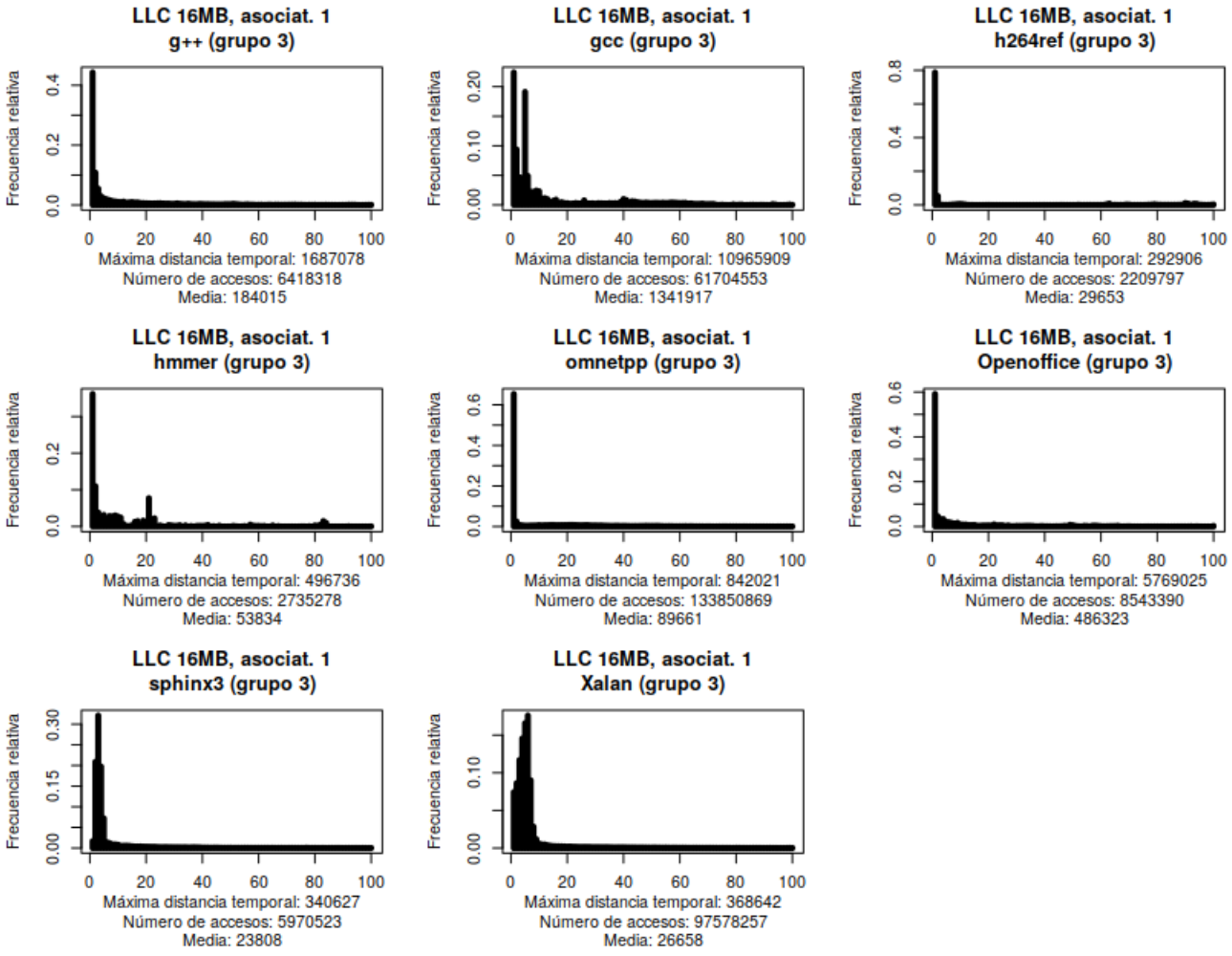


Figura 4.5: Aplicaciones del grupo 3 para el clustering cuando la LLC es de 16MB y la asociatividad por conjuntos de nivel 1

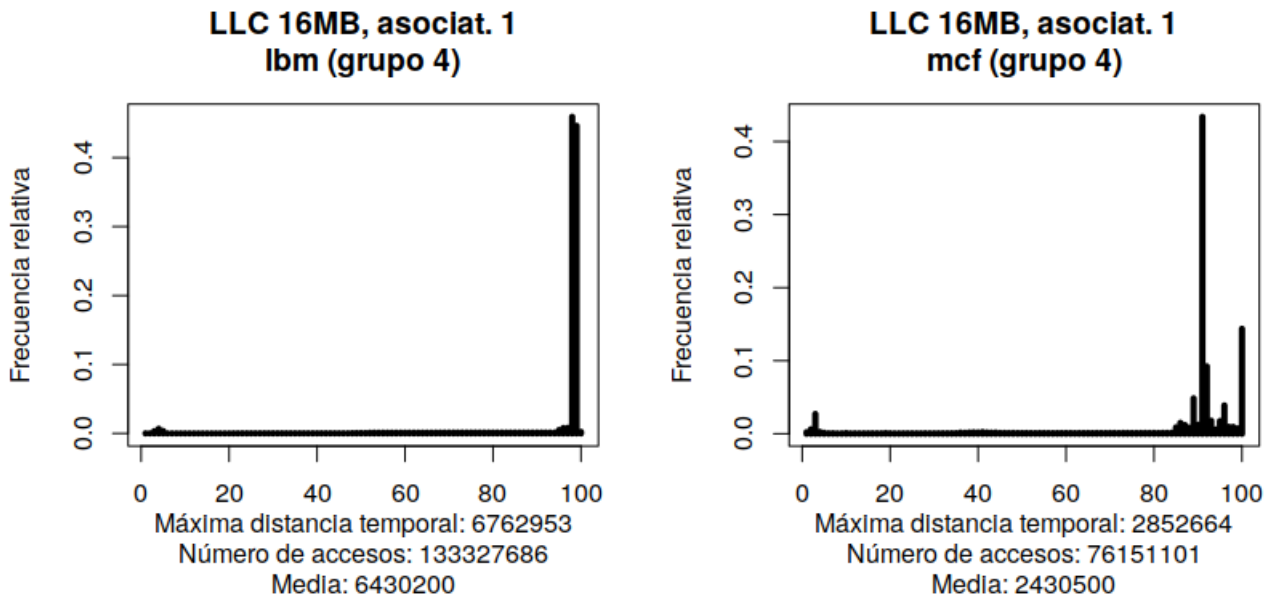


Figura 4.6: Aplicaciones del grupo 4 para el clustering cuando la LLC es de 16MB y la asociatividad por conjuntos de nivel 1

máxima distancia temporal es muy pequeña: 293 accesos intermedios. Por otro lado, 'bzip2" y 'gobmk" no presentan esta tendencia decreciente.

- El grupo 3 caracteriza a aplicaciones con una fuerte reutilización de direcciones de memoria a corto plazo. Se diferencian de las aplicaciones del grupo 2 en que la altura del histograma es prácticamente cero a partir de un cierto tiempo. Esto motiva separarlas de aplicaciones como 'firefox" y 'namd".

- El último grupo contiene dos aplicaciones donde la mayoría de accesos repetidos se producen tras un número muy elevado de accesos intermedios, aunque hay un pequeño número de accesos tempranos.

4.1.2. Agrupamiento considerando un umbral de olvido

La segunda aproximación para la categorización de aplicaciones en función de su localidad temporal pasa por un umbral de olvido. Cuando se accede a una posición de memoria, se elevan esos y los datos cercanos a la memoria caché, pues existe una alta probabilidad de volverlos a utilizar en un futuro próximo. Si no se vuelven a utilizar, acabarán siendo desechados por el algoritmo de selección de víctima. Es posible que tiempo después se vuelva a acceder a la misma posición de memoria, pero ha habido tantos accesos intermedios que no queda ningún residuo en la caché, y este acceso se trata a todos los efectos como una posición de memoria no vista hasta ese momento.

En este contexto tiene sentido definir un parámetro de olvido que se denotará como T_u. Se considera que el acceso a una posición de memoria es el primero de una posible secuencia de

accesos repetidos en cualquiera de los dos siguientes casos:

a. Es, realmente, la primera vez que se accede a dicha posición de memoria.

b. Se ha accedido anteriormente a esa posición, pero el número de accesos intermedios m verifica que $m > T_u$.

La elección de T_u depende del tamaño de la caché, puesto que en función de éste se tardará más o menos tiempo en eliminar los residuos de una posición de memoria. Nótese que siempre que haya espacio, no será necesario desalojar un bloque de datos. No debemos olvidar el propósito de este trabajo: encontrar una buena configuración para que una memoria SDRAM pueda actuar como caché de una RRAM que trabaje inmediatamente por debajo en la jerarquía de memoria.

Consideremos una cache SDRAM con un tamaño de datos de 4GB y un tamaño de línea de 64 Bytes. El número de líneas será, aproximadamente, de 64 millones. Si la asociatividad fuera por conjuntos de nivel 16, por ejemplo, tendríamos 4 millones de conjuntos de 16 líneas cada uno. En un caso óptimo podríamos recibir 4 millones de accesos sin repetir ningún conjunto, aunque la realidad es diferente y habrá varios conjuntos con muchos accesos, lo que da lugar a reemplazos. Podríamos considerar que la vida media de una línea en esta cache podría ser del orden de un millón de accesos, aunque es simplemente una suposición. Por ello, un valor de $T_u = 1.000.000$ parece un buen candidato.

El preprocesamiento que se necesita realizar en el fichero de localidad temporal para incluir la información del parámetro de olvido es muy sencillo. Para cada línea del fichero, si el valor de la distancia temporal es superior a T_u, no debemos considerarlo como un acceso repetido y, por tanto, borrarlo del fichero. El resultado es que para todas las aplicaciones y escenarios, se tienen unos datos que informan sobre la localidad temporal y en ningún caso se supera el valor T_u. Para hacer el clustering, podemos de nuevo quedarnos con las alturas del correspondiente histograma que resuma la distribución.

Sea b el número de alturas del histograma que resume la muestra de la localidad temporal considerando un parámetro de olvido. El histograma se crea de forma que el valor máximo es T_u independientemente de la muestra, por lo que ahora los histogramas comparten escala y el eje x es comparable entre sí. El problema que dificultaba la interpretabilidad en la solución anterior, debido a la escala relativa definida por la máxima distancia temporal, desaparece por completo.

En la figura 4.7 se muestra un histograma con $b = 100$ para la aplicación "Firefox", con LLC de 16MB y asociatividad de nivel 1. La forma no es muy diferente del que se representa en la figura 4.1, aunque puede verse cómo representa una ampliación para valores pequeños, resultado de eliminar las observaciones mayores que T_u. Además, ahora podemos proporcionar una interpretación más natural del polígono de frecuencias acumuladas. Casi el 60 % de las veces, se vuelve a acceder a memoria para consultar una dirección repetida con menos de 200.000 accesos intermedios a otras posiciones. De hecho, casi el 20 % de los accesos repetidos involucran menos de 10.000 accesos intermedios. En este caso apenas se nota, pero veremos como en otras aplicaciones, la forma del histograma sí cambia por cuestiones de escala.

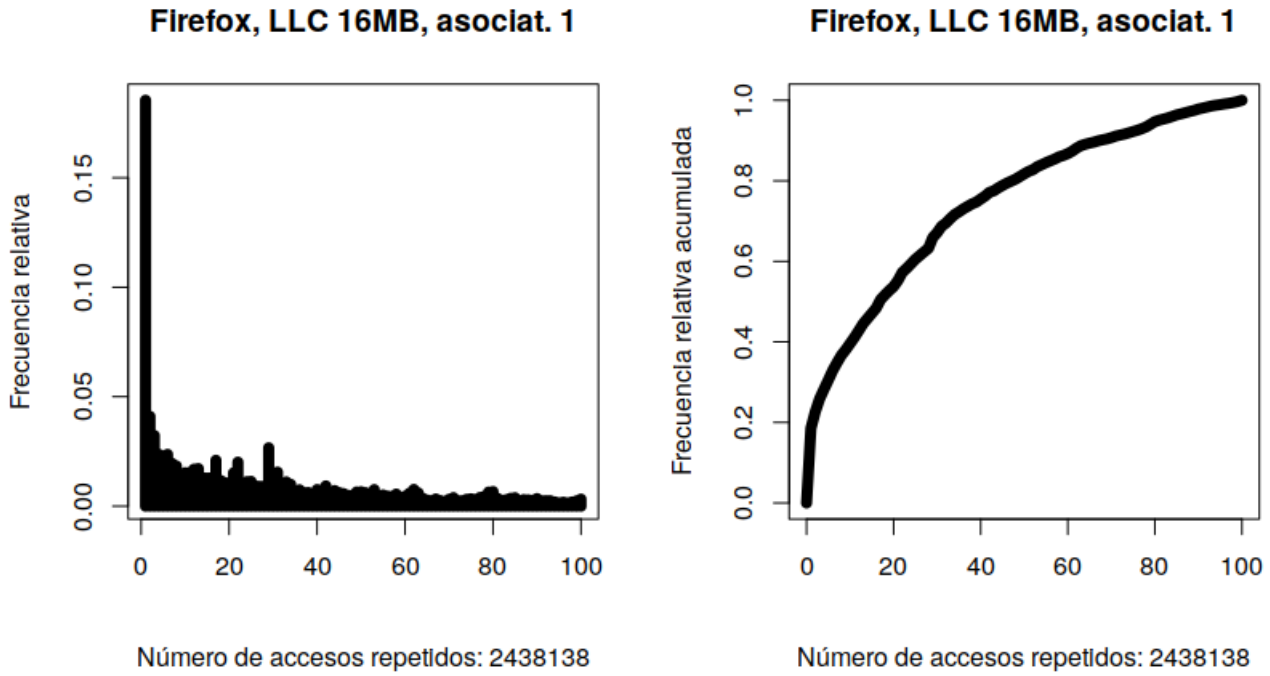


Figura 4.7: Histograma de la distribución de la localidad temporal para Firefox y polígono de frecuencias relativas acumuladas, con LLC de tamaño 16MB y asociatividad de nivel 1, considerando un parámetro de olvido.

Para el clustering, se tomará $b = 10$ de la misma forma que en el caso anterior. En todas las representaciones gráficas se utiliza $b = 100$. También se han eliminado aquellas aplicaciones que acceden menos de 50 veces a memoria, y se ha utilizado la métrica JSD para valorar las diferencias en los histogramas. De igual forma que antes, se propone un clustering jerárquico basado en el método de Ward para el escenario con 16MB de LLC y asociatividad de nivel 1.

El dendograma resultante se muestra en la figura 4.8. A diferencia del anterior, no hay un número de clusters que resulte evidente a primera vista. Podría pensarse que 4 clusters son adecuados. Nótese que el comportamiento de "libquantum" es tan peculiar que cuesta mucho agruparla con otro cluster. Vamos a elegir 5 clusters, separando también el comportamiento de "sjeng" del resto, pues ahora veremos que tanto él como "libquantum" merecen mención aparte.

Las figuras 4.9, 4.10, 4.11, 4.12 y 4.13 contienen los histogramas de todas las aplicaciones, separados por grupos. Debajo de cada histograma se incluye el número de accesos repetidos a posiciones de memoria (esto es, el número de datos con los que se ha representado el histograma).

A diferencia del anterior clustering, ahora sí se pueden comparar entre sí las distintas aplicaciones en la escala de tiempo. Si dos aplicaciones tienen unos histogramas similares donde el mayor peso se sitúa a la izquierda, es correcto decir que "las dos aplicaciones tienden a volver a acceder tempranamente a los datos una vez utilizados". La altura de la barra i, h_i, $\forall i \in \{1, \ldots, b\}$, indica el porcentaje relativo de veces en las que, entre dos accesos a una misma posición de memoria, hay entre $\frac{b}{T_u}(i-1)$ y $\frac{b}{T_u}i$ accesos intermedios, independientemente de la aplicación.

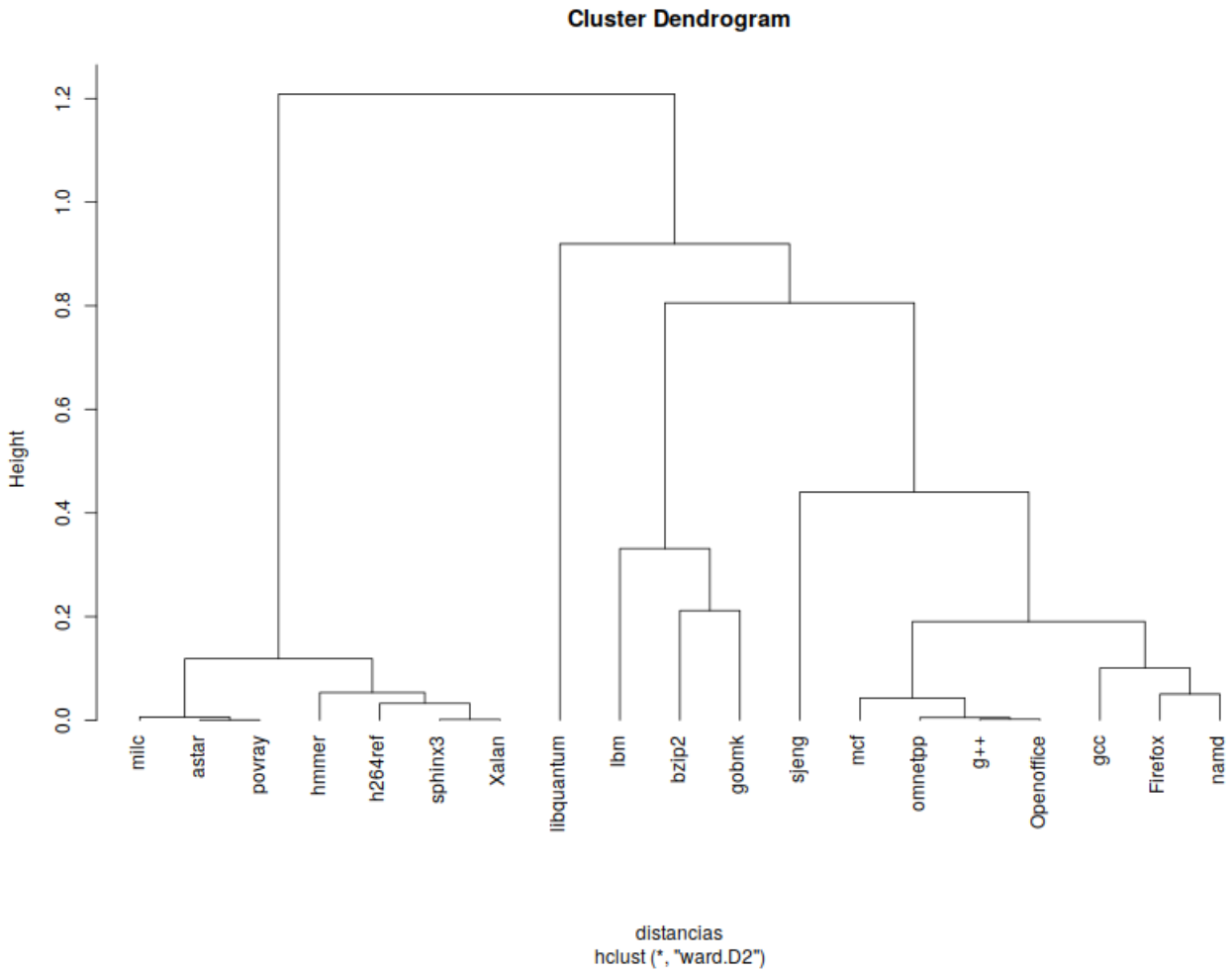


Figura 4.8: Dendograma para las aplicaciones con LLC de 16MB y asociatividad de nivel 1, considerando umbral de olvido

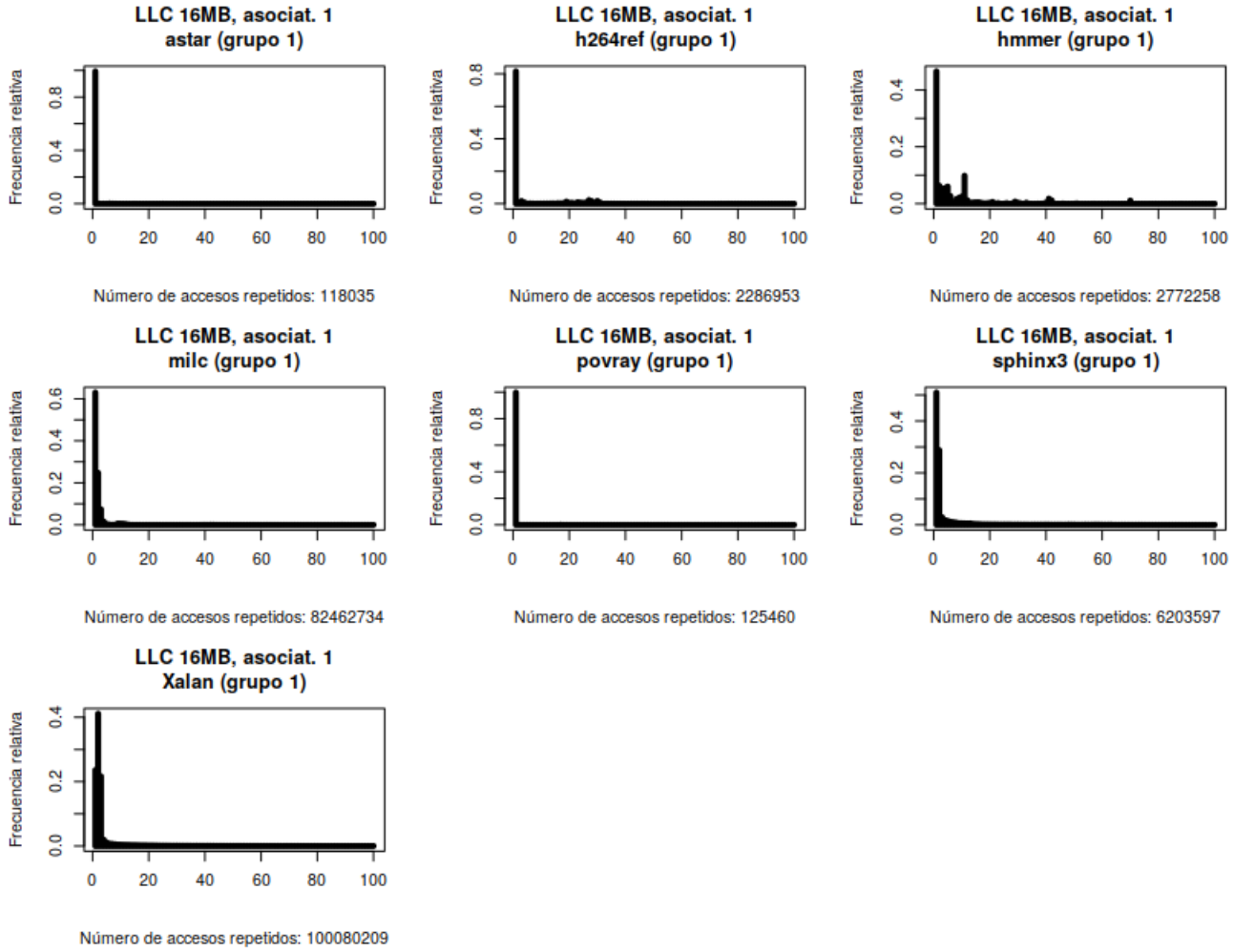


Figura 4.9: Aplicaciones del grupo 1 para el clustering cuando la LLC es de 16MB y la asociatividad por conjuntos de nivel 1, considerando un umbral de olvido

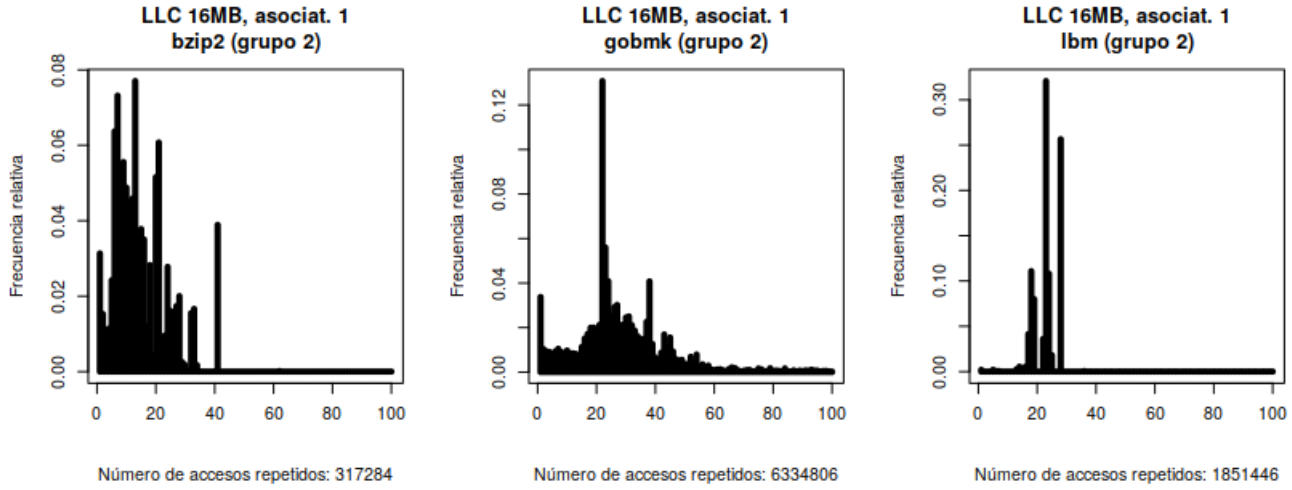


Figura 4.10: Aplicaciones del grupo 2 para el clustering cuando la LLC es de 16MB y la asociatividad por conjuntos de nivel 1, considerando un umbral de olvido

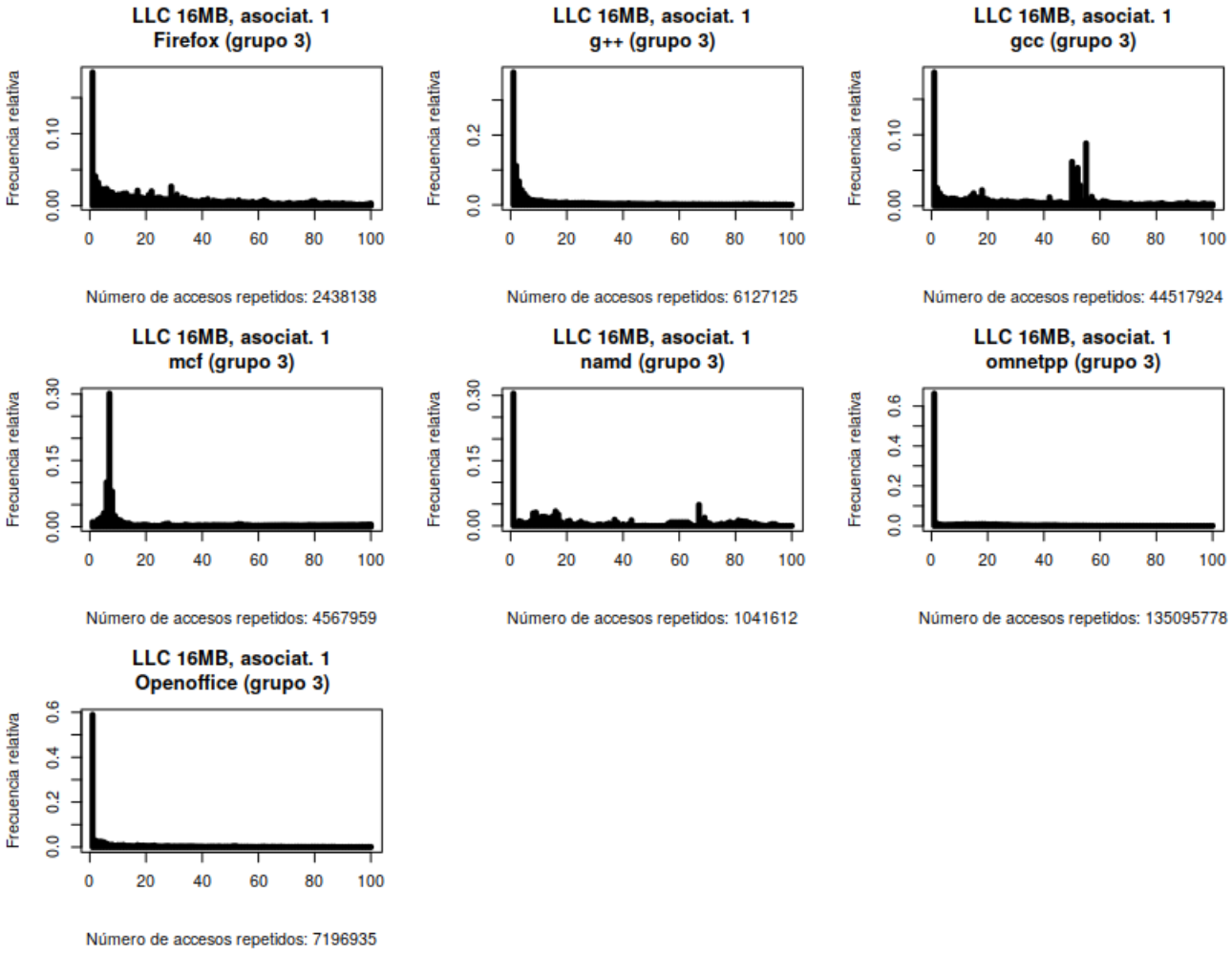


Figura 4.11: Aplicaciones del grupo 3 para el clustering cuando la LLC es de 16MB y la asociatividad por conjuntos de nivel 1, considerando un umbral de olvido

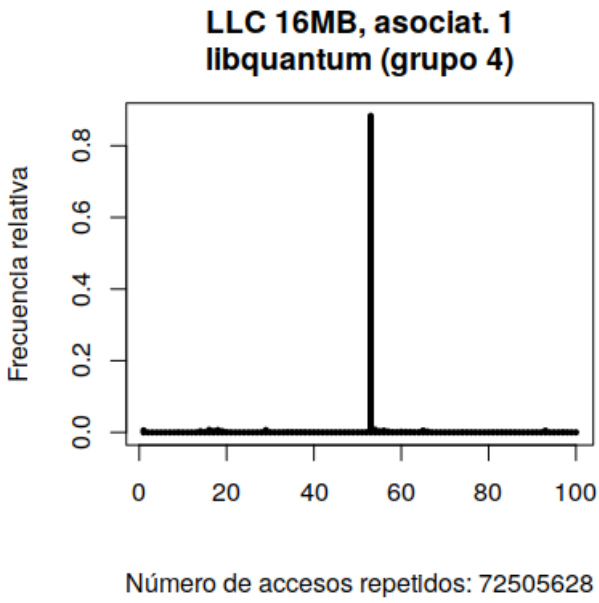


Figura 4.12: Aplicaciones del grupo 4 para el clustering cuando la LLC es de 16MB y la asociatividad por conjuntos de nivel 1, considerando un umbral de olvido

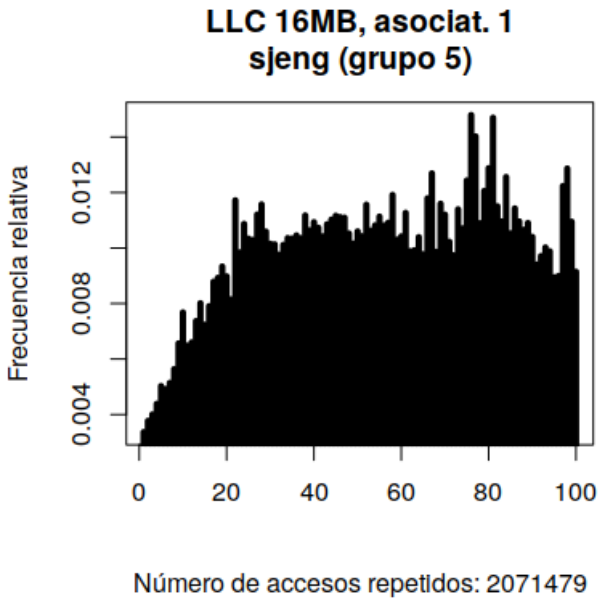


Figura 4.13: Aplicaciones del grupo 5 para el clustering cuando la LLC es de 16MB y la asociatividad por conjuntos de nivel 5, considerando un umbral de olvido

A la vista de los histogramas y el agrupamiento podemos hacer algunos comentarios sobre cada grupo.

- El grupo 1 está formado por 7 aplicaciones, en las que el número de accesos intermedios es muy bajo. En todas estas aplicaciones se tarda muy poco en volver a acceder a posiciones de memoria ya visitadas. En el caso de "Xalan" esto supone más de 100 millones de accesos repetidos tempranos. Este grupo está formado por aplicaciones que saturan la LLC, el conjunto de trabajo no cabe, y se producen numerosos fallos de caché. A partir de un cierto momento, se puede apreciar como apenas hay accesos repetidos (en "hmmer" aún se pueden apreciar unos pocos).

- El grupo 2 está formado por 3 aplicaciones donde todos o la gran mayoría de los accesos repetidos tienen lugar entre 0 y 500.000 unidades de tiempo. El tiempo que se tarda en volver a acceder muestra un patrón irregular, aunque muy alejado del patrón del grupo 1. Son aplicaciones que no sobrecargan tanto la LLC como las anteriores.

- El grupo 3 tiene un patrón que recuerda al del grupo 1. El decrecimiento del histograma es muy rápido, con alta probabilidad de accesos repetidos en cortos intervalos de tiempo. La diferencia con el grupo 1 radica en que este decrecimiento no es tan lento, sino que sigue accediéndose a posiciones repetidas tras algún tiempo.

- El grupo 4 y el grupo 5 están formados por una única aplicación cada uno, alejadas de los tres patrones vistos anteriormente. La aplicación "libquantum" tarda prácticamente siempre el mismo tiempo (entre 520.000 y 530.000 accesos intermedios) en volver a repetir un acceso. Por otro lado, "sjeng" muestra una tendencia creciente y uso casi uniforme de la caché.

Es interesante también comentar las diferencias existentes más marcadas entre los dos clustering.

- Anteriormente se había clasificado a "astar" como una aplicación del grupo 1 (aplicaciones que no acceden tempranamente de nuevo a la misma información, sino que hay un número de accesos no despreciable entre medias). Sin embargo, la máxima distancia temporal era de 6 accesos intermedios, lo que ha provocado que aquí se le clasifique en el grupo de las aplicaciones que acceden muy pronto a posiciones repetidas.

- La aplicación "libquantum" también había sido clasificada en el grupo 1, y al establecer un umbral de olvido, es la única que mantiene este comportamiento. La otra aplicación del grupo 1 del clustering anterior, "sjeng" también ha sido clasificada en un grupo aparte por su comportamiento anómalo. En la figura 4.3 puede apreciarse un tiempo uniforme al cortar la muestra por el umbral de olvido.

- La aplicación "povray" fue anteriormente clasificada en el grupo 2 (aplicaciones cuyo tiempo entre accesos a una misma dirección está distribuido a lo largo del rango, es decir, hay

accesos repetidos en casi cualquier intervalo de tiempo). Debido a que su rango de distancia temporal era muy pequeño, ha sido ahora clasificada como una aplicación que utiliza muy pronto la memoria para accesos repetidos, al igual que "astar".

- Las aplicaciones del anterior grupo 3 (aplicaciones con una fuerte reutilización de direcciones de memoria a corto plazo) pasan ahora al dividirse en el grupo 1 y 3 (aplicaciones con un alto ratio de reutilización con pocos accesos intermedios, con una tendencia decreciente más marcada en el grupo 1) en función del decrecimiento en la distribución del tiempo.

Gracias a este agrupamiento, se abre la posibilidad de diseñar técnicas de optimización en la caché diferentes en función del comportamiento de la aplicación. Se podría monitorizar el comportamiento de una aplicación de forma dinámica, y obtener un perfil similar a los histogramas que se han representado anteriormente. Para cada uno de los grupos detectados se podría obtener un perfil promedio y asignar dicha aplicación, durante la ejecución del programa, a uno de los grupos en función de su distancia al patrón promedio. Dependiendo de qué grupo esté más cerca, se aplicarían unas políticas de optimización u otras.

Los resultados del clustering para las otras tres configuraciones se encuentran en los anexos A (32MB, asociatividad 1), B (16MB, asociatividad 8) y C (32MB, asociatividad 8).

4.2. Perfil de utilización de memoria

Otra característica que puede resultar de interés a la hora de categorizar las aplicaciones es el perfil de utilización de la memoria. Puede resultar interesante conocer en qué instantes se accede a memoria, y cómo son esos patrones. En un gráfico adecuado que resuma este perfil para una aplicación concreta se pueden detectar también comportamientos propios de la localidad algorítmica.

Para evaluar este aspecto se va a crear un **fichero de densidad** a partir del fichero de traza. Este nuevo archivo tendrá la extensión *.opkc*. Se define el **número de operaciones por kilociclo (OPKC)** como el número medio de operaciones de memoria (tanto de escritura y lectura como de reemplazo) que se producen en un intervalo de tiempo determinado, utilizando como referencia 1000 ciclos. Formalmente, el OPKC entre el instante T_i y T_j (expresado en ciclos) se define como:

$$OPKC(T_i, T_j) = 1000\frac{O(T_i, T_j)}{T_j - T_i}$$

donde $O(T_i, T_j)$ es el número de operaciones de memoria solicitadas entre T_i y T_j.

El fichero de densidad contiene el OPKC "instantáneo", calculado con una granularidad de 1000 ciclos. Esto significa que, si consideramos una sección como un intervalo que tiene una duración temporal de 1000 ciclos, el fichero de densidad contiene el OPKC de todas las secciones ejecutadas por una aplicación y configuración. En este trabajo se utiliza el concepto de ciclo como

aproximación de cada instrucción ejecutada en el procesador, suponiendo una instrucción por ciclo.

Un ejemplo de lo que contiene el fichero de densidad puede ser examinado a continuación con el ejemplo de Firefox y la configuración habitual de 16MB y asociatividad 1.

91.0	127.0	69.0	94.0	75.0	23.0	55.0	93.0

Tabla 4.2: Comienzo del fichero de densidad para Firefox, con la configuración de asociatividad 1 y LLC de 16MB

Puesto que el fichero de densidad contiene información sobre la carga de trabajo a la que está sometida la memoria principal, pueden realizarse gráficos asociados para poder comprobar los patrones de accesos a memoria y poder clasificar las aplicaciones en función de los mismos. Además, también se pueden valorar las diferencias que existen entre las diferentes configuraciones, así como la existencia o no de localidad algorítmica.

El primer ejemplo que aquí se comenta tiene que ver con "lbm", una aplicación que muestra una clara estacionalidad en el perfil de utilización y que se traduce en una localidad algorítmica evidente (véase la figura 4.14). En los gráficos podemos observar que el comportamiento de los accesos a memoria es muy similar en todas las configuraciones y que el OPKC es un valor muy elevado si lo comparamos con los gráficos que siguen a continuación. Además, es llamativo que ni aumentar el tamaño de la caché, ni modificar la asociatividad, provoca una mejora en el número o el patrón de accesos a memoria.

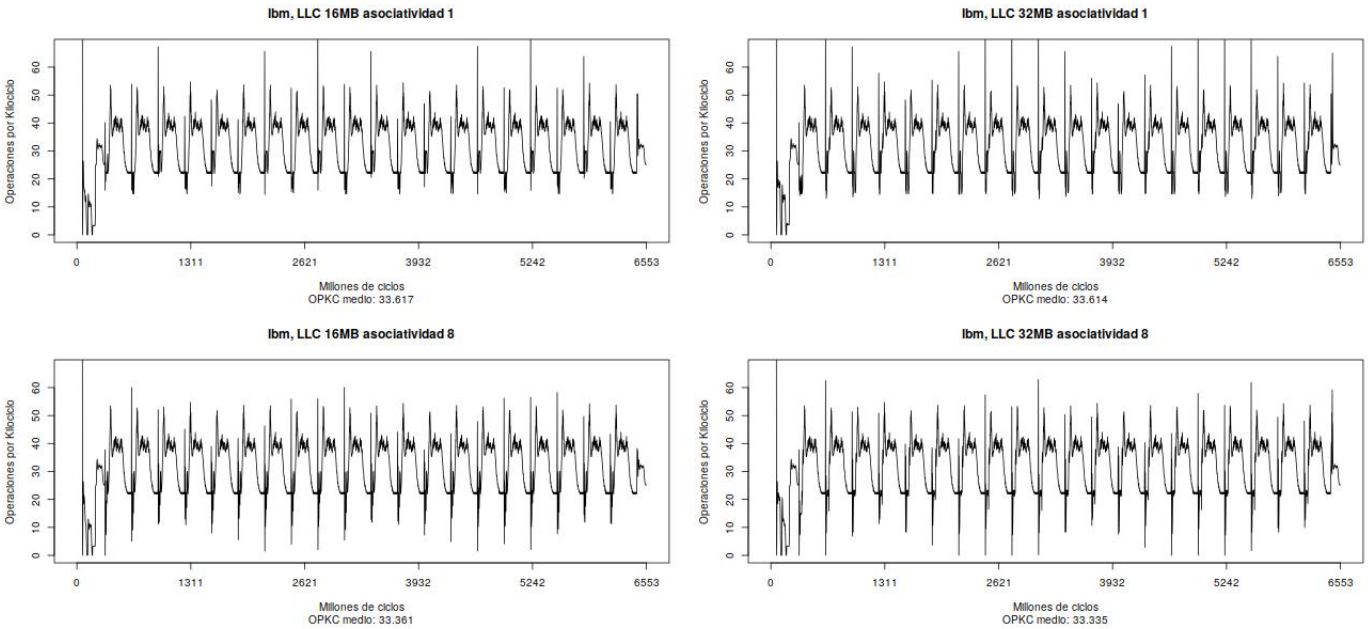


Figura 4.14: Perfil de memoria de lbm

Por otro lado, "libquantum" es una aplicación que se ve muy influida por el tamaño de la caché. Si la configuración es de 16MB se tiene un valor muy grande para el OPKC medio (más de 20),

pero al doblar el tamaño de la caché, la utilización de la memoria principal desciende hasta 10 veces. Esto es una evidencia clara de que con una caché de 16MB el conjunto de trabajo habitual de la aplicación no puede ser almacenado en la misma.

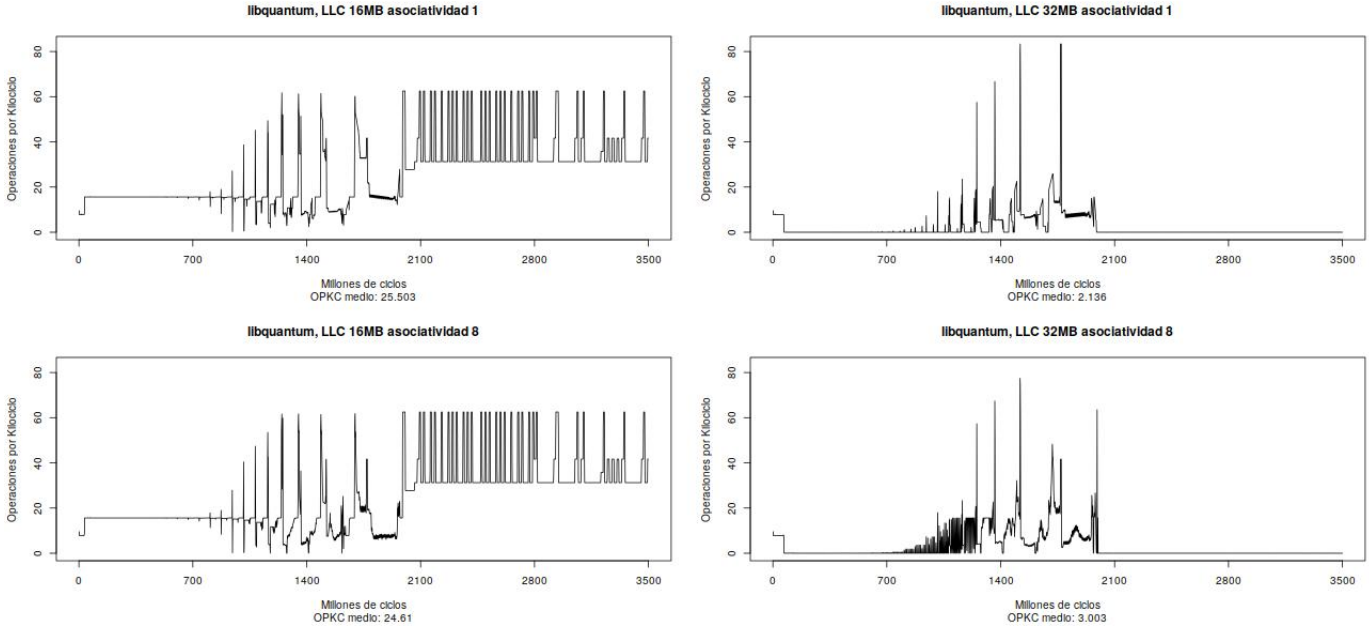


Figura 4.15: Perfil de memoria de libquantum

De forma similar, un ejemplo de aplicación que no se ve tan influido por el tamaño de la caché pero sí por la asociatividad es "omnetpp" (figura 4.16). Al pasar de asociatividad 1 a asociatividad 8, se reduce el OPKC medio y el perfil de utilización de memoria filtra algunos accesos que ahora no son necesarios.

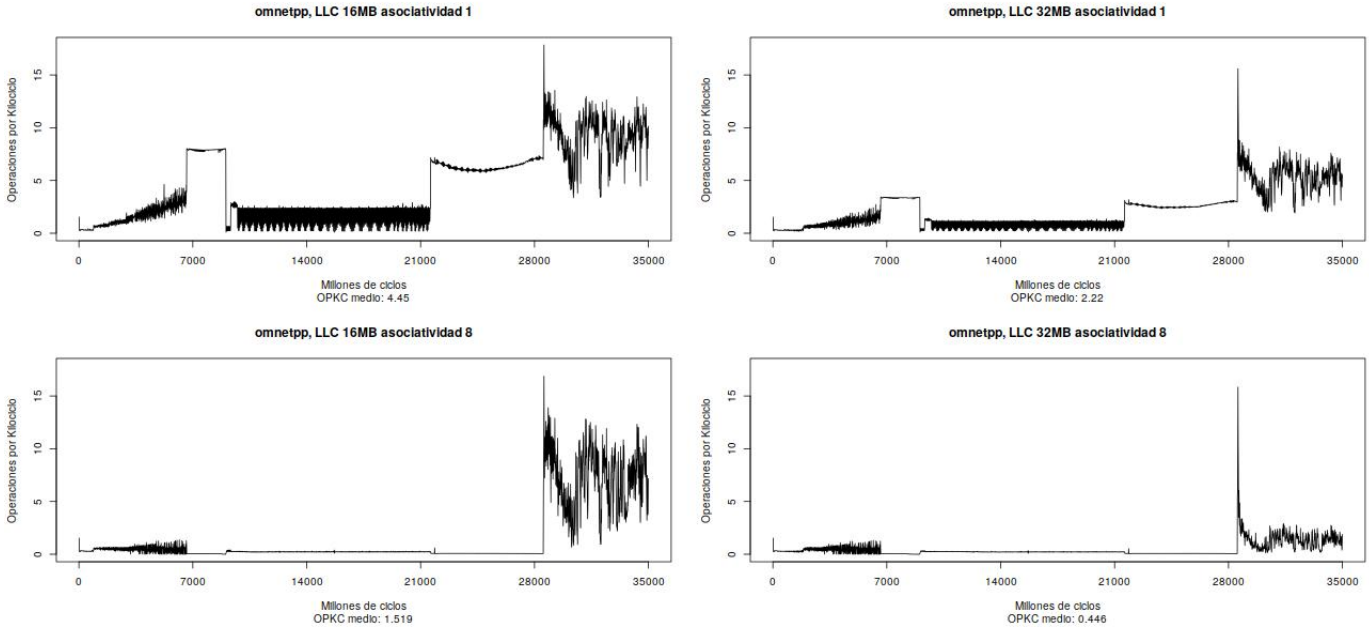


Figura 4.16: Perfil de memoria de omnetpp

Una aplicación que es muy sensible tanto al cambio de tamaño como de asociatividad es "milc" (figura 4.17). La configuración de 16MB y asociatividad 1 resulta ser patológica, produciendo muchos accesos innecesarios.

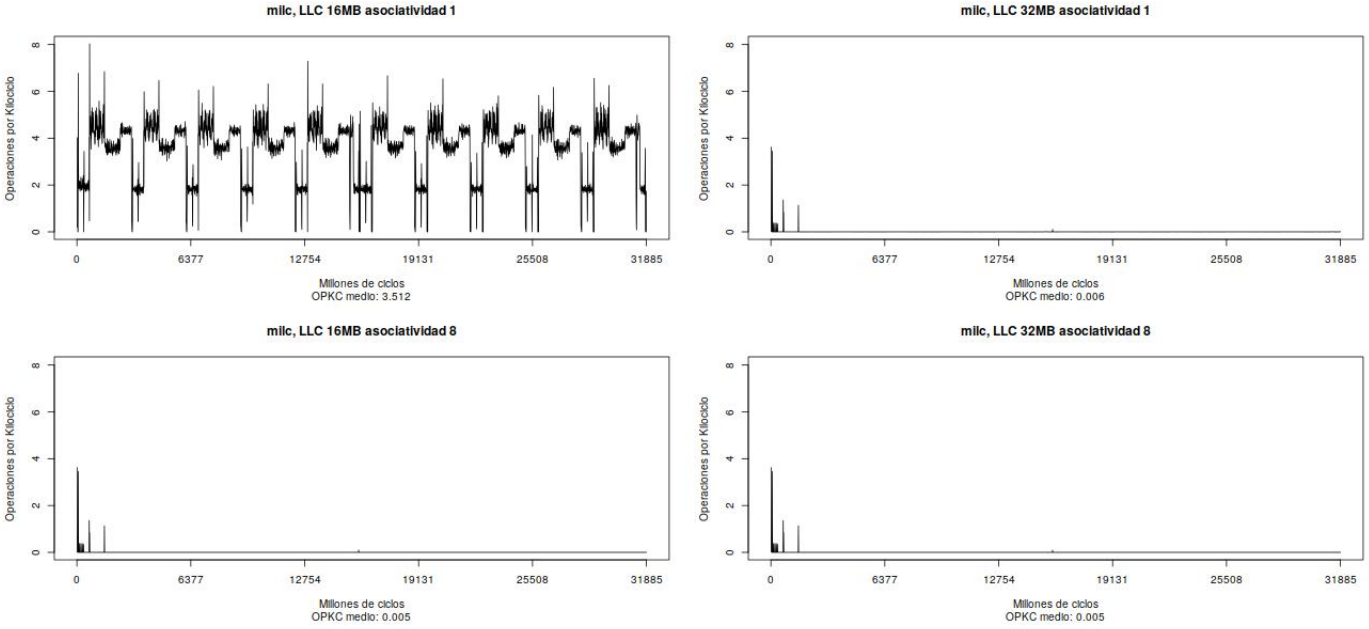


Figura 4.17: Perfil de memoria de milc

Todos los gráficos se han realizado tras un preprocesado simple de los datos sobre la densidad. Puesto que las aplicaciones se ejecutan durante miles de millones de ciclos, es necesario, para representar gráficamente los resultados, calcular el OPKC medio en secciones de mayor tamaño. En todos los casos el tamaño de las secciones se ha ajustado para que la curva que refleja el perfil de acceso conste de 3000 puntos. En el anexo D se encuentra información sobre los perfiles de memoria de las demás aplicaciones.

Capítulo 5

Mecanismo de prebúsqueda

Hasta ahora hemos examinado el comportamiento de la localidad temporal y algorítmica. La localidad temporal nos ha servido para agrupar aplicaciones en función de lo que tardan en volver a acceder a una misma posición de memoria, mientras que la localidad algorítmica ha sido estudiada a través del perfil de accesos a memoria. Sin embargo, todo el estudio que se ha hecho hasta ahora es más bien descriptivo. En este capítulo examinaremos algo más complicado: la localidad espacial.

La localidad espacial de los programas se refiere, como ya hemos comentado anteriormente, a la elevada probabilidad de acceder a una zona cercana de memoria en un corto intervalo de tiempo. Esto es evidentemente más complicado de medir, puesto que no hay que vigilar una única posición de memoria, sino toda la zona de alrededor. También se vuelve algo subjetivo, pues para proporcionar medidas cuantitativas de la localidad espacial no hay que definir sólo un umbral de olvido, sino también delimitar la zona de memoria que se considera cercana.

El propósito de este trabajo es determinar una buena forma de organizar el sistema de memoria principal en dos niveles: una memoria SDRAM que actúa como caché de una memoria RRAM. En este apartado daremos, precisamente, la respuesta a esta pregunta.

Utilizar una memoria SDRAM como caché de otra más grande tiene varias implicaciones. Por un lado, la memoria SDRAM será una memoria grande, por lo que la caché podrá almacenar gran cantidad de información que la RRAM mantiene. Está destinado fundamentalmente a aplicaciones muy grandes y exigentes, con un gran volumen de utilización de memoria. Un ejemplo de estas aplicaciones son las destinadas al tratamiento de grandes cantidades de datos (*Big Data*), de las que tanto se habla hoy en día. El verdadero interés de esta situación pasa por suponer una memoria RRAM grande (por ejemplo, del orden de 256GB), que sabemos es más lenta que las memorias RAM dinámicas, y colocar una SDRAM como caché en el nivel superior (por ejemplo, del orden de 16GB). En el Capítulo 2 pudimos comprobar que la creación de una jerarquía de memoria nos permite disfrutar de las ventajas de ambas partes: mantenemos el carácter no volátil de la RRAM, pero permitimos un acceso más rápido a través de una caché SDRAM.

Por otro lado, este sistema será siempre más lento que uno que sólo posea una SDRAM de suficiente tamaño. Al añadir más niveles y sustituir la SDRAM por una RRAM funcionando de

apoyo principal, se pierde la rapidez de acceder únicamente a una memoria y aumenta el consumo de energía, pues hay que añadir el mecanismo de caché, definir la política de reemplazo, crear nuevas conexiones, etc.

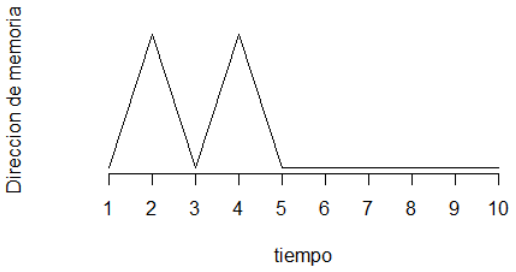
El estudio de la localidad espacial que se realiza en este apartado es mucho más que descriptivo: veremos cómo podemos utilizar esta información para diseñar un mecanismo de **prebúsqueda**. Un sistema de prebúsqueda ubicado en la SDRAM se encarga de analizar la secuencia de peticiones de acceso que proceden de los niveles superiores de la jerarquía y predecir las próximas. Por simplicidad y por no añadir ruido innecesario, nos centraremos únicamente en los accesos de lectura y escritura, dejando de lado los reemplazos. Si la memoria SDRAM dispusiera de un método eficaz de prebúsqueda, podría realizar peticiones de acceso a la RRAM situada en la capa inferior mientras está ociosa, de manera que, si finalmente la predicción se cumple, el acceso a la línea será más rápido al no tener por qué llegar hasta el nivel de RRAM. Suponiendo una caché SDRAM suficientemente grande, podemos permitirnos que la prebúsqueda sea menos fina, tomar menos riesgos, y traer más porciones de memoria aunque la probabilidad de usarlas no sea tan alta.

Sin embargo, analizar la secuencia de accesos reciente no es para nada sencillo. El comportamiento de las aplicaciones está lejos de ser reconocible, al menos si nos fijamos únicamente en el comportamiento reciente sin realizar ninguna discriminación. En la figura 5.1 podemos comprobar los 10 primeros accesos que llegan al sistema de memoria RAM suponiendo una LLC de 16MB y una asociatividad por conjuntos de nivel 1. En la izquierda, los diez accesos se muestran a escala real. Da la impresión de que los accesos 2 y 4 son extraños, y que el resto se mantiene en una línea horizontal. Sin embargo, en la figura de la derecha se ha representado únicamente la parte inferior. Podemos comprobar como, lejos de ser una línea recta, se repite la distinción entre direcciones de memoria superiores e inferiores. Adicionalmente se muestra la tabla con las direcciones virtuales exactas utilizadas.

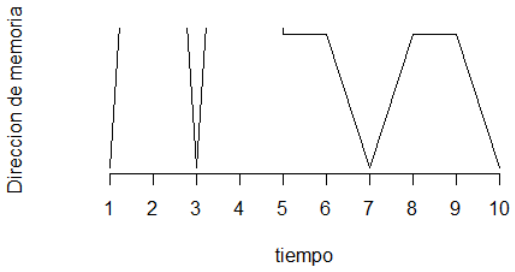
Podemos distinguir fácilmente tres zonas diferentes en una simple secuencia de diez accesos:

- Los accesos 1, 3, 7 y 10 corresponden al rango de direcciones 0x7FB01E60XXXX. Son las direcciones más bajas.
- Los accesos 2 y 4 corresponden al rango 0x7FFEDCDFXXXX y son las direcciones más altas. Tanto, que en la figura 5.1 (b) no aparecen.
- Los accesos 5, 6, 8 y 9 corresponden al rango 0x7FB01E83XXXX, el intermedio entre los tres.

Las principales causas de que se produzcan accesos a tres zonas diferentes de memoria es que un proceso rara vez está dedicado exclusivamente a una tarea, sino que realiza varias intercaladamente. El comienzo de la ejecución de un programa es especialmente interesante, pues aún no se ha estabilizado su comportamiento.



(a) Sin ampliar



(b) Ampliado

Número de acceso	Dirección de memoria
1	0x7FB01E60BC00
2	0x7FFEDCDF2400
3	0x7FB01E60C980
4	0x7FFEDCDF23C0
5	0x7FB01E830E40
6	0x7FB01E830C40
7	0x7FB01E60C9C0
8	0x7FB01E831000
9	0x7FB01E8319C0
10	0x7FB01E60CA00

(c) Direcciones físicas

Figura 5.1: Primeros diez accesos a memoria de la aplicación astar (asociatividad 1, tamaño LLC 16MB)

Si en una simple secuencia de 10 accesos ya aparecen tres zonas de memoria, vamos a necesitar diseñar un procedimiento sofisticado de prebúsqueda que atienda a la posibilidad de acceso alternado a diferentes zonas en la memoria principal.

Este capítulo se ha dividido en diferentes secciones, cada una detallando diferentes aproximaciones para enfrentarnos a este problema. En la primera sección se aborda un modelo basado en una distribución binomial, con el objetivo de predecir el número de accesos a zonas cercanas de memoria. En la segunda sección exploraremos la posibilidad de utilizar la información inmediatamente anterior para decidir si un bloque de memoria merece ser cacheado o no. La última aproximación y la más útil de todas se basa en un modelo de Markov oculto para separar las zonas de memoria.

Por último, se diseñará un mecanismo de prebúsqueda basado en el modelo oculto de Markov que permita predecir accesos futuros a memoria. Para comprobar su funcionamiento, se propondrá un modelo de simulación de una caché SDRAM en el que se aplique la prebúsqueda. Por último, se mostrarán los resultados obtenidos para las 20 aplicaciones que se han venido considerando hasta ahora.

5.1. Primera propuesta: modelo binomial

Sea l_i la línea asociada a la i-ésima posición de memoria asignada a una determinada aplicación. Para caracterizar la localidad espacial debemos definir:

- Una ventana espacial, S, que distinga las líneas que pertenecen a la zona de memoria cercana a una porción.
- Una ventana temporal, T, que permita caracterizar si el acceso es cercano o no en el tiempo.

Definición 1. (Localidad espacial de una aplicación). Sea t_{ij} el tiempo en el que se accede a la línea l_i por j-ésima vez. Se considera que existe **localidad espacial** si se accede a otra línea cualquiera, l_k, de forma que $i - S \leq k \leq i + S$, con $k \neq i$, siempre que el momento de acceso se encuentre en el intervalo temporal definido, es decir, existe algún j' tal que $t_{ij} - T \leq t_{kj'} \leq t_{ij} + T$. En este caso, l_k pertenece al grupo de la localidad espacial de la línea l_i accedida por j-ésima vez: $l_k \in locspa(l_i, j)$.

Nótese que, según los principios de localidad temporal, suelen producirse varios accesos a una misma línea, de ahí la necesidad del doble subíndice.

La definición anterior nos dice que existe localidad espacial si accedemos a una ventana espacial en una determinada ventana temporal. Ambas ventanas se tienen en cuenta por los dos sentidos: la ventana espacial delimita la zona de memoria tanto por arriba como por abajo, y la ventana temporal tanto a lo que ha ocurrido antes como lo que ocurrirá después. Para nuestras tareas de predicción, será mucho más útil distinguir entre este pasado y futuro.

Definición 2. (Localidad espacial pasada y futura). Sea t_{ij} el tiempo en el que se accede a la línea l_i por j-ésima vez. Se considera que existe **localidad espacial futura** si se accede a otra línea cualquiera, l_k, de forma que $i - S \leq k \leq i + S$, con $k \neq i$, siempre que el momento de acceso se encuentre en el intervalo temporal definido, es decir, existe algún j' tal que $t_{ij} < t_{kj'} \leq t_{ij} + T$, y $l_k \in locspa_{fut}(l_i, j)$. Por otro lado, se considera **localidad espacial pasada** si $t_{ij} - T \leq t_{kj'} < t_{ij}$, y $l_k \in locspa_{pas}(l_i, j)$.

Sea Y_{ij} la variable aleatoria que indica el número de líneas que pertenecen al conjunto de la localidad espacial futura para el j-ésimo acceso de la línea i, para $i = 1, \ldots, K$, $j = 1, \ldots, n_i$. El valor K es el número de líneas de memoria asociadas al proceso, mientras que n_i es el número de veces que se ha accedido a la línea l_i. Formalmente, $Y_{ij} = |locspa_{fut}(l_i, j)|$. Siempre se cumple que $0 \leq Y_{ij} \leq 2S$. Si $Y_{ij} = 0$, entonces no se ha accedido a ninguna línea cercana en el espacio y en el tiempo; si $Y_{ij} = 2S$ se han accedido a todas las que conforman la zona de localidad espacial.

Supongamos que Y_{ij} sigue una distribución binomial: $Y_{ij} \sim B(2S, p_{ij})$. Esto significa que todas las líneas que se encuentran dentro de la ventana espacial tienen la misma probabilidad de ser usadas. Esta probabilidad, p_{ij}, es la probabilidad de acceder a cualquiera de las líneas dentro de la ventana espacial y temporal, considerando sólo el futuro. Si somos capaces de modelar p_{ij}, podemos discernir si resulta rentable traer **el bloque entero** de direcciones de memoria a la caché. Si p_{ij} es grande, se espera que al llevar todo el bloque de direcciones desde l_{i-S} hasta l_{i+S} acertemos en gran parte.

Sin embargo, esta suposición está lejos de ser cierta. Algo más realista sería suponer que no es igual de probable acceder a todas las líneas cercanas, y que la probabilidad va decreciendo a medida que nos alejamos. De esta manera, pasaríamos a modelar variables aleatorias de Bernouille y no un proceso binomial directamente.

Bajo este supuesto, sea Y_{ijk} una variable indicadora que vale 1 si la línea $l_k \in locspa_{fut}(l_i, j)$, es decir, se accede a l_k en la ventana espacial y temporal definida, y sea p_{ijk} la probabilidad de que este suceso ocurra. Ahora $Y_{ij} = \sum_{k=i-S, k\neq i}^{i+S} Y_{ijk}$. En lugar de tomar $p_{ijk} = p_{ij}$ $\forall k \in \{i-S, \ldots, i-1, i+1, \ldots, i+S\}$ como hacíamos antes, parece más razonable modelar $p_{ijk} = \alpha^k p_{ij}$, con $\alpha \in (0, 1)$. Esto garantiza que la probabilidad de acceder a una línea va disminuyendo conforme nos alejamos.

Para modelar y obtener una estimación de p_{ij} y α podríamos utilizar un enfoque parecido a la regresión logística. Sea X_{ij} otra variable aleatoria asociada a la localidad espacial, pero esta vez a la localidad espacial pasada. X_{ij} indica el número de líneas que pertenecen al conjunto de la localidad espacial pasada para el j-ésimo acceso de la línea i, con $i = 1, \ldots, K$, $j = 1, \ldots, n_i$. Formalmente, $X_{ij} = |locspa_{pas}(l_i, j)|$. Siempre se cumple que $0 \leq X_{ij} \leq 2S$.

Un modelo razonable para las probabilidades futuras sería modelar de forma lineal el *logit* de la probabilidad:

$$log\left(\frac{p_{ij}}{1 - p_{ij}}\right) = \beta_0 + \beta_1 X_{ij}$$

De esta forma,

$$p_{ijk} = \frac{e^{\beta_0+\beta_1 X_{ij}}}{1+e^{\beta_0+\beta_1 X_{ij}}} \cdot \alpha^k$$

Antes de entrar en el proceso de estimación de los parámetros del modelo, es necesario comprobar si las hipótesis distribucionales son ciertas o, por el contrario, no es posible continuar con un modelo así. Para ello vamos a crear un nuevo tipo de fichero dedicado a la localidad espacial.

5.1.1. Fichero de localidad espacial

La obtención de los **ficheros de localidad espacial**, de extensión *.locspa*, tal y como se describen a continuación, es una tarea muy ardua. El objetivo es disponer de los valores de las variables X_{ij} e Y_{ij} para poder desarrollar la estimación del modelo.

Tal y como está definido, X_{ij} contiene información sobre la localidad espacial pasada, mientras que Y_{ij} almacena lo relevante al futuro. Además de la separación en dos de la ventana temporal, también es interesante dividir en dos zonas la ventana espacial: las líneas situadas antes y después de la considerada. El comportamiento de los procesos puede tener una tendencia creciente o decreciente en la utilización de la memoria en función de la tarea que se esté realizando, y es interesante caracterizar este comportamiento.

De esta manera, la variable X_{ij}^1 se refiere a la localidad espacial pasada para las líneas sucedidas antes de l_i, mientras que X_{ij}^2 se utiliza para las líneas posteriores. Siguiendo la notación anterior:

$$X_{ij}^1 = \left|\{l_k : l_k \in locspa_{pas}(l_i, j) \wedge k < i\}\right| \tag{5.1}$$

$$X_{ij}^2 = \left|\{l_k : l_k \in locspa_{pas}(l_i, j) \wedge k > i\}\right| \tag{5.2}$$

De forma que, combinando (5.1) y (5.2), se tiene de forma trivial

$$X_{ij} = X_{ij}^1 + X_{ij}^2 \tag{5.3}$$

Análogamente, se realiza la misma distinción para la localidad espacial futura que denotábamos como Y_{ij}.

$$Y_{ij}^1 = \left|\{l_k : l_k \in locspa_{fut}(l_i, j) \wedge k < i\}\right| \tag{5.4}$$

$$Y_{ij}^2 = \left|\{l_k : l_k \in locspa_{fut}(l_i, j) \wedge k > i\}\right| \tag{5.5}$$

$$Y_{ij} = Y_{ij}^1 + Y_{ij}^2 \tag{5.6}$$

El fichero de localidad espacial parte del procesamiento del fichero de accesos, tiene tantas filas como accesos a memoria haya producido la aplicación, y 7 columnas:

- El momento de acceso a la línea, medido en número de accesos.

- La dirección de memoria involucrada.
- El número de veces que se ha accedido a esta línea, contando con éste.
- El valor de X_{ij}^1.
- El valor de X_{ij}^2.
- El valor de Y_{ij}^1.
- El valor de Y_{ij}^2.

Puesto que para cada acceso individual tienen que examinarse posiciones cercanas de memoria y contar cuántos ha habido, se requiere mucho tiempo para obtener un fichero de localidad espacial. En el ejemplo de Firefox que venimos manejando, hay 7.248.108 accesos a memoria, aunque otras aplicaciones manejan ficheros con mayor número de ellos.

En la tabla 5.1 se encuentra un ejemplo de fichero de localidad espacial. Concretamente, se muestra el comienzo para la aplicación astar tomando una LLC de 16MB y asociatividad 1. La ventana temporal se ha establecido como $T = 1000$, y la ventana espacial a $S = 100$. Este fichero está ordenado alfabéticamente por *tag* y no por t, ya que de esta forma resultaba mucho más rápido el cálculo (aunque en este ejemplo el orden para los dos coincide).

t	**tag**	**numAcceso**	X_{ij}^1	X_{ij}^2	Y_{ij}^1	Y_{ij}^2
194	**0x558de3770040**	1	0	0	0	16
197	**0x558de3770080**	1	1	0	0	15
201	**0x558de37700c0**	1	2	0	0	14
202	**0x558de3770100**	1	3	0	0	13
204	**0x558de3770140**	1	4	0	0	12
205	**0x558de3770180**	1	5	0	0	11
207	**0x558de37701c0**	1	6	0	0	10
208	**0x558de3770200**	1	7	0	0	9
246	**0x558de3770280**	1	8	0	0	8

Tabla 5.1: Comienzo del fichero de localidad espacial para la aplicación astar, con LLC de 16MB y asociatividad 1.

En esta tabla podemos comprobar varias cosas. En primer lugar, la zona de memoria que aparece descrita conlleva una lectura en orden creciente. Esto se deduce de que el valor de X_{ij}^1 es cada vez mayor, pues cada vez hay más líneas anteriores que han sido leídas en un pasado reciente; de la misma forma, el valor de Y_{ij}^2 cada vez es menor, puesto que las líneas posteriores futuras se van leyendo poco a poco. Concretamente, esto marca una zona de 16 líneas de memoria que se leen en orden creciente.

Por otro lado, puesto que el valor de X_{ij}^2 y de Y_{ij}^1 es 0 siempre, no hay líneas posteriores leídas en el pasado ni tampoco líneas anteriores futuras, es decir, no se está leyendo de adelante hacia atrás.

Ahora que ya tenemos un fichero adecuado con el que podemos poner en práctica este modelo, examinemos si resulta factible.

5.1.2. Violación de las suposiciones

De los dos modelos propuestos al comienzo de esta sección, el primero tomaba una distribución binomial para Y_{ij}, y el segundo asume que las probabilidades decrecen a medida que nos alejamos de l_i. Con el objetivo de comprobar si Y_{ij} sigue una binomial, representaremos su histograma para el caso de astar (figura 5.2).

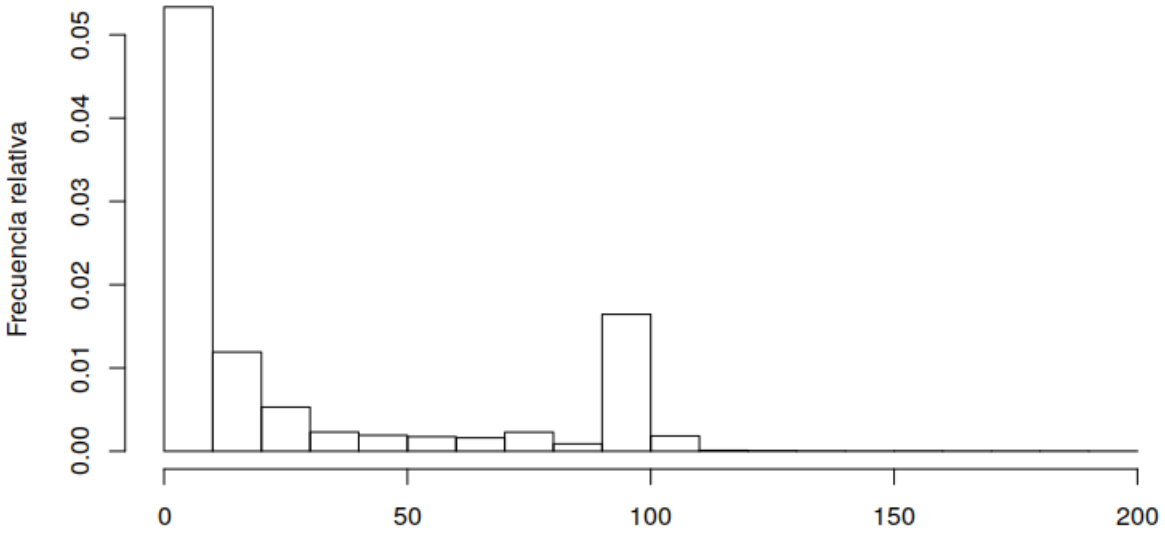
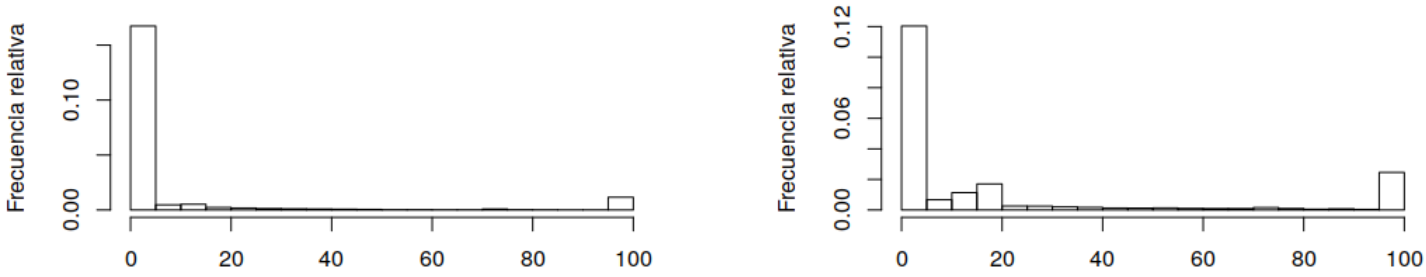


Figura 5.2: Histograma para Y_{ij} en el caso de astar, LLC 16BM y asociatividad 1.

A la vista del gráfico es imposible suponer que Y_{ij} sigue una distribución binomial, pues se presenta una distribución bimodal con un pico cercano a 0 y otro pico cercano a S. Situaciones similares se presentan en otras aplicaciones. Esto invalida el primer modelo presentado.

En la figura 5.3 podemos comprobar que, además de no poder suponer un modelo binomial, la distribución no es la misma para Y_{ij}^1 y Y_{ij}^2.

En cuanto a las suposiciones del segundo modelo, que establecen que la probabilidad de acceder a una línea decrece con la distancia, tampoco se pueden suponer ciertas. Que los histogramas de la figura 5.3 muestren picos en el valor $S = 100$ indica que hay determinadas zonas en la



(a) Histograma para Y_{ij}^1

(b) Histograma para Y_{ij}^2

Figura 5.3: Histograma para Y_{ij} en el caso de astar, LLC 16BM y asociatividad 1, separando por zona de memoria.

memoria que se leen consecutivamente, tanto en una orientación como en la otra. En estas zonas, la probabilidad de leer una línea posterior (cercana) es la misma independientemente de que se encuentre al principio o al final de la ventana espacial. Por tanto, no se verifica que $p_{ijk} = p_{ij}\alpha^k$. Un modelo alternativo con $p_{ijk} = p_{ij}\alpha_i^k$ podría explorarse, pero parece demasiado complejo, y ya hemos comprobado que éste no es un buen camino.

Por tanto, una vez desechados los modelos binomiales, pasamos a otra aproximación, que utiliza igualmente el fichero de localidad temporal.

5.2. Segunda propuesta: información inmediata anterior

Como hemos podido comprobar en el ejemplo de astar de la tabla 5.1, existe una fuerte relación entre X_{ij}^1 y Y_{ij}^2, así como también entre X_{ij}^2 y Y_{ij}^1. Esto parece lógico, pues el pasado de lo que ha sucedido detrás puede ayudarnos a pedecir el futuro de lo las líneas posteriores, y viceversa.

Si estamos en una zona de memoria grande donde leemos secuencialmente hacia delante y la ventana espacial es lo suficientemente pequeña, es bastante probable que $X_{ij}^1 = Y_{ij}^2 = S$. En este caso, podríamos utilizar directamente X_{ij}^1 para decidir si merece la pena cachear todo el bloque posterior, y lo mismo se aplica con X_{ij}^2 para cachear el bloque anterior.

La tabla 5.2 muestra las correlaciones entre estas cuatro variables, medidas para astar, con una LLC de 16MB y asociatividad 1. Aquello que podíamos intuir sobre la relación de X_{ij}^1 y Y_{ij}^2, y entre X_{ij}^2 y Y_{ij}^1, se confirma conociendo la estructura de correlación. Este par de casos son los únicos con una correlación significativa.

En la figura 5.4 se encuentra un gráfico de dispersión para Y_{ij}^2 frente a X_{ij}^1. Cuanta más densidad de puntos haya en una zona, quiere decir que mayor es la frecuencia de esa situación. Por ejemplo,

	X_{ij}^1	X_{ij}^2	Y_{ij}^1	Y_{ij}^2
X_{ij}^1	1.0000000	-0.1597534	-0.1858305	0.9274238
X_{ij}^2	-0.1597534	1.0000000	0.9160947	-0.1767372
Y_{ij}^1	-0.1858305	0.9160947	1.0000000	-0.1541357
Y_{ij}^2	0.9274238	-0.1767372	-0.1541357	1.0000000

Tabla 5.2: Correlaciones entre las variables X_{ij}^1, X_{ij}^2, Y_{ij}^1 y Y_{ij}^2, para astar, LLC 16MB, asociatividad 1.

en la zona cercana al $(0, 0)$ hay muchos puntos, lo que nos dice que si X_{ij}^1 es pequeño, con gran probabilidad Y_{ij}^2 también lo será. Además, parece haber una cierta tendencia creciente.

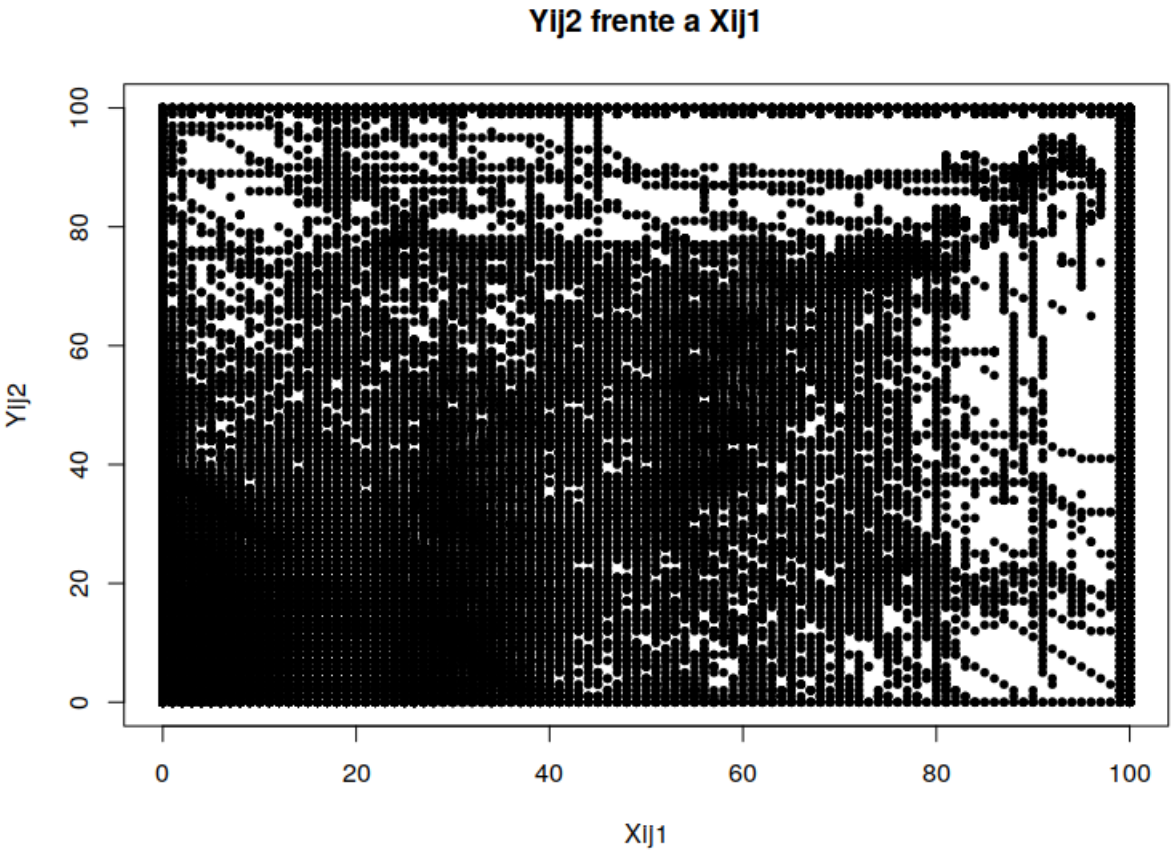


Figura 5.4: Gráfico de dispersión de Y_{ij}^2 sobre X_{ij}^1, para astar, LLC 16MB, asociatividad 1.

Sin embargo, hay muchos aspectos de este gráfico que resultan negativos para nuestro propósito:

- Aunque hay una mayor probabilidad de encontrar el par (X_{ij}^1,Y_{ij}^2) en torno a la bisectriz del primer cuadrante, un número no despreciable de veces tiene un comportamiento totalmente alejado del esperado. Por ello, no siempre X_{ij}^1 va a resultar un buen predictor de Y_{ij}^2.
- Es común que $Y_{ij}^2 = 100$, independientemente del valor de X_{ij}^1. Lo mismo ocurre en la situación contraria.

Por estas dos razones, no resulta adecuado utilizar X_{ij}^1 como predictor de Y_{ij}^1. Tampoco parece adecuada una regresión lineal, dada la distribución de los puntos. Aunque no se hayan mostrado explícitamente en este trabajo, ocurren situaciones similares al representar las distribuciones de otras aplicaciones, o al mostrar el gráfico de Y_{ij}^1 frente a X_{ij}^2.

Hasta ahora hemos intentado construir modelos en base a la localidad espacial de la definición 1. En la siguiente sección exploraremos un método que nos llevará a resultados mucho mejores, así como al mecanismo de prebúsqueda anunciado al comienzo de este capítulo.

5.3. Tercera propuesta: modelo oculto de Markov

A lo largo de la ejecución de cualquier proceso se producen comportamientos sistemáticos en la memoria. Al examinar detenidamente los accesos, podemos darnos cuenta de que existen ciertos patrones, pero que normalmente se encuentran entremezclados entre sí y con un cierto ruido totalmente no informativo. Hay que tener en cuenta que estamos examinando los accesos a memoria que caen fuera de la jerarquía de caché (figura 3.1, página 65), y que por tanto, muchas solicitudes quedan enmascaradas por los aciertos de la caché superior. A este nivel, debemos trabajar con cierta información que, fuera de contexto, no podemos considerar relevante. De hecho, cuanto mejor sea la caché de niveles superiores, menos información podremos extraer.

Antes de entrar en materia, presentaremos un ejemplo motivador que continuaremos utilizando para ilustrar el funcionamiento de los procedimientos desarrollados.

Supongamos un vector A de tamaño t_A que contiene números decimales almacenados en memoria, y que queremos obtener la suma de todos los elementos que forman parte de A. Un algoritmo muy sencillo para este objetivo pasaría por recorrer todos los elementos de A siguiendo un orden secuencial. Supongamos $t_A = 1000$. Si el proceso no está realizando ninguna actividad adicional, se espera que los accesos a memoria durante el cálculo de la suma sean secuenciales desde la primera posición de A hasta la última. Por simplicidad, vamos a suponer que A está almacenado en porciones contiguas y que cada línea de memoria tiene el tamaño exacto para almacenar un número en punto flotante; por tanto, hay 1000 líneas dedicadas al vector A.

Sea $Y(t)$ la línea de memoria a la que se ha accedido en tiempo t, medido siempre en número de accesos. Si representamos la ventana de $Y(t)$ frente a t en el tramo donde se realiza el cálculo descrito, debería ser algo similar a lo que se muestra en la figura 5.5. Nótese que $Y(t)$ ha sido transformado y que el mínimo valor que toma en la representación es 0, aunque esto no tiene por qué ser así. En este caso la predicción de los futuros accesos es más que evidente, puesto que al ser lectura secuencial, cae todo en la misma línea recta.

Considérese ahora otro vector B de longitud $t_B = 1000$ guardado en bloques contiguos de memoria, pero almacenado en orden inverso al vector A. Imagínese que tanto A como B son vectores ordenados y que se han calculado como parte de la recursividad del conocido algoritmo *mergesort* para ordenar un vector. Este algoritmo de ordenación rápida, con una complejidad

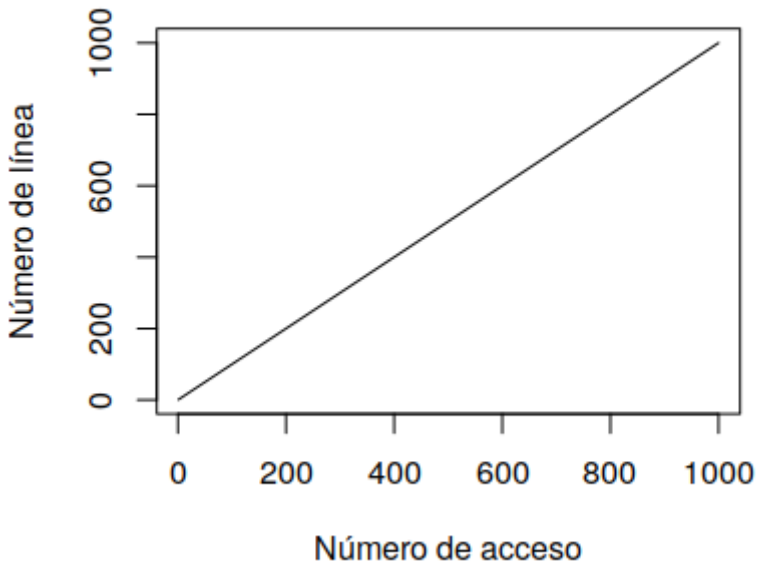


Figura 5.5: Representación simplificada de accesos a memoria para el cálculo de la suma de los elementos de A.

temporal de $O(n \log n)$, se divide en dos partes [20]. En la primera, se ordenan recursivamente dos vectores de la mitad de longitud; en la segunda, se combinan estos dos vectores ordenados.

El paso de combinación en el algoritmo *mergesort* es sencillo. Como se parte de que tanto A como B están ordenados, sólo es necesario comparar el primer elemento de cada uno de ellos, elegir el menor, y colocarlo en un vector que contendrá el resultado. Una vez eliminado el elemento seleccionado, se procede de igual forma hasta que se hayan considerado todos ellos (véase la figura 5.6 para más detalles).

En términos de accesos a memoria, estaremos accediendo a los dos vectores simultáneamente, por lo que el resultado al representar $Y(t)$ frente a t no será algo tan sencillo como lo de la figura 5.5. En la figura 5.7 (a) se puede comprobar cómo podría cambiar la estructura de accesos a memoria al considerar la lectura secuencial de A y B intercalada (recuérdese que B está almacenado en orden inverso). Por otro lado, no es improbable que el proceso se encuentre realizando otras tareas en segundo plano y genere otros accesos a memoria nada relevantes con el algoritmo principal. En el caso de la figura 5.7 (b) se ha añadido un cierto ruido para poder simular de forma más realista el patrón de accesos a memoria. En ambas tenemos dos grupos claros de direcciones de memoria que nos gustaría clasificar.

Así pues, queda claro que el reconocimiento de patrones no es una tarea precisamente simple, y en la mayoría de aplicaciones es más complejo que lo representado en la figura 5.7. En esta sección, describiremos un método que permite **reconocer zonas de memoria**, para poder separar comportamientos diferentes con una estructura más simple, y poder analizar cada uno de ellos por

```
algorithm merge(A, B) is
    inputs A, B : list
    returns list

    C := new empty list
    while A is not empty and B is not empty do
        if head(A) ≤ head(B) then
            append head(A) to C
            drop the head of A
        else
            append head(B) to C
            drop the head of B

    // By now, either A or B is empty. It remains to empty the other input list.
    while A is not empty do
        append head(A) to C
        drop the head of A
    while B is not empty do
        append head(B) to C
        drop the head of B

    return C
```

Figura 5.6: Pseudocódigo de la combinación de dos vectores ordenados para *mergesort*.

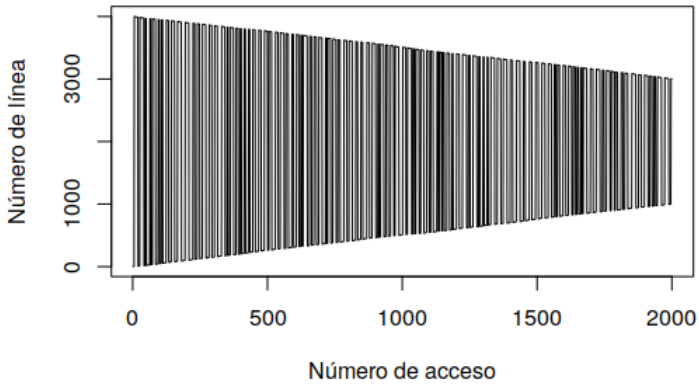
separado.

Sea $Y(t)$ la línea accedida en tiempo t. Cómo referencia, se toma la primera línea que ha sido asignada para el programa como la línea número 0. Así mismo, vamos a suponer que todas las líneas pertenecen a un grupo. Sea $S(t)$ el grupo al que pertenece la línea que se ha leído en el instante t. Volviendo al ejemplo introductorio, tendríamos dos grupos que consideramos relevantes. Sin contar con el ruido, podríamos asignar $S(t) = q_1$ si en el instante t se ha accedido a una posición del vector A y $S(t) = q_2$ si ha sido para el vector B. En este apartado consideraremos un **modelo oculto de Markov** para explicar y clasificar cada línea en un grupo de memoria.

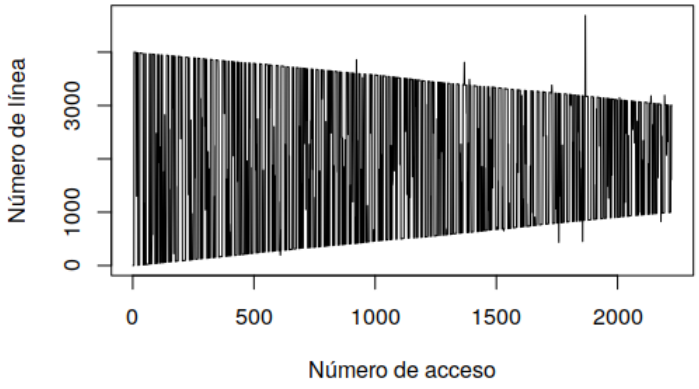
Un modelo oculto de Markov [21] es un modelo estadístico que supone un proceso de Markov sobre el sistema que se quiere describir, pero con parámetros desconocidos. El objetivo suele ser la estimación de estos parámetros desconocidos, a partir de otros parámetros observables. Una vez determinados, pueden servir para realizar un **reconocimiento de patrones**. Entre sus aplicaciones más sonadas, se encuentran las series temporales, reconocimiento del habla, o la bioinformática.

Mientras que en un modelo de Markov normal, el estado es visible, no ocurre así en un modelo oculto. Sólo se observan las variables que se ven influidas por el estado, y cada estado tiene una distribución de probabilidad asociada. En nuestro caso, $S(t)$ es el estado oculto, mientras que $Y(t)$ es la variable observada asociada a ese estado, y que depende evidentemente del valor de $S(t)$.

El diagrama representado en la figura 5.8 muestra la arquitectura general de un modelo de Markov oculto. El valor de la variable oculta $S(t)$ sólo depende de lo que ha ocurrido en el



(a) Sin ruido



(b) Con ruido

Figura 5.7: Representación simplificada de accesos a memoria en la combinación de un *mergesort*

instante anterior, es decir, en $S(t-1)$. Esto es lo que se conoce comúnmente como la **propiedad de Markov**.

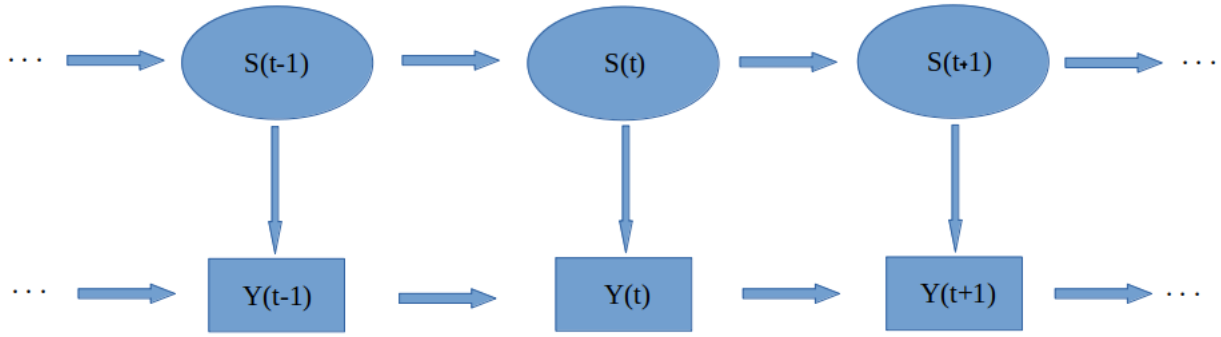


Figura 5.8: Esquema del funcionamiento de un modelo oculto de Markov

A continuación se define formalmente lo que es un Modelo Oculto de Markov.

Definición 3. (Modelo oculto de Markov). Un modelo oculto de Markov es una quíntupla (Q, V, π, A, B) donde:

- $Q = \{q_1, \ldots, q_N\}$ es el conjunto de estados.
- $V = \{v_1, \ldots, v_M\}$ es el conjunto de los valores observables.
- $\pi = \{\pi_1, \ldots, \pi_N\}$ es el conjunto que contiene las probabilidades iniciales; π_i dennota la probabilidad de que el estado inicial sea el estado q_i.
- $A = \{a_{ij} : q_i, q_j \in Q\}$, donde $a_{ij} = P(S(t) = q_j | S(t-1) = q_i)$ es la probabilidad de pasar al estado q_j en el instante t si el estado anterior era el q_i.
- $B = \{b_j(v_k) : q_j \in Q \wedge v_k \in V\}$, donde $b_j(v_k) = P(Y(t) = v_k | S(t) = q_j)$, es decir, la probabilidad de observar v_k cuando estamos en el estado q_j en el momento t.

Dada una secuencia de valores de salida $Y(t), t = 1, \ldots, T$, el problema de encontrar el conjunto de estados de transición más probable $S(t), t = 1, \ldots, T$ se resuelve con el algoritmo de Baum-Welch. No obstante, el modelo que nosotros tenemos no se adecúa exactamente a la definición 3, puesto que no es posible conocer de antemano el número de estados que conforman el conjunto Q. Además, este procedimiento tiene una complejidad temporal $O(n^2)$, lo que resulta muy contraproducente dado el enorme número de accesos a memoria a analizar (en algunos casos, millones). Por estos dos motivos, se propone una alternativa para la estimación de la secuencia $S(t), t = 1, \ldots, T$.

Para nosotros, el modelo oculto de Markov, siguiendo la definición 3, tiene las siguientes características:

- $Q = \{q_1, \ldots, q_N\}$, con N desconocido de antemano.

- El conjunto V contiene las direcciones de memoria que pueden ser accedidas. Como se direccionan líneas, y cada línea está compuesta por 64B, se verifica que $V \subseteq \{0x000000000000, 0x000000000040, \ldots, 0xFFFFFFFFFF80, 0xFFFFFFFFFFC0\}$
- Las probabilidades $\pi_i,\ i = 1, \ldots, N$ son desconocidas.
- Las probabilidades de transición a_{ij} también son desconocidas.
- Supondremos normalidad para la distribución de $Y(t)$, de forma que $Y(t)|S(t) = q_k \sim \mathcal{N}(\mu_k(t), \sigma_k^2(t))$ El valor de $Y(t)$ se toma como un número en base hexadecimal, y ya queda definida $b_j(v_k)$.

El procedimiento que se va a construir para estimar la secuencia $S(t), t = 1, \ldots, T$ es un procedimiento iterativo y voraz con una complejidad temporal $O(n)$. En el paso t, elegiremos el valor más plausible para $S(t)$, y se denotará como $\widehat{S(t)}$. Una vez que la asignación a un grupo haya sido hecha, no será reconsiderada más adelante.

El algoritmo para la estimación de la secuencia de grupos se describe mediante dos partes: hay que definir tanto los **valores iniciales** del procedimiento, como el **paso iterativo**. El paso iterativo se refiere a cómo estimamos el valor de $S(t+1)$, una vez que conocemos $\widehat{S(1)}, \ldots, \widehat{S(t)}$. Al comienzo se debe determinar cómo estimar los primeros valores de la secuencia para poder comenzar con el procedimiento iterativo.

5.3.1. Paso iterativo

Empezaremos describiendo el **paso iterativo**. Supongamos que estamos en el paso $t+1$, y que conocemos el valor $y(t+1)$. El problema que hay que resolver es: ¿a qué grupo asignamos el instante $t+1$? ¿Cuál es el valor más probable para $\widehat{S(t+1)}$? Definiremos en primer lugar aquel conjunto de estados de los que conocemos su existencia en tiempo t, para después mostrar la elección de $\widehat{S(t+1)}$.

Definición 4. (**Conjunto de estados conocidos**). Si $Q = \{q_1, \ldots, q_N\}$ es el conjunto total de estados, con N desconocido, se define el conjunto de estados conocidos en tiempo t, $Q_c(t) \subseteq Q$, como $Q_c(t) = \left\{ q \in Q : q \in \left\{ \widehat{S(1)}, \ldots, \widehat{S(t)} \right\} \right\}$

Podemos basar la decisión para $\widehat{S(t+1)}$ en cálculo de probabilidades. Sea A_{t+1} el suceso "obtener un valor para $Y(t+1)$ tan o más extremo que $y(t+1)$". Si conocemos el estado $S(t+1)$, entonces formalmente

$$\begin{aligned} A_{t+1}|\,[S(t+1) = q_k] = &\left[Y(t+1) - \mu_k(t+1) \geq |y(t+1) - \mu(t+1)|\,\right] \cup \\ &\left[Y(t+1) - \mu_k(t+1) \leq -|y(t+1) - \mu(t+1)|\,\right] \end{aligned} \tag{5.7}$$

Si conseguimos obtener, al menos una aproximación razonable, para $P(S(t+1) = q \mid A_{t+1}),\ \forall\ q \in Q_c(t)$, podremos decidir el estado más plausible teniendo

en cuenta la información observada (es decir, la línea de memoria). Podemos aproximar esta probabilidad con el **teorema de Bayes**.

$$\begin{aligned} P\left(S(t+1)=q_k \mid A_{t+1}\right) = & \frac{P\left(S\left(t+1\right)=q_k \wedge A_{t+1}\right)}{P(A_{t+1})} \propto P\left(S\left(t+1\right)=q_k \wedge A_{t+1}\right) = \\ & = P(S(t+1)=q_k) \cdot P(A_{t+1}|S(t+1)=q_k) \end{aligned} \tag{5.8}$$

donde el símbolo proporcional $\propto$ se utiliza para eliminar constantes de proporcionalidad que no dependan de k.

Para continuar con la ecuación (5.8) necesitamos estimar tanto $P(S(t+1)=q_k)$ como $P(A_{t+1}|S(t+1)=q_k)$. En cuanto a lo primero, no conocemos el número de estados que hay en total. Como no disponemos de más información, parece razonable tomar

$$P(S(t+1)=q_k) = \frac{1}{N} \tag{5.9}$$

aunque no podemos conocer su valor exacto, al desconocer N.

Con respecto a la estimación de $P(A_{t+1}|S(t+1)=q_k)$, lo que queremos obtener es una medida acerca de cuán verosímil es que los parámetros de la Normal de la que procede $y(t+1)$ sean $\mu_k(t+1)$ y $\sigma_k^2(t+1)$. El cálculo es exactamente igual al del p-valor en un test normal de dos colas. Es bien sabido que un test de dos colas es adecuado en el caso en que podamos considerar tan inverosímil alejarnos por arriba o por debajo de la media (como en este caso), y que en dicha situación el p-valor es el doble que el de un test unilateral (figura 5.9).

$$P(A_{t+1}|S(t+1)=q_k) = P\left(Z \geq \frac{|y(t+1)-\mu_k(t+1)|}{\sigma_k(t+1)}\right) + P\left(Z \leq \frac{-|y(t+1)-\mu_k(t+1)|}{\sigma_k(t+1)}\right) \tag{5.10}$$

donde $Z \sim \mathcal{N}(0,1)$.

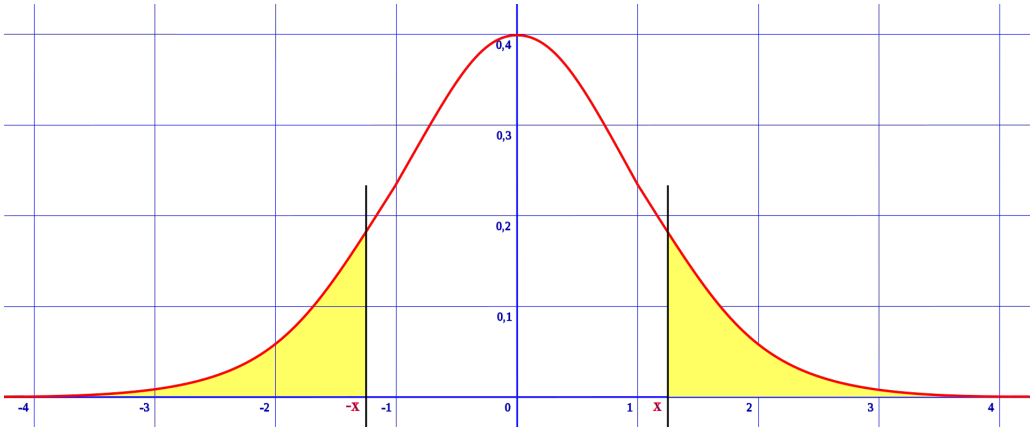


Figura 5.9: Test de dos colas aplicado a la distribución Normal.

Si Φ se refiere a la función de distribución de una $\mathcal{N}(0,1)$, y teniendo en cuenta que $\Phi(x) = 1 - \Phi(-x)$ por simetría (figura 5.9), el cálculo se simplifica a

$$P(A_{t+1}|S(t+1) = q_k) = 2\Phi\left(-\frac{|y(t+1) - \mu_k(t+1)|}{\sigma_k(t+1)}\right) \tag{5.11}$$

Combinando la información de (5.8), (5.9) y (5.11), podemos continuar desarrollando la expresión:

$$\begin{aligned} P\left(S(t+1) = q_k \mid A_{t+1}\right) &\propto P(S(t+1) = q_k) \cdot P(A_{t+1}|S(t+1) = q_k) = \\ &= \frac{2}{N}\Phi\left(-\frac{|y(t+1) - \mu_k(t+1)|}{\sigma_k(t+1)}\right) \propto \Phi\left(-\frac{|y(t+1) - \mu_k(t+1)|}{\sigma_k(t+1)}\right) \end{aligned} \tag{5.12}$$

La ecuación (5.12) muestra que la probabilidad de obtener un valor de $y(t+1)$ o más extremo, suponiendo que $S(t+1) = q_k$, es directamente proporcional a la probabilidad buscada en un principio. Hemos podido reducir la ecuación inicial utilizando símbolos proporcionales en lugar de igualdades porque lo que nos interesa determinar es el estado q_k donde se alcanza el **máximo** para $P\left(S(t+1) = q_k \mid A_{t+1}\right)$, y para ello no necesitamos el valor exacto. La relación de igualdad exacta es

$$P\left(S(t+1) = q_k \mid A_{t+1}\right) = \frac{2}{N \cdot P(A_{t+1})}\Phi\left(-\frac{|y(t+1) - \mu_k(t+1)|}{\sigma_k(t+1)}\right) \tag{5.13}$$

Como la primera parte no depende de k, queda claro así que

$$\arg\max_{q_k \in Q_c(t)} P\left(S(t+1) = q_k \mid A_{t+1}\right) = \arg\max_{q_k \in Q_c(t)} \Phi\left(-\frac{|y(t+1) - \mu_k(t+1)|}{\sigma_k(t+1)}\right) \tag{5.14}$$

De esta forma, elegiremos como valor para $\widehat{S(t+1)}$ aquel **estado conocido** solución de (5.14), pero siempre que se supere un umbral. Cabe la posibilidad de que la línea $y(t+1)$ sea una que pertenece a un estado aún no conocido, de ahí la necesidad de abrir la puerta para un nuevo estado y de establecer un umbral mínimo. Si no se llega a una mínima probabilidad, consideraremos que es el estado inicial de un nuevo grupo, ya que no resulta plausible ubicarlo en ninguno de los grupos actuales.

Definición 5. (**Estado más probable**). Sea $Q_c(t)$ el conjunto de estados conocidos en tiempo t, $p_{max}(t+1) = \text{máx}_{q \in Q_c(t)} P\left(S(t+1) = q \mid A_{t+1}\right)$, y p_{min} un umbral mínimo para la probabilidad. Se define el estado más probable en tiempo $t+1$, $q_{mp}(t+1)$, como sigue.

$$q_{mp}(t+1) = \begin{cases} \arg\max_{q \in Q_c(t)} P\left(S(t+1) = q \mid A_{t+1}\right) & \text{si } p_{max}(t+1) \geq p_{min} \\ \text{un nuevo estado } q_n, \text{con } q_n \notin Q_c(t) & \text{si } p_{max}(t+1) < p_{min} \end{cases}$$

Como estado asignado en el instante $t+1$ elegimos el estado más probable de la definición anterior: $\widehat{S(t+1)} = q_{mp}(t+1)$.

No obstante, según lo especificado en la ecuación (5.13), no podemos conocer directamente $p_{max}(t+1)$ al desconocer N y $P(A_{t+1})$, por lo que difícilmente podemos comprobar si $p_{max}(t+1) \geq p_{min}$.

Consideremos en su lugar $p'_{max}(t+1) = \text{máx}_{q_k \in Q_c(t)} \Phi\left(-\frac{|y(t+1)-\mu_k(t+1)|}{\sigma_k(t+1)}\right)$ y p'_{min} un umbral mínimo para la probabilidad $p'_{max}(t+1)$. Entonces es evidente que existe una biyección entre p_{min} y p'_{min} dada por la relación de igualdad de (5.13):

$$p_{min} = \frac{2p'_{min}}{P(A_{t+1}) \cdot N} \tag{5.15}$$

Esta biyección permite determinar el estado más probable de una manera práctica,

$$q_{mp}(t+1) = \begin{cases} \arg\max_{q_k \in Q_c(t)} \Phi\left(-\frac{|y(t+1)-\mu_k(t+1)|}{\sigma_k(t+1)}\right) & \text{si } p'_{max}(t+1) \geq p'_{min} \\ \text{un nuevo estado } q_n, \text{con } q_n \notin Q_c(t) & \text{si } p'_{max}(t+1) < p'_{min} \end{cases} \tag{5.16}$$

utilizando los valores de $\Phi\left(-\frac{|y(t+1)-\mu_k(t+1)|}{\sigma_k(t+1)}\right)$, sin necesidad de conocer las probabilidades reales, y asegurando la definición 5 (aunque sin conocer el valor exacto de p_{min}). La desventaja principal es que, si utilizamos el valor de p'_{min} y la equivalencia de la ecuación 5.16, el valor de p_{min} depende de t (a través de A_{t+1}). Esto provoca que, al fijar p'_{min} en el procedimiento iterativo de calcular el estado más probable, estamos considerando *diferentes* umbrales de probabilidad p_{min} en cada paso.

Para solucionar este problema podemos aprovecharnos de la estructura de probabilidad de la normal. Previsiblemente, $P(A_{t+1})$ será un valor no muy grande, pero que no podemos conocer al no conocer todos los estados. Las colas de la distribución normal no son pesadas, de manera que encontrar valores muy alejados de la media (digamos, por ejemplo, más de tres veces su desviación estándar) es poco probable. Si un valor está muy alejado de todos los grupos conocidos hasta el momento, todas sus probabilidades serán enormemente pequeñas, por lo que podemos permitirnos un umbral pequeño para p_{min}. Si la elección de p'_{min} es lo suficientemente pequeña, los cambios que sufre p_{min} no serán muy relevantes, pues serán de órdenes muy pequeños. No parece demasiado importante si $p_{min} = 0.001$ o $p_{min} = 0.0005$, puesto que cuando no pertenece a ningún grupo esperamos valores de $p_{max}(t+1)$ por debajo de ambos.

Teniendo esto en cuenta, queda la elección de un umbral adecuado p'_{min} que proporcione buenos resultados. Se ha comprobado empíricamente que un valor adecuado puede rondar en torno a $p'_{min} = 0.000025$. Éste será el valor utilizado en el procedimiento iterativo.

Otro de los asuntos que resta resolver para el cálculo del estado más probable según la ecuación 5.16 es la estimación de $\mu_k(t+1)$ y $\sigma_k(t+1)$, $\forall q_k \in Q_c(t)$. La estimación de estos valores es crítica para poder aproximar correctamente $\Phi\left(-\frac{|y(t+1)-\mu_k(t+1)|}{\sigma_k(t+1)}\right)$ y, por tanto, elegir de forma adecuada el estado más probable. Para ello introducimos una nueva definición, así como nueva notación que simplifique la expresión.

Definición 6. (**Estado más cercano**). Sea $Q_c(t)$ el conjunto de estados conocidos en tiempo t, y $q_k \in Q_c(t)$ un estado. Se define el estado más cercano a q_k en tiempo t como aquel estado $q_{k'}$ que verifica

$$q_{k'} = \arg\min_{q_i \in Q_c(t)} |\mu_k(t) - \mu_i(t)|$$

Sea $n_k(t)$ el número de elementos asignados al estado q_k en tiempo t, $n_k(t) = \sum_{i=1}^{t} I(\widehat{S(i)} = q_k)$. Los elementos de la secuencia $Y(t), t = 1, \ldots, t$ asignados al estado q_k se denotan como $y_{k,1}, y_{k,2}, \ldots, y_{k,n_k(t)}$.

La estimación de $\mu_k(t+1)$ y $\sigma_k(t+1)$, $\widehat{\mu_k(t+1)}$ y $\widehat{\sigma_k(t+1)}$ respectivamente, se actualiza cada vez que un nuevo elemento es asignado al estado correspondiente. El cálculo del estimador distingue entre varias situaciones.

- Si $n_k(t) = 1$ estamos hablando de un grupo con un sólo elemento. En este caso, se toma

$$\widehat{\mu_k(t')} = y_{k,1}, \quad t' \geq t \tag{5.17}$$

Por otro lado, obtener un buen valor para $\widehat{\sigma_k(t')}$, $t' \geq t$ es más complicado. Sea t_k el instante en el que se asignó la primera observación al estado q_k. Esto ocurrió porque la probabilidad de pertenencia a cualquier estado conocido en tiempo t_k no era lo suficientemente grande, por lo que se asigna dicha observación a un estado no conocido en ese momento. Sea $q_{k'}$ el estado más cercano a q_k en tiempo t_k. De esta forma se toma

$$\widehat{\sigma_k(t')} = \text{mín}\,(10\widehat{\sigma_{k'}(t_k)}, \sigma_{max}), \quad t' \geq t_k \tag{5.18}$$

donde σ_{max} es una cota superior para la estimación de $\sigma_k(t')$ y no puede superarse en ningún momento. La justificación de esta decisión es simple: se elige la desviación estándar del grupo más cercano porque se espera que el comportamiento sea similar al comienzo. Se multiplica por un factor de 10 para asegurar que la desviación estándar es lo suficientemente amplia como para permitir que las observaciones cercanas caigan en el nuevo grupo con una probabilidad razonable. Cuando haya más observaciones asignadas a este estado, entonces la estimación para la desviación estándar será más pequeña y se irá adaptando a las nuevas observaciones.

La cota superior σ_{max} se utiliza para evitar que la desviación estándar crezca más de la cuenta. Un valor plausible es $\sigma_{max} = 200$. Esto implica que el acceso a una línea con más de 400 líneas de diferencia con respecto a lo esperado se considera extraño (se toma $2\sigma_{max}$ al ser un umbral característico cuando se habla de observaciones atípicas en una distribución Normal).

Por otro lado, para evitar que la desviación estándar sea muy grande y se solape con las distribuciones de estados cercanos, se diseña una pequeña heurística de adaptación de la

desviación típica. En el momento de la creación del nuevo grupo, se comprueba si $\widehat{\sigma_k(t_k)} > \frac{|\mu_k(t_k)-\mu_{k'}(t_k)|}{2}$. Si se cumple esta condición, se toma como nueva estimación

$$\widehat{\sigma_k^{new}(t')} = \frac{\widehat{\sigma_k(t_k)}}{2}, \quad t' \geq t_k \tag{5.19}$$

Si la condición vuelve a violarse, el resultado vuelve a dividirse entre dos y así sucesivamente hasta que se verifique. Esto asegura que una distribución con una desviación estándar estimada muy grande no "robe" elementos a otro grupo.

- Si $1 < n_k(t) \leq 5$, hablamos de estados que tienen más de una observación, pero aún son grupos pequeños. En este caso, se actualiza la estimación de la media y se utiliza la media muestral:

$$\widehat{\mu_k(t')} = \frac{1}{n_k(t)} \sum_{i=1}^{n_k(t)} y_{k,i}, \quad t' \geq t \tag{5.20}$$

Sin embargo, la estimación de la desviación estándar no cambia y sigue siendo la definida en el momento de la creación del grupo.

- Si $n_k(t) > 5$, la estimación es más complicada. Sea $Y_{k,1}, \ldots, Y_{k,n_k(t)}$ las variables aleatorias que definen la secuencia accedida en el estado q_k. Consideramos un modelo de regresión lineal de la forma

$$Y_{k,i} = \beta_0 + \beta_1 \cdot i + \epsilon_{k,i}, \qquad i = 1, \ldots, n_k(t) \tag{5.21}$$

con $\epsilon_{k,i} \sim \mathcal{N}(0, \sigma_k^2)$.

El proceso de estimación de los coeficientes por el método de mínimos cuadrados para obtener $\hat{\beta}_0$ y $\hat{\beta}_1$ es ampliamente conocido [23]. Una vez conocida esta estimación, la predicción para una observación del análisis viene dada por

$$\widehat{\mu_{k,i}} = \hat{\beta}_0 + \hat{\beta}_1 \cdot i, \qquad i = 1, \ldots, n_k(t)$$

Puede comprobarse [23] que un estimador insesgado para σ_k se obtiene de

$$\widehat{\sigma_k} = \sqrt{\frac{1}{n_k(t) - 2} \sum_{i=1}^{n_k(t)} \left(Y_{k,i} - \widehat{\mu_{k,i}}\right)^2}$$

De esta manera, la actualización de los estimadores relativos al estado q_k utilizan la información del modelo lineal propuesto y

$$\widehat{\mu_k(t')} = \hat{\beta}_0 + \hat{\beta}_1 \cdot (n_k(t) + 1), \quad t' > t \tag{5.22}$$

$$\widehat{\sigma_k(t')} = \begin{cases} 1 & \text{si } \widehat{\sigma_k} \leq 1 \\ \widehat{\sigma_k} & \text{si } 1 < \widehat{\sigma_k} < \sigma_{max} \\ \sigma_{max} & \text{si } \widehat{\sigma_k} \geq \sigma_{max} \end{cases}, \quad t' > t \tag{5.23}$$

La estimación de la media según la ecuación 5.22 permite adaptarse a una tendencia en el uso de las líneas de memoria. Como ya hemos comentado, es habitual ir leyendo en un orden

secuencial, y esta forma de estimar la media permitiría esta adaptación. Por otro lado, la estimación de la desviación típica según la ecuación 5.23 es la más razonable, puesto que el modelo lineal ha separado la tendencia del ruido. Adicionalmente, se considera una cota inferior y superior. La cota superior sirve al mismo propósito que el explicado en el primer caso: evita que el valor se dispare. La cota inferior evita que se tome un valor tan pequeño que sea imposible considerar observaciones cercanas a un mismo grupo, además de evitar una estimación muy reducida para la desviación estándar de un hipotético nuevo grupo más cercano (nótese que en la creación de este nuevo grupo, se comenzaría con un valor muy pequeño para $\widehat{\sigma_k(t')}$ según la ecuación 5.18).

En la práctica, veremos como en ocasiones es conveniente ajustar un modelo de regresión robusta en lugar de un modelo de regresión lineal normal. Las ventajas de utilizar estos modelos fundamentalmente radican en la detección de *outliers* y puntos muy influyentes en el modelo. Existen procedimientos para ajustar un modelo reduciendo el peso de estas observaciones "raras", y en ocasiones provoca unos mejores resultados (para nuestro propósito) que un modelo lineal habitual. No es el objetivo de este trabajo profundizar en los modelos de regresión robusta, y por tanto no se mostrarán los fundamentos teóricos del mismo. No obstante, veremos como en la práctica resulta conveniente ajustar un modelo de este tipo, y elegir dinámicamente cuál de los dos es más apropiado. Puede consultar más información en [24].

Por último, se considera una limitación en el tamaño del modelo lineal. El cómputo empieza a hacerse pesado a medida que crece el valor de $n_k(t)$, así que en la práctica no se utilizan todos los valores en la regresión, sino que se limita a los últimos 500.

Con esto, ya tenemos completo el procedimiento para el paso iterativo. A grandes rasgos, los pasos que se siguen, suponiendo que nos encontramos en el instante $t+1$, son los siguientes:

1. Determinar el estado más probable, $q_{mp}(t+1)$, siguiendo las directrices de la ecuación 5.16.
2. Si el estado más probable es un estado conocido en tiempo t, entonces actualizamos la estimación de la media y la desviación típica relativa a dicho estado según lo especificado anteriormente.
3. Si el estado más probable es un estado nuevo, estamos creando un nuevo grupo. Debemos buscar el estado más cercano para obtener una estimación inicial de la desviación estándar y utilizar las ecuaciones 5.18 y 5.19 para determinar su valor.

5.3.2. Paso inicial

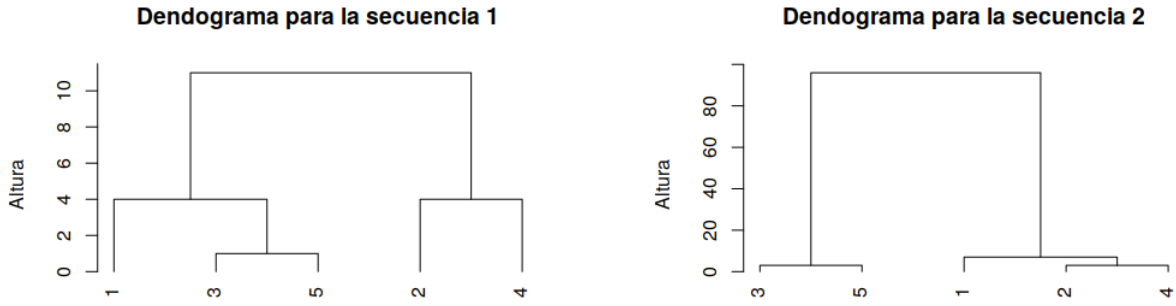
Ahora que está definido el proceso de actualización y asignación de estados de forma iterativa, queda por especificar cómo creamos el primer grupo. Una vez que hay un grupo, el procedimiento continúa de forma natural. Mientras sea plausible asignar observaciones a dicho estado, se

continuará haciendo. En el momento en que una observación quede lo suficientemente alejada, crearemos un nuevo grupo.

La elección del primer grupo es especialmente importante. Debemos asegurarnos de que realmente las observaciones escogidas forman parte de un mismo estado, pues si no estaremos cimentando un procedimiento sobre una mala base. Por ello, no está de más ser un poco más estrictos a la hora de elegir el primero.

Consideremos los cinco primeros valores de la secuencia de accesos, $y(1), y(2), y(3), y(4), y(5)$. Nos preguntamos si todos ellos pertenecen a un mismo estado. Una consecuencia directa de grandes diferencias en estas cinco observaciones sería la obtención de un dendograma que muestre claramente la existencia de más de un grupo.

Siguiendo un proceso similar al agrupamiento de aplicaciones (véase la sección 4), se propone un clustering ascendente jerárquico con el método de Ward para esta primera secuencia de observaciones. Para que sea más sencillo entender el criterio de decisión, se proponen dos ejemplos. Consideremos dos secuencias de valores, $\boldsymbol{x}_1 = (1, 8, 4, 12, 5)$ y $\boldsymbol{x}_2 = (1, 8, 97, 5, 94)$. En la primera, parece que se pueden considerar como un grupo, mientras que en la segunda parece haber dos. En la figura 5.10 se muestran los dos dendogramas asociados.



(a) Secuencia 1 (b) Secuencia 2

Figura 5.10: Dendogramas para las secuencias de ejemplo

A la vista de la figura 5.10 podemos considerar que la secuencia 2 tiene dos grupos bastante marcados, no ocurriendo lo mismo en la secuencia 1. Sabemos esto gracias a que el **cociente de alturas** es mucho mayor en el segundo caso.

Para la primera secuencia, las alturas (ordenadas) donde se producen las uniones en el dendograma son $h_{1,1} = 1, h_{1,2} = 4, h_{1,3} = 4, h_{1,4} = 11$. Si calculamos los cocientes de cada altura con la inmediata anterior, obtenemos $\frac{h_{1,2}}{h_{1,1}} = 4, \frac{h_{1,3}}{h_{1,2}} = 1, \frac{h_{1,4}}{h_{1,3}} = 2.75$. Al no haber grandes diferencias, se puede considerar como un único grupo. Por otro lado, para la segunda secuencia, las alturas son $h_{2,1} = 3, h_{2,2} = 3, h_{2,3} = 7, h_{2,4} = 96$. Esto provoca unos cocientes entre alturas más grandes: $\frac{h_{2,2}}{h_{2,1}} = 1, \frac{h_{2,3}}{h_{2,2}} = 2.3333, \frac{h_{2,4}}{h_{2,3}} = 13.71429$.

A la vista del dendograma y de las diferencias en las alturas, concluiríamos que la primera secuencia se considera un grupo, pero la segunda no. Para automatizar esto, podemos considerar un umbral de inicio, u_{ini}, de manera que si **todos los cocientes de alturas** son menores que u_{ini} se considera un estado válido.

Definición 7. (**Primera secuencia reconocida**). Sea $Y(t), t = 1, \ldots, T$ una secuencia de accesos a memoria. Se define la primera secuencia reconocida de longitud n a la secuencia $Y(i), \ldots, Y(i+n-1)$, con i mínimo, de forma que si $h_1, \ldots, h_{n-1}$ representan las alturas ordenadas calculadas para el dendograma según el método de Ward, se cumple que $\frac{h_i}{h_{i-1}} \leq u_{ini}$, $\forall\ i \in \{2, \ldots, n-1\}$

En nuestro procedimiento, buscaremos la primera secuencia reconocida de longitud 5 para formar el primer estado conocido. Sea $Y(i), \ldots, Y(i+4)$ esta secuencia. La asignación de estados para el comienzo del procedimiento es la siguiente:

$$\begin{aligned} \widehat{S(j)} &= \emptyset, \quad j = 1, \ldots, i-1 \\ \widehat{S(j)} &= q_1, \quad j = i, \ldots, i+4 \end{aligned} \tag{5.24}$$

donde el símbolo $\emptyset$ se utiliza para mostrar que dicha observación no se asigna a ningún estado, y q_1 se utiliza para representar el primer estado.

Por otro lado, la estimación de la media y la desviación estándar se hace como sigue:

$$\widehat{\mu_1(t')} = \frac{1}{5} \sum_{j=0}^{4} Y(i+j), \quad t' \geq i \tag{5.25}$$

$$\widehat{\sigma_1(t')} = \sqrt{\frac{1}{4} \sum_{j=0}^{4} \big(Y(i+j) - \mu_1(i+j)\big)^2}, \quad t' \geq i \tag{5.26}$$

que no es más que el cálculo de la media y cuasi-desviación muestral.

Un buen umbral, comprobado empíricamente, ha resultado ser $u_{ini} = 3$. Nótese que éste es un valor muy estricto: según este criterio, ni siquiera la secuencia 1 se considera como un grupo. Es preferible ser muy exigentes y despreciar más observaciones al principio, que ser permisivos, tomar un valor alto para u_{ini}, y que el primer estado sea asignado a una secuencia incorrecta.

5.3.3. Procedimiento completo

Ahora que está definido tanto el paso inicial como el paso iterativo, queda completado el procedimiento de reconocimiento para el Modelo Oculto de Markov. En la figura 5.11 se encuentra el pseucodódigo del procedimiento completo.

Cuando hemos comenzado a describir esta técnica, hemos comentado la posibilidad de que exista ruido en los accesos a memoria: ciertas peticiones de lectura o escritura que no se encuadren bien dentro de los demás grupos. Sin embargo, el procedimiento que hemos detallado no permite

Entrada: Secuencia de accesos a memoria (en número de línea), $Y(t),\ t = 1 \ldots, T$.
Salida: Estimación de los estados, $\widehat{S(t)},\ t = 1 \ldots, T$.
Algoritmo:

Encontrar la primera secuencia reconocida de longitud 5, $Y(i), \ldots, Y(i+4)$
Asignar $\widehat{S(j)} = \emptyset,\ j = 1, \ldots, i-1$
Asignar $\widehat{S(j)} = q_1,\ j = i, \ldots, i+4$
Calcular $\widehat{\mu_1(t')} = \frac{1}{5}\sum_{j=0}^{4} Y(i+j), \quad t' \geq i$
Calcular $\widehat{\sigma_1(t')} = \sqrt{\frac{1}{4}\sum_{j=0}^{4}\left(Y(i+j) - \mu_1(i+j)\right)^2}, \quad t' \geq i$
Para cada $t = i+5, \ldots, T$, **hacer**
Obtener el conjunto de estados conocidos, $Q_c(t-1)$
Determinar $p'_{max}(t) = \text{máx}_{q_k \in Q_c(t-1)}\ \Phi\left(-\frac{\left|y(t) - \widehat{\mu_k(t)}\right|}{\widehat{\sigma_k(t+1)}}\right)$
Obtener el estado más probable, $q_k = q_{mp}(t)$ (ecuación 5.16)
Asignar $\widehat{S(t)} = q_k$
Si $q_k \in Q_c(t-1)$ (probabilidad de pertenencia suficientemente alta), **entonces**
Si $n_k(t) \leq 5$, **entonces**
Actualizar $\widehat{\mu_k(t')} = \frac{1}{n_k(t)}\sum_{i=1}^{n_k(t)} y_{k,i}, \quad t' \geq t$
Si no, **entonces**
Plantear el modelo $Y_{k,i} = \beta_0 + \beta_1 \cdot i + \epsilon_{k,i}$
Estimar β_0 y β_1 considerando una regresión lineal: $\widehat{\beta_0^1}$ y $\widehat{\beta_1^1}$
Estimar β_0 y β_1 considerando una regresión robusta: $\widehat{\beta_0^2}$ y $\widehat{\beta_1^2}$
Calcular $\widehat{\mu_{k,i}^1} = \widehat{\beta_0^1} + \widehat{\beta_1^1} \cdot i,\ i = 1, \ldots, n_k(t)$
Calcular $\widehat{\mu_{k,i}^2} = \widehat{\beta_0^2} + \widehat{\beta_1^2} \cdot i,\ i = 1, \ldots, n_k(t)$
Calcular $\widehat{\sigma_k^1} = \sqrt{\frac{1}{n_k(t)-2}\sum_{i=1}^{n_k(t)}\left(Y_{k,i} - \widehat{\mu_{k,i}^1}\right)^2}$
Calcular $\widehat{\sigma_k^2}$ (se omiten detalles)
Determinar el mejor modelo, $j = \arg\min_{z=1,2}\widehat{\sigma_k^z}$
Actualizar $\widehat{\mu_k(t')} = \widehat{\beta_0^j} + \widehat{\beta_1^j} \cdot (n_k(t)+1), \quad t' > t$
Actualizar $\widehat{\sigma_k(t')} = \begin{cases} 1 & \text{si } \widehat{\sigma_k^j} \leq 1 \\ \widehat{\sigma_k^j} & \text{si } 1 < \widehat{\sigma_k^j} < \sigma_{max} \\ \sigma_{max} & \text{si } \widehat{\sigma_k^j} \geq \sigma_{max} \end{cases}, \quad t' > t$
Fin-si
Si no (creación de un nuevo grupo):
Asignar $\widehat{\mu_k(t')} = y_{k,1}, \quad t' \geq t$
Determinar el estado más cercano a q_k en tiempo t, $q_{k'}$
Calcular $\widehat{\sigma_k(t')} = \text{mín}\left(10\widehat{\sigma_{k'}(t_k)}, \sigma_{max}\right), \quad t' \geq t$
Mientras $\widehat{\sigma_k(t)} > \frac{|\mu_k(t) - \mu_{k'}(t)|}{2}$, **hacer**
Asignar $\widehat{\sigma_k(t')} = \frac{\widehat{\sigma_k(t)}}{2}, \quad t' \geq t_k$
Fin-mientras
Fin-si
Fin-para
Fin

Figura 5.11: Pseudocódigo para el reconocimiento de estados en el Modelo Oculto de Markov

detectar este ruido: si un acceso a memoria está alejado de los demás y no se puede encuadrar en ningún grupo, se creará un grupo nuevo para dicha observación. Si, por otro lado, es ruido pero está cerca de un grupo, entonces se asignará a un grupo ya existente empañando los accesos de dicho estado.

Tratar esta última situación es complicado: no es sencillo distinguir si un acceso pertenece realmente a un estado o si es ruido. Sin embargo, tampoco es muy relevante, pues el valor de los parámetros se estima, entre otras técnicas, con un procedimiento de regresión robusta. En este caso, las observaciones más extrañas tendrán menos peso en la estimación, de forma que aunque una esté alejada de las demás no influirá demasiado.

Por otro lado, sí que podemos solucionar la asignación de peticiones extrañas a estados aislados. Consideremos un umbral, n_g, que corresponde al mínimo número de observaciones que debe tener un grupo para considerarse un estado válido. Un valor adecuado para n_g se ha comprobado que es $n_g = 100$. Consideremos un nuevo estado q_0, que representa aquellas líneas de memoria que no siguen ningún patrón. Los accesos a memoria asociados a estados con menos de n_g observaciones, o asociados a ningún estado, pasan a formar parte del estado q_0.

De forma resumida, una vez que el pseudocódigo de la figura 5.11 ha devuelto la estimación de los estados, $\widehat{S(t)},\ t = 1\ldots, T$, se modifican aquellos con la siguiente regla:

$$\text{Si } \widehat{S(t)} = \emptyset \text{ ó } \widehat{S(t)} = q_k \text{ con } n_k(T) < n_g, \text{ entonces } \widehat{S^{new}(t)} = q_0, \text{ para cada } t = 1\ldots, T \quad (5.27)$$

Como medida del éxito del procedimiento de reconocimiento de patrones, se toma el porcentaje de observaciones que han sido clasificados a un estado diferente al q_0.

Definición 8. (**Proporción de accesos reconocidos**). Sea $\widehat{S(t)},\ t = 1\ldots, T$ la secuencia de asignación de estados tras la modificación propuesta (5.27), y $n_k(T)$ el número de elementos que pertenecen al estado q_k en tiempo T. La proporción de accesos reconocidos, en adelante PAR, se toma como

$$PAR = \frac{T - n_0(T)}{T}$$

Antes de poner de manifiesto el potencial de esta técnica, vamos a continuar con el ejemplo motivador que ha introducido este capítulo. En la figura 5.7 (b) se representaba una posible secuencia de accesos a memoria durante el paso final de un *mergesort*. Al pasar esta secuencia de accesos a memoria por el procedimiento de reconocimiento de patrones, podemos comprobar que los resultados son muy buenos. En la figura 5.12 se representan de nuevo los accesos a memoria frente al tiempo, pero ahora se distinguen colores. El color negro representa observaciones clasificadas en el estado q_0, mientras que los demás colores representan a grupos informativos. Podemos comprobar como, en esta situación teórica y preparada, encontramos sin problemas el patrón de acceso a los vectores A y B, mientras que el ruido (incluso el cercano) se desestima. El $PAR = 0.8981$ obtenido es prometedor, y más teniendo en cuenta que la probabilidad de ruido es de 0.1 en este ejemplo.

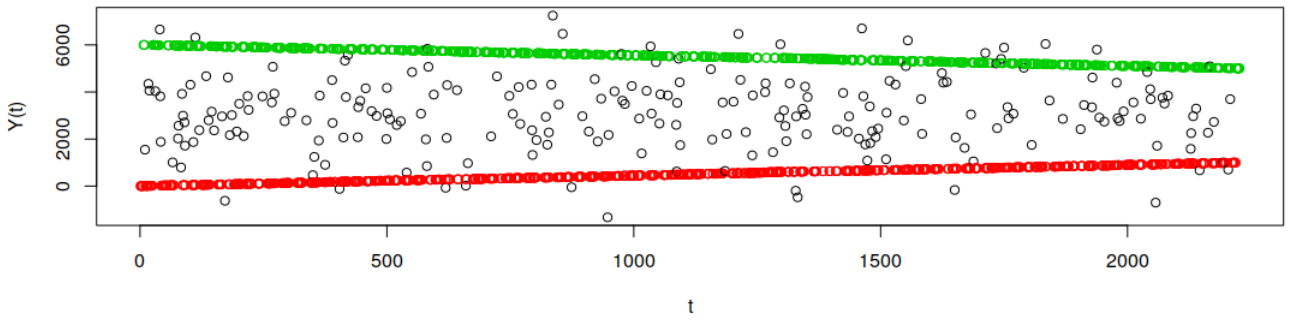


Figura 5.12: Secuencia de accesos a memoria de la combinación de un *mergesort* separados por grupo

5.4. Resultados del reconocimiento de patrones

En esta sección exploraremos el rendimiento del procedimiento de reconocimiento de patrones para las veinte aplicaciones que hemos venido considerando. En la tabla 5.3 se encuentran las aplicaciones, junto con el número de accesos a memoria, que se han examinado, así como el porcentaje de accesos reconocidos. Para aquellas aplicaciones pesadas se ha limitado el número de accesos explorados a 5.000.000, puesto que el tiempo de las simulaciones puede llevar varios días. Todos los accesos han sido examinados suponiendo una LLC con una configuración 16/1 (tamaño 16MB, asociatividad 1).

En general, la proporción de accesos reconocidos en aplicaciones reales es prometedora. Es muy habitual reconocer al menos el 80 % de las líneas de memoria, pero hay algunas aplicaciones que suponen excepciones. Aunque specrand tiene un PAR de tan sólo 49.34 %, no podemos considerarlo un fracaso, ya que sólo se examinan 3.508 accesos a memoria. La aplicación gcc también tiene un porcentaje menor que el resto, reconociendo aproximadamente el 62.5 % de las líneas correctamente. Le sigue Firefox (77.59 %) y Openoffice (78.21 %), dos aplicaciones cuya interactividad puede justificar una mayor dificultad en el reconocimiento de patrones.

Sin tener en cuenta specrand por su pequeño número de accesos, gcc es una de las aplicaciones de la que podemos extraer menos información. De hecho, es la que tiene un mayor número de estados (7458) y pocas observaciones asignadas a cada uno de ellos, lo que indica un comportamiento muy fragmentado. En la figura 5.13 podemos ver el histograma para el número de elementos (líneas de memoria) asignadas a cada grupo. La inmensa mayoría de los grupos tienen un tamaño muy pequeño.

Por otro lado, las aplicaciones que son más fáciles de reconocer son, fundamentalmente, lbm, libquantum y sjeng. En todas ellas, el porcentaje de accesos reconocidos es superior al 99 %. Además, todas tienen una característica común: los grupos que se reconocen tienden a ser muy grandes.

Aplicación	**Número de accesos explorados**	**Porcentaje accesos reconocidos**
Astar	249.164	88.4951 %
Bzip2	615.821	91.5649 %
Firefox	5.000.000	77.5941 %
g++	5.000.000	81.326 %
gcc	5.000.000	62.5091 %
gobmk	5.000.000	95.8011 %
h264ref	2.648.497	81.7642 %
hmmer	3.351.430	95.7508 %
lbm	5.000.000	99.9639 %
libquantum	5.000.000	99.9587 %
mcf	5.000.000	97.836 %
milc	5.000.000	97.3394 %
namd	1.961.439	86.8271 %
Omnetpp	5.000.000	87.9315 %
Openoffice	5.000.000	78.2115 %
povray	185.871	89.5261 %
sjeng	5.000.000	99.941 %
specrand	3.508	49.3444 %
sphinx3	5.000.000	94.9355 %
Xalan	5.000.000	90.0227 %

Tabla 5.3: Resultados del reconocimiento de patrones para las veinte aplicaciones

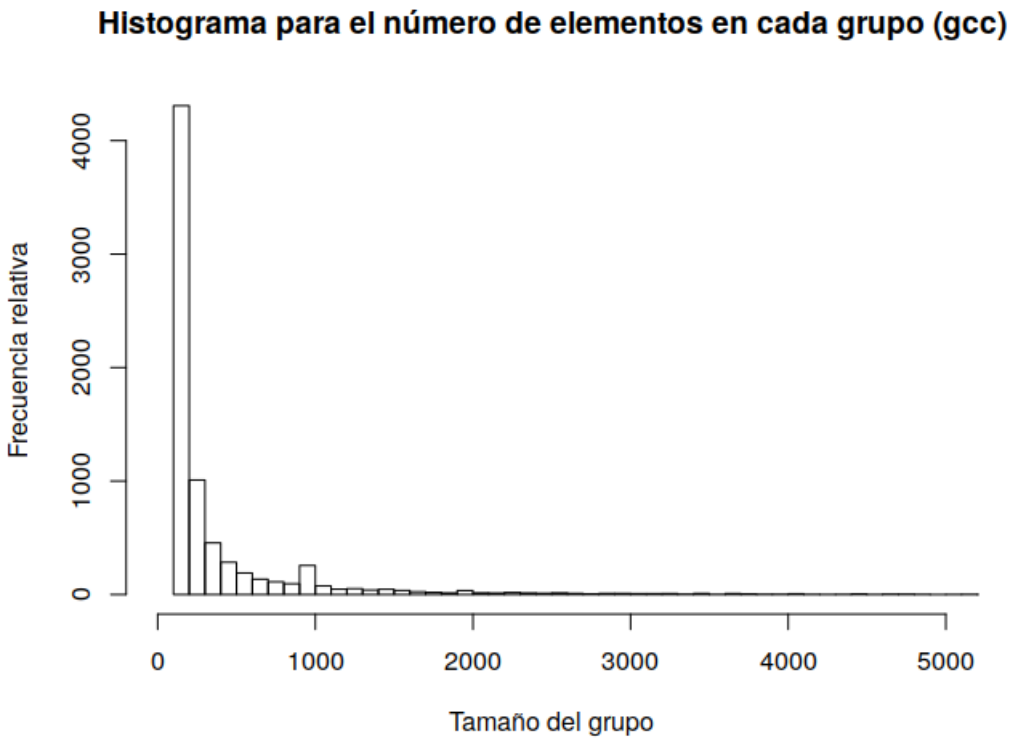


Figura 5.13: Histograma para el número de elementos que hay en cada grupo reconocido de la aplicación gcc

- En el caso de lbm, se encuentran 7 grupos tras examinar 5 millones de accesos. El grupo más grande (etiquetado con el número 6) está formado por 3.350.001 accesos a memoria. El siguiente grupo más grande lo componen 1.646.420 accesos. Además, como veremos más adelante, todos estos accesos tienen una tendencia muy marcada, o incluso son secuenciales.
- El comportamiento de libquantum es especialmente curioso, pues hay varios grupos, pero muchos de ellos tienen características idénticas. Por ejemplo, existen dos grupos, ambos con exactamente 524.289 accesos asignados. Si seguimos explorando, encontramos más curiosidades, como que existen 25 grupos, etiquetados desde el valor 51 hasta el 75, cada uno de ellos con 6.888 accesos.
- Por último, hablaremos de mcf. Esta aplicación también tiene grupos muy grandes; lo habitual es que ronden los 937.500 accesos. Además, hay otros grupos más pequeños, con menos de 1.000 accesos en cada uno.

Adicionalmente, veremos algunos gráficos llamativos que muestren algunos de los patrones reconocidos. Concretamente, se presentará:

- Un gráfico que muestre una lectura secuencial de líneas de memoria.
- Un gráfico que muestre una tendencia en las direcciones de memoria accedidas, pero sin ser lectura secuencial.

- Un gráfico que muestre accesos a una zona sin tendencia.
- Un gráfico de una zona reconocida que no tenga tendencia y sea impredecible.

Los gráficos se encuentran en la figura 5.14. Cada uno de ellos representa las direcciones de memoria de un determinado grupo $Y(t)$ frente al tiempo t. El número de línea al que se ha accedido comienza la escala en el 0 para mejorar la legibilidad. No obstante, que un gráfico indique que una línea ha sido accedida después de otra no quiere decir que este último acceso haya sido inmediato. Sólo se están representando las direcciones asociadas a un estado, pero puede haber accesos intermedios a otros grupos que no se estén representando. El rango indicado en cada gráfico es la diferencia entre la línea de memoria con la dirección más alta y la línea más baja.

El gráfico del primer caso (lectura secuencial) lo podemos encontrar con la aplicación lbm (figura 5.14 a). Tras un pequeño salto inicial, el resto de los accesos son secuenciales, pues se aprecia una recta a lo largo del tiempo. Esto quiere decir que tenemos una porción de memoria a la que se está accediendo secuencialmente, una línea detrás de otra. No obstante, es posible que haya accesos intercalados con otro grupo.

El gráfico del grupo con tendencia (figura 5.14 b) procede de astar. Es una zona de memoria que manifiesta una clara tendencia creciente, aunque no es un acceso secuencial. Existen ciertos "altibajos", como un pequeño ruido, pero manteniendo aproximadamente la misma tendencia. Tanto en este grupo como en el anterior, es perfectamente predecible qué es lo que sucedería a continuación, en caso de encontrar un acceso a memoria perteneciente a dicho grupo.

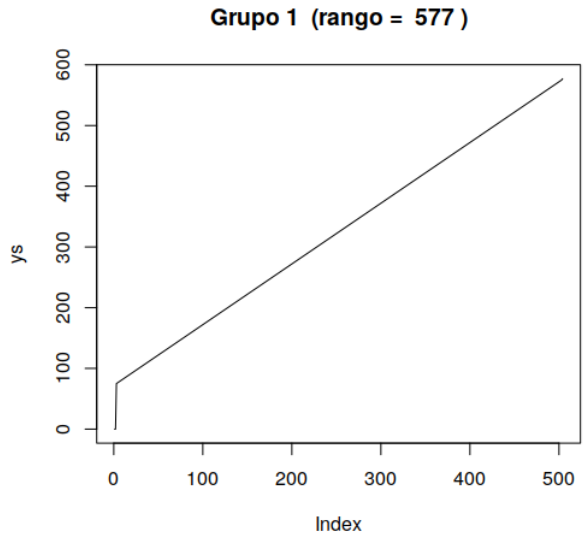
El ejemplo de un grupo de memoria sin tendencia viene de la mano de Firefox (figura 5.14 c). Los accesos que llegan a memoria principal en este grupo se encuentran en torno a una misma línea y parece que siguen un patrón regular. En la mayoría de casos, parecen accesos repetidos, fruto de los desalojos tempranos de la caché LLC, que sólo tiene asociatividad 1.

El último ejemplo corresponde a otro grupo de Firefox (figura 5.14 d). Este grupo es característico por ser uno de los más difíciles para predecir su comportamiento futuro. A lo largo de las líneas de memoria accedidas en esta zona, existen varios cambios de comportamiento y de tendencia. Por ejemplo, los 200 primeros accesos parecen tener un patrón diferente a los 300 que los siguen. A partir del acceso 500, vuelve a aparecer un comportamiento extraño.

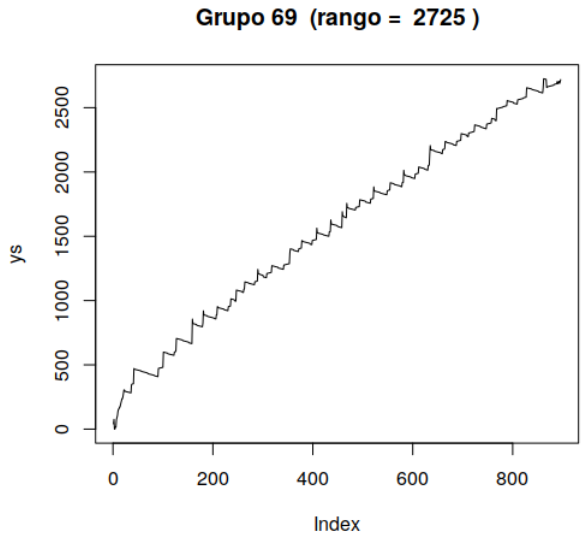
En líneas generales, conseguida la separación de comportamiento en los accesos a memoria, podemos comenzar a idear un sencillo mecanismo de prebúsqueda, tal y como se describe en la sección que sigue.

5.5. Prebúsqueda

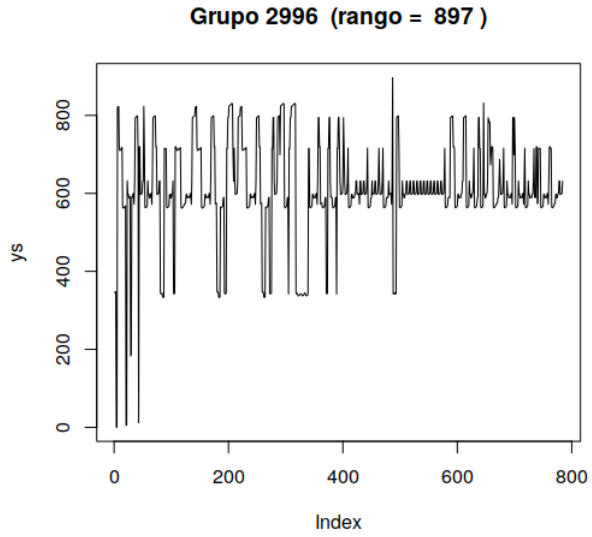
Si el comportamiento de los procesos se desglosa y se separa en función de los accesos a memoria, obtenemos varios perfiles de grupos, tal y como acabamos de ver. En la mayoría de casos, se están produciendo accesos intercalados a varios estados, junto con ruido que no puede ser clasificado a



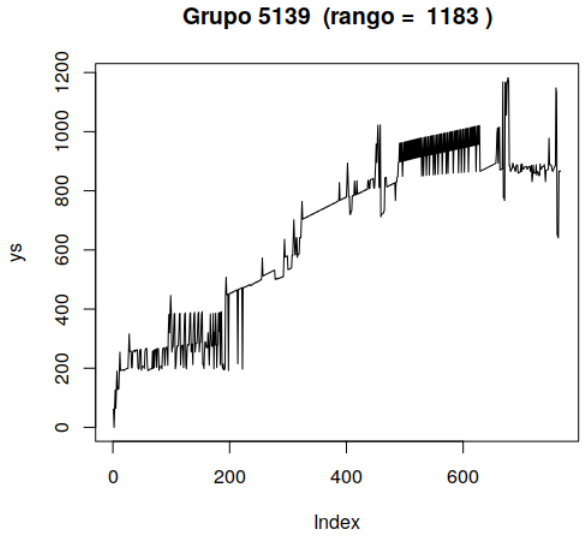
(a) Grupo secuencial (lbm)



(b) Grupo con tendencia (astar)



(c) Grupo sin tendencia (Firefox)



(d) Grupo impredecible (Firefox)

Figura 5.14: Ejemplos de grupos reconocidos por el procedimiento oculto de Markov

ninguno. Por este motivo es necesario separar los accesos en zonas antes de proponer el mecanismo de prebúsqueda, que marcará el fin de esta memoria y este trabajo.

Consideremos un grupo cualquiera que ha sido reconocido y que está compuesto por varias observaciones. Por construcción, la secuencia de accesos asociada debe tener un comportamiento, tendencia, y formas parecidos. Examinando uno y sólo un grupo simultáneamente, y suponiendo que dicho grupo va a continuar en un futuro, podemos elaborar predicciones sobre los accesos posteriores.

El abanico de opciones que se abre de cara a las predicciones futuras de un grupo de accesos a memoria es enorme. Entre las técnicas que podríamos usar, destacan las redes neuronales recurrentes, modelos ARIMA, suavizado exponencial, etc. Todos ellos tienen la característica fundamental de incorporar el tiempo en la predicción. Nótese que este tiempo debe ser entendido como si únicamente existiera esa secuencia de accesos; aunque haya habido accesos anteriores, la primera observación en un nuevo grupo se encuentra en tiempo 1, y de igual forma, aunque existan accesos intercalados a otros grupos, la segunda observación clasificada en dicho grupo se encontrará en tiempo 2.

Si recordamos el propósito de este TFG, buscamos utilizar una memoria SDRAM como una caché sobre una RRAM mucho más grande. En cuánto a tamaños, podría pensarse en una caché SDRAM de 16 o 32GB, sobre una RRAM mucho mayor. En la figura 5.15 se muestra un diagrama de la jerarquía de memoria propuesta para acoplar una SDRAM a la RRAM subyacente. El controlador de la SDRAM es una unidad que se encarga de los cálculos necesarios para el reconocimiento de grupos y de la prebúsqueda. Nótese que, dada la naturaleza del procedimiento oculto de Markov, se soportan varias tareas simultáneamente trabajando en zonas diferentes de memoria.

Según el diagrama de la figura 5.15, las peticiones a memoria SDRAM serían capturadas por el controlador de la SDRAM, que dispone del mecanismo de prebúsqueda. Mientras se satisface la operación recibida del anterior nivel en la jerarquía de memoria, se realizan las operaciones necesarias para encuadrar el nuevo acceso en algún estado o grupo. Una vez ubicado, se determina si es necesario traer alguna línea con el mecanismo de prebúsqueda (que se expondrá a continuación), y se traslada la orden a la RRAM situada en el nivel inferior.

Al disponer de una SDRAM más grande, podemos permitirnos un cierto grado de desperdicio en la caché: podemos traer ciertas líneas aunque no tengamos una certeza grande sobre su uso futuro. Esto no implica traer toda la aplicación, o todas las zonas colindantes a una línea cada vez que su uso, pero nos permite cierta libertad a la hora de diseñar el mecanismo de prebúsqueda.

De entre todas las posibilidades, vamos a optar por un enfoque sencillo y utilizar la regresión lineal sobre el tiempo. Sea $\boldsymbol{Y}_k = \{Y_{k,1}, \ldots, Y_{k,n_k}\}$ el vector aleatorio que contiene la secuencia de observaciones asignadas al estado q_k. Se asume un modelo de regresión lineal (similar al considerado anteriormente) como sigue.

$$Y_{k,i} = \beta_0 + \beta_1 \cdot i + \epsilon_{k,i}, \qquad i = 1, \ldots, n_k \tag{5.28}$$

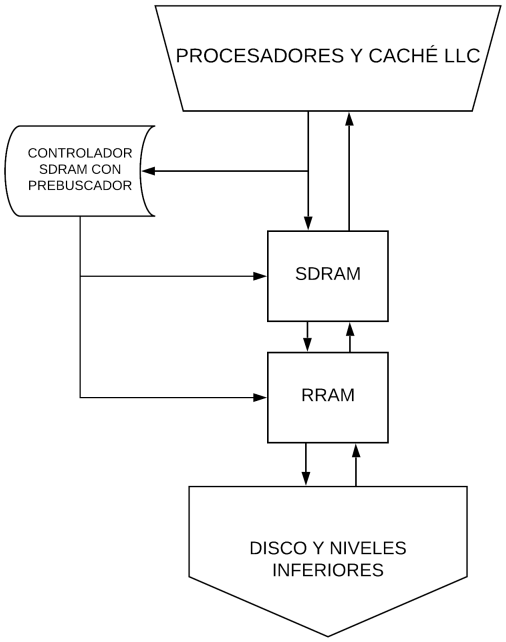


Figura 5.15: Diagrama de la arquitectura SDRAM-RRAM propuesta.

con $\epsilon_{k,i} \sim \mathcal{N}(0, \sigma_k^2)$.

Una vez estimados los coeficientes $\widehat{\beta_0}$ y $\widehat{\beta_1}$ por el método de mínimos cuadrados, podemos elaborar predicciones. En concreto, nos interesa la predicción para el siguiente instante en que un acceso a memoria sea asignado a q_k. El valor buscado es:

$$\widehat{Y_{k,n_k+1}} = \hat{\beta}_0 + \hat{\beta}_1 \cdot (n_k + 1) \tag{5.29}$$

Sin embargo, de esta estimación puntual surgen varios problemas:

- El siguiente valor para Y_{k,n_k+1} será un número entero, correspondiente a la línea de memoria accedida, pero esta restricción no existe en la ecuación 5.29.
- La predicción puntual, una vez redondeada, tiene una probabilidad muy baja de acertar debido al ruido que existe.

En lugar de utilizar una estimación puntual, se propone utilizar un intervalo de predicción para el modelo lineal de la ecuación 5.28 [23]. Es bien sabido que, en el caso general, si X es la matriz de diseño del modelo, un intervalo de predicción para $Y_{k,i}$ viene dado por

$$\widehat{Y_{k,i}} \pm t_{n_k-2,1-\frac{\alpha}{2}} \sqrt{\hat{\sigma_k}^2 \left(1 + x_i^t \left(X^t X\right)^{-1} x_i\right)} \tag{5.30}$$

donde $\widehat{\sigma_k} = \sqrt{\frac{1}{n_k-2} \sum_{i=1}^{n_k} \left(Y_{k,i} - \widehat{\mu_{k,i}}\right)^2}$ es una estimación insesgada para σ_k, $x_i = \begin{bmatrix} 1 \\ i \end{bmatrix}$, y $t_{n_k-2,1-\frac{\alpha}{2}}$ es el cuantil $1 - \frac{\alpha}{2}$ de una distribución t-student con $n_k - 2$ grados de libertad. El intervalo de

predicción garantizaría una confianza de aproximadamente un $100(1-\alpha)\,\%$ de que el verdadero valor se encuentra dentro del intervalo.

La matriz de diseño del modelo propuesto es muy sencilla, pues tan sólo es necesario estimar dos parámetros:

$$X = \begin{bmatrix} 1 & 1 \\ 1 & 2 \\ \vdots & \vdots \\ 1 & n_k \end{bmatrix} \tag{5.31}$$

Los cálculos necesarios para obtener una expresión particular de la ecuación 5.30 pasan por obtener $(X^tX)^{-1}$. En este caso particular es muy sencillo:

$$X^tX = \begin{bmatrix} \sum_{i=1}^{n_k} 1 & \sum_{i=1}^{n_k} i \\ \sum_{i=1}^{n_k} i & \sum_{i=1}^{n_k} i^2 \end{bmatrix} = \begin{bmatrix} n_k & \frac{n_k(n_k+1)}{2} \\ \frac{n_k(n_k+1)}{2} & \frac{n_k(n_k+1)(2n_k+1)}{6} \end{bmatrix} \tag{5.32}$$

La inversa de esta matriz existe y es sencilla de hallar, siempre que $n_k > 1$:

$$(X^tX)^{-1} = \frac{1}{n_k(n_k-1)} \begin{bmatrix} 2(2n_k+1) & -6 \\ -6 & \frac{12}{n_k+1} \end{bmatrix} \tag{5.33}$$

En concreto nos interesa un intervalo de predicción para la siguiente línea de memoria, Y_{k,n_k+1}. En este caso $x_{n_k+1} = \begin{bmatrix} 1 \\ n_k+1 \end{bmatrix}$, por lo que continuamos la particularización de la expresión 5.30 de la forma que sigue.

$$x_{n_k+1}^t (X^tX)^{-1} x_{n_k+1} = \frac{2(2n_k+1)}{n_k(n_k-1)} \tag{5.34}$$

El intervalo de predicción buscado resulta, finalmente,

$$\widehat{Y_{k,n_k+1}} \pm t_{n_k-2,1-\frac{\alpha}{2}} \sqrt{\hat{\sigma_k}^2 \left(1 + \frac{2(2n_k+1)}{n_k(n_k-1)}\right)} \tag{5.35}$$

con una confianza del $100(1-\alpha)\,\%$ de que el verdadero valor para Y_{k,n_k+1} esté dentro. Sea $\widehat{Y^l_{k,n_k+1}}$ el extremo inferior del intervalo, y $\widehat{Y^u_{k,n_k+1}}$ el extremo superior.

$$\begin{aligned} \widehat{Y^l_{k,n_k+1}} &= \widehat{Y_{k,n_k+1}} - t_{n_k-2,1-\frac{\alpha}{2}} \sqrt{\hat{\sigma_k}^2 \left(1 + \tfrac{2(2n_k+1)}{n_k(n_k-1)}\right)} \\ \widehat{Y^u_{k,n_k+1}} &= \widehat{Y_{k,n_k+1}} + t_{n_k-2,1-\frac{\alpha}{2}} \sqrt{\hat{\sigma_k}^2 \left(1 + \tfrac{2(2n_k+1)}{n_k(n_k-1)}\right)} \end{aligned} \tag{5.36}$$

Sean $\lfloor x \rfloor$ y $\lceil x \rceil$ las funciones de redondeo por abajo y por arriba, respectivamente. Consideremos la secuencia de valores $\lfloor \widehat{Y^l_{k,n_k+1}} \rfloor, \ldots, \lceil \widehat{Y^u_{k,n_k+1}} \rceil$, que representa un grupo de posibilidades para la dirección de la siguiente línea accedida, con una confianza aproximadamente del $100(1-\alpha)$. El mecanismo de prebúsqueda que se plantea **trata todas estas líneas con la misma**

probabilidad de ser elegidas. Por tanto, y aprovechando que la caché SDRAM dispondrá de una capacidad razonable, se trae a la caché SDRAM todo el bloque colindante de direcciones.

No obstante, existen ciertas restricciones. Recordemos que en la sección 5.3 hemos descrito un valor umbral n_g que se exige a todos los estados reconocidos para ser considerado como un grupo válido. Anteriormente habíamos establecido $n_g = 100$. De esta forma, al pseudocódigo de la figura 5.11 le añadimos los siguientes pasos:

1. Si q_k es el estado al que se ha asignado la observación $Y(t)$, y se cumple que $n_k(t) \geq n_g$, se realiza una prebúsqueda y se prosigue en el paso siguiente. En caso contrario, no se continúa.

2. Se escogen las 100 últimas observaciones que han sido asignadas a q_k, $Y_{k,n_k(t)-99}, \ldots, Y_{k,n_k(t)}$.

3. Se plantea el modelo de la ecuación 5.28, se estiman los coeficientes β_0 y β_1, y se calcula la estimación puntual $\widehat{Y_{k,n_k+1}} = \hat{\beta}_0 + \hat{\beta}_1 \cdot (n_k + 1)$.

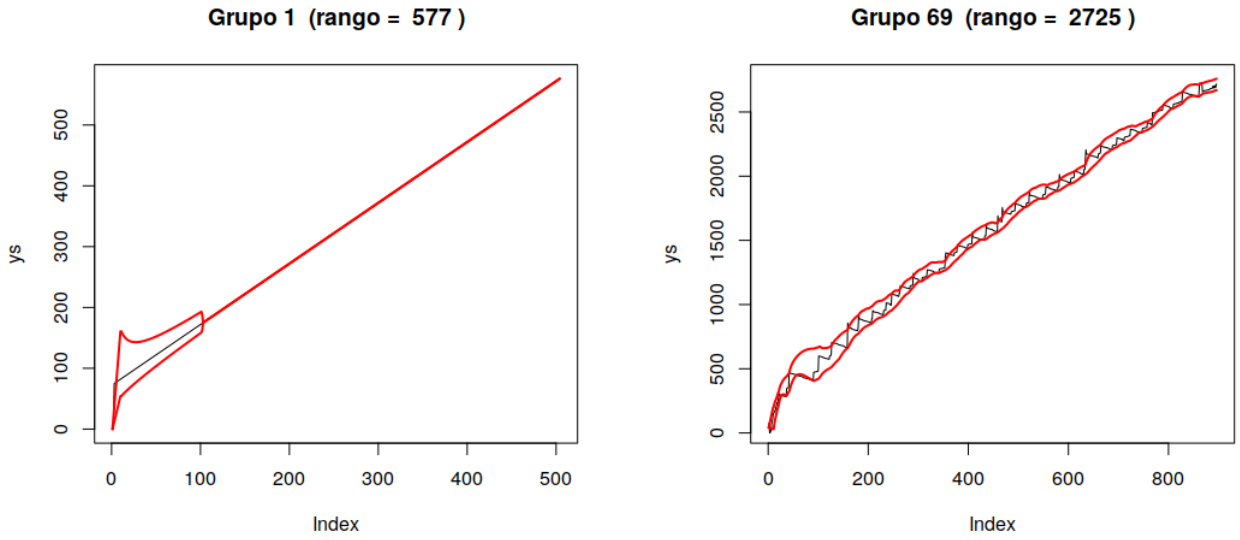
4. Obtenemos la secuencia $\lfloor \widehat{Y^l_{k,n_k+1}} \rfloor, \ldots, \lceil \widehat{Y^u_{k,n_k+1}} \rceil$ utilizando la expresión 5.35, y con una confianza del 90 % ($\alpha = 0.1$).

5. El prebuscador efectúa la orden de traer a la SDRAM todas las líneas de la secuencia anterior.

Para ilustrar de forma gráfica cuáles son las posibilidades de este procedimiento, se presentan de nuevo los grupos mostrados en la sección anterior (figura 5.14). Esta vez, se muestra adicionalmente una banda roja que corresponde al intervalo de predicción, calculado con el procedimiento descrito anteriormente, y utilizando los últimos 100 valores del grupo. Nótese que según los pasos especificados no se realiza una prebúsqueda hasta que el grupo está compuesto por lo menos por 100 accesos, aunque en los gráficos se muestra el intervalo desde el principio. Para estas primeras observaciones, no se realizaría la prebúsqueda.

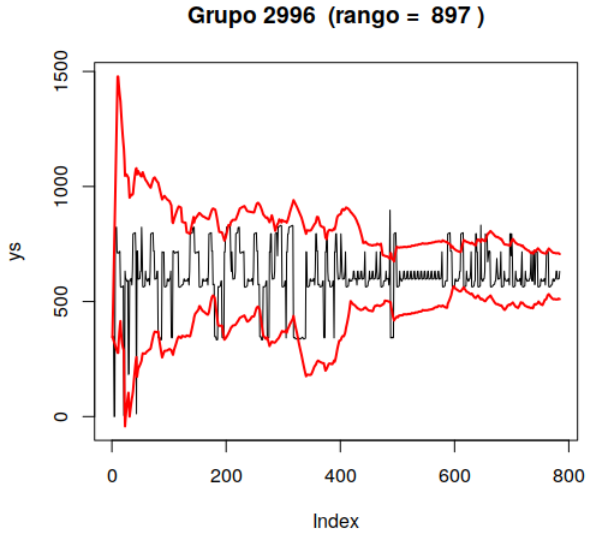
En el caso del grupo secuencial (figura 5.14 a), el intervalo de predicción comienza siendo elevado debido al salto inicial. A partir de la observación número 101, la primera no se tiene en cuenta para realizar la prebúsqueda, por lo que la amplitud del intervalo decrece. Además, al tratarse de un grupo perfectamente secuencial, el intervalo de predicción contiene exactamente un único punto. Es por esto que en grupos secuenciales, el mecanismo de prebúsqueda tiene un excelente potencial. Por otro lado, es lógico, pues son los grupos más predecibles que nos podremos encontrar.

Para el grupo con tendencia creciente (figura 5.14 b) podemos comprobar cómo el intervalo de predicción permite ajustar una pequeña curva que va adaptándose a la forma de los datos. Esto se debe a la utilización de los últimos 100 valores para el cálculo, lo que anula la influencia del principio, y permite ir adaptándose a una velocidad aceptable. Al utilizarse $\alpha = 0.1$, se estima que aproximadamente el 10 % de los accesos no son capturados correctamente por el método de prebúsqueda. Variaciones en el valor de α provocan directamente un cambio en la amplitud del intervalo de predicción, que repercute a su vez en un aumento o disminución del número de líneas traídas a la SDRAM.

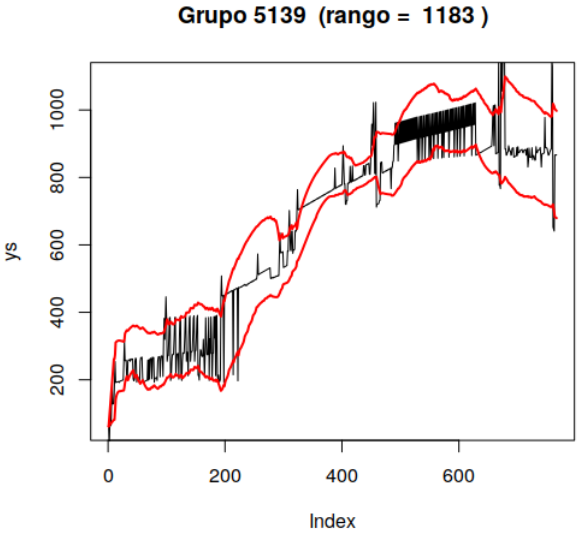


(a) Grupo secuencial (lbm)

(b) Grupo con tendencia (astar)



(c) Grupo sin tendencia (Firefox)



(d) Grupo impredecible (Firefox)

Figura 5.16: Ejemplos de grupos reconocidos por el procedimiento oculto de Markov junto con intervalos de predicción

En el caso del grupo sin tendencia (figura 5.14 c) las bandas rojas sugieren un estrechamiento progresivo en las direcciones de memoria accedidas, pero siempre en torno a un rango de valores específico. Para el último grupo (5.14 d), los cambios en el comportamiento de los accesos a memoria ocasionan grandes cambios en la estructura de los intervalos de predicción. No obstante, aún así hay un gran número de accesos que son capturados dentro de las bandas.

Puede parecer que el mecanismo de prebúsqueda que se propone tiene un alto desperdicio de la caché, pues por cada línea accedida se traen numerosas líneas colindantes. Sin embargo, aquí entra en juego el intervalo de predicción inmediato anterior, por lo que en la mayoría de ocasiones el número de líneas nuevas traídas por cada acceso será bastante reducido. Ilustremos esto con los cuatro ejemplos de la figura 5.14.

- En el caso del grupo secuencial, tras cada acceso examinado, el procedimiento de prebúsqueda determinará que la siguiente línea es la única candidata para ser prebuscada, por lo que se trae exactamente un elemento extra.
- En el caso del grupo con tendencia, el intervalo de predicción parece tener una amplitud de unas 100 líneas intermedias. Supongamos que en un cierto instante t deben traerse a mayores unas 100 líneas extra. Muchas de ellas habrán sido traídas en el instante anterior, ya que la intersección de ambos intervalos de predicción contendrá un número muy grande de líneas. Por tanto, aunque el mecanismo de prebúsqueda determine 100 líneas extra, muchas de ellas ya se encontrarán en la SDRAM y para las que sólo es necesario actualizar ciertas estadísticas (como el instante en el que fue traída).
- Algo similar ocurre en el grupo sin tendencia y en el grupo impredecible. Con respecto al primero, el intervalo parece estrecharse con el tiempo, pero siempre en torno a un valor, por lo que casi todas las líneas ya han sido traídas antes. En el grupo impredecible será necesario traer alguna más.

Nótese además que los cálculos necesarios en este mecanismo de prebúsqueda no son triviales. Todo ello consume tiempo y recursos, y mientras tanto la SDRAM puede seguir recibiendo peticiones de los niveles superiores. No es tanto problema la complejidad de los cálculos a este nivel, puesto que en los niveles superiores ya hay una jerarquía de caché a la que el procesador podrá acceder, y por ende pasará más tiempo entre dos accesos consecutivos a la SDRAM. Según la jerarquía propuesta, el controlador de la SDRAM va recibiendo peticiones de acceso y, mientras la SDRAM sirve la petición a los niveles superiores, el controlador ubica el acceso en un grupo y solicita a la RRAM traer las líneas que sean necesarias. En esta arquitectura será necesario añadir alguna estructura de datos para poder almacenar una cola de peticiones tanto a la SDRAM como a la RRAM inferior.

Tan sólo resta una simulación de la arquitectura propuesta para conocer las posibilidades que ofrece esta técnica en las aplicaciones reales. En la siguiente sección se propone un marco de simulación y los resultados producidos, con lo que concluiremos este capítulo y también esta memoria.

5.6. Resultados de la prebúsqueda

Para finalizar esta sección, proponemos un marco de simulación que permita conocer cuán bueno es el mecanismo de prebúsqueda sobre las aplicaciones reales consideradas. Para cada una de las 20 aplicaciones, simularemos que las peticiones de lectura y escritura llegan a una caché SDRAM. Si ésta puede satisfacer las peticiones al nivel superior lo hará inmediatamente, pero en caso negativo solicitará a la RRAM inferior el dato correspondiente.

Se proponen tres condiciones para simular la caché:

- En la primera simulación consideraremos una caché SDRAM ideal, sin límite de tamaño, pero con un umbral de olvido de 25.000 accesos. Esto quiere decir que, aunque no existe un número máximo para las líneas que pueden almacenarse simultáneamente en la SDRAM, todas aquellas que lleven más de 25.000 peticiones intermedias sin ser utilizadas son desechadas. Esta caché ideal se supone con un sólo conjunto.

- Una caché SDRAM con 16MB de capacidad y asociatividad por conjuntos de nivel 4. No se considera umbral de olvido.

- Una caché SDRAM con 64MB de capacidad y asociatividad por conjuntos de nivel 8. Tampoco hay umbral de olvido.

La razón para utilizar cachés SDRAM de pequeño tamaño, a pesar de que los objetivos iniciales se referían a cachés de gran capacidad, es el número de peticiones consideradas en la simulación. Habitualmente vamos a considerar unos 5 millones de accesos, por lo que si realizamos la prueba con una SDRAM muy grande, todo el conjunto de trabajo se podrá localizar en la caché, no habrá reemplazos, y la simulación no contendrá ningún tipo de información válida. El tamaño de la caché simulada se ha elegido teniendo en cuenta el número de accesos recibidos.

Para cada uno de estos tres tipos de caché, se realizan dos simulaciones:

- En una de ellas, no se considera el mecanismo de prebúsqueda. Una línea sólo es traída a la caché SDRAM cuando es requerida, y se almacena por si en un futuro vuelve a ser necesitada. Sólamente se basa en la **localidad temporal**.

- En la otra, además de mantener una línea en la caché cuando se utiliza, también se considera el mecanismo de prebúsqueda descrito. Cuando una línea se encuadra dentro de un grupo y hay un número suficiente de ellas, la cache SDRAM elabora una predicción basada en un modelo lineal y solicita a la RRAM el contenido de todas las líneas resultantes. Por tanto, en esta simulación, estamos aprovechando la **localidad temporal y espacial**.

Adicionalmente, en todos los casos, consideraremos las siguientes características:

- Los bits menos significativos de la línea, despreciando el *offset*, sirven para elegir el número de conjunto al que pertenece dicha línea en la caché SDRAM, según se explicó en el Capítulo 2. En la figura 2.4 (página 34) pueden encontrarse más detalles sobre esta división. Esto garantiza que líneas consecutivas elegidas en el procedimiento de prebúsqueda vayan a conjuntos diferentes, evitando así reemplazos innecesarios. Todas las líneas elegidas en el mecanismo de prebúsqueda acabarán en la caché, pues es imposible que se solapen los números de conjuntos. Nótese que esto convierte a la asociatividad en el principal límite de la caché: se espera un correcto funcionamiento siempre que la asociatividad sea de al menos el número de grupos a los que se accede simultáneamente. Si una aplicación accede intercaladamente a 4 zonas de memoria, pero la asociatividad sólo es de nivel 2, pueden producirse reemplazos entre la prebúsqueda de estas 4 zonas y disminuir notablemente el rendimiento.

- Se utiliza una política de reemplazo LRU en cada conjunto. Cuando una línea deba ser traída a la caché SDRAM, primero se comprueba si ya se encontraba allí. En caso afirmativo, se actualiza el instante de uso de dicha línea, pero si no es así debe ser traída de la RRAM. Si no hay más espacio en el conjunto, la víctima que será desalojada es aquella que lleva más tiempo sin usarse. Véase el Capítulo 2 para más detalles sobre las políticas de reemplazo.

- Se considera un acierto de caché siempre que una petición pueda ser satisfecha a nivel de SDRAM, sin tener que recurrir al nivel inferior para conseguir la información. El sistema de memoria llevará un registro sobre las peticiones de memoria en todo momento, registrando el uso de cada línea cuando sea necesario. Una línea se puede encontrar en la caché por dos motivos: fue traída porque se necesitaba, o porque fue prebuscada. En caso de que se necesite una línea pero no se encuentre en la SDRAM se considera un fallo.

El controlador de la SDRAM llevará todas las estructuras de datos necesarias para la prebúsqueda: los grupos reconocidos hasta el momento por el modelo oculto de Markov, los estimadores de la media y la desviación estándar, el historial de la caché y los modelos de predicción.

Todas estas simulaciones se realizan en R [25], por la facilidad de este lenguaje para el ajuste de los modelos necesarios, así como la versatilidad y flexibilidad en cuánto al tipado de las estructuras de datos y variables (tiene un tipado muy débil). R es un entorno y lenguaje de programación con un enfoque al análisis estadístico. Apareció en 1993 (*Ross Ihaka y Robert Gentleman*) y adopta un enfoque de alto nivel. Se trata de un software libre diseñado para la comunidad estadística con enfoques de minería de datos, investigación biomédica, bioinformática y matemáticas financieras. Permite una gran flexibilidad, como cualquier usuario de R conoce, pero a costa de una menor eficiencia en la ejecución donde, la interpretación en lugar de la compilación es, en gran medida, su causa.

Para la simulación de la caché ideal se utilizan, como máximo, 500.000 accesos a memoria. Esta simulación es rápida y permite hacernos una idea sobre el funcionamiento del mecanismo

de prebúsqueda, y compararlo con el que no lo tiene. Para las simulaciones de las cachés 16/4 y 64/8 se tendrán en cuenta un máximo de 5.000.000 accesos de cada aplicación. Estas simulaciones requieren de un tiempo considerable dado el rendimiento de R, y a fecha de entrega de este TFG, existe una aplicación que aún se encuentra en examen.

Para cada simulación individual, dada la configuración de la caché SDRAM, queremos conocer las siguientes estadísticas:

1. El **número de accesos** a memoria que se han examinado.

2. El **porcentaje de accesos reconocidos**, que deberá ser el mismo independientemente del modelo de caché. El único motivo por el que pueden diferir estos porcentajes es por una modificación de los parámetros (umbral de inicio, probabilidad mínima), o por una elección aleatoria en caso de empate de estados más probables.

3. El **número de entradas totales** que han pasado por la caché, teniendo en cuenta incluso las que se han desechado por la política de reemplazo. En definitiva, el número de veces que se ha tenido que crear una entrada.

4. El **número de entradas actuales** que hay en la caché en el momento de finalización de la simulación.

5. El **número de reemplazos** que han ocurrido en la caché. En la ideal, se considera un reemplazo eliminar la línea transcurrido el umbral de olvido.

6. El **porcentaje prebuscado**, que indica cuátas de las entradas de la caché SDRAM han sido traídas por orden del mecanismo de prebúsqueda y no por el requerimiento inmediato de una línea de memoria. En el caso de que no exista un mecanismo de prebúsqueda, se tiene un 0 %.

7. El **porcentaje de desperdicio**, que indica cuántas de las entradas que se han creado en la caché no han sido finalmente utilizadas. Si no hay mecanismo de prebúsqueda, se tendrá un 0 %.

8. El **porcentaje de éxitos** de caché.

9. El **porcentaje de aciertos por prebúsqueda**, que indica, de todos los aciertos existentes, cuántos se deben al mecanismo de prebúsqueda en lugar de a un acceso repetido basado en la localidad temporal (0 % si no hay prebúsqueda).

10. El **porcentaje de prebúsquedas útiles**, que indica cuántas de las entradas que se han creado en la caché como orden del mecanismo de prebúsqueda han sido finalmente utilizadas. No debe confundirse con el porcentaje de desperdicio, que incluye también las líneas traídas por efecto de la localidad temporal. Si no hay mecanismo de prebúsqueda, se tendrá un 0 %.

5.6.1. Caché SDRAM ideal con umbral de olvido

Aquí examinaremos los resultados de una caché SDRAM ideal con un umbral de olvido de 25.000 accesos, para cada una de las 20 aplicaciones, limitando el número de accesos examinados a 500.000. El objetivo es tener una primera comparación sobre las diferencias entre utilizar y no utilizar el mecanismo de prebúsqueda. Puesto que la prebúsqueda trata tanto la localidad temporal como la espacial, es de esperar que en todos los casos funcione mejor una caché con este mecanismo. Lo realmente interesante es determinar si dicha mejora es significativa teniendo en cuenta la complejidad de la técnica.

Para la presentación de los resultados incluiremos una tabla con los resultados numéricos, y un gráfico. La figura 5.17 contiene información sobre el porcentaje de éxito de la caché ideal con y sin el mecanismo de prebúsqueda (representado con un círculo y una cruz, respectivamente). Además, como sabemos, incluir esta prebúsqueda ocasiona que cierta parte de la caché se desperdicie, pues habrá ciertas líneas que se traigan pero nunca sean requeridas. El porcentaje de desperdicio de la caché se representa con un triángulo. Una medida de eficiencia de la técnica podría ser derivada de la comparación de las diferencias en los porcentajes de éxito con la tasa de desperdicio.

Por otro lado, en la tabla 5.4 podemos encontrar las 10 características que hemos descrito arriba. La numeración de la columna de información corresponde al mismo orden en que estas estadísticas fueron especificadas. Como es sabido, el porcentaje de accesos reconocidos será el mismo haya o no prebúsqueda, pues no se ve influido por ello. Para que la comparación se ajuste a la realidad lo máximo posible, se ha examinado el mismo número de accesos para cada posibilidad dentro de una misma aplicación. Se espera que el número de entradas totales de la caché, el número de entradas actuales, y el número de reemplazos aumenten al utilizar la prebúsqueda. Además, el porcentaje prebuscado, el porcentaje de desperdicio, el porcentaje de aciertos por prebúsqueda, y el porcentaje de prebúsquedas útiles, será siempre de 0 % si ésta no se ha realizado. Especialmente interesante es la diferencia entre el porcentaje de éxitos de caché, tal y como aparece en la figura 5.17.

Las aplicaciones que parecen beneficiarse mucho más del mecanismo de prebúsqueda son bzip2, gobmk, lbm, libquantum, mcf, namd y sjend. En todos estos casos, la tasa de éxito aumenta notablemente y a costa de una baja tasa de desperdicio. Otras aplicaciones, como h264ref, milc, povray y Xalan, tienen una ganancia mucho menor, y a costa de un cierto desperdicio en la caché. Esto es porque estas aplicaciones tienen un comportamiento más difícil de reconocer, los intervalos de predicción son más amplios, y en consecuencia hay líneas prebuscadas que no se usan. La aplicación que tiene un mayor desperdicio de la caché es astar, llegando casi a un 50 %.

Por otro lado, aunque para algunas aplicaciones la prebúsqueda no proporcione una mejora tan grande, como milc, podemos comprobar en la tabla que el porcentaje de aciertos debido a la prebúsqueda no es despreciable. La prebúsqueda aumenta la tasa de éxito de un 74.13 % a un 94.47 % a costa de desperdiciar un 44.89 % de la caché. Sin embargo, sólo el 52.5 % de las prebúsquedas se utilizan. Esta aplicación, junto con astar, es la que tiene un menor índice de prebúsquedas útiles.

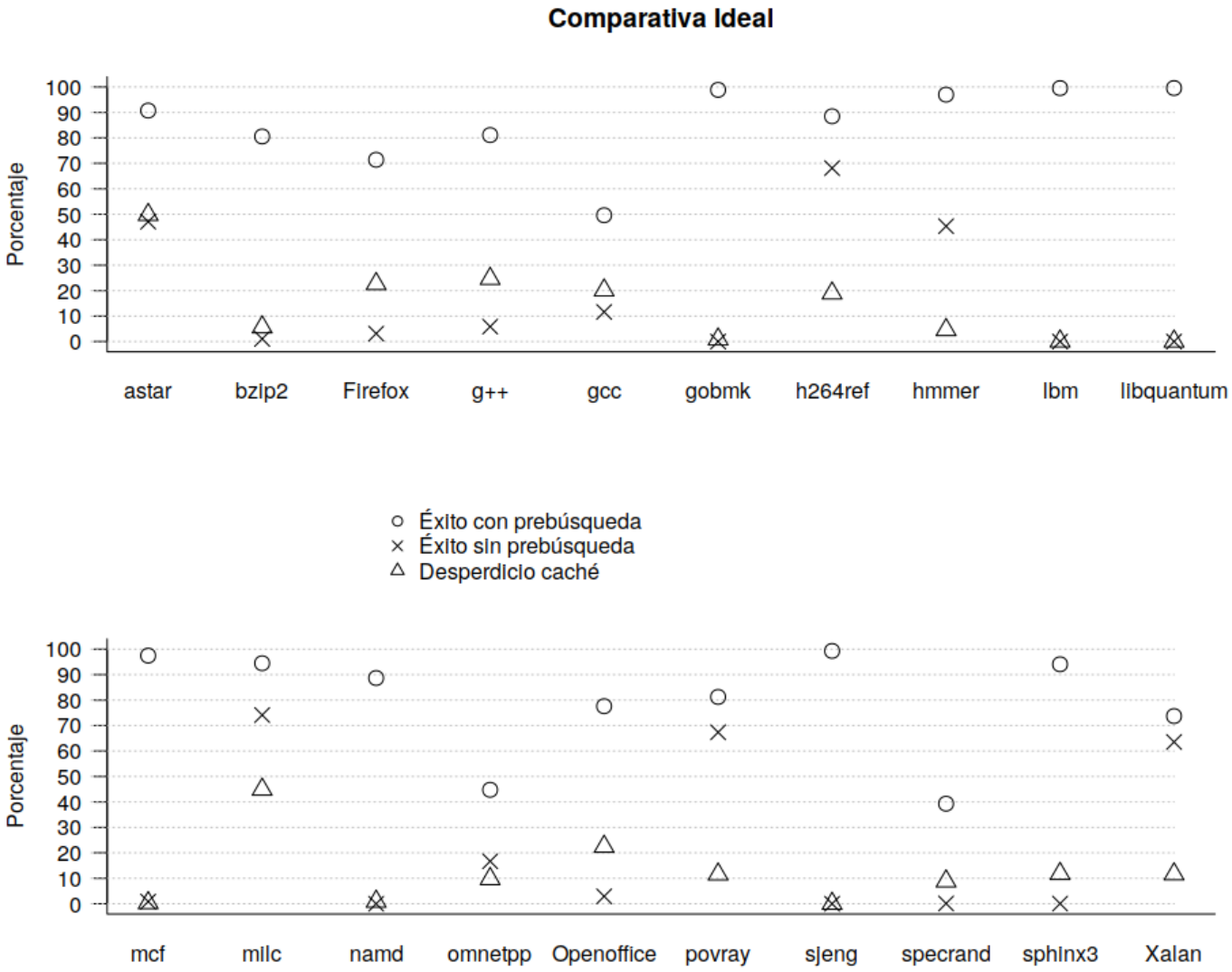


Figura 5.17: Gráfico de comparación de la simulación con y sin prebúsqueda, suponiendo una caché ideal con umbral de olvido

	astar		bzip2		Firefox		g++		gcc	
Info	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.
1	249.164	249.164	5e+05	5e+05	5e+05	5e+05	5e+05	5e+05	5e+05	5e+05
2	88.4983 %	88.4983 %	88.7768 %	88.7768 %	79.4088 %	79.4088 %	82.778 %	82.778 %	56.6724 %	56.6724 %
3	131.824	262.264	494.720	523.765	484.347	625.818	470.657	622.627	441.807	529.719
4	209	1.863	24.938	37.331	23.781	64.332	22.758	60.134	23.479	34.614
5	131.615	260.401	469.782	486.434	460.566	561.486	447.899	562.493	418.328	495.105
6	0 %	91.1825 %	0 %	81.4367 %	0 %	77.1475 %	0 %	84.8251 %	0 %	52.4374 %
7	0 %	49.8143 %	0 %	5.6606 %	0 %	22.6512 %	0 %	24.6693 %	0 %	20.1467 %
8	47.0935 %	90.719 %	1.056 %	80.5544 %	3.1306 %	71.397 %	5.8686 %	81.1034 %	11.6386 %	49.6104 %
9	0 %	74.8163 %	0 %	99.3512 %	0 %	97.3058 %	0 %	96.9989 %	0 %	83.7187 %
10	0 %	45.3686 %	0 %	93.0491 %	0 %	70.6392 %	0 %	70.9174 %	0 %	61.5795 %
	gobmk		**h264ref**		**hmmer**		**lbm**		**libquantum**	
Info	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.
1	5e+05	5e+05	5e+05	5e+05	5e+05	5e+05	5e+05	5e+05	5e+05	5e+05
2	99.478 %	99.478 %	81.0128 %	81.0128 %	90.1632 %	90.1632 %	99.6386 %	99.6386 %	99.6506 %	99.6506 %
3	499.978	504.586	159.387	193.786	273.455	285.959	499.990	500.579	499.993	500.319
4	25.001	25.003	1.546	4.656	3.573	4.436	25.001	25.007	25.001	25.002
5	474.977	479.583	157.841	189.130	269.882	281.523	474.989	475.572	474.992	475.317
6	0 %	98.8743 %	0 %	70.2873 %	0 %	94.7178 %	0 %	99.5313 %	0 %	99.5589 %
7	0 %	0.9132 %	0 %	18.9761 %	0 %	4.5996 %	0 %	0.1177 %	0 %	0.0652 %
8	0.0044 %	98.864 %	68.1226 %	88.4842 %	45.309 %	96.979 %	0.002 %	99.5308 %	0.0014 %	99.5586 %
9	0	99.9955 %	0 %	52.2787 %	0 %	84.8429 %	0 %	99.998 %	0 %	99.9986 %
10	0 %	99.0764 %	0 %	73.0021 %	0 %	95.1439 %	0 %	99.8818 %	0 %	99.9346 %
	mcf		**milc**		**namd**		**omnetpp**		**Openoffice**	
Info	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.
1	5e+05	5e+05	5e+05	5e+05	5e+05	5e+05	5e+05	5e+05	5e+05	5e+05
2	97.9274 %	97.9274 %	89.09 %	89.09 %	96.912 %	96.912 %	60.1954 %	60.1954 %	86.1142 %	86.1142 %
3	495.722	497.387	129.344	215.910	499.956	504.417	416.564	461.676	485.388	623.756
4	24.931	25.122	4.946	7.491	25.001	25.003	14.434	18.308	24.826	36.691
5	470.791	472.265	124.398	208.419	474.955	479.414	402.130	443.368	460.562	587.065
6	0 %	97.4507 %	0 %	87.1905 %	0 %	88.7553 %	0 %	40.1751 %	0 %	82.0386 %
7	0 %	0.3356 %	0 %	44.8983 %	0 %	0.8844 %	0 %	9.7724 %	0 %	22.4639 %
8	0.8556 %	97.464 %	74.1312 %	94.4686 %	0.0088 %	88.656 %	16.6872 %	44.7606 %	2.9224 %	77.593 %
9	0 %	99.6087 %	0 %	52.5051 %	0 %	99.9928 %	0 %	75.3958 %	0 %	96.9054 %
10	0 %	99.6557 %	0 %	48.5055 %	0 %	99.0036 %	0 %	75.6754 %	0 %	72.6179 %
	povray		**sjeng**		**specrand**		**sphinx3**		**Xalan**	
Info	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.
1	185.871	1858.71	5e+05	5e+05	3.508	3.508	5e+05	5e+05	5e+05	5e+05
2	89.7418 %	89.7418 %	99.4822 %	99.4822 %	49.3444 %	49.3444 %	96.2442 %	96.2442 %	79.0316 %	79.0316 %
3	60.688	68.633	499.995	500.306	3.504	3.845	499.692	566.486	182.274	205.361
4	468	1.997	25.001	25.002	3.504	3.845	25.001	25.002	7.577	13.367
5	60.220	66.636	474.994	475.304	0	0	474.691	541.484	174.697	191.994
6	0 %	49.1804 %	0 %	99.2912 %	0 %	44.6294 %	0 %	94.7554 %	0 %	36.0025 %
7	0 %	11.5775 %	0 %	0.0622 %	0 %	8.8687 %	0 %	11.7909 %	0 %	11.6351 %
8	67.3494 %	81.2348 %	0.001 %	99.2908 %	0.114 %	39.3101 %	0.0616 %	94.058 %	63.5452 %	73.7148 %
9	0 %	25.6795 %	0 %	99.999 %	0 %	99.7099 %	0 %	99.976 %	0 %	32.6689 %
10	0 %	76.4591 %	0 %	99.9374 %	0 %	80.1282 %	0 %	87.5564 %	0 %	67.6824 %

Tabla 5.4: Comparación de la caché SDRAM con y sin prebúsqueda, suponiendo una SDRAM ideal con umbral de olvido

Nótese que aquellas aplicaciones cuya diferencia en la tasa de éxito es muy pequeña es porque tienen una localidad temporal muy marcada. Gracias a la caracterización de aplicaciones y el agrupamiento en base a la localidad temporal del Capítulo 4, podemos distinguir las aplicaciones que tardan poco y que tardan mucho en repetir accesos. Si la aplicación tarda poco, el mecanismo de prebúsqueda ideado debería tener menos peso, por ejemplo, limitando la amplitud del intervalo de predicción. Estas optimizaciones se pueden adaptar dinámicamente al comportamiento de las aplicaciones de forma sencilla.

Cabe destacar que todas estas conclusiones son aproximadas. En los resultados que se presentan a continuación se examinan 5 millones de accesos, 10 veces más de lo que hemos comprobado aquí. Aunque esto nos ha servido para determinar si esta técnica funciona o no funciona bien, a continuación podremos establecer con más precisión una comparación entre ambas situaciones.

5.6.2. Caché SDRAM asociativa por conjuntos

Ahora presentaremos los resultados para las dos cachés siguientes:

- SDRAM de 16MB, configurada asociativa por conjuntos de nivel 4.
- SDRAM de 64MB, configurada asociativa por conjuntos de nivel 8.

Evidentemente, es de esperar que una caché de 64MB se comporte mejor que una de 16MB en cuanto a porcentaje de éxito y tasa de desperdicio. Al tener una mayor capacidad global y por conjuntos, podemos almacenar más información antes de que sea necesario reemplazar una entrada de la caché por otra. No obstante, la diferencia entre el comportamiento de estas dos cachés puede ser útil para valorar cómo se comporta el mecanismo de prebúsqueda cuando está incluido en una SDRAM sometida a mucho estrés. En todos los casos, se ha establecido un límite máximo de 5 millones para el número de accesos examinados.

De nuevo, presentaremos los resultados en formato gráfico, además de una tabla con todas las estadísticas. Las figuras 5.18 y 5.19 presentan la tasa de éxito para el caso con y sin prebúsqueda, y la tasa de desperdicio necesaria para alcanzar esa diferencia, de las configuraciones SDRAM 16/4 y 64/8 respectivamente. Las tablas 5.5 y 5.6 contienen la información exacta sobre los diez puntos mencionados anteriormente.

En el último gráfico, existe una aplicación señalada en rojo. Como ya se ha comentado, la simulación es muy pesada y se realiza íntegramente en R. Ésta concretamente aún se encuentra en ejecución, de forma que lo que se muestra en la tabla y el gráfico son resultados parciales, y el número de accesos examinados es ligeramente diferente en el caso con y sin prebúsqueda.

En general, podemos establecer que:

- Al considerar un mayor número de accesos en comparación con los que se tenían en la caché ideal, la tasa de éxitos sin prebúsqueda (correspondiente a la localidad temporal) aumenta.

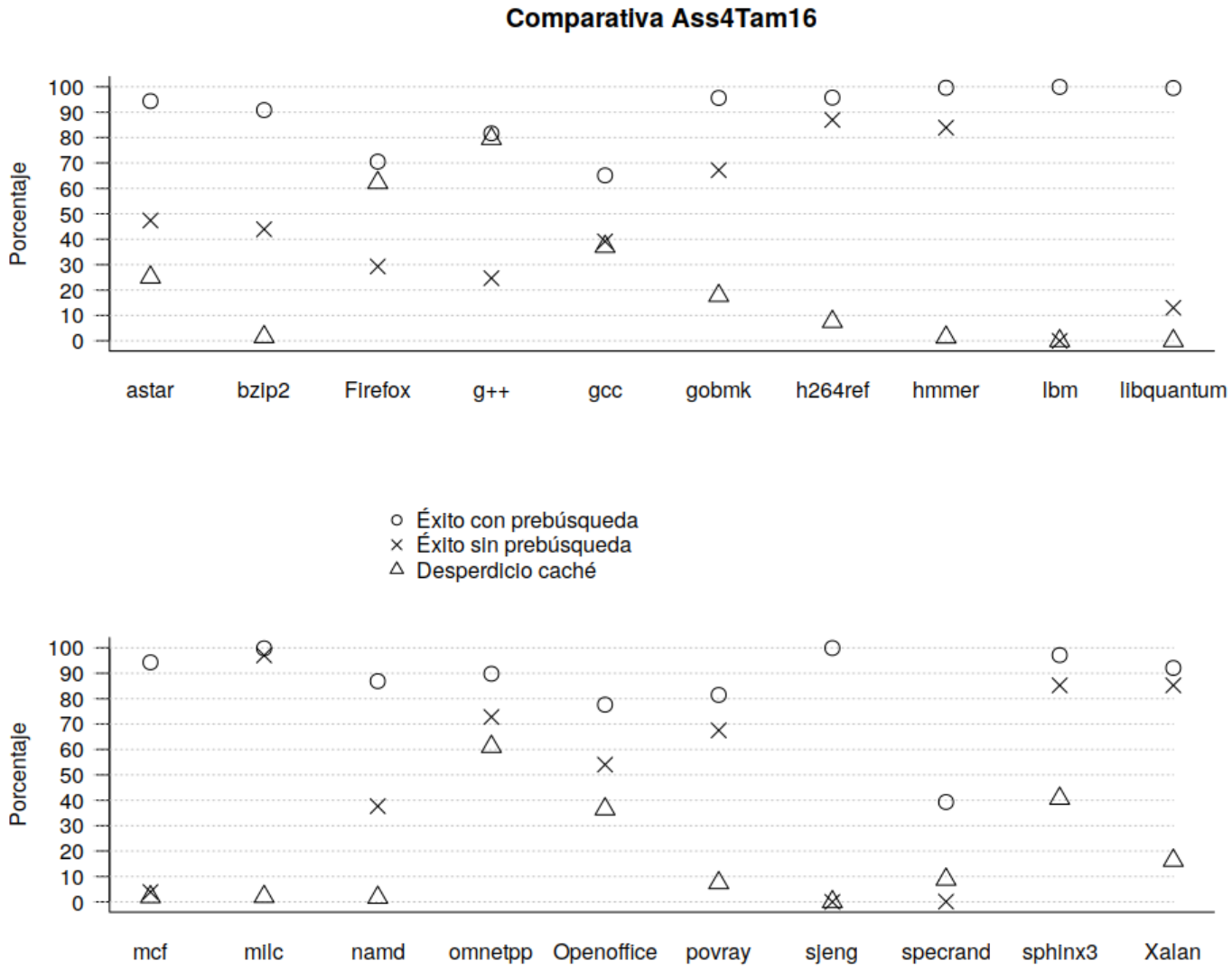


Figura 5.18: Gráfico de comparación de la simulación con y sin prebúsqueda, suponiendo una caché SDRAM 16/4

	astar		**bzip2**		**Firefox**		**g++**		**gcc**	
Info	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.
1	249.164	249.164	615.821	615.821	5e+06	5e+06	3.500.000	3.500.000	5e+06	5e+06
2	88.4951 %	88.4951 %	91.5649 %	91.5649 %	77.5941 %	77.5941 %	81.5245 %	81.5245 %	62.5091 %	62.5091 %
3	131.129	174.957	345.338	352.689	3.536.553	10.687.108	2.636.385	14.22.9164	3.042.364	5.681.922
4	131.111	174.844	247.881	249.553	262.144	262.144	262.144	262.144	262.144	262.144
5	18	113	97.457	103.136	3.274.409	10.424.964	2.374.241	13.967.020	2.780.220	5.419.778
6	0 %	91.9769 %	0 %	83.9533 %	0 %	86.2164 %	0 %	95.4904 %	0 %	69.3054 %
7	0 %	25.0507 %	0 %	1.5767 %	0 %	62.2513 %	0 %	79.5546 %	0 %	37.1228 %
8	47.3724 %	94.3664 %	43.9223 %	90.8098 %	29.2689 %	70.5387 %	24.6747 %	81.6663 %	39.1527 %	65.1191 %
9	0 %	75.5179 %	0 %	81.0086 %	0 %	88.6058 %	0 %	94.358 %	0 %	81.4927 %
10	0 %	72.7641 %	0 %	98.1219 %	0 %	27.7964 %	0 %	16.6884 %	0 %	46.4359 %
	gobmk		**h264ref**		**hmmer**		**lbm**		**libquantum**	
Info	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.
1	5e+06	5e+06	2.648.497	2.648.497	3.351.430	3.351.430	5e+06	5e+06	5e+06	5e+06
2	95.8011 %	95.8011 %	81.7642 %	81.7642 %	95.7508 %	95.7508 %	99.9639 %	99.9639 %	99.9587 %	99.9587 %
3	1.641.539	2.369.320	345.580	375.885	539.189	549.664	4.999.990	5.000.583	4.348.492	4.348.039
4	262.144	262.144	258.627	259.453	262.144	262.144	262.144	262.144	262.144	262.144
5	1.379.395	2.107.176	86.953	116.432	277.045	287.520	4.737.846	4.738.439	4.086.348	4.085.895
6	0 %	90.7586 %	0 %	70.0055 %	0 %	97.5259 %	0 %	99.9511 %	0 %	99.4216 %
7	0 %	17.7935 %	0 %	7.5701 %	0 %	1.3916 %	0 %	0.0119 %	0 %	0.0121 %
8	67.1692 %	95.6208 %	86.9518 %	95.7431 %	83.9117 %	99.5942 %	2e-04 %	99.9511 %	13.0302 %	99.497 %
9	0 %	96.4031 %	0 %	53.0196 %	0 %	92.1073 %	0 %	99.9998 %	0 %	99.9706 %
10	0 %	80.3946 %	0 %	89.1864 %	0 %	98.5731 %	0 %	99.9881 %	0 %	99.9878 %
	mcf		**milc**		**namd**		**omnetpp**		**Openoffice**	
Info	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.
1	5e+06	5e+06	5e+06	5e+06	1.961.439	1.961.439	5e+06	5e+06	5e+06	5e+06
2	97.836 %	97.836 %	97.3394 %	97.3394 %	86.8271 %	86.8271 %	87.9315 %	87.9315 %	78.2115 %	78.2115 %
3	4.805.488	4.903.580	149.504	152.659	1.222.709	1.247.516	1.360.975	3.937.055	2.298.016	3.797.078
4	262.144	262.144	149.504	152.659	262.144	262.144	262.144	262.144	262.144	262.144
5	4.543.344	4.641.436	0	0	960.565	985.372	1.098.831	3.674.911	2.035.872	3.534.934
6	0 %	94.1664 %	0 %	93.5195 %	0 %	79.364 %	0 %	87.0196 %	0 %	70.5257 %
7	0 %	1.9804 %	0 %	2.0667 %	0 %	1.7254 %	0 %	61.1063 %	0 %	36.5457 %
8	3.8902 %	94.2789 %	97.0099 %	99.8021 %	37.6627 %	86.875 %	72.7805 %	89.7791 %	54.0397 %	77.6168 %
9	0 %	99.0722 %	0 %	95.0254 %	0 %	83.02 %	0 %	46.2416 %	0 %	68.1725 %
10	0 %	97.8969 %	0 %	97.7901 %	0 %	97.8259 %	0 %	29.7787 %	0 %	48.181 %
	povray		**sjeng**		**specrand**		**sphinx3**		**Xalan**	
Info	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.
1	185.871	185.871	5e+06	5e+06	3.508	3.508	5e+06	5e+06	5e+06	5e+06
2	89.5261 %	89.5261 %	99.941 %	99.941 %	49.3444 %	49.3444 %	94.9335 %	94.9335 %	90.0227 %	90.0227 %
3	60.411	65.293	4.999.984	5.000.863	3.504	3.845	738.842	1.455.900	738.807	911.172
4	60.411	65.293	262.144	262.144	3.504	3.845	262.144	262.144	262.144	262.144
5	0	0	4.737.840	4.738.719	0	0	476.698	1.193.756	476.663	649.028
6	0 %	47.2118 %	0 %	99.9058 %	0 %	44.6294 %	0 %	90.121 %	0 %	56.4889 %
7	0 %	7.4771 %	0 %	0.0177 %	0 %	8.8687 %	0 %	40.711 %	0 %	16.3114 %
8	67.4984 %	81.4565 %	3e-04 %	99.9057 %	0.114 %	39.3101 %	85.2232 %	97.1234 %	85.2239 %	92.0708 %
9	0	24.1348 %	0 %	99.9999 %	0 %	99.7099 %	0 %	70.6594 %	0 %	46.1561 %
10	0 %	84.1627 %	0 %	99.9823 %	0 %	80.1282 %	0 %	54.8263 %	0 %	71.1246 %

Tabla 5.5: Comparación de la caché SDRAM con y sin prebúsqueda, suponiendo una SDRAM con configuración 16/4.

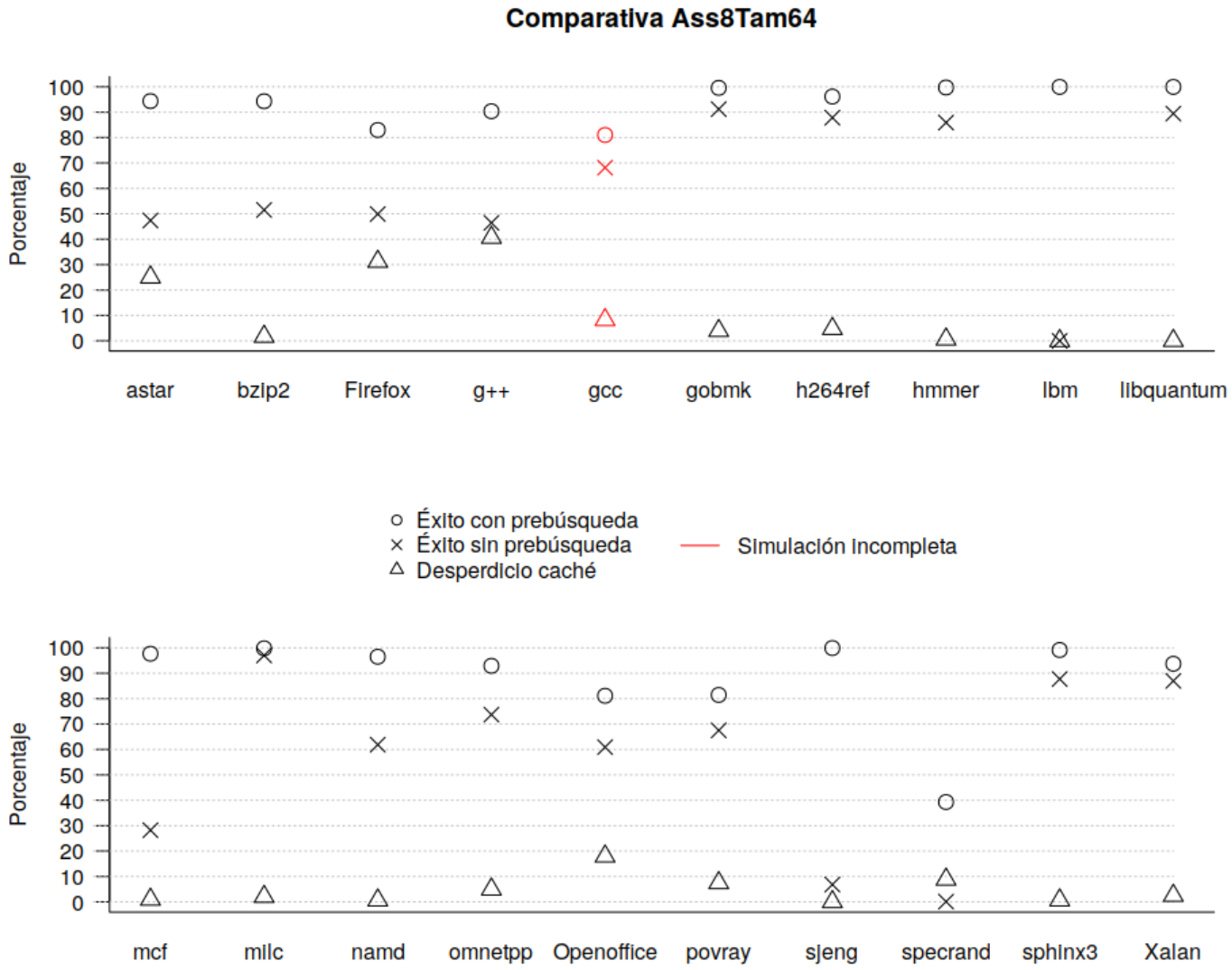


Figura 5.19: Gráfico de comparación de la simulación con y sin prebúsqueda, suponiendo una caché SDRAM 64/8

	astar		bzip2		Firefox		g++		gcc	
Info	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.
1	249.164	249.164	615.821	615.821	5e+06	5e+06	5e+06	5e+06	5e+06	3.500.000
2	88.4951 %	88.4951 %	91.5649 %	91.5649 %	77.5941 %	77.5941 %	81.326 %	81.326 %	62.5091 %	52.5763 %
3	131.129	174.957	298.537	303.736	2.504.406	4.002.909	2.677.753	4.790.973	1.591.902	1.526.758
4	131.129	174.957	298.537	303.351	1.046.783	1.048.560	1.048.482	1.048.576	1.048.482	1.048.566
5	0	0	0	385	1.457.623	2.954.349	1.629.271	3.742.397	543.420	478.192
6	0 %	91.9769 %	0 %	88.4703 %	0 %	78.7517 %	0 %	89.9538 %	0 %	56.4989 %
7	0 %	25.0507 %	0 %	1.7114 %	0 %	31.2104 %	0 %	40.6804 %	0 %	8.2658 %
8	47.3724 %	94.3664 %	51.5221 %	94.3133 %	49.9119 %	82.989 %	46.4449 %	90.3738 %	68.162 %	81.0241 %
9	0 %	75.5179 %	0 %	86.7597 %	0 %	78.3968 %	0 %	86.6124 %	0 %	58.005 %
10	0 %	72.7641 %	0 %	98.0656 %	0 %	60.3686 %	0 %	54.7764 %	0 %	85.3701 %
	gobmk		**h264ref**		**hmmer**		**lbm**		**libquantum**	
Info	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.
1	5e+06	5e+06	2.648.497	2.648.497	3.351.430	3.351.430	5e+06	5e+06	5e+06	5e+06
2	95.8011 %	95.8011 %	81.7642 %	81.7642 %	95.7508 %	95.7508 %	99.9639 %	99.9639 %	99.9587 %	99.9587 %
3	439.968	458.208	322.396	338.547	472.291	475.281	4.999.990	5.000.583	528.043	528.370
4	439.968	458.208	322.396	338.547	472.291	475.281	1.048.576	1.048.576	528.043	528.370
5	0	0	0	0	0	0	3.951.414	3.952.007	0	0
6	0 %	95.2281 %	0 %	69.8417 %	0 %	97.9681 %	0 %	99.9511 %	0 %	99.5359 %
7	0 %	3.9807 %	0 %	4.7707 %	0 %	0.6291 %	0 %	0.0119 %	0 %	0.0619 %
8	91.2006 %	99.5627 %	87.8272 %	96.145 %	85.9078 %	99.7119 %	2e-04 %	99.9511 %	89.4391 %	99.951 %
9	0 %	95.7853 %	0 %	49.0224 %	0 %	92.4977 %	0 %	99.9998 %	0 %	99.9804 %
10	0 %	95.8198 %	0 %	93.1693 %	0 %	99.3579 %	0 %	99.9881 %	0 %	99.9378 %
	mcf		**milc**		**namd**		**omnetpp**		**Openoffice**	
Info	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.
1	5e+06	5e+06	5e+06	5e+06	1.961.439	1.961.439	5e+06	5e+06	5e+06	5e+06
2	97.836 %	97.836 %	97.3394 %	97.3394 %	86.8271 %	86.8271 %	87.9315 %	87.9315 %	78.2115 %	78.2115 %
3	3.589.335	3.641.941	149.504	152.659	747.689	752.750	1.314.557	1.384.853	1.954.530	2.443.343
4	1.048.576	1.048.576	149.504	152.659	747.689	752.750	1.048.573	1.048.576	1.048.576	1.048.576
5	2.540.759	2.593.365	0	0	0	0	265.984	336.277	905.954	1.394.767
6	0 %	96.814 %	0 %	93.5195 %	0 %	90.86 %	0 %	74.3714 %	0 %	61.3628 %
7	0 %	0.9628 %	0 %	2.0667 %	0 %	0.6723 %	0 %	4.969 %	0 %	17.935 %
8	28.2133 %	97.6794 %	97.0099 %	99.8021 %	61.8806 %	96.4923 %	73.7089 %	92.9016 %	60.9094 %	81.1192 %
9	0 %	98.8851 %	0 %	95.0254 %	0 %	91.7542 %	0 %	72.5325 %	0 %	59.6202 %
10	0 %	99.0055 %	0 %	97.7901 %	0 %	99.26 %	0 %	93.3187 %	0 %	70.7722 %
	povray		**sjeng**		**specrand**		**sphinx3**		**Xalan**	
Info	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.	Sin pr.	Con pr.
1	185.871	185.871	5e+06	5e+06	3.508	3.508	5e+06	5e+06	5e+06	5e+06
2	89.5261 %	89.5261 %	99.941 %	99.941 %	49.3444 %	49.3444 %	94.9335 %	94.9335 %	90.0227 %	90.0227 %
3	60.411	65.293	4.658.414	4.659.350	3.504	3.845	614.407	618.672	651.582	668.189
4	60.411	65.293	1.048.576	1.048.576	3.504	3.845	614.407	618.672	651.582	668.189
5	0	0	3609838	3610774	0	0	0	0	0	0
6	0 %	47.2118 %	0 %	99.9008 %	0 %	44.6294 %	0 %	93.0272 %	0 %	52.9723 %
7	0 %	7.4771 %	0 %	0.0189 %	0 %	8.8687 %	0 %	0.6894 %	0 %	2.4854 %
8	67.4984 %	81.4565 %	6.8317 %	99.9076 %	0.114 %	39.3101 %	87.7119 %	99.1372 %	86.9684 %	93.7153 %
9	0 %	24.1348 %	0 %	99.998 %	0 %	99.7099 %	0 %	93.4953 %	0 %	35.4075 %
10	0 %	84.1627 %	0 %	99.9811 %	0 %	80.1282 %	0 %	99.2589 %	0 %	95.3082 %

Tabla 5.6: Comparación de la caché SDRAM con y sin prebúsqueda, suponiendo una SDRAM con configuración 64/8.

Esto es debido a que, en muchos casos, se reutilizan líneas en instantes futuros. Además, al eliminar la restricción del umbral de olvido, ya no es necesario re-acceder a ella en menos de 25.000 accesos, puesto que ahora está limitado por el tamaño real de la caché; en muchos casos evitamos un reemplazo innecesario.

- La tasa de éxitos derivada de la localidad temporal también aumenta al pasar de una configuración SDRAM 16/4 a 64/8, puesto que hay menos reemplazos, y una línea vive más en la caché, teniendo más tiempo para ser aprovechada.
- La tasa de éxitos en el caso con prebúsqueda suele aumentar al pasar de la caché ideal a la caché 16/4, aunque en algunos casos se produce una ligera disminución. De nuevo, al pasar a una 64/8 también se produce una mejora.
- Hay muchas variaciones en cuanto a la tasa de desperdicio, tendiendo a aumentar al considerar más accesos y la caché 16/4. Sin embargo, al aumentar el tamaño, el desperdicio es cada vez menor.

Con una configuración SDRAM 16/4, podemos distinguir los siguientes grupos, en función del impacto de la prebúsqueda:

- Las aplicaciones libquantum, lbm, mcf y sjeng funcionan extraordinariamente bien. Para la primera, la tasa de éxito pasa de un 0.0002 % a un 99.9511 %, debido en gran parte al buen reconocimiento en grupos con el modelo de Markov, con un desperdicio de tan sólo 0.0119 %. Situaciones similares se producen en las demás.
- Existen otras aplicaciones donde ya había una componente de localidad temporal notoria, pero aún así la técnica ofrece estupendos resultados. Es el caso, por ejemplo, de astar, bzip2, gobmk y namd. La tasa de éxito aumenta, aunque a costa de algunas líneas desperdiciadas.
- Existen otras aplicaciones, como Firefox y g++, que también aumentan notablemente la tasa de éxito. En éstas, sin embargo, no llega a valores excesivamente altos, y la tasa de desperdicio es enorme. Para Firefox, el 62.25 % de las entradas creadas en la caché no se utilizan nunca, mientras que para g++ este porcentaje es de casi el 80 %.
- También podemos encontrar un grupo de aplicaciones cuya mejora es mínima, pues la localidad temporal ya hacía casi todo el trabajo (h264ref, hmmer, Xalan, sphinx3).

Por otro lado, la configuración 64/8, en comparación, tiene unas tasas de desperdicio menores, mejores tasas de éxito, y proporciona mejores resultados en general. Aunque esto es esperable, la diferencia en el rendimiento de estas dos cachés puede darnos una idea sobre cuán exhaustiva es la técnica. En algunas aplicaciones podemos observar un gran número de reemplazos en la caché 16/4. Es el ejemplo de Firefox, donde la prebúsqueda lleva en total a desalojar más de 10 millones de entradas. En la configuración 64/8, el número de entradas desalojadas es de apenas 3 millones. La causa de esta diferencia será, probablemente, la propia prebúsqueda. Al utilizar una caché pequeña

en una aplicación exigente habrá colisiones, especialmente si los grupos están fragmentados y los intervalos de predicción son muy amplios. La prebúsqueda sobrecarga demasiado la caché, y muchas son desalojadas antes de tener tiempo de ser utilizadas. Con un tamaño mayor, la caché no está sometida a este estrés y puede implementar mejor la técnica.

Con una configuración 64/8 podemos concluir que:

- lbm, mcf y sjeng siguen manteniendo un comportamiento muy bueno. Algo diferente ocurre en libquantum, donde el aumento de la caché hace que la localidad temporal tenga mucho más peso y que no se reemplacen las líneas tan fácilmente. La tasa de éxito sin prebúsqueda pasa de un 13.03 % con una SDRAM 16/4, a un 89.43 % con una 64/8.
- Las aplicaciones bzip2, astar, y namd mantienen un comportamiento similar. De forma similar a limquantum, la localidad temporal toma más peso en gobmk.
- Firefox y g++ mantienen su comportamiento, con una mejora general del rendimiento.
- Apenas se producen cambios donde las aplicaciones ya mostraban una fuerte localidad temporal.

El número total de simulaciones es muy alto. Para cada una de las veinte aplicaciones se han tomado 6 escenarios distintos, y se presentan diez campos numéricos como resumen de cada par aplicación-escenario. Sería muy arduo e innecesario realizar una interpretación completa de todas las estadísticas presentadas en las tablas 5.5 y 5.6. En lugar de ello, vamos a elegir dos aplicaciones, una con un buen funcionamiento de la técnica, y otra donde se den algunos problemas.

Vamos a considerar en primer lugar el ejemplo de Firefox, una de las aplicaciones con una menor tasa de éxito en la simulación con prebúsqueda. La técnica de Markov es capaz de reconocer tan sólo el 77 % de los accesos, de forma que un número no despreciable (más de 1 millón) no se encuadra en ningún grupo. La caché 16/4 está siendo completamente utilizada: las 262.144 entradas que hay disponibles se encuentran en uso. Existe un gran número de reemplazos en los dos casos. Sin tener en cuenta la prebúsqueda, hay más de 3 millones de reemplazos, que ascienden hasta 10 millones al utilizar la técnica. El 86.21 % de la caché está compuesta por entradas que han sido prebuscadas. Desgraciadamente, el 62.25 % de las entradas creadas no se utilizan nunca, y sólo el 27.8 % de las prebúsquedas son realmente utilizadas. A pesar de este desperdicio, la técnica permite aumentar la tasa de éxito notablemente, pasando de un 29.2689 % al no haber prebúsqueda, a un 70.53 %. Otra medida que asegura el beneficio del prebuscador es el porcentaje de aciertos debidos a la prebúsqueda, que asciende a un 88.61 %. A pesar de que Firefox es una aplicación compleja e interactiva, y de que no se reconocen todos los accesos, podemos estar satisfechos con el funcionamiento global.

Con una caché SDRAM 64/8 obtenemos mejores resultados. Cuando no hay prebúsqueda se producen aproximadamente 1 millon y medio de reemplazos, cantidad que duplica cuando sí la hay. En este caso no se llena por completo la SDRAM de entradas de caché. La tasa de éxito pasa

de un 50 % a un 83 % al considerar la prebúsqueda, y la tasa de desperdicio se reduce a la mitad: el 31.2 % de las entradas no se utilizan. La cantidad de prebúsquedas que resultan útiles aumenta hasta un 60.36 %, lo que confirma que unas prebúsquedas reemplazan a otras si la caché no tiene el tamaño suficiente. No obstante, al ser también más complicado eliminar las entradas creadas por localidad temporal, el porcentaje de aciertos por prebúsqueda desciende de un 88.61 % a un 78.4 %.

La causa de que Firefox sea una aplicación más difícil de clasificar en grupos es la fragmentación de los accesos a memoria. Como hablamos de una aplicación interactiva, que dispara muchos hilos, y realiza muchas tareas simultáneamente, es mucho más complicado distinguir zonas claras de uso de la memoria. De forma similar a lo que ocurría con gcc, la mayoría de los grupos reconocidos tienen un tamaño muy pequeño. En la figura 5.20 se encuentra el histograma para el tamaño de los grupos. Es claramente apreciable cómo la mayoría de grupos tiene menos de 2.000 elementos, muchos de ellos menos que 500 accesos. La prebúsqueda sólo se lanza cuando hay al menos 100 accesos en un mismo estado, por lo que un número muy elevado de grupos dificulta la tarea.

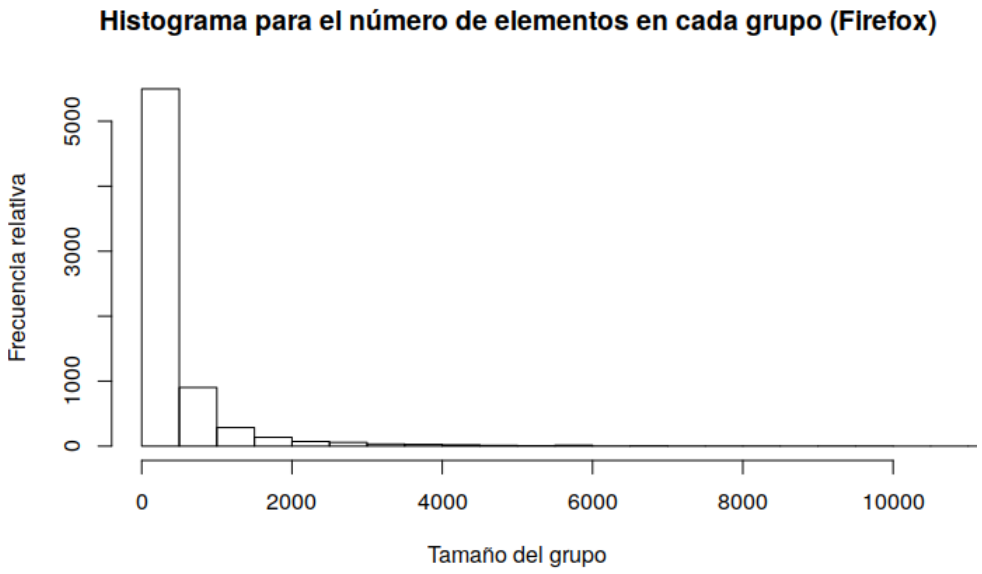
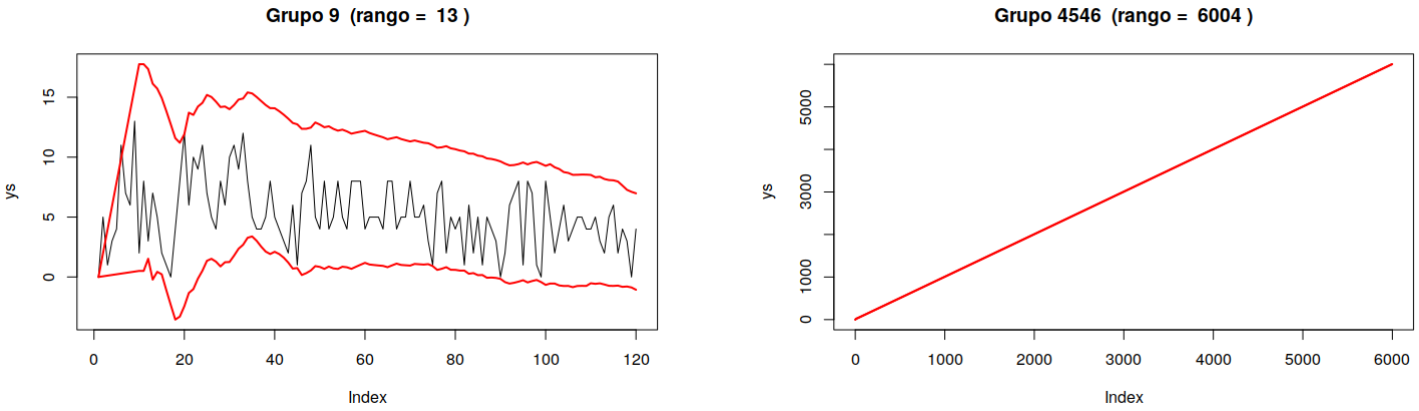


Figura 5.20: Histograma para el número de elementos que hay en cada grupo reconocido de la aplicación Firefox.

En la figura 5.21 se muestran dos ejemplos de grupos reconocidos para Firefox. En la parte de la izquierda se representa un grupo pequeño, de 120 accesos, con una amplitud de 13 líneas de memoria. Podemos ver que muchas de las líneas se repiten con el tiempo, provocado por una LLC en el nivel superior que elige mal a sus víctimas. Además, muchas de las líneas que se deban traer con la prebúsqueda ya se encontrarán en la SDRAM. En la parte de la derecha tenemos un grupo fácil de reconocer, grande, y con una tendencia creciente. La prebúsqueda aquí tiene un potencial mucho mayor.

Adicionalmente, en la figura 5.22 se han representado trozos de las peticiones de acceso en

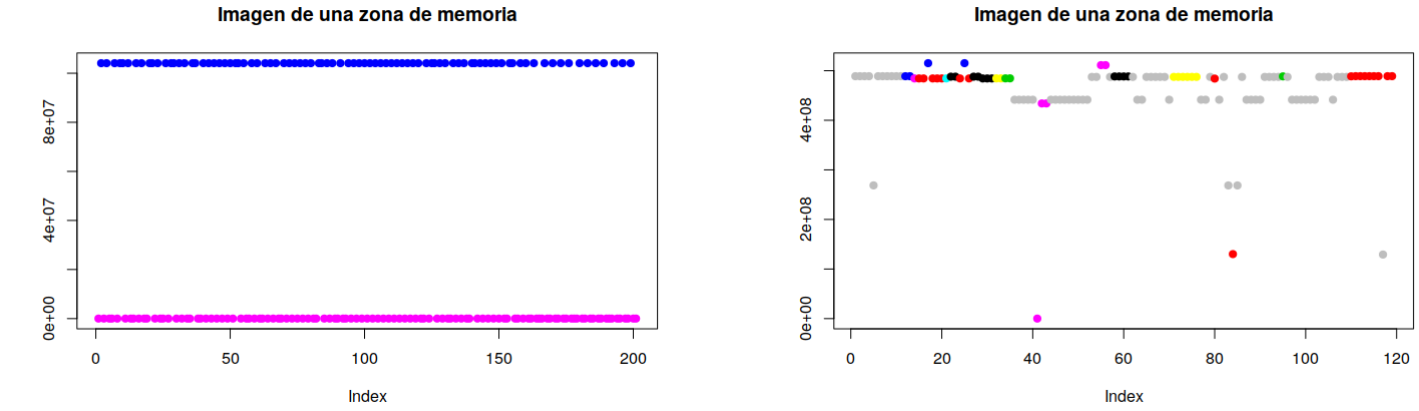


(a) Grupo pequeño

(b) Grupo grande

Figura 5.21: Ejemplos de grupos reconocidos para Firefox

diferentes puntos de la ejecución. Los colores indican una separación por los estados reconocidos en el modelo de Markov. En la izquierda tenemos una imagen donde los accesos se van alternando a dos zonas de memoria. En la derecha, los accesos están repartidos a lo largo de un espacio muy grande de direcciones, y es imposible detectar patrones.



(a) Zona reconocible

(b) Zona irreconocible

Figura 5.22: Ejemplos de zonas de memoria para Firefox

Como ejemplo de aplicación con excelente rendimiento en la arquitectura propuesta se elige mcf. El modelo oculto de Markov reconoce con éxito el 97.836 % de los accesos a memoria producidos. En la figura 5.23 se encuentra el histograma del tamaño de los grupos. En total se reconocen 2391 estados diferentes, cuando en Firefox se tenían más de 7000. Aunque en el histograma parece que los grupos tienen un tamaño pequeño, existen tres grupos especialmente grandes que no aparecen en esta figura. El estado más grande tiene asignados 2.495.933 accesos de memoria, y los siguientes más destacados 934.729 y 102.405. Esto indica una gran facilidad para reconocer accesos, lo que repercute en el rendimiento del prebuscador.

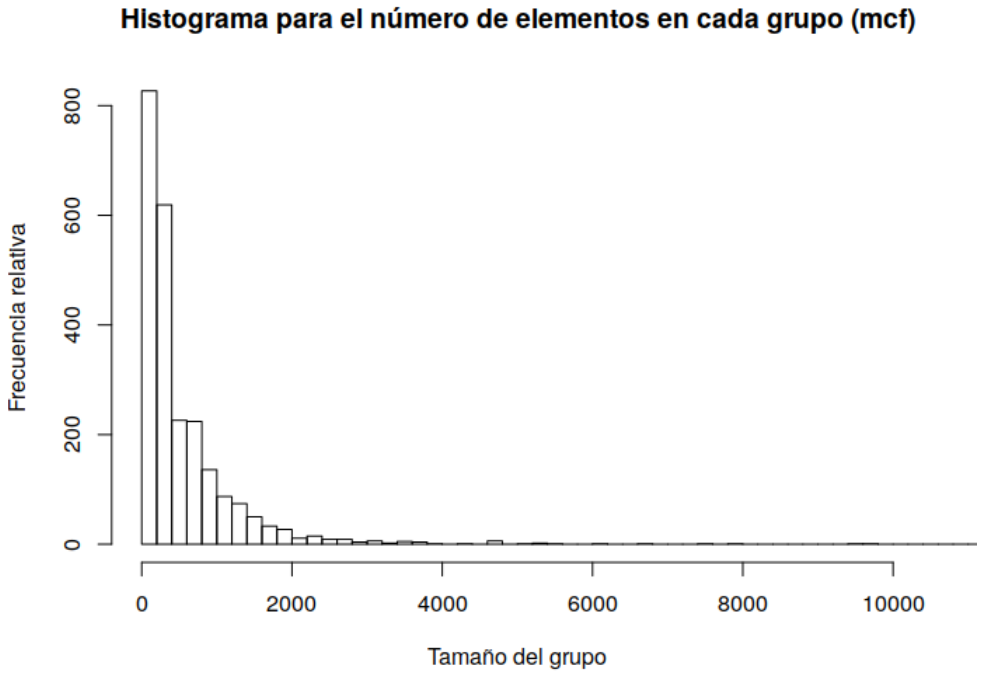


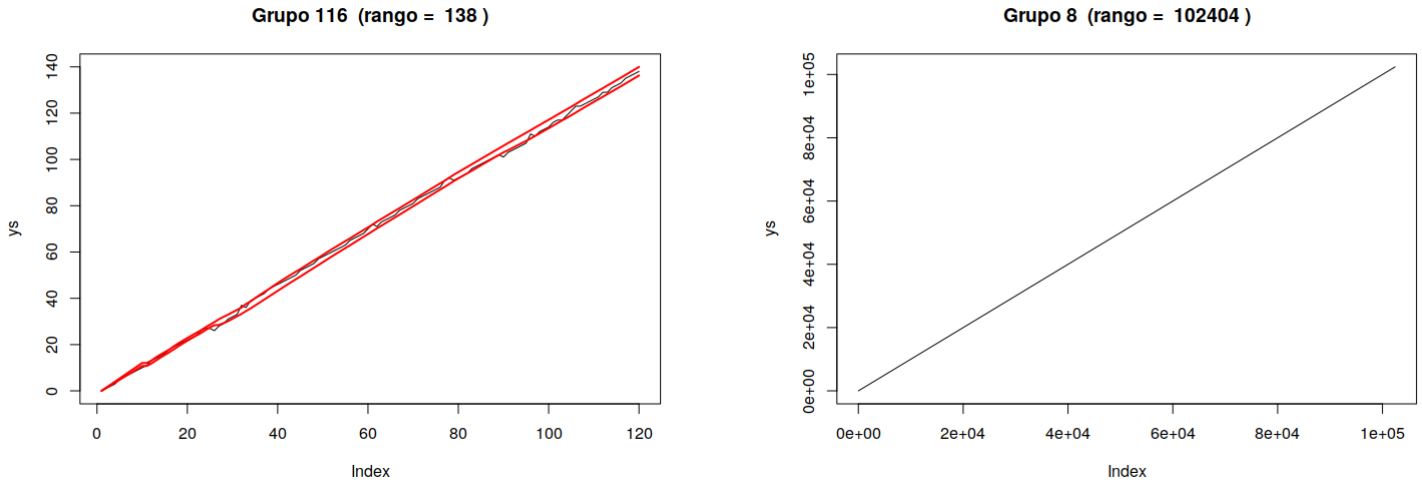
Figura 5.23: Histograma para el número de elementos que hay en cada grupo reconocido de la aplicación mcf.

Tomando la configuración 16/4, la caché se utiliza completamente, y tanto cuando existe prebúsqueda como cuando no, hay aproximadamente 4 millones y medio de reemplazos. El número de entradas totales es ligeramente inferior al número de accesos examinados porque hay repeticiones en los accesos. Que el número de reemplazos no sea significativamente más grande cuando el prebuscador actúa significa que los intervalos de predicción son muy estrechos y que hay muchos aciertos de caché. Concretamente, pasamos de un 3.8 % a un 94.27 % con la técnica. El desperdicio de la caché es mínimo, tan sólo un 1.98 % de las entradas no se utilizan nunca. El 94.16 % de ellas se crean por orden del mecanismo de prebúsqueda, y el 97.89 % de todas estas entradas resultan finalmente útiles. La inmensa mayoría de aciertos (99.07 %) se deben a la técnica.

En la configuración 64/8, la tasa de éxito tomando sólo la localidad temporal pasa a un 28.21 %, previsiblemente porque ya no hay tantos desalojos. La tasa de acierto asciende a un 97.67 % al utilizar la prebúsqueda. Esta SDRAM de 64MB también está llena, aunque en total se crean aproximadamente un millón menos de entradas con respecto a la 16/4. La tasa de desperdicio también disminuye, siendo ahora de 0.9628 %, y la tasa de prebúsquedas útiles supera el 99 %.

En la figura 5.24 se encuentran dos imágenes que corresponden a un grupo pequeño y otro grande de los reconocidos por el procedimiento de Markov. El grupo pequeño en este caso presenta una tendencia creciente, con el intervalo de predicción muy estrecho, lo que agiliza la prebúsqueda. El grupo grande, con aproximadamente 100.000 accesos, muestra un comportamiento creciente también. Los grupos son, en general, mucho más fáciles de reconocer que en Firefox.

En la figura 5.25 se encuentra la imagen de un par de tramos en el recorrido de la aplicación.

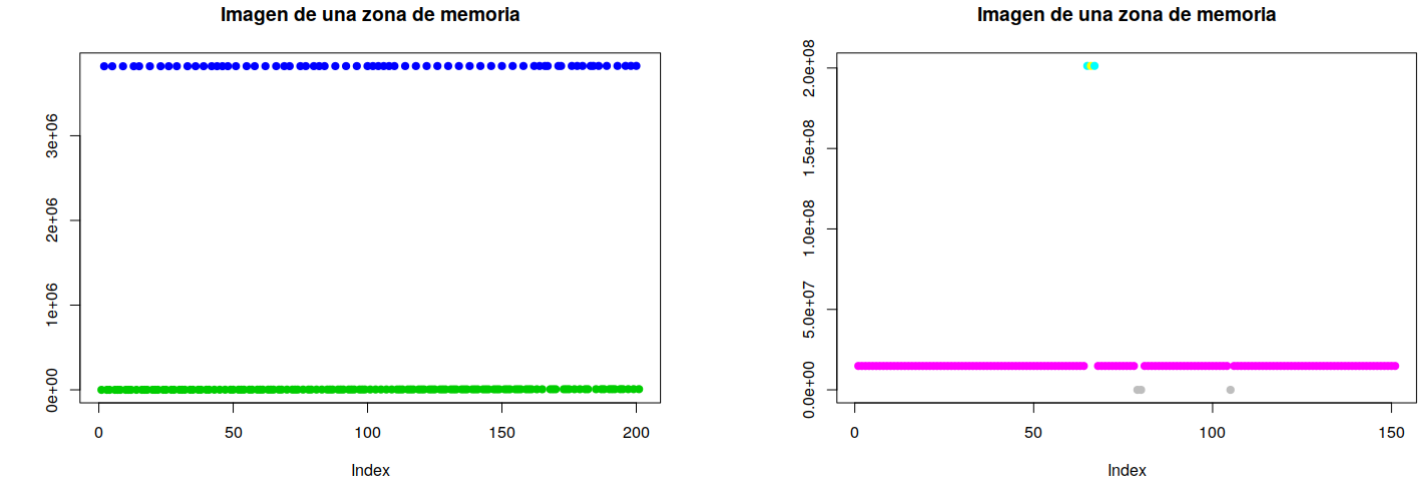


(a) Grupo pequeño

(b) Grupo grande

Figura 5.24: Ejemplos de grupos reconocidos para mcf

Las zonas son, en general, mucho menos caóticas que Firefox. En la izquierda tenemos una porción similar a la representada en la figura 5.22, mientras que en la derecha podemos ver un comportamiento regular, aunque con ligeros accesos a otras zonas separadas.



(a) Zona 1

(b) Zona 2

Figura 5.25: Ejemplos de zonas de memoria para mcf

Queda entonces mostrado el enorme potencial de la técnica desarrollada, en general para las 20 aplicaciones, y en particular para estos dos ejemplos. Esta técnica, compuesta en dos fases (la fase de separación de estados, y la fase de prebúsqueda), es capaz de aprender de las peticiones de acceso que llegan al sistema de memoria principal, y anticiparse a las necesidades de la LLC preparando el contenido con anterioridad. Incluso en aplicaciones donde no es posible reconocer muchos patrones, la tasa de éxitos aumenta significativamente, aunque a costa muchas veces de

un alto desperdicio.

Todas las simulaciones realizadas en este apartado suponen una LLC con una configuración 16/1. Bajo este escenario, con una LLC de la que no se espera un comportamiento bueno, disponemos de mucha información a nivel de SDRAM. En una configuración 32/8, no siempre podremos esperar estos buenos resultados, pues muchas peticiones se resuelven en el nivel superior y la información nunca llega a la SDRAM. Cuanto peor sea la caché en niveles superiores, se espera que la técnica funcione mejor. Si por el contrario, las cachés de niveles superiores tienen un comportamiento muy bueno, entonces es menos probable que los resultados a este nivel tengan un potencial grande. Sea como sea, el conjunto de la jerarquía de memoria en general asegura un nivel de rendimiento muy bueno.

Conclusiones y trabajo futuro

A lo largo de este Trabajo Fin de Grado hemos realizado un análisis exhaustivo de la localidad temporal, espacial y algorítmica de ciertas aplicaciones, a nivel de memoria principal. Para ello hemos utilizado ciertas aplicaciones de la suite SPEC2006, además de Firefox, Openoffice, gcc y g++. Utilizando una herramienta de Intel, llamada Intel® Pin, podemos monitorizar las peticiones de acceso a memoria que realiza la LLC y aprender de estos datos.

Por un lado, la localidad temporal nos ha servido para agrupar aplicaciones con un comportamiento similar. Hemos utilizado algunas técnicas de clustering para diferenciar entre aquellas que volvían a acceder muy tempranamente a los datos y aquellas que no. Como la información de la que se disponía era muy abundante, la hemos resumido en histogramas de igual longitud, y hemos utilizado la medida de divergencia de Jensen-Shannon para valorar diferencias entre estos histogramas. Una vez realizado el agrupamiento, se ha comprobado que aquellas aplicaciones que acceden muy tempranamente a los datos tienen un comportamiento patológico de la caché de nivel superior, que puede estar producido por un tamaño demasiado pequeño o una mala elección de la asociatividad. Si la memoria de nivel superior ofreciera un mejor rendimiento, no sería necesario acceder a líneas repetidas.

La localidad algorítmica ha sido estudiada en menor profundidad por la dificultad para ser comprendida y analizada, y por la fuerte dependencia de cada aplicación. Hemos examinado el perfil de utilización de memoria, entendido como el número de operaciones solicitadas por unidad de tiempo, para cada una de las veinte aplicaciones, y hemos podido comprobar la existencia de ciertas estructuras y patrones regulares en el número de peticiones. En todos los casos, una caché LLC en el nivel superior con una configuración 16/1 ofrece peores resultados que las demás. Hay ciertas aplicaciones para las que aumentar la asociatividad resulta enormemente beneficioso, pues se reduce el número de fallos por conflicto. Otras, por contra, mejoran al aumentar el tamaño, evitando fallos de capacidad. Además, hay algunas aplicaciones que sólo mejoran su rendimiento al aumentar tanto la asociatividad como el tamaño.

El análisis más detenido ha sido el correspondiente a la localidad espacial. En el Capítulo 5 hemos propuesto una arquitectura SDRAM-RRAM para mejorar la velocidad global del sistema, sin renunciar a las ventajas que ofrecen las memorias resistivas, entre las que se encuentra la no volatilidad. Junto a la memoria SDRAM hemos incluido un pequeño controlador, que se encarga de ir analizando las peticiones de memoria que se reciben del nivel superior, y aplica técnicas de reconocimiento de patrones para realizar una prebúsqueda. Aunque hemos examinado algunos

modelos basados en la distribución binomial, finalmente se ha optado por un modelo oculto de Markov para distinguir grupos en los accesos. Con esta división en grupos, es mucho más sencillo desarrollar técnicas de prebúsqueda realmente efectivas, incluso con una simple regresión lineal. Aunque han sido probados otros modelos para esta segunda fase de predicción (redes neuronales recurrentes, modelos para series temporales), ninguno de ellos obtiene resultados tan buenos que compensen un aumento de complejidad del controlador SDRAM.

Finalmente se ha discutido la posible aplicación de la agrupación realizada en el Capítulo 4 para futuras optimizaciones de la técnica de prebúsqueda. Como hemos comprobado, la prebúsqueda tiene especial interés en el caso donde la localidad espacial sea mucho más marcada que la localidad temporal. Si esto no ocurre, desperdiciaremos una parte muy grande de la caché SDRAM para un aumento poco significativo de la tasa de éxito. El desperdicio se puede reducir limitando la amplitud del intervalo de predicción, pues así se traerán menos líneas a la SDRAM. Esta optimización se puede implementar de forma dinámica y sin mucho esfuerzo; tan sólo es necesario examinar la localidad temporal de la aplicación y decidir a cuál de los grupos creados con las técnicas de clustering está más cerca.

La arquitectura presentada podría considerarse cuando las memorias resistivas sustituyan a las convencionales memorias DRAM. Dada su alta densidad y no volatilidad, pueden llegar a construirse memorias RRAM de terabytes de capacidad, sustituyendo incluso a los discos duros SSD (*Solid State Driver*) y memorias *Flash*, logrando velocidades de acceso mucho mayores. Las investigaciones para que esto sea posible continúan hoy en día, centradas fundamentalmente en caracterizar los estados intermedios dentro de la conmutación entre el estado de alta y baja resistencia. Lo desarrollado en esta memoria permite no sólo incrementar aún más las prestaciones de un sistema RRAM, sino que también proporciona un marco general para el diseño de mecanismos de prebúsqueda, que no tienen por qué estar atados a una jerarquía SDRAM-RRAM. La única restricción es disponer del tiempo suficiente entre accesos como para realizar el análisis y reconocimiento de patrones.

Como trabajo futuro, se presentan numerosas alternativas. Por un lado, sería interesante aplicar las optimizaciones basadas en la localidad temporal en el mecanismo de prebúsqueda. Además, sería igualmente provechoso repetir las simulaciones para determinar el rendimiento de la prebúsqueda cuando la caché de nivel superior es mejor. En esta memoria sólo se han presentado los resultados cuando la LLC tiene una configuración 16/1, pero igual de interesante es conocer su comportamiento bajo una supuesta LLC 32/8. De esta manera, tendríamos una idea más cercana del impacto de la parte superior de la jerarquía en la técnica desarrollada.

Además, también podría desarrollarse la arquitectura SDRAM-RRAM con mayor detalle. Hasta ahora sólo hemos presentado una SDRAM que actúa sobre una RRAM y que tiene un controlador asociado. No hemos ofrecido detalles, sin embargo, de cómo está compuesto este controlador. Previsiblemente necesitará un pequeño pero rápido procesador para realizar todos estos cálculos, además de utilizar una cierta porción de memoria SDRAM para almacenar la información necesaria. Junto a esto, sería conveniente desarrollar un estudio sobre el coste aproximado de

la implementación de la técnica, para conocer las restricciones de tiempo y espacio que tiene este procesador al realizar los cálculos. También se abre la posibilidad del desarrollo de una simulación más realista, teniendo en cuenta la arquitectura desarrollada y las restricciones temporales, para evaluar el rendimiento completo de una aplicación en términos del tiempo de ejecución.

Por último, sería interesante realizar un estudio comparativo de las diferentes técnicas de prebúsqueda existentes. Hasta ahora, la mayoría de técnicas desarrolladas son mucho más simples porque están destinadas a actuar a nivel de caché de procesador, una memoria mucho más rápida, con menor capacidad de almacenamiento y poco tiempo para realizar un análisis exhaustivo. Estas técnicas son muy diferentes de la presentada aquí, aunque puede resultar interesante comparar algún aspecto concreto entre ellas. Finalmente, cabe la posibilidad de preparar un artículo detallando la técnica y los resultados obtenidos, en revistas tanto de índole estadística como las versadas en la rama de arquitectura de computadoras.

Anexos

Apéndice A

Agrupamiento de la localidad temporal con umbral de olvido para LLC de 32MB y asociatividad 1.

El dendograma sugiere que podemos dividir a las aplicaciones en cinco grupos.

- En el grupo 1 están aquellas que reutilizan posiciones de memoria a muy corto plazo.
- En el grupo 2 están aquellas que reutilizan posiciones a corto plazo, aunque a veces también a largo plazo.
- En el grupo 3 están las aplicaciones en las que la mayoría de reutilizaciones se producen en los primeros 400.000 accesos intermedios, incluso a veces con tendencia creciente.
- En el grupo 4 están las aplicaciones para las que la distancia temporal está repartida en todo el eje.
- En el grupo 5 está una aplicación con un comportamiento extraño, donde las reutilizaciones se producen de forma tardía.

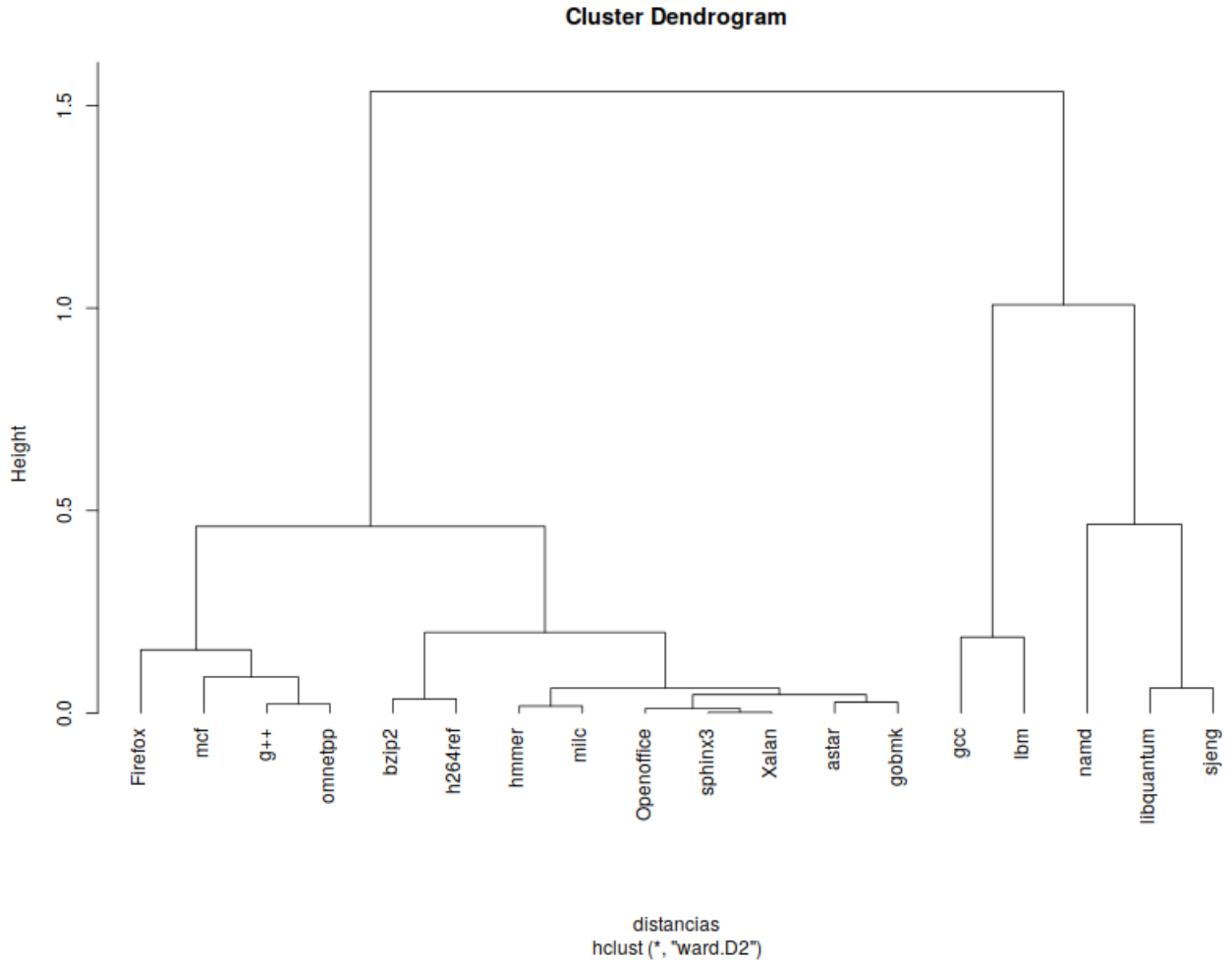


Figura A.1: Dendograma para las aplicaciones con LLC de 32MB y asociatividad de nivel 1, considerando umbral de olvido

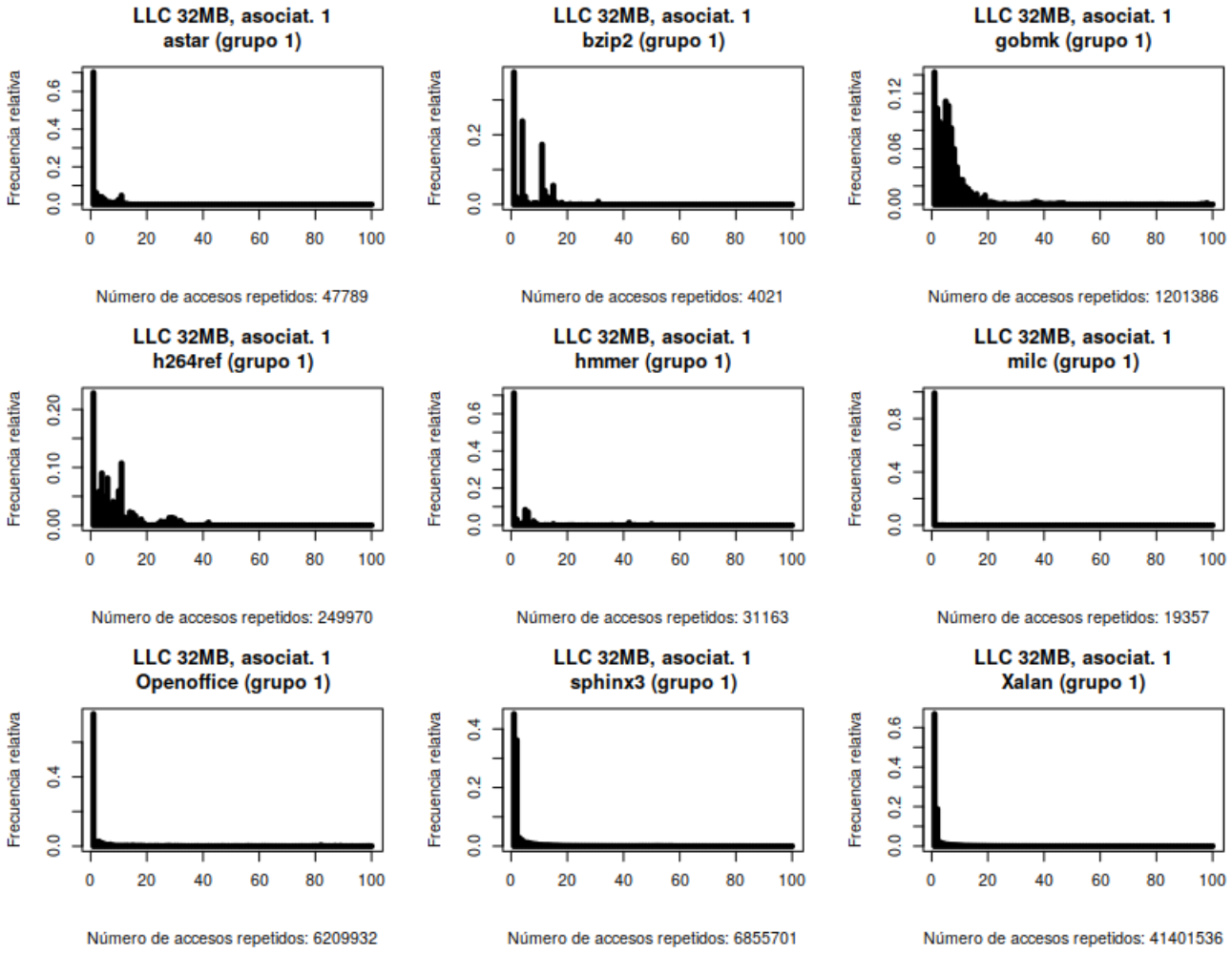


Figura A.2: Aplicaciones del grupo 1 para el clustering cuando la LLC es de 32MB y la asociatividad por conjuntos de nivel 1, considerando un umbral de olvido

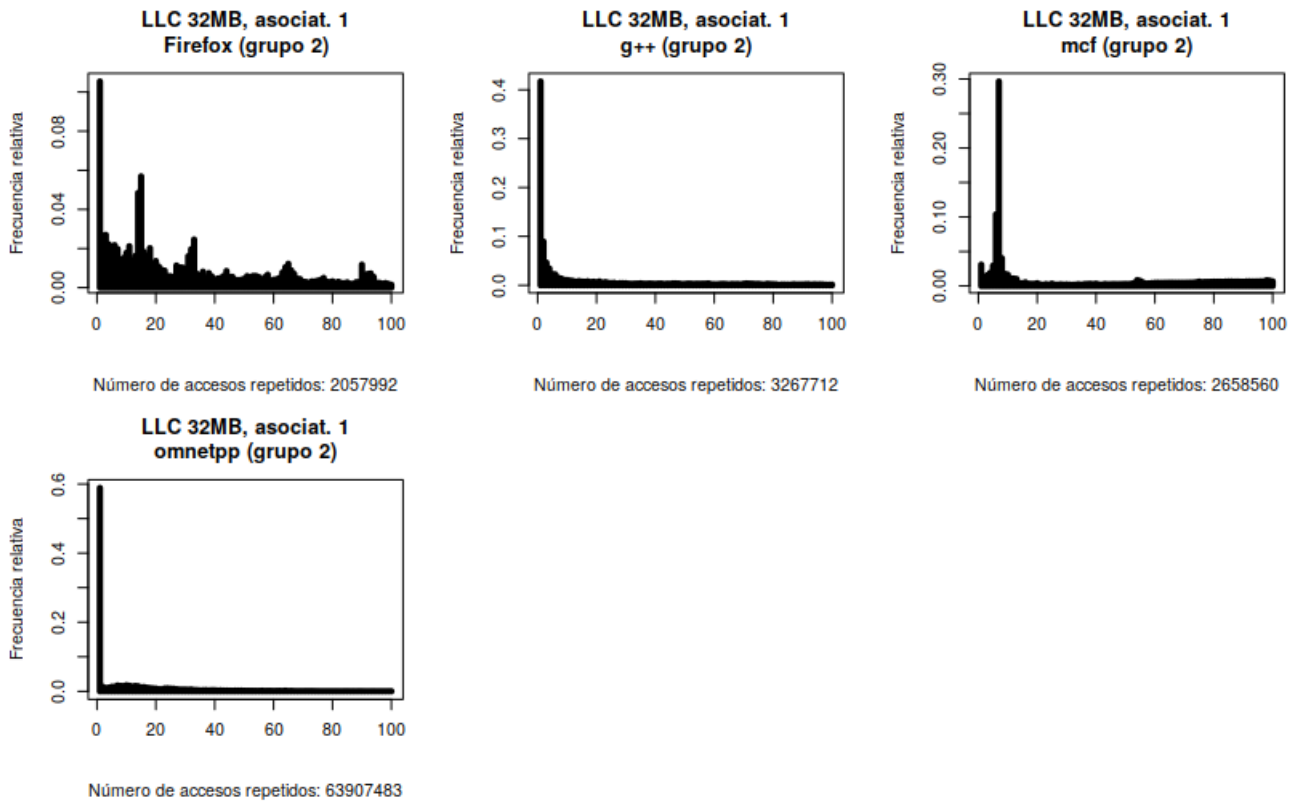


Figura A.3: Aplicaciones del grupo 2 para el clustering cuando la LLC es de 32MB y la asociatividad por conjuntos de nivel 1, considerando un umbral de olvido

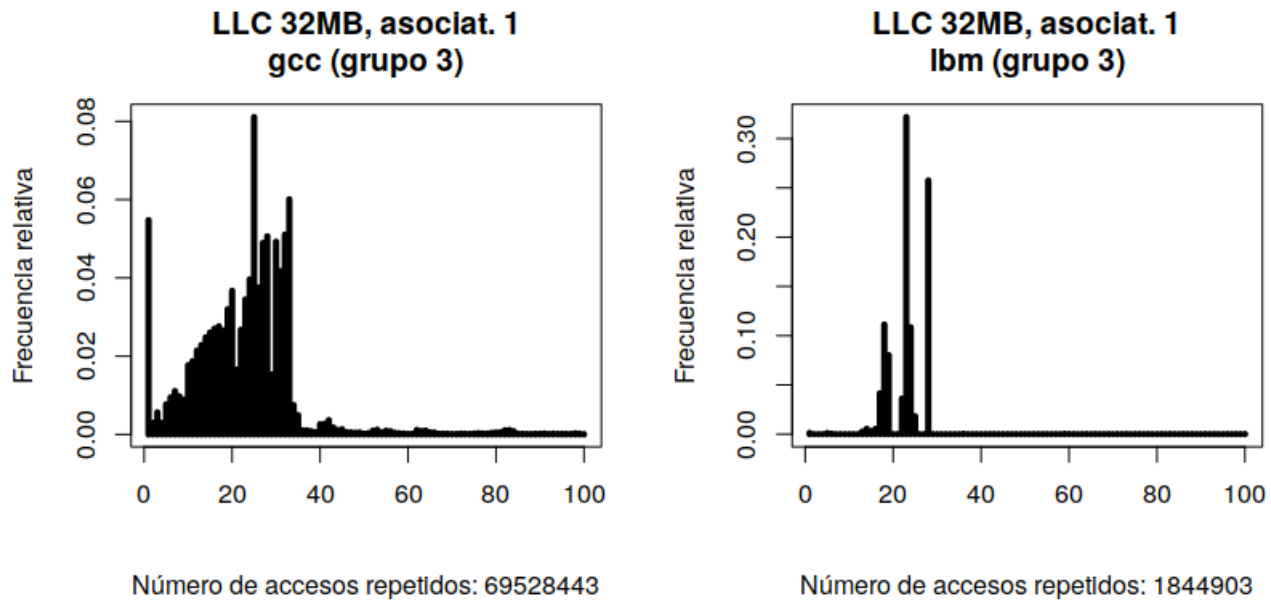


Figura A.4: Aplicaciones del grupo 3 para el clustering cuando la LLC es de 32MB y la asociatividad por conjuntos de nivel 1, considerando un umbral de olvido

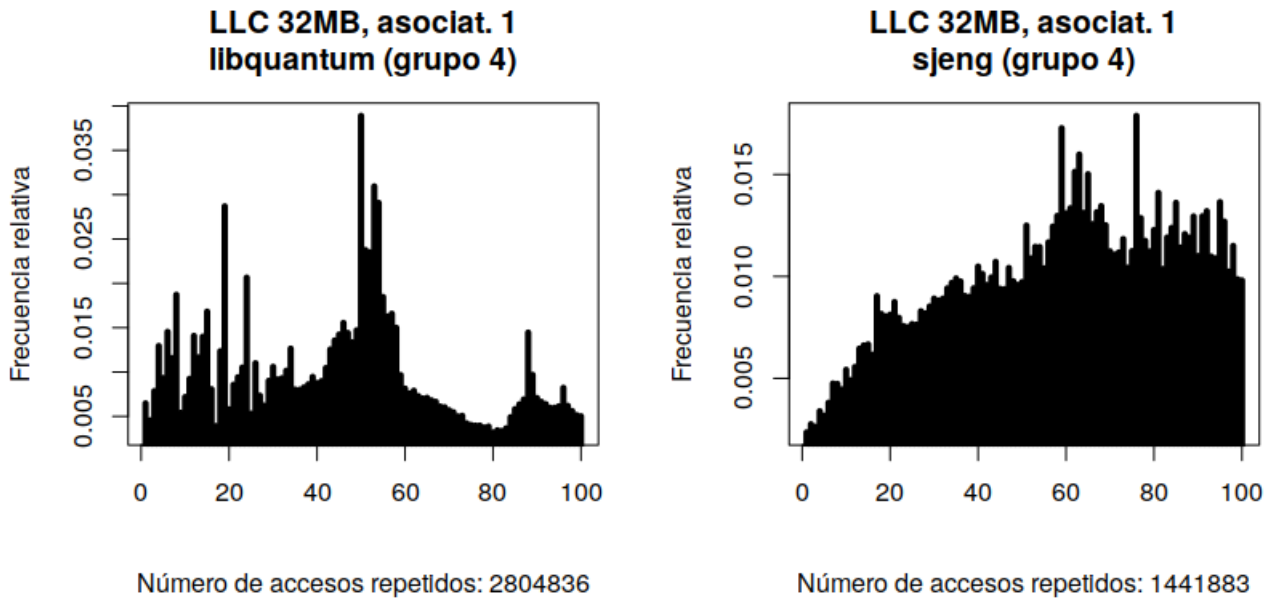


Figura A.5: Aplicaciones del grupo 4 para el clustering cuando la LLC es de 32MB y la asociatividad por conjuntos de nivel 1, considerando un umbral de olvido

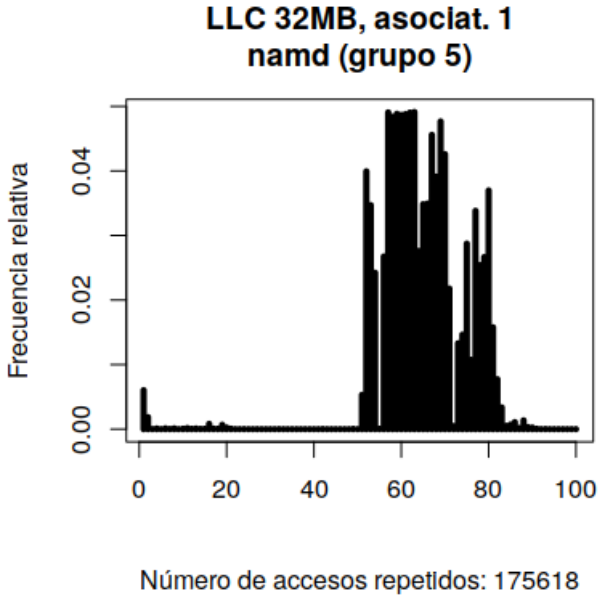


Figura A.6: Aplicaciones del grupo 5 para el clustering cuando la LLC es de 32MB y la asociatividad por conjuntos de nivel 5, considerando un umbral de olvido

Apéndice B

Agrupamiento de la localidad temporal con umbral de olvido para LLC de 16MB y asociatividad 8.

El dendograma no aporta demasiada información. Una posibilidad es clasificar en cuatro grupos.

- El grupo 1 contiene aplicaciones con una distancia temporal temprana, aunque no en exceso.
- El grupo 2 está formado por aplicaciones con una distancia temporal entre accesos muy repartida, aunque con formas variadas.
- El grupo 3 tiene aplicaciones con una distancia temporal con gran peso al principio, aunque no muy diferente del grupo 2.
- El grupo 4 contiene una aplicación con un comportamiento anómalo.

Los grupos 2 y 3 podrían fusionarse, obteniendo una clasificación similar. Nótese que con esta asociatividad ya no se tienen los casos patológicos en los que se reutilizan muy rápido posiciones ya accedidas.

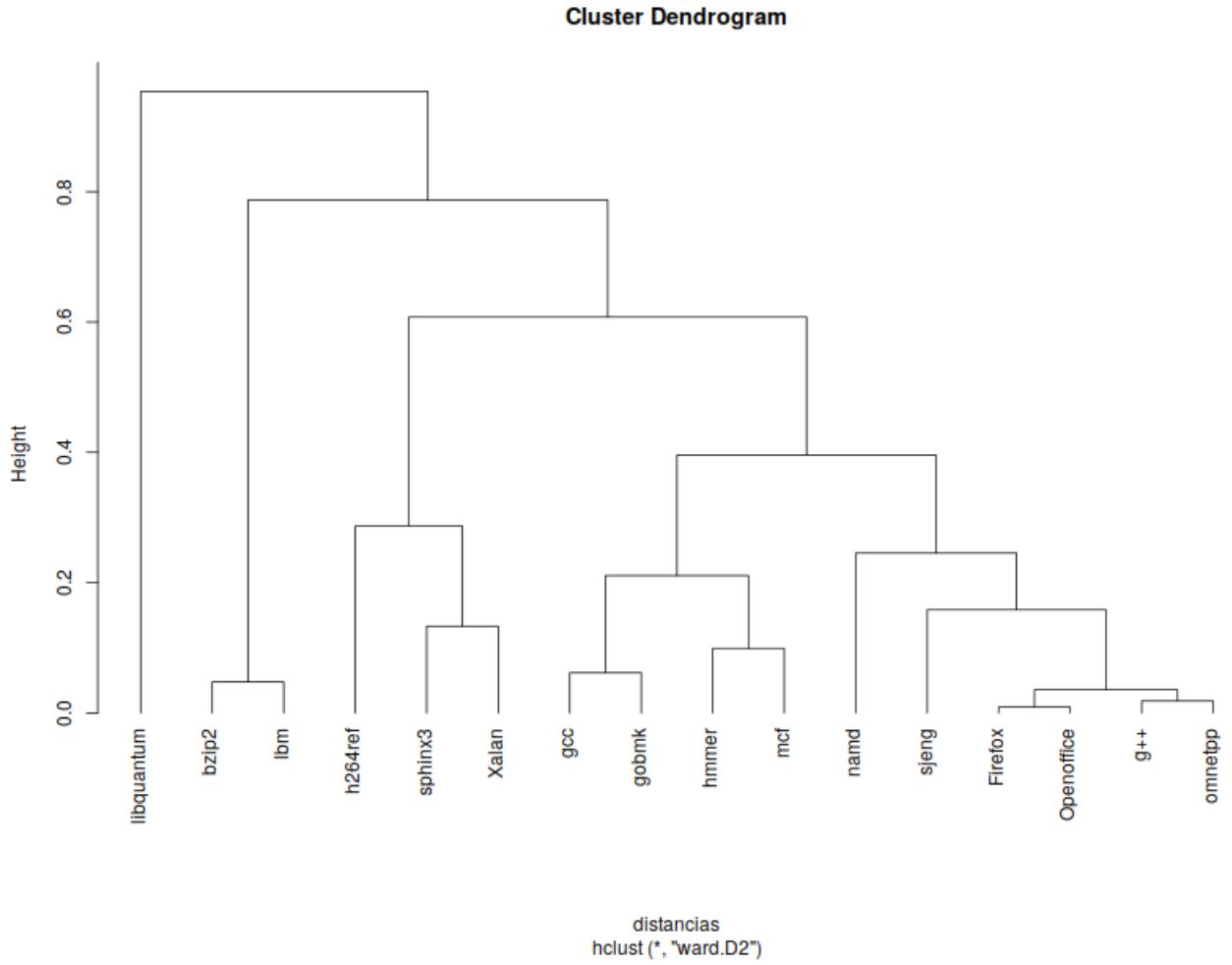


Figura B.1: Dendograma para las aplicaciones con LLC de 16MB y asociatividad de nivel 8, considerando umbral de olvido

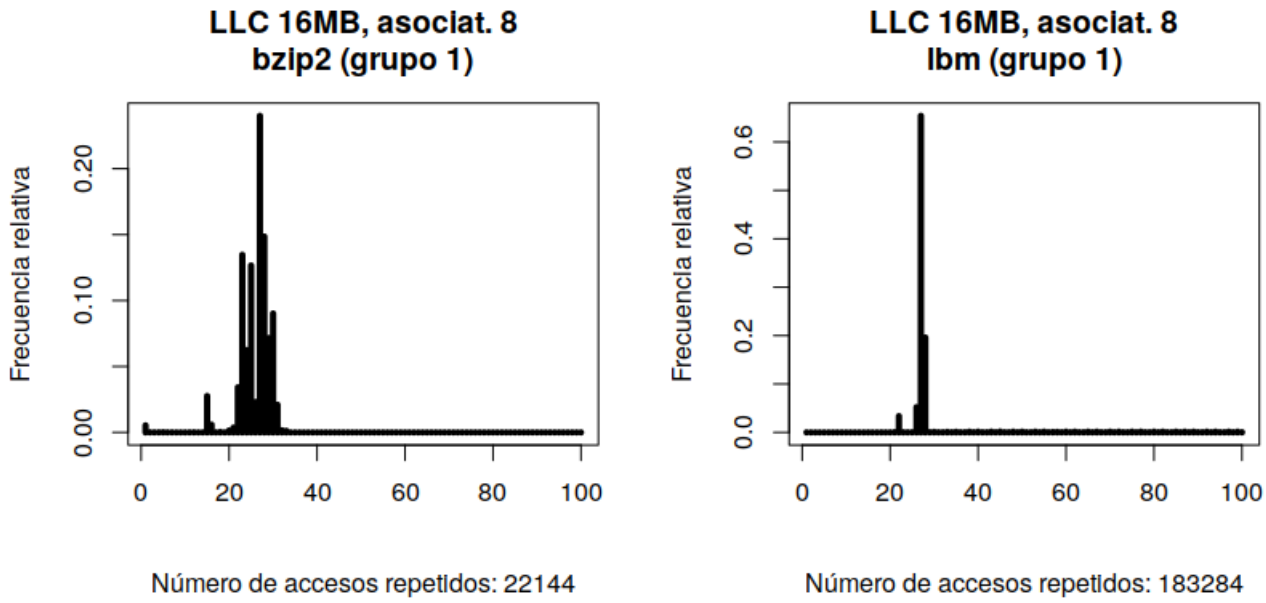


Figura B.2: Aplicaciones del grupo 1 para el clustering cuando la LLC es de 16MB y la asociatividad por conjuntos de nivel 8, considerando un umbral de olvido

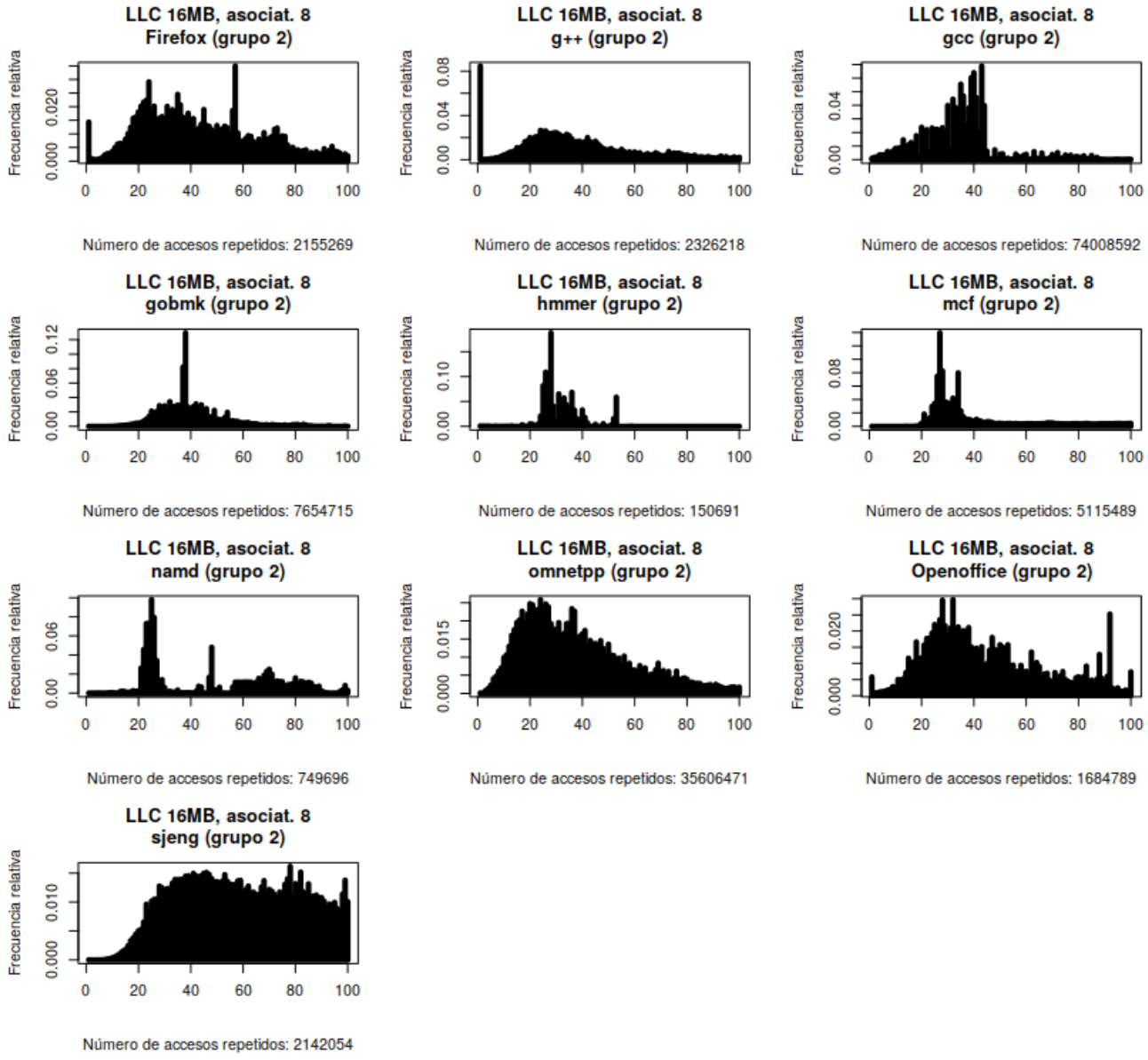


Figura B.3: Aplicaciones del grupo 2 para el clustering cuando la LLC es de 16MB y la asociatividad por conjuntos de nivel 8, considerando un umbral de olvido

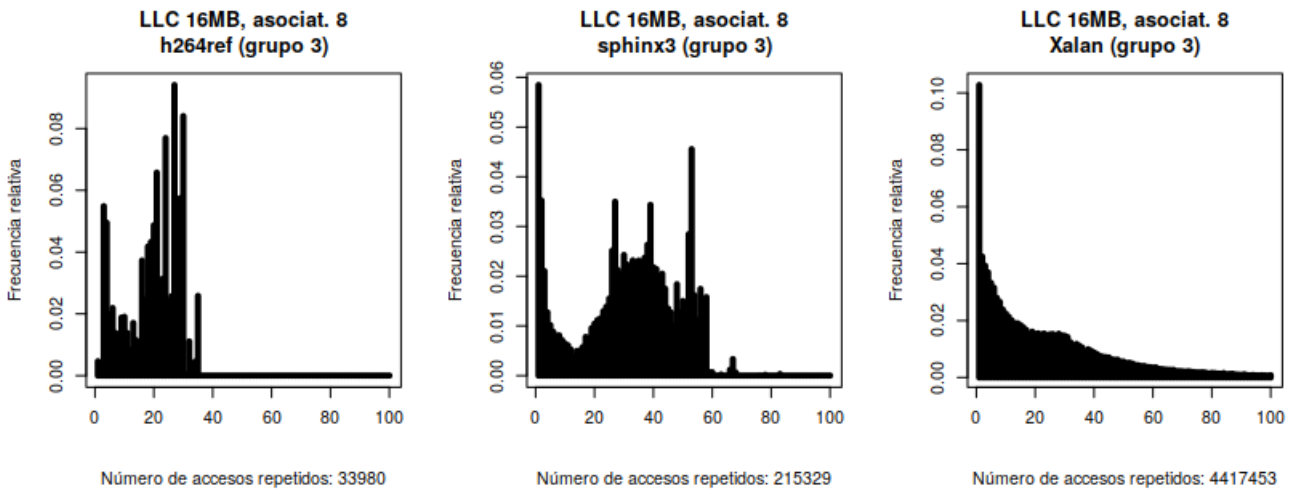


Figura B.4: Aplicaciones del grupo 3 para el clustering cuando la LLC es de 16MB y la asociatividad por conjuntos de nivel 8, considerando un umbral de olvido

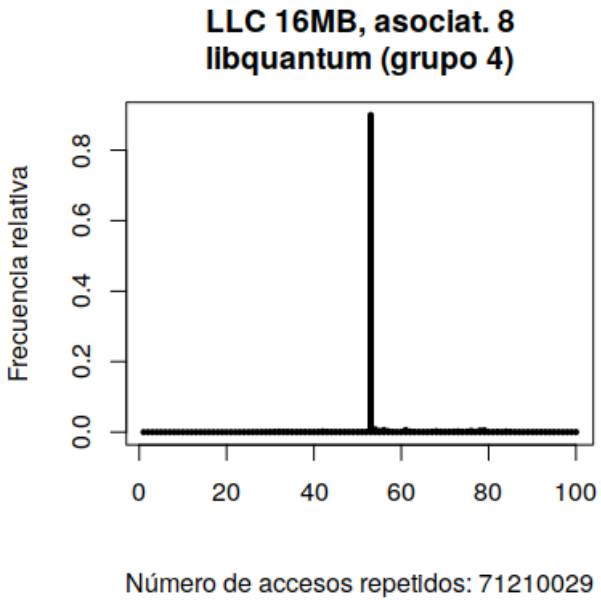


Figura B.5: Aplicaciones del grupo 4 para el clustering cuando la LLC es de 16MB y la asociatividad por conjuntos de nivel 8, considerando un umbral de olvido

Apéndice C

Agrupamiento de la localidad temporal con umbral de olvido para LLC de 32MB y asociatividad 8.

El dendograma sugiere claramente dos grupos:

- El grupo 1 está formado por aplicaciones con una tendencia decreciente en la localidad temporal.
- El grupo 2 está formado por aplicaciones cuya reutilización de posiciones de memoria se puede producir en cualquier momento.

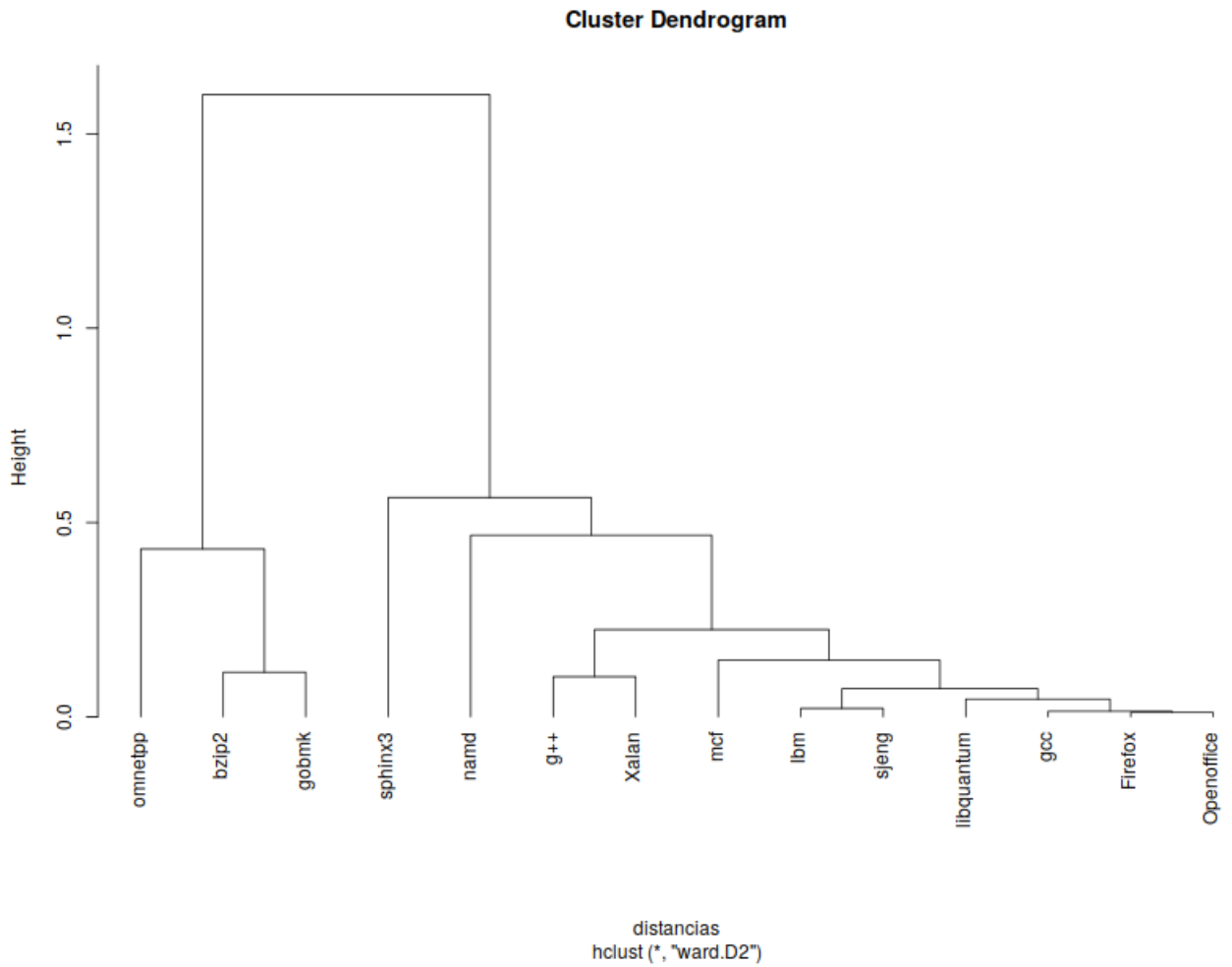


Figura C.1: Dendograma para las aplicaciones con LLC de 32MB y asociatividad de nivel 8, considerando umbral de olvido

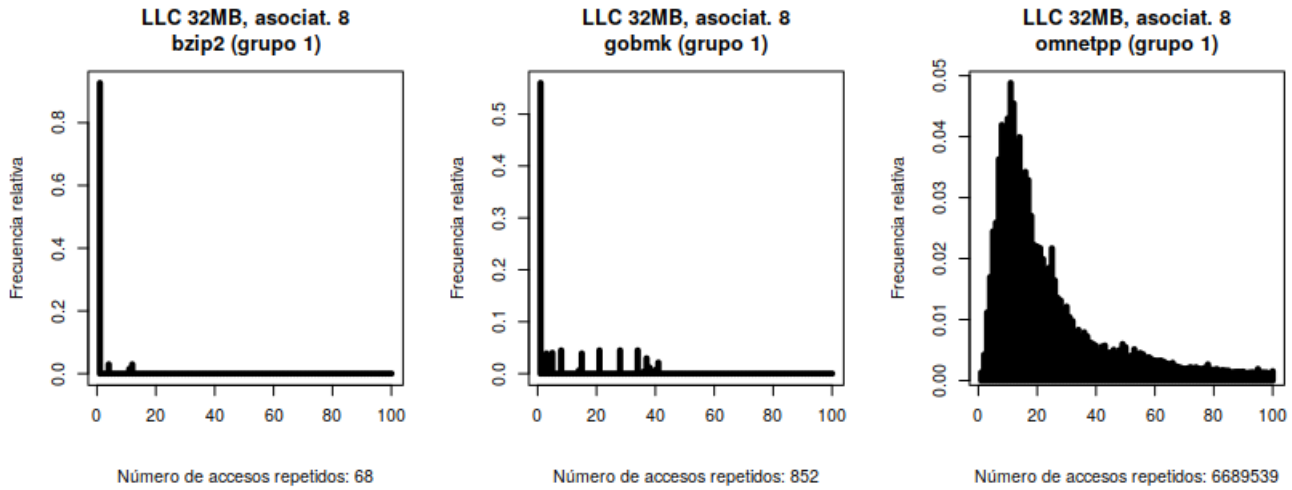


Figura C.2: Aplicaciones del grupo 1 para el clustering cuando la LLC es de 32B y la asociatividad por conjuntos de nivel 8, considerando un umbral de olvido

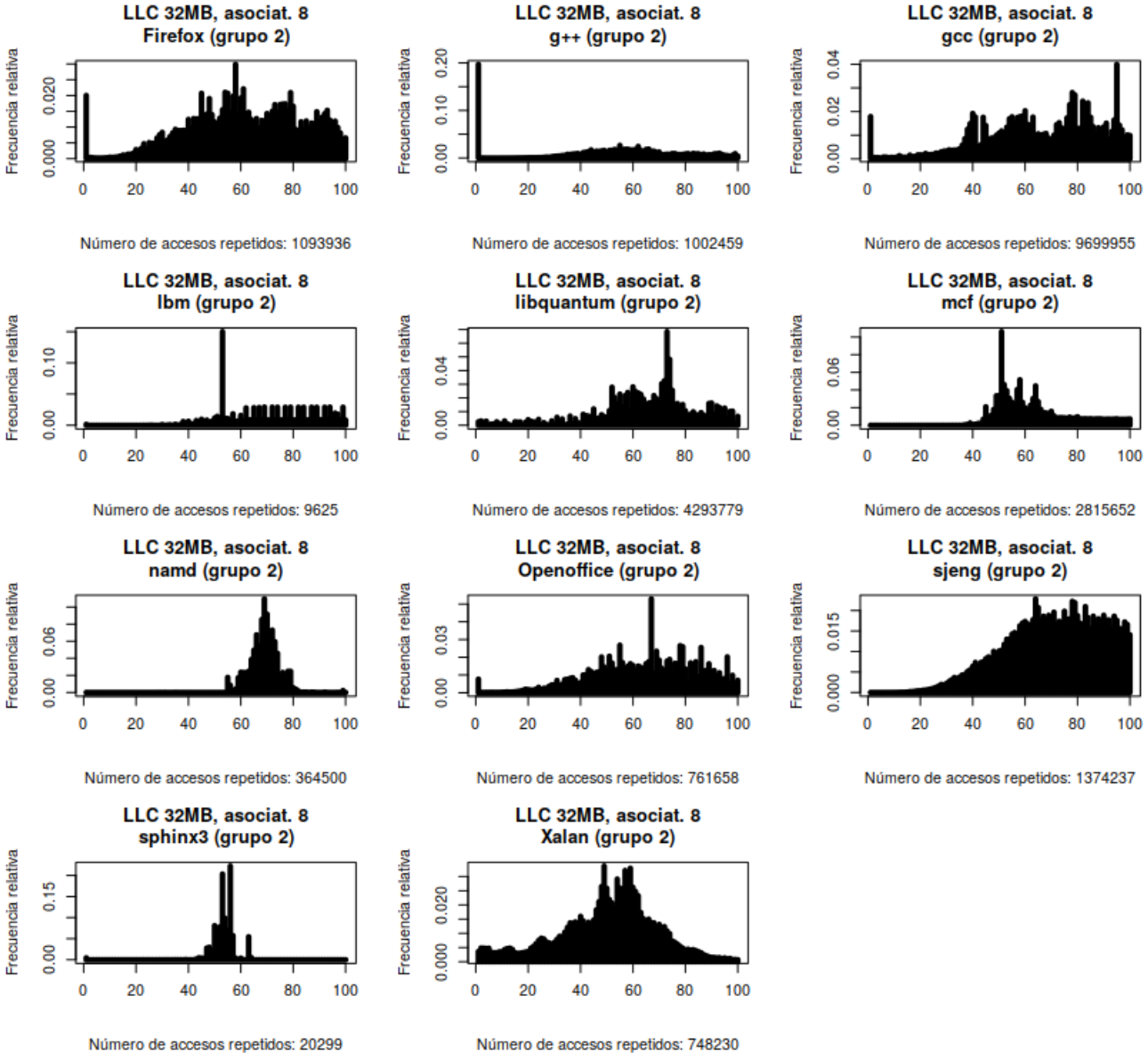


Figura C.3: Aplicaciones del grupo 2 para el clustering cuando la LLC es de 32MB y la asociatividad por conjuntos de nivel 8, considerando un umbral de olvido

Apéndice D

Perfil de memoria de las aplicaciones restantes

En la tabla D.1 se encuentra información sobre el perfil de memoria para cada aplicación y cada configuración de la caché. Entre los datos que se incluyen están:

- El OPKC medio.
- La varianza del OPKC.
- El porcentaje de actividad de la memoria, entendido como el porcentaje de secciones de 1.000 ciclos que han tenido alguna operación de acceso a memoria.
- El porcentaje de secciones que se pueden considerar "outliers". La sección i se considera un outlier si $OPKC_i >= Q_{75} + 3IQR$, donde Q_{75} representa el tercer cuartil de todos los OPKC instantáneos,y IQR es el rango intercuartil.
- La duración de la ráfaga (número de secciones consecutivas con actividad en memoria) más larga.
- La duración de la ráfaga más larga, pero contando las secciones consecutivas de alta densidad. Una sección i se considera de alta densidad si $OPKC_i >= 10 \cdot \overline{OPKC}$.

	Media	**Varianza**	**Porcentaje actividad**	**Porcentaje outliers**	**Rafaga**	**Ráfaga alta densidad**
astar_Ass1Tam16	0.0146	0.2777	0.59 %	0.59 %	641	641
astar_Ass1Tam32	0.0095	0.2629	0.30 %	0.30 %	730	730
astar_Ass8Tam16	0.0060	0.2029	0.19 %	0.19 %	641	641
astar_Ass8Tam32	0.0060	0.2029	0.19 %	0.19 %	641	641
bzip2_Ass1Tam16	0.0554	4.3715	1.05 %	1.05 %	6828	6828

	Media	**Varianza**	**Porcentaje actividad**	**Porcentaje outliers**	**Rafaga**	**Ráfaga alta densidad**
bzip2_Ass1Tam32	0.0192	0.5053	0.67 %	0.67 %	6828	6828
bzip2_Ass8Tam16	0.0216	0.7741	0.66 %	0.66 %	6828	6828
bzip2_Ass8Tam32	0.0170	0.4934	0.65 %	0.65 %	6828	6828
Firefox_Ass1Tam16	1.3732	44.5964	40.13 %	4.26 %	4868	4249
Firefox_Ass1Tam32	1.3010	46.9789	32.42 %	4.54 %	5065	4250
Firefox_Ass8Tam16	1.3715	43.0621	33.33 %	5.16 %	4838	4246
Firefox_Ass8Tam32	1.0607	28.1669	30.07 %	3.91 %	13455	4246
g++_Ass1Tam16	0.9189	25.5365	21.66 %	21.66 %	3241	854
g++_Ass1Tam32	0.6743	20.4941	15.93 %	15.93 %	947	946
g++_Ass8Tam16	0.6178	18.8129	13.91 %	13.91 %	1841	960
g++_Ass8Tam32	0.4766	14.3098	11.45 %	11.45 %	1359	959
gcc_Ass1Tam16	2.8377	174.7519	28.99 %	13.67 %	47246	35007
gcc_Ass1Tam32	2.5306	69.4875	39.92 %	1.16 %	39533	15678
gcc_Ass8Tam16	5.2570	49.5026	73.57 %	0.43 %	152423	4040
gcc_Ass8Tam32	1.4668	112.1290	12.75 %	12.75 %	51186	51186
gobmk_Ass1Tam16	0.3013	18.9515	6.39 %	6.39 %	19293	6301
gobmk_Ass1Tam32	0.0612	2.4781	2.03 %	2.03 %	6301	6301
gobmk_Ass8Tam16	0.3822	27.7546	6.42 %	6.42 %	25479	6472
gobmk_Ass8Tam32	0.0128	0.8287	0.05 %	0.05 %	6301	6301
h264ref_Ass1Tam16	0.1063	2.7107	2.69 %	2.69 %	2422	1844
h264ref_Ass1Tam32	0.0205	0.8434	0.43 %	0.43 %	2422	2422
h264ref_Ass8Tam16	0.0126	0.6552	0.17 %	0.17 %	2422	2422
h264ref_Ass8Tam32	0.0093	0.4057	0.11 %	0.11 %	2422	2422
hmmer_Ass1Tam16	0.1693	0.7525	6.41 %	6.41 %	158	154
hmmer_Ass1Tam32	0.0153	0.0972	0.75 %	0.75 %	158	158
hmmer_Ass8Tam16	0.0280	0.1447	0.91 %	0.91 %	212	212
hmmer_Ass8Tam32	0.0136	0.0481	0.70 %	0.70 %	158	158
lbm_Ass1Tam16	33.6169	1137.3787	99.21 %	17.48 %	6231323	4
lbm_Ass1Tam32	33.6140	1138.6693	99.21 %	15.32 %	6231323	0
lbm_Ass8Tam16	33.3614	1143.5562	98.67 %	17.75 %	411241	5
lbm_Ass8Tam32	33.3349	1142.4888	98.25 %	17.68 %	411241	5
libquantum_Ass1Tam16	25.5031	215.8603	99.78 %	0.01 %	1873700	6
libquantum_Ass1Tam32	2.1363	43.8432	21.07 %	21.07 %	125837	6293
libquantum_Ass8Tam16	24.6097	234.2326	99.15 %	0.04 %	1544176	14
libquantum_Ass8Tam32	3.0035	56.6293	25.44 %	19.99 %	101220	6293
mcf_Ass1Tam16	28.5738	2030.5884	50.28 %	0.89 %	209951	11078

	Media	Varianza	Porcentaje actividad	Porcentaje outliers	Rafaga	Ráfaga alta densidad
mcf_Ass1Tam32	27.7978	2018.8459	50.93 %	0.87 %	209939	13872
mcf_Ass8Tam16	28.5189	2014.2425	50.24 %	0.90 %	209951	10571
mcf_Ass8Tam32	27.7412	1996.3306	49.18 %	0.86 %	209951	13872
milc_Ass1Tam16	3.5124	61.8578	45.57 %	8.40 %	2250	189
milc_Ass1Tam32	0.0056	0.2103	0.11 %	0.11 %	1464	1464
milc_Ass8Tam16	0.0047	0.2071	0.07 %	0.07 %	1464	1464
milc_Ass8Tam32	0.0047	0.2071	0.07 %	0.07 %	1464	1464
namd_Ass1Tam16	0.0836	6.4266	1.27 %	1.27 %	10051	10051
namd_Ass1Tam32	0.0385	3.2750	0.27 %	0.27 %	10000	10000
namd_Ass8Tam16	0.0737	6.6292	0.56 %	0.56 %	10051	10051
namd_Ass8Tam32	0.0484	3.7899	0.30 %	0.30 %	9648	9648
omnetpp_Ass1Tam16	4.4504	49.1534	54.97 %	0.92 %	13605	302
omnetpp_Ass1Tam32	2.2205	25.8514	48.58 %	2.48 %	9568	215
omnetpp_Ass8Tam16	1.5188	27.0391	18.52 %	18.52 %	13455	401
omnetpp_Ass8Tam32	0.4465	11.8314	10.98 %	10.98 %	13316	433
Openoffice_Ass1Tam16	0.4172	6.3128	12.60 %	12.60 %	6068	6060
Openoffice_Ass1Tam32	0.3627	11.4953	11.12 %	11.12 %	3678	3277
Openoffice_Ass8Tam16	0.2377	10.8720	4.11 %	4.11 %	4385	3672
Openoffice_Ass8Tam32	0.1948	9.5531	3.38 %	3.38 %	3276	3276
povray_Ass1Tam16	0.0899	0.9879	1.98 %	1.98 %	758	758
povray_Ass1Tam32	0.0228	0.4340	0.39 %	0.39 %	408	408
povray_Ass8Tam16	0.0228	0.4323	0.39 %	0.39 %	798	798
povray_Ass8Tam32	0.0228	0.4340	0.39 %	0.39 %	434	434
sjeng_Ass1Tam16	2.0058	475.7037	15.55 %	15.55 %	20552	20517
sjeng_Ass1Tam32	1.9358	462.2348	13.07 %	13.07 %	20552	20517
sjeng_Ass8Tam16	2.0091	475.6611	15.33 %	15.33 %	20541	20517
sjeng_Ass8Tam32	1.9425	461.0596	13.01 %	13.01 %	20544	20517
specrand_Ass1Tam16	0.0064	0.2626	0.04 %	0.04 %	150	150
specrand_Ass1Tam32	0.0064	0.2625	0.04 %	0.04 %	150	150
specrand_Ass8Tam16	0.0064	0.2625	0.04 %	0.04 %	150	150
specrand_Ass8Tam32	0.0064	0.2626	0.04 %	0.04 %	150	150
sphinx3_Ass1Tam16	1.1030	6.7257	33.11 %	8.12 %	9591	9581
sphinx3_Ass1Tam32	1.2075	7.1946	35.74 %	8.99 %	9584	6131
sphinx3_Ass8Tam16	0.1599	4.4272	4.14 %	4.14 %	9590	9583
sphinx3_Ass8Tam32	0.1107	3.9992	2.83 %	2.83 %	9591	9583
Xalan_Ass1Tam16	3.1111	29.9119	51.73 %	1.06 %	2414	195

	Media	**Varianza**	**Porcentaje actividad**	**Porcentaje outliers**	**Rafaga**	**Ráfaga alta densidad**
Xalan_Ass1Tam32	1.3512	12.2894	36.34 %	2.11 %	414	264
Xalan_Ass8Tam16	0.2657	5.1009	6.87 %	6.87 %	437	328
Xalan_Ass8Tam32	0.1453	4.1268	2.05 %	2.05 %	414	284

Tabla D.1: Algunos resúmenes sobre el perfil de utilización de memoria de las aplicaciones

Así mismo, los gráficos que muestran los perfiles de memoria se muestran a continuación.

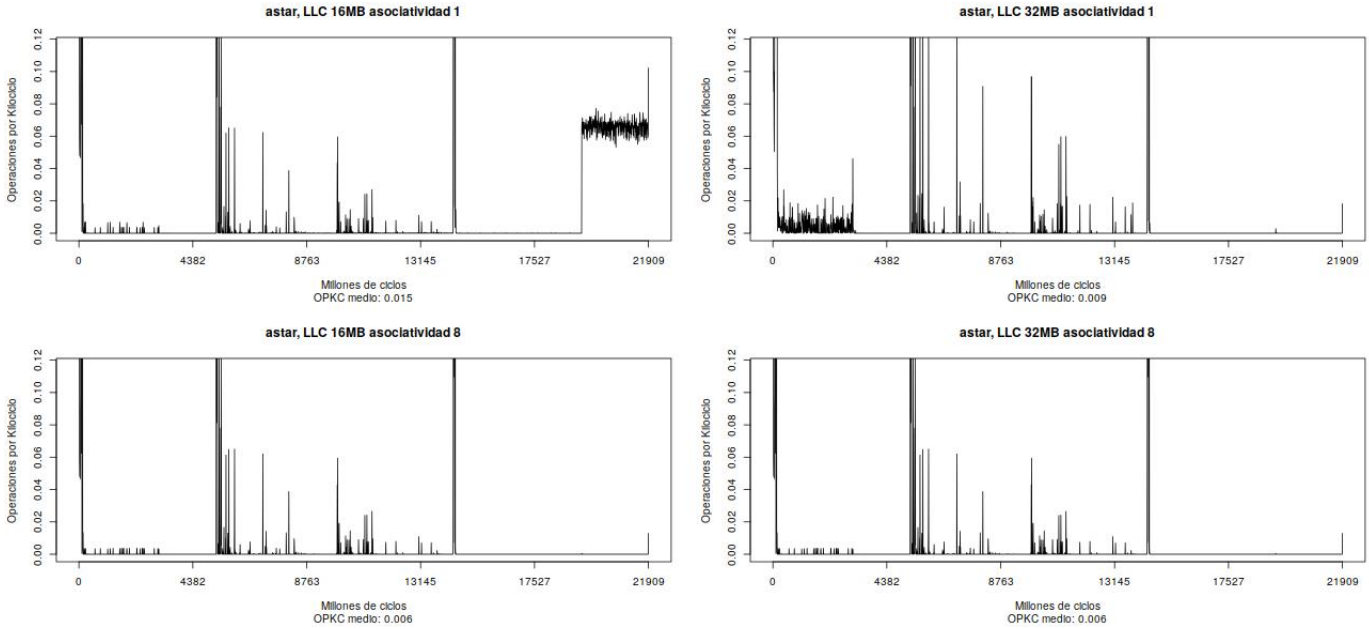


Figura D.1: Perfil de memoria de astar

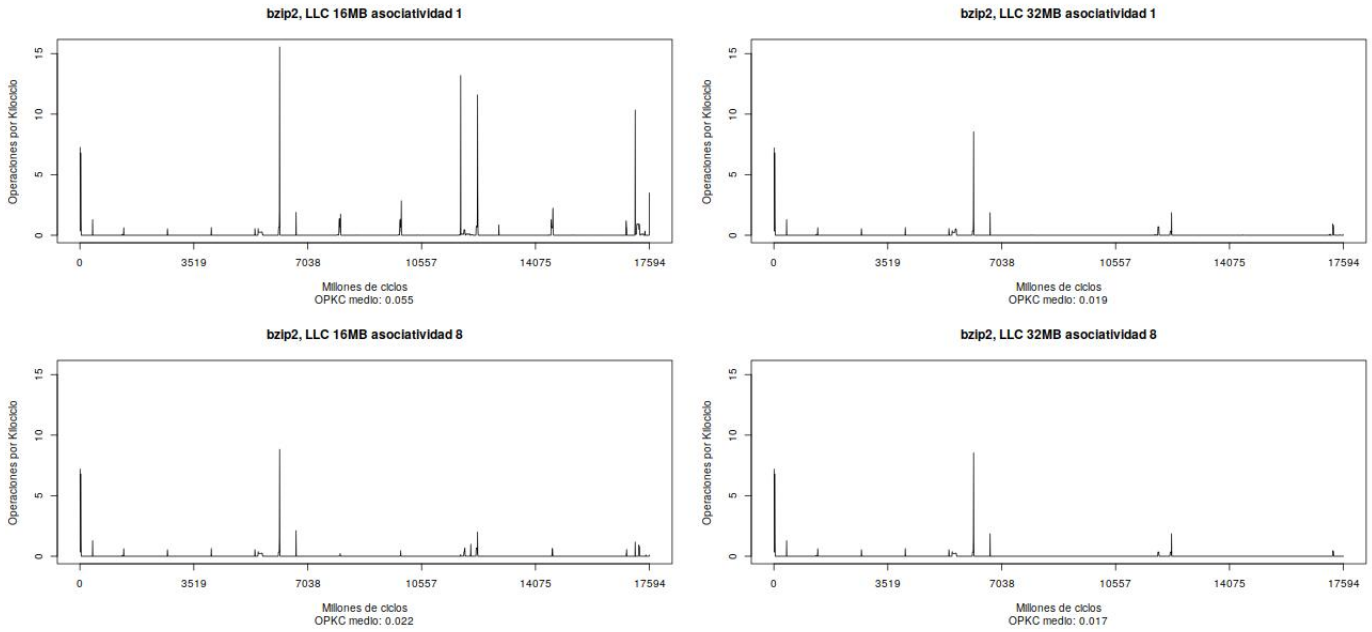


Figura D.2: Perfil de memoria de bzip2

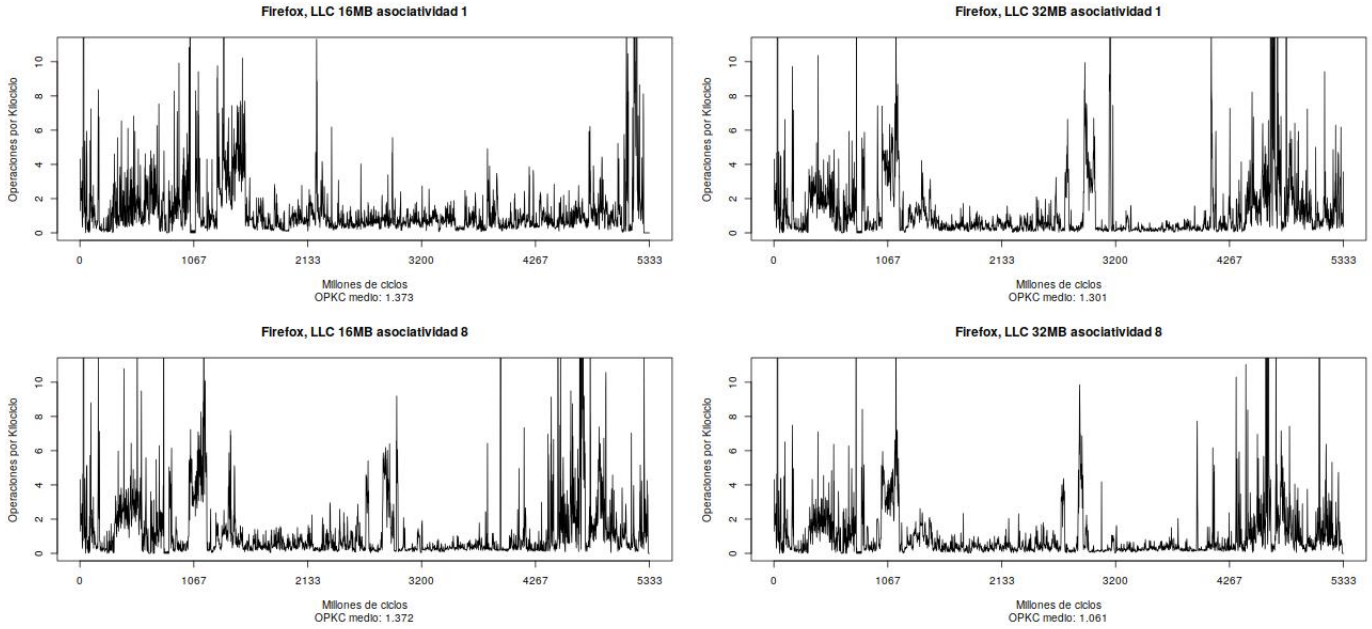


Figura D.3: Perfil de memoria de firefox

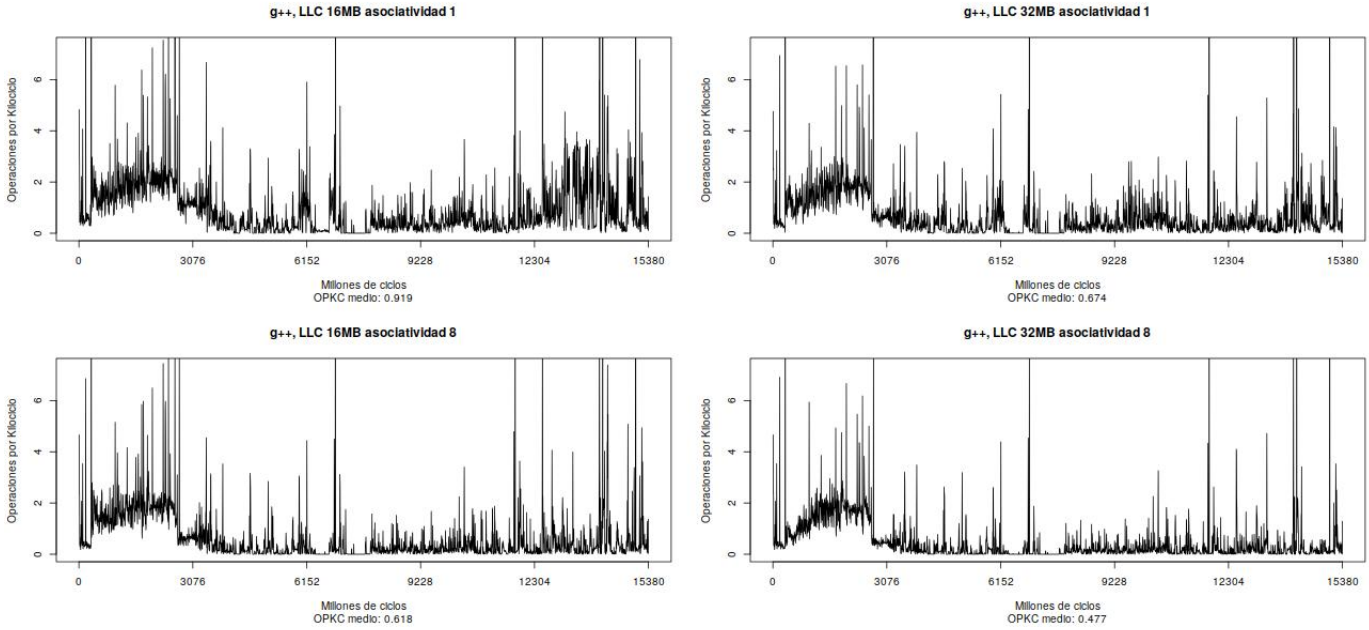


Figura D.4: Perfil de memoria de g++

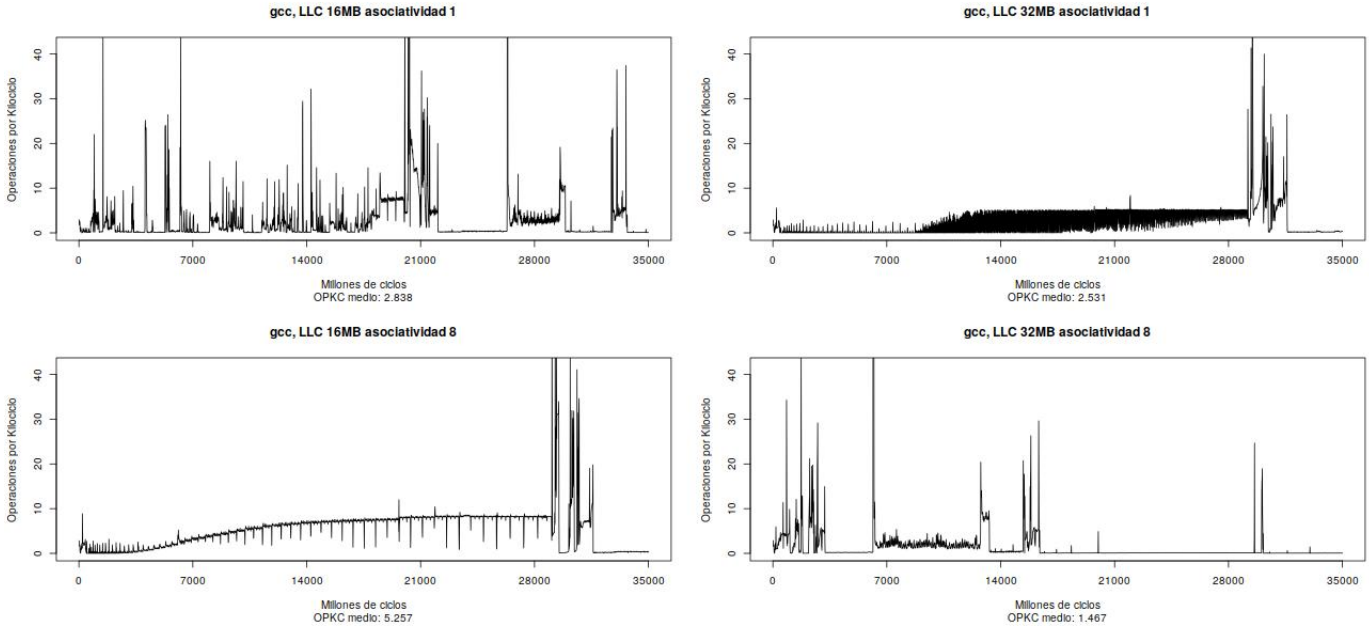


Figura D.5: Perfil de memoria de gcc

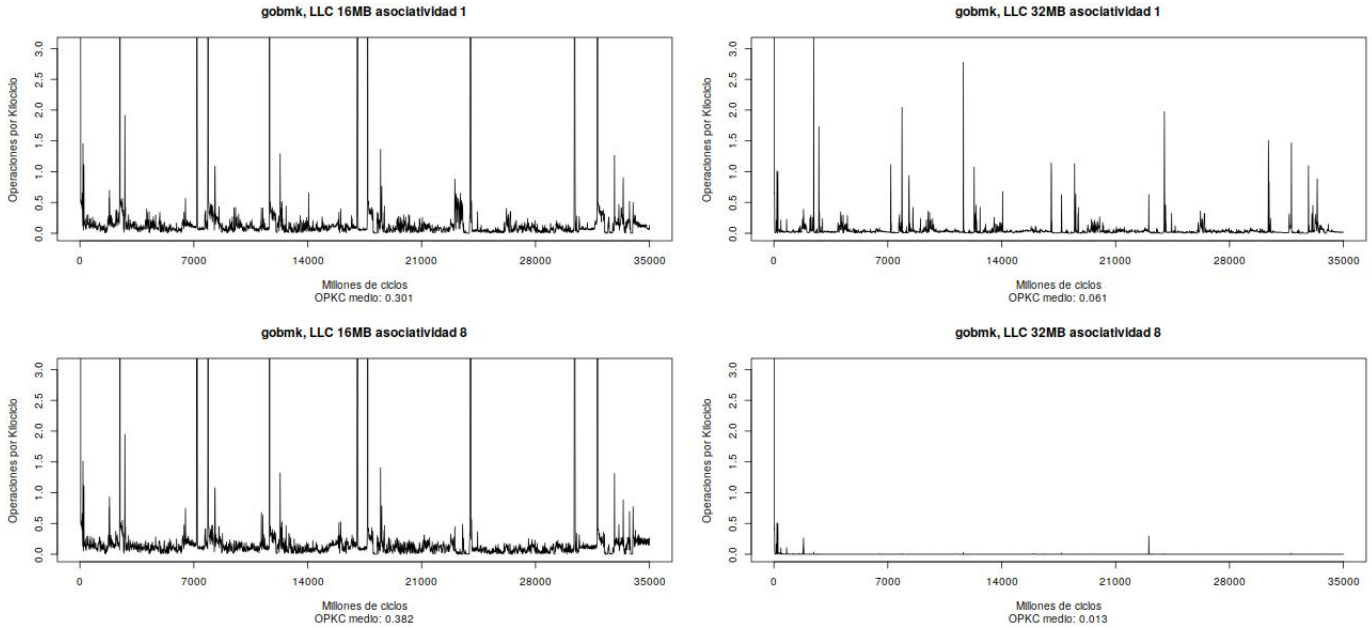


Figura D.6: Perfil de memoria de gobmk

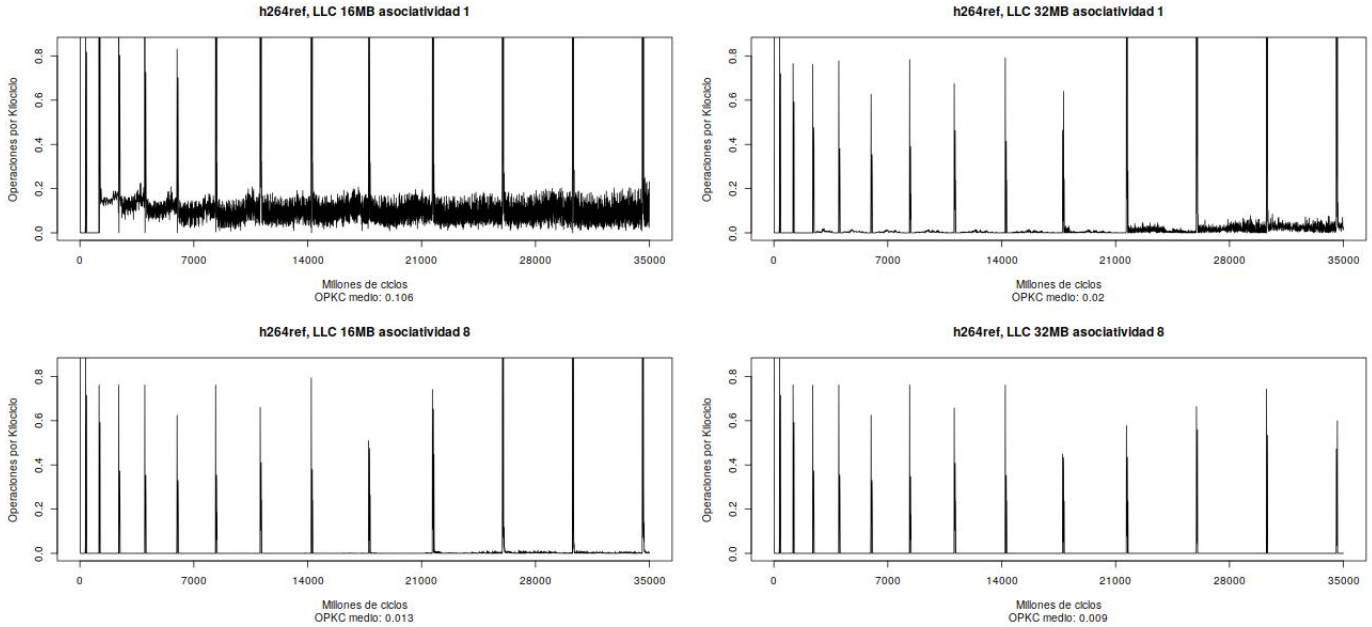


Figura D.7: Perfil de memoria de h264ref

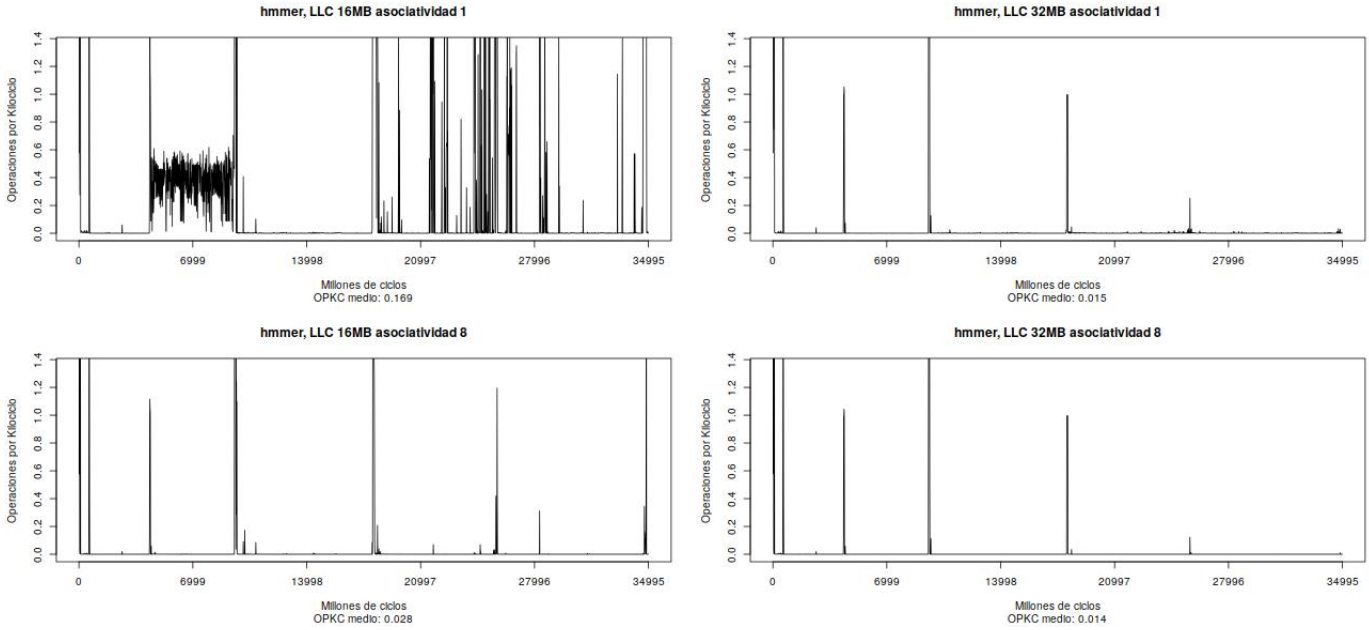


Figura D.8: Perfil de memoria de hmmer

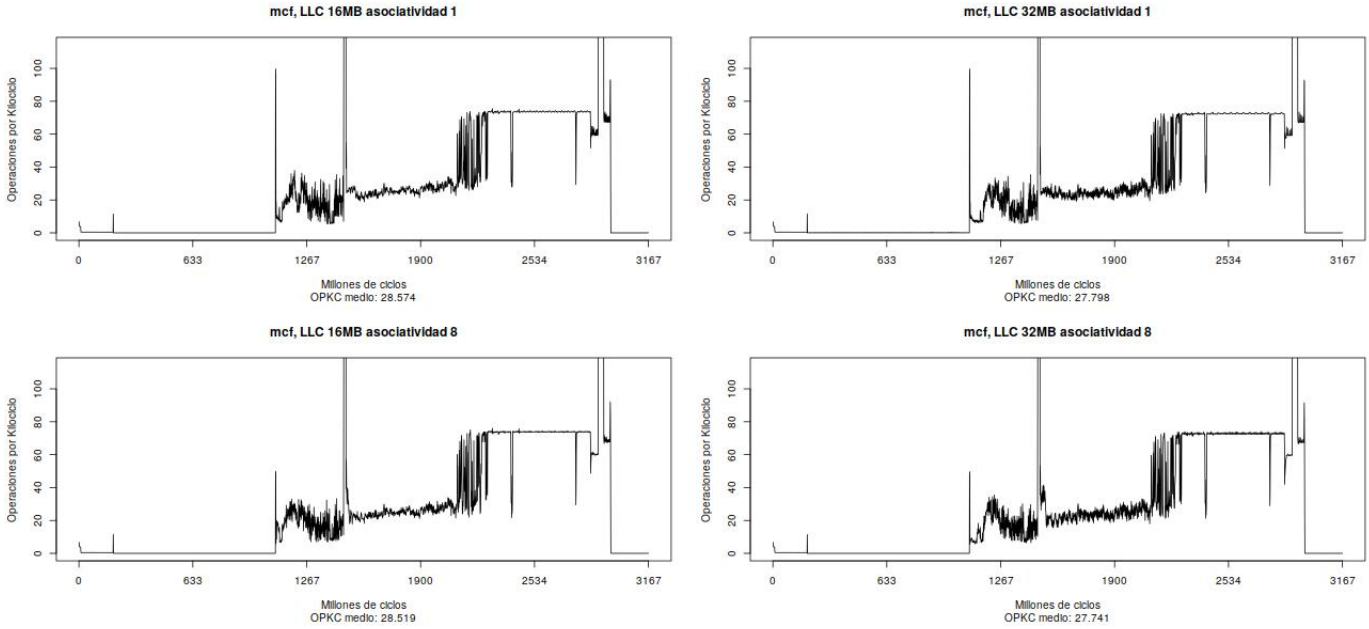


Figura D.9: Perfil de memoria de mcf

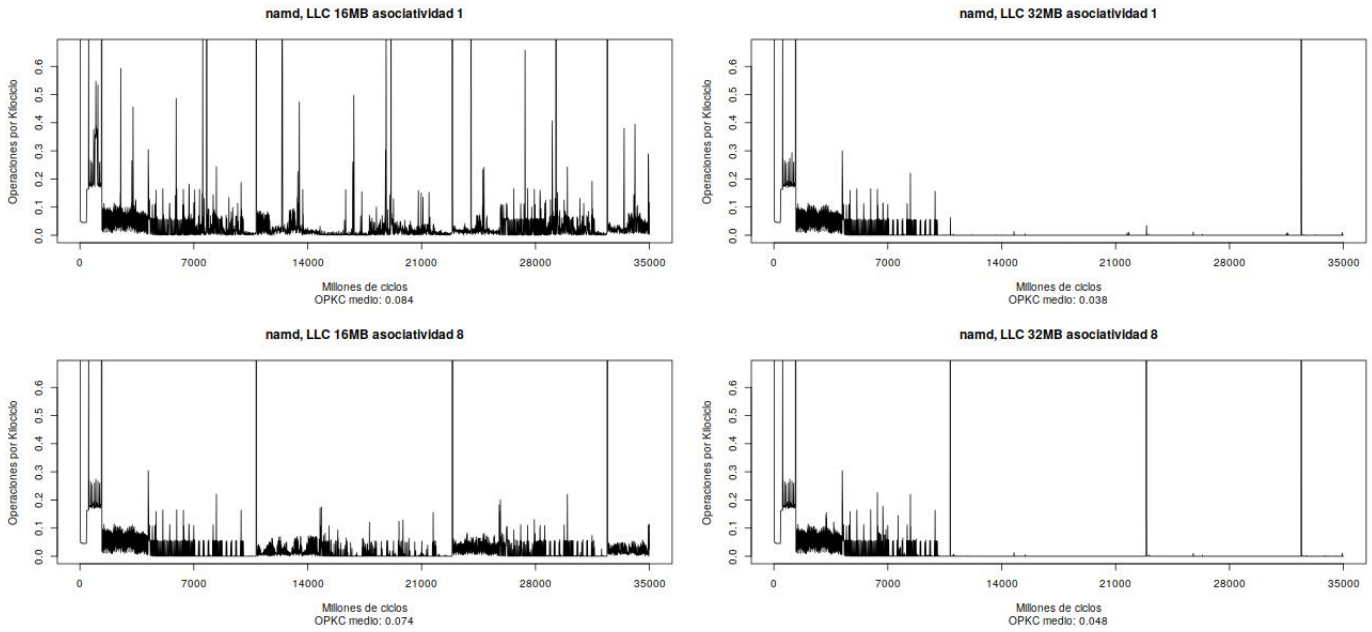


Figura D.10: Perfil de memoria de namd

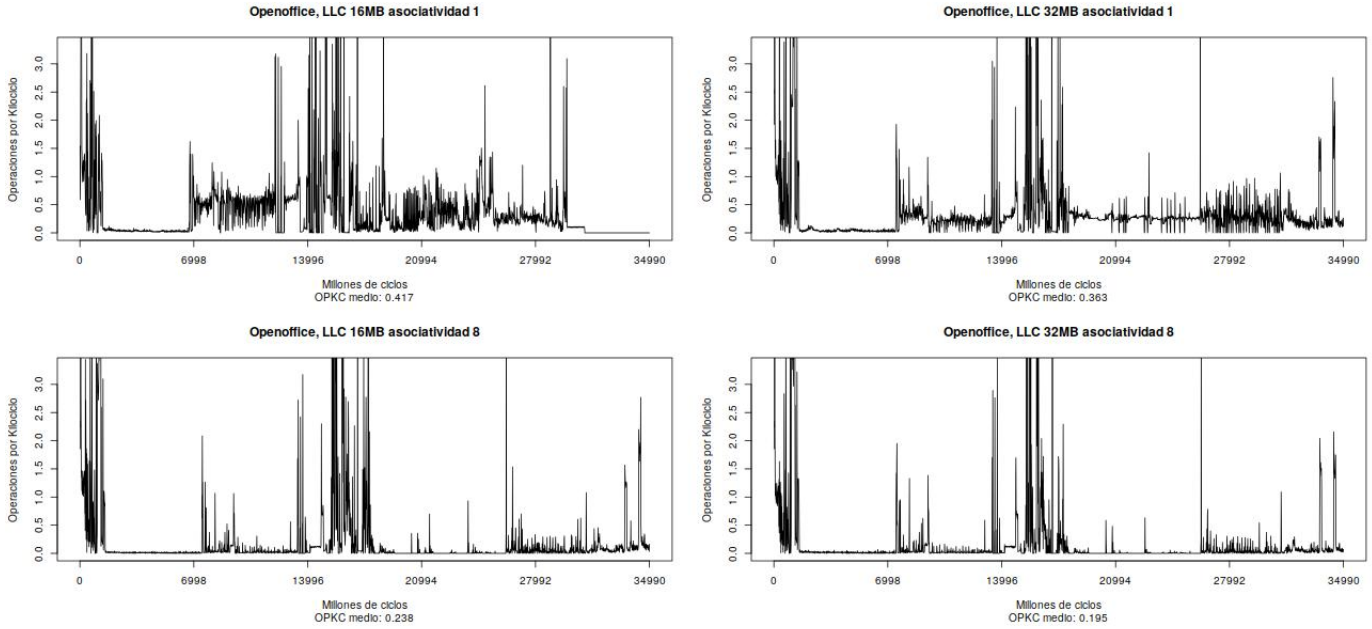


Figura D.11: Perfil de memoria de Openoffice

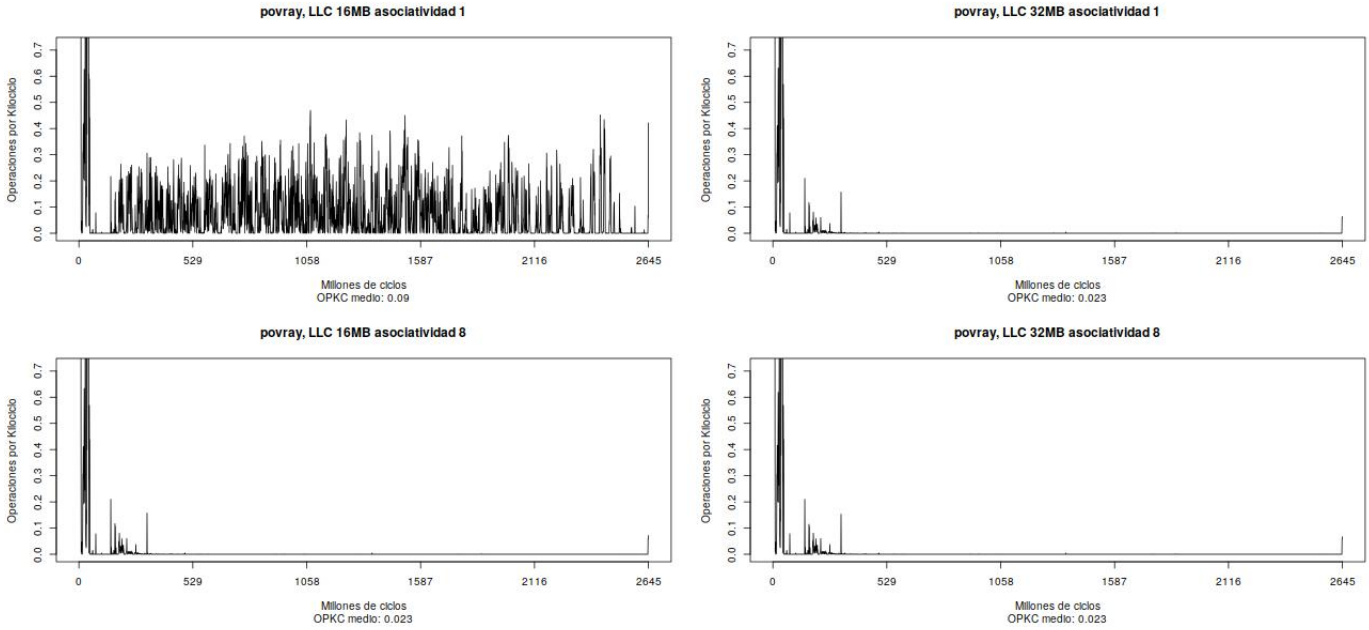


Figura D.12: Perfil de memoria de povray

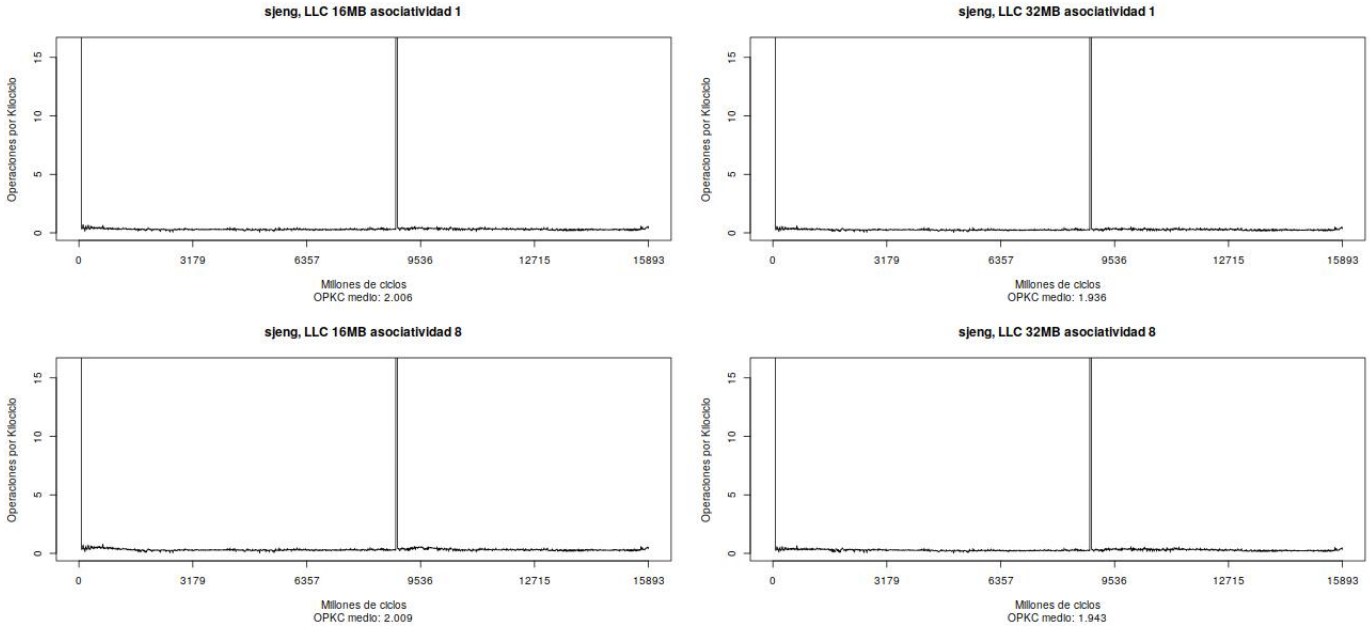


Figura D.13: Perfil de memoria de sjeng

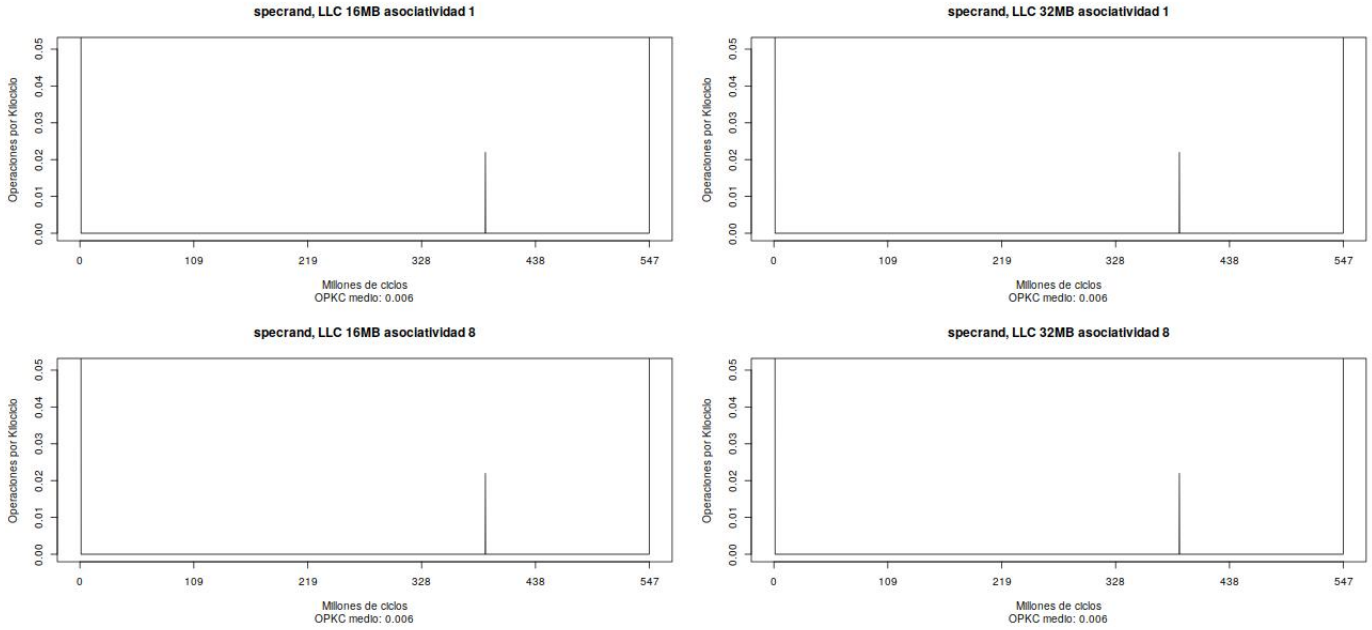


Figura D.14: Perfil de memoria de specrand

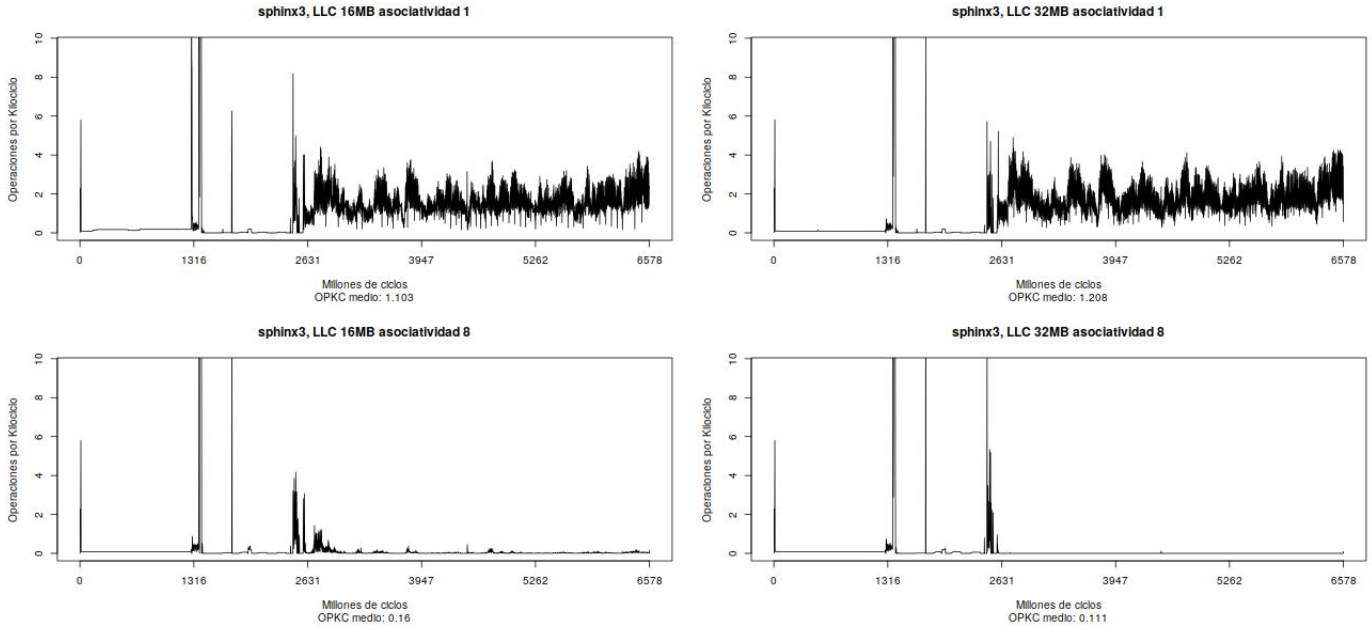


Figura D.15: Perfil de memoria de sphinx3

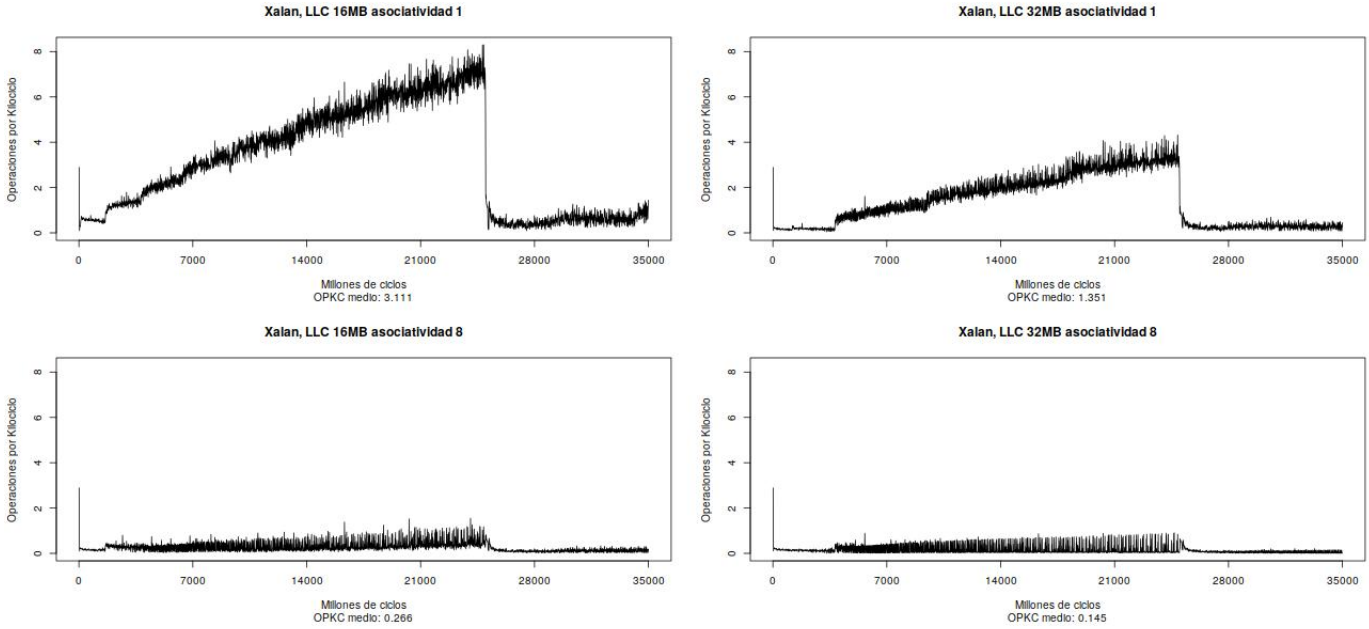


Figura D.16: Perfil de memoria de Xalan

Bibliografía

[1] H. Castán, S. Dueñas, A. Sardiña, H. García, T. Arroval, A. Tamm, T. Jõgiaas, K. Kukli and J. Aarik. "RRAM memories with ALD high-k dielectrics: electrical characterization and analytical modelling". En Thin Film Processes - Artifacts on Surface Phenomena and Technological Facets. Ed. Intech (2017). ISBN: 978-953-51-3067-3. pp:165 - 178 (Chapter 9).

[2] Salvador Dueñas, Helena Castán, Kaupo Kukli, Mats Mikkor, Kristjan Kalam, Tonis Arroval and Aile Tamm. "Memory Maps: Reading RRAM Devices withour Power Consumption". En ECS Transactions. The Electrochemical Society 2018. Vol. 85, issue 8, pp. 201 - 205

[3] H. Castán, S. Dueñas, H. García, O. G. Ossorio, L. A. Domínguez, B. Sahelices, E. Miranda, M. B. González and F. Campabadal. "Analysis and control of the intermediate memory states of RRAM devices by means of admittance parameters". En Journal of Applied Physics. Vol. 124, 152101 (2018).

[4] H.García, O.G.Ossorio, S.Dueñas and H.Castán. "Controlling the intermediate conductance states in RRAM device for synaptic applicatiobs". En Microelectronic Engineering (July 2019). Vol. 215, pp: 110984. (PREPRINT)

[5] Luis Manuel Ramos Martínez, José Luis Briz Velasco y Pablo Ibáñez Martín. "Alternativas de Diseño en Sistemas de Prebúsqueda Hardware de Datos". (2009). Memoria de Tesis Doctoral. Universidad de Zaragoza.

[6] Jacob, Bruce, Ng, Spencer & Wang, David (2007). 'Memory Systems: Cache, DRAM, Disk". Morgan Kaufmann Publishers Inc.

[7] Marco Antonio Castro Churampi. (s.f.). "Memoria caché". Recuperado el 22 de enero de 2019 de https://www.monografias.com/trabajos37/memoria-cache/memoria-cache.shtml

[8] "Nehalem (microarchitecture)". (s.f.). En Wikipedia. Recuperado el 23 de enero de 2019 de https://en.wikipedia.org/wiki/Nehalem_(microarchitecture)

[9] Robert G. Plantz. (s.f.). "Introduction to Computer Organization with x86-64 Assembly Language & GNU/Linux". Recuperado el 24 de enero de 2019 de https://bob.cs.sonoma.edu/IntroCompOrg-x64/book.html#bookch16.html

[10] Ericka Drane. (s.f.). "DRAM: Dynamic RAM". Recuperado el 24 de enero de 2019 de https://slideplayer.com/slide/4139713/

[11] http://www.m5sim.org/Coherence-Protocol-Independent_Memory_Components

[12] "DDR SDRAM". (s.f.). En Wikipedia. Recuperado el 25 de enero de 2019 de https://es.wikipedia.org/wiki/DDR_SDRAM.

[13] Mario D. Marino. (2015). "DRAM background Fully-Buffered DIMM Memory Architectures: Understanding Mechanisms, Overheads and Scaling". Recuperado de https://slideplayer.com/slide/1537064/

[14] Heechul Yun. (2013). "MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multicore Platforms". Recuperado el 1 de febrero de https://www.slideshare.net/saiparan/mem-guard-rtas13web-25212473.

[15] "Cache Replacement Policies". (s.f.). En Wikipedia. Recuperado el 14 de febrero de 2019 de https://en.wikipedia.org/wiki/Cache_replacement_policies.

[16] Osnat Levi (Intel). (2018). "Pin - A Dynamic Binary Instrumentation Tool". Recuperado el 6 de febrero de 2019 de https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[17] L. Billard & Jaejik Kim. (2017). "Hierarchical clustering for histogram data". WIREs Comput Stat 2017, 9:e1405. doi: 10.1002/wics.1405.

[18] "Jensen-Shannon divergence". (s.f.). En Wikipedia. Recuperado el 28 de febrero de https://en.wikipedia.org/wiki/Jensen–Shannon_divergence.

[19] Hong-Tai Chou and David J. DeWitt. 1985. "An Evaluation of Buffer Management Strategies for Relational Database Systems". VLDB.

[20] "Merge algorithms". (s.f.). En Wikipedia. Recuperado el 10 de mayo de 2019 de https://en.wikipedia.org/wiki/Merge_algorithm.

[21] "Modelo oculto de Markov". (s.f.). En Wikipedia. Recuperado el 10 de mayo de 2019 de https://es.wikipedia.org/wiki/Modelo_oculto_de_Márkov.

[22] "One- and two-tailed tests". (s.f.). En Wikipedia. Recuperado el 15 de mayo de 2019 de https://en.wikipedia.org/wiki/One-_and_two-tailed_tests

[23] Montgomery, D.C., Peck, E.A. y Vining, G.G. (2006). "Introduction to Linear Regression Analysis (4ª Ed.). Wiley.

[24] Rousseeuw, P. J. and Leroy, A. M. (1987). "Robust Regression and Outlier Detection". J. Wiley, New York.

[25] R Core Team (2018). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.