Departamento de Informática de Sistemas y Computadores

Universitat Politècnica de València

Arquitectura de memoria para explotar la localidad y el paralelismo a nivel de banco

Trabajo Fin de Master

Máster en Ingeniería de Computadores

Autor

José Duro Gómez

Directores

Prof. María Engracia Gómez Requena

Prof. Julio Sahuquillo Borrás

4 de septiembre de 2015

Agradecimientos

En primer lugar me gustaría agradecer a mis tutores María Engracia y Julio por darme la oportunidad de realizar este trabajo y por enseñarme y ayudarme en lo que necesitaba. A Salvador Petit junto a mis tutores, en esas largas reuniones discutiendo sobre los primeros desastrosos resultados, buscando el componente que retrasaba todo el sistema. Tengo que agradecer a Vicent Selfa por su paciencia en esas horas que de su trabajo que ha perdido explicándome esas "cosas triviales" necesarias para avanzar. También agradecer a los compañeros del laboratorio que me acogiesen como uno más tan rápidamente, creando el buen ambiente y ese ánimo que ayuda a trabajar mas cómodamente. Por último, agradecer a mis padres su ayuda para poder estar aquí, y como no, a mi pareja por soportar mis largos monólogos y cabreos cuando no todo iba bien y por apoyarme y animarme a seguir.

Resumen

La memoria principal es un importante cuello de botella en las prestaciones de los chips multiprocesador actuales. Actualmente, los bancos de DRAM guardan la última fila accedida de cada banco en un registro interno llamado row buffer (RB), el cuál permite accesos posteriores más rápidos a la fila que almacenan. Esta arquitectura se diseñó originalmente para procesadores de un hilo de ejecución, y consigue mejorar notablemente las prestaciones en muchas aplicaciones gracias a que permite explotar la localidad espacial de las aplicaciones en ejecución individual.

Sin embargo, en los procesadores multinúcleo, las distintas aplicaciones en ejecución compiten por los bancos de memoria y, por tanto, por el RB de cada banco. Nuestros experimentos demuestran que estas interferencias afectan a la localidad del RB, lo que reduce la tasa de acierto en éstos y puede penalizar en gran medida a su tiempo de ejecución. Estos resultados manifiestan que es necesario un rediseño de la arquitectura con el fin de aumentar el ancho de banda y reducir la latencia de acceso a memoria en los procesadores multinúcleo donde múltiples aplicaciones se ejecutan concurrentemente. Inicialmente se estudia cómo el uso de varios row buffers por banco permite mejorar las prestaciones en la mayoría de aplicaciones incluso cuando se ejecutan en solitario.

En este trabajo se propone utilizar un conjunto de row buffers por banco para aumentar las prestaciones de los procesadores multinúcleo. Para ello, se plantea una implementación de los row buffers mediante una tabla hardware compartida por las distintas aplicaciones. En vez de realizar una asignación estática, en este trabajo se propone una asignación dinámica en tiempo de ejecución de row buffers a las aplicaciones, en función de las necesidades de las mismas. La propuesta, llamada Dynamic (DYN), mejora el rendimiento de los distintos esquemas de mapeo de memoria. Los resultados obtenidos para un sistema con 4 núcleos muestran que, en promedio, DYN mejora el IPC en los dos esquemas de mapeo de memoria utilizados un 20 % y un 30 %. Además reduce el consumo de energía, en promedio, un 30 %.

Palabras clave: organización DRAM, row buffer, localidad de página, paralelismo a nivel de banco

Abstract

In recent years, the main memory bottleneck has aggravated with the adoption of chip multiprocessors. The increasing number of applications accessing simultaneously the memory has changed its access pattern. Nevertheless current DRAM banks only latch the last accessed row in an internal buffer, namely row buffer (RB), which allows much faster subsequent accesses to the latched row. This architecture was originally designed for single-thread processors and pursues to take advantage of the spatial locality that individual applications exhibit.

However, in multicore processors, the different running applications compete for memory banks and for their RBs. Our experiments demonstrate that interference will affect the RBs locality, which can reduce the hit rate and may penalize largely its runtime. These results shows the need of redesigning the architecture in order to increase bandwidth and reduce latency memory access where multiple applications run concurrently in a multicore processors.

First, this work shows that performance of most individual applications improves with additional row buffers per bank.

This work proposes *row tables* as a pool of row buffers shared among threads to increase the multicore processors performance. Instead of performing a static mapping, in this work a dynamic allocation of row buffers proposed, depending on the needs of each thread. This proposal called Dynamic (DYN) improves performance for any memory mapping scheme. Results for a 4-core system show that, on average, DYN improves IPC in memory mapping schemes by 20% and 30%, and saving energy by 30%.

Keywords: DRAM, row buffer, bank locality, bank paralelism

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1 Descripción del problema

Los avances tecnológicos han permitido aumentar la capacidad de memoria en cada nueva generación de memoria DRAM (Dynamic Random Access Memory). Se trata de un avance significativo para hacer frente a la creciente demanda de memoria de las aplicaciones. Sin embargo, el ancho de banda y el tiempo de acceso no han escalado al mismo ritmo, por lo que la memoria principal se ha convertido en el cuello de botella para los multiprocesadores en chip (CMPs) actuales. Para entender el problema, veamos previamente la organización y funcionamiento básico de los módulos actuales.

Las memorias DRAM están organizadas en matrices de bloques dispuestos en filas y columnas. Normalmente, la mayoría de accesos a memoria solicitan un único bloque de caché, pero las memorias DRAM actuales acceden a la matriz de celdas leyendo una fila completa (del orden de varios KB) que incluye un elevado número de bloques de caché. La fila leída se almacena en un registro interno del banco, llamado *row buffer* (RB). Este registro guarda la última fila accedida y su contenido no cambia hasta que no se lee una nueva fila. El bloque solicitado se envía desde el RB al controlador de memoria. Como el row buffer almacena una fila o página de memoria, en caso de que las peticiones siguientes requieran bloques almacenados en la misma fila, los datos se leen directamente del row buffer. De esta forma se evita un nuevo acceso a la matriz de datos de la DRAM, con el consiguiente ahorro en latencia y energía. En los módulos actuales, si el bloque se lee desde la matriz de datos, el tiempo de acceso es aproximadamente tres veces mayor que si se lee desde el row buffer. En consecuencia, accediendo al RB se puede mejorar significativamente las prestaciones de las aplicaciones. Para ello, mucha investigación se ha centrado en el diseño de políticas orientadas a explotar la localidad del row buffer.

Del funcionamiento descrito es fácil deducir que la efectividad del row buffer puede verse reducida en gran medida cuando se ejecutan múltiples programas. En efecto, el cuello de botella que la memoria principal ejerce sobre las prestaciones aumenta con el número de núcleos, que es la tendencia actual en los CMPs. Esto es debido a que las peticiones de acceso a memoria de todos las aplicaciones ejecutándose en los distintos núcleos compiten entre sí por los recursos de memoria, por lo que los conflictos de banco y de bus pueden tener un impacto significativo en el tiempo de acceso a memoria. En consecuencia, es necesario el diseño de nuevas políticas de planificación de peticiones de acceso a memoria en los CMPs.

1.2 Motivación

Para motivar la idea propuesta de ese trabajo en primer lugar se van a explorar los beneficios potenciales que podría aportar.

En los procesadores multinúcleo actuales se suelen ejecutar varias aplicaciones (cargas multiprogramadas) simultáneamente en el sistema. Esto quiere decir que la localidad de accesos consecutivos al RB se ve reducida con respecto a las ejecuciones de las aplicaciones individuales ya que los accesos de distintas aplicaciones se intercalan entre sí en el acceso a un determinado banco.

La figura 1.1 ilustra este problema. En ella se aprecia como los accesos de cuatro aplicaciones se intercalan entre sí en tiempo de ejecución en un procesador con cuatro núcleos. Estos cuatro benchmarks se corresponden con la mezcla 7 (tabla 5.2) presentada en la sección 5.5. En la figura cada punto muestra un acceso al número de página indicado en el eje Y. Las accesos de distintas aplicaciones se muestran en diferentes colores. Del funcionamiento descrito en el apartado anterior, se deduce que cada vez que se realiza un acceso a una página distinta, cambia el contenido del row buffer. Así, si posteriormente se necesita acceder a la página anterior se deberá volver a leer la página de nuevo desde la matriz de datos. En la figura se aprecia que los accesos de una aplicación se intercalan con los accesos de las demás, cambiando el contenido del row buffer y, en consecuencia, reduciendo la localidad del row buffer. Ésta es la situación típica en los módulos actuales y los resultados se han obtenido mediante una simulación detallada de un módulo de memoria comercial.

Una posible solución para aumentar la localidad sería disponer de varios row buffers. En efecto, si se utilizan varios RBs, la localidad espacial de una aplicación dada podría explotarse mejor ya que varias páginas de distintas aplicaciones podrían estar abiertas

(en los distintos RBs) al mismo tiempo y por lo tanto se podrían mejorar la localidad de row buffer de las aplicaciones individuales.

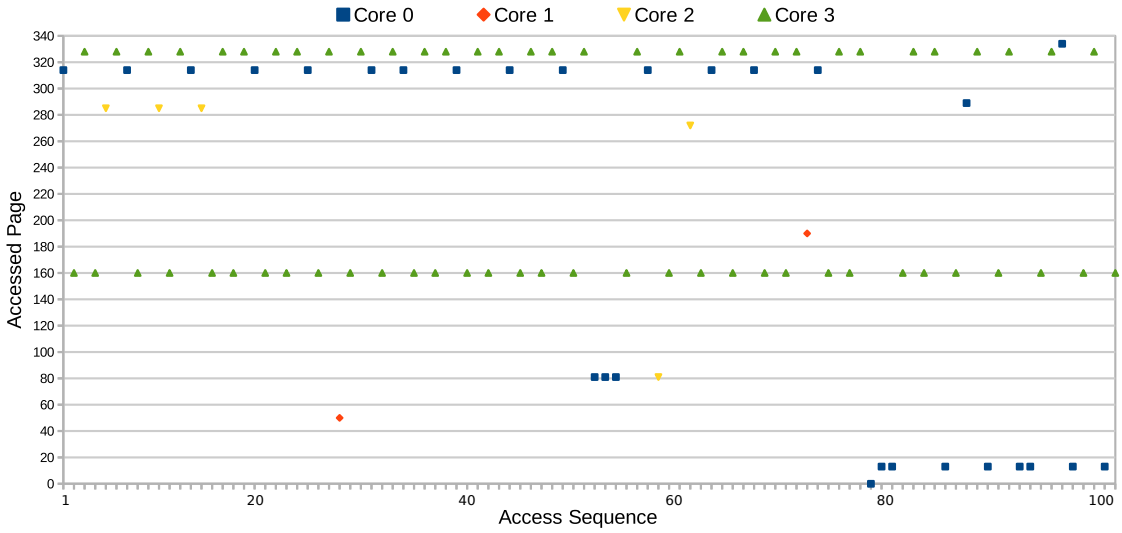


Figura 1.1: Muestra de accesos a memoria durante una ejecución de la mezcla 7.

El razonamiento anterior evidencia la necesidad de disponer de módulos de memoria con varios row buffers, sin embargo cabe realizarse la siguiente pregunta: cuando una aplicación se ejecuta, ya sea de manera individual o junto a otras aplicaciones, ¿podría beneficiarse de disponer solo para ella de varios row buffers? Ante esta cuestión, realizamos experimentos concluyendo que, efectivamente, el hecho de disponer de varios row buffers ayuda a mejorar las prestaciones de algunas aplicaciones en ejecución individual. El experimento persigue verificar si una aplicación vuelve a acceder a una página de memoria en un espacio de tiempo relativamente corto una vez el row buffer se ha cargado con el contenido de otra página. La figura 1.2 muestra los accesos que llegan a memoria para la aplicación `leslie3d` en un periodo de 100 ciclos de ejecución. Como se aprecia, a la memoria llega una ráfaga de 64 accesos, desde el acceso 20 hasta el 84, en los cuales se accede a las páginas 1929 y 1963 de manera intercalada. Esto significa que si sólo se dispusiese de un row buffer, y asumiendo una política que no se puede acceder más de 4 veces seguidas a la misma página, qué es una política implementada en muchos módulos actuales para evitar inanición en las políticas de planificación de accesos a memoria del controlador, el contenido del row buffer se iría cargando un total de 16 veces con las páginas 1929 y 1063 de manera alternativa. Esto significaría un aumento tanto en la latencia como en consumo energético.

Una vez visto que el comportamiento de las aplicaciones individuales justifica la necesidad de disponer de varios row buffers, cabe plantearse en qué medida se beneficiarían las distintas aplicaciones. Para ello, hemos medido el row buffer hit ratio

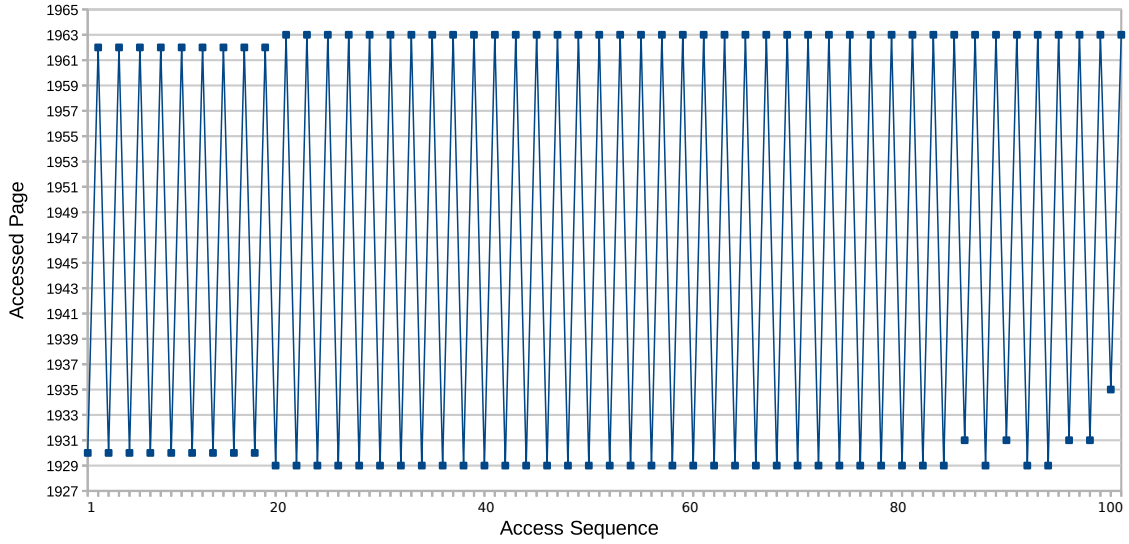


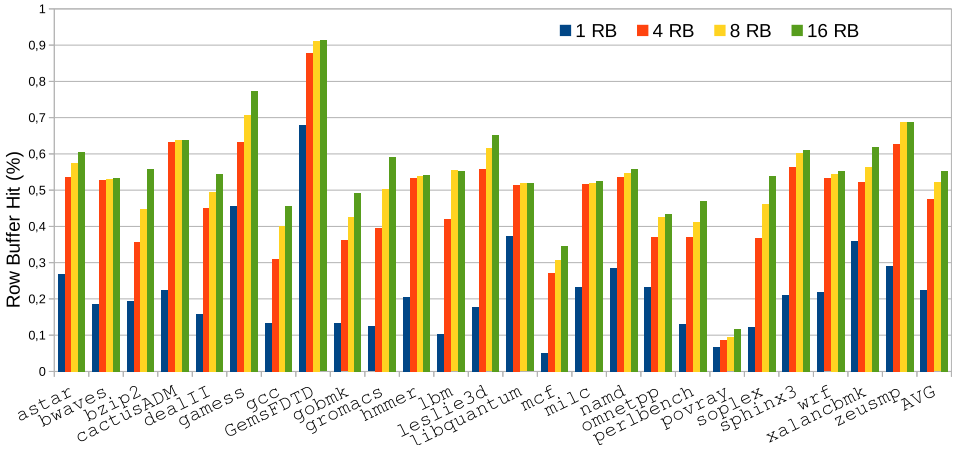
Figura 1.2: Muestra de accesos a memoria durante una ejecución de *leslie3d*.

(RBHR) de las aplicaciones en ejecución individual (sin interferencias con otras aplicaciones) variando el número de row buffers para dos funciones de mapeo distintas *rcb* y *brc* (ver la sección 2.2 para detalles de estas funciones de correspondencia). La figura 1.3 presenta los resultados de RBHR variando el número de row buffers de 1 a 16 para los benchmarks *SPEC-CPU 2006* en ejecución individual considerando ambas funciones de mapeo. Como se puede observar, independientemente de la función utilizada, los diferentes benchmarks presentan comportamientos distintos cuando incrementa el número de row buffers. En general, la tasa de aciertos en ambas funciones mejora al aumentar el número de RBs. El paso de 1 RB a 4 RBs influye de forma apreciable. Respecto a la función de mapeo, se aprecia que hay aplicaciones que presentan i) resultados similares con ambas funciones, ii) mejores resultados con *rcb* como `astar`, y iii) mejores resultados con *brc* como `leslie3d`. Sobre la media, ambas funciones de mapeo proporcionan unos resultados de RBHR similares en todas las configuraciones.

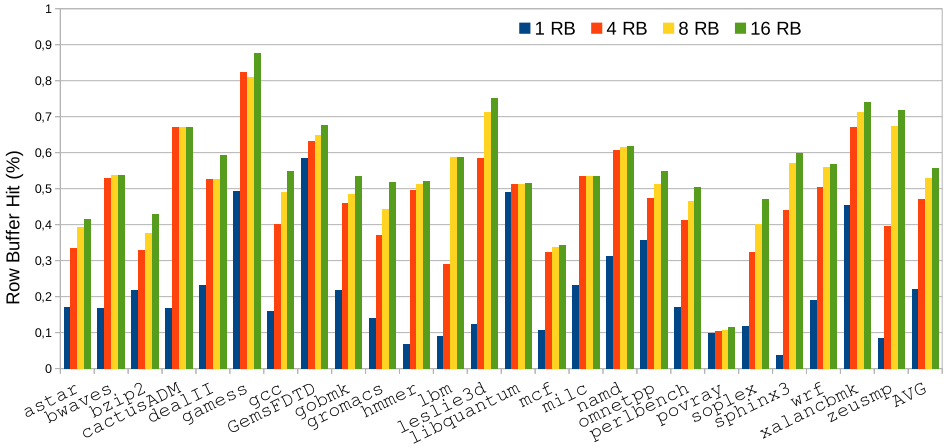
Atendiendo a los patrones de comportamiento identificados en el experimento anterior, los benchmarks se pueden agrupar y clasificar en las siguientes cuatro categorías:

- Categoría 1. En esta categoría se incluyen las aplicaciones con una gran mejora en el RBHR con el aumento del número de row buffers. De 1 RB a 4 RBs el RBHR aumenta considerablemente pero además sigue mejorando con más, como puede ser el caso de *gromacs* en *rcb* o `soplex` y `zeus` en *brc*.

- Categoría 2. En esta categoría se incluyen las aplicaciones que obtienen una gran mejora con el aumento de 1 a 4 RBs, pero luego el RBH se mantiene con un



(a) Mapeo *rcb*



(b) Mapeo *brc*

Figura 1.3: Row buffer hit variando el número de row buffers.

número mayor de row buffers, como puede ser el caso de `bwaves` en *rcb*, `cactusADM` en *brc* o `milc` en cualquiera de los dos mapeos.

- Categoría 3. Esta categoría incluye las aplicaciones que presentan una leve mejora con el aumento del número de RBs. Presentan un RBH por encima de la media con 1 RB. Un ejemplo es `GemsFDTD` en cualquiera de los dos esquemas de mapeo.

- Categoría 4. Por último, en esta categoría se incluyen aplicaciones con baja localidad espacial, por lo que presentan un bajo RBHR, el cual no mejora con el aumento de row buffers, como puede ser el caso de la aplicación `povray`.

1.3 Objetivos

Actualmente existen varias técnicas que tienen como objetivo reducir los tiempos de acceso a memoria. Desde un punto de vista ideal, el acceso a memoria sería instantáneo, pero los retrasos que existen dependen del nivel de caché en el que se encuentre el dato o, en su caso, en la memoria principal; siendo ésta la más costosa desde un punto de vista energético y de tiempo.

El objetivo de este trabajo es diseñar una organización de memoria que permita reducir estos dos puntos negativos aprovechando el paralelismo en los accesos por parte de las diferentes aplicaciones a memoria principal.

El tiempo de acceso al sistema de memoria depende principalmente de si el bloque de caché solicitado se encuentra en el RB o no, siendo 3 veces más rápido un acceso al RB que al banco de memoria. En consecuencia, es importante tratar de maximizar los aciertos para reducir el tiempo de acceso y obtener una mejora en el rendimiento. En este trabajo se propone aumentar el número de *row buffers* para maximizar el número de accesos que encuentren el bloque en ellos. Para aumentar el número de *row buffers* es necesario remodelar el sistema de memoria e introducir lógica adicional para gestionar estos nuevos recursos.

Aunque en principio, las prestaciones se benefiaciarían disponiendo la mayor cantidad de *row buffers* posibles, esta opción no sería eficiente debido al espacio que ocuparían los RBs en memoria, la escalabilidad de la lógica y el posible incremento en latencia que esto supondría. Por ello se realizará un estudio para encontrar el número óptimo de *row buffers*.

Por otro lado, tal y como hemos visto, no todas las aplicaciones se benefician de igual manera para un número dado de row buffers por lo que se diseñará un esquema que asigne dinámicamente los row buffers a las aplicaciones en función de sus necesidades en tiempo de ejecución. Los resultados se analizarán para distintos esquemas de mapeo de memoria que aprovechen este comportamiento de las aplicaciones ubicadas en las categorías 1 y 2, sin que ello afecte negativamente al rendimiento de las aplicaciones pertenecientes a otras categorías.

1.4 Estructura del trabajo

El resto del trabajo se organiza de la siguiente manera. En el capítulo 2 se explicarán los fundamentos teóricos básicos sobre los que se ha desarrollado el trabajo y, por consiguiente, como está organizado un sistema convencional así como las funciones de

correspondencia típicas que se utilizan en la memoria principal. En el capítulo 3 se hace referencia a trabajos previos relacionados con la propuesta que se presenta. En el capítulo 4 se presentará la propuesta de mejora. En el capítulo 5 se explicará el entorno de simulación y los benchmarks utilizados para la evaluación de la propuesta. En el capítulo 6 se mostrarán y analizarán los resultados de rendimiento y energía obtenidos de las diferentes propuestas. Por último, en el capítulo 7 se resumirán las conclusiones obtenidas así como el trabajo futuro.

Capítulo 2

Conceptos Previos

En este capítulo se describen los aspectos fundamentales de un sistema multinúcleo. Además, para una mejor comprensión de la totalidad del sistema, se describe el procesador, la jerarquía de memoria, con especial énfasis en la memoria principal, y la red de interconexión que interconecta estos elementos.

El sistema modelado está basado en un procesador multinúcleo donde los distintos núcleos se modelan como procesadores superescalares agresivos de manera análoga a la mayoría de los procesadores reales.

2.1 Estructura básica de un sistema

2.1.1 Procesador

El procesador ha sido tradicionalmente el foco de los diseñadores de sistemas ya que proporciona la potencia de cálculo. La mejora de éste se puede lograr sobre todo siguiendo diversas direcciones principales: la mejora de la potencia de cálculo de los núcleos individuales, el aumento del número de núcleos o combinar ambas acciones conjuntamente. La mayoría de los fabricantes de procesadores optan por procesadores multinúcleo con el objetivo de proporcionar un buen equilibrio entre rendimiento y potencia. Para este fin, incluyen varios núcleos relativamente simples. Además, los recientes avances de la tecnología y las mejoras a nivel microarquitectural han permitido utilizar técnicas más agresivas para ejecuciones fuera de orden, lo cual beneficia el uso de estos procesadores multinúcleo. Por lo tanto, desde un punto de vista de alto nivel, los dos rasgos principales que caracterizan el procesador actual son la cantidad de núcleos y su agresividad en la ejecución fuera de orden.

2.1.2 Red de interconexión

La red de interconexión es la encargada de comunicar los elementos del sistema entre sí, y posibilitar que intercambien peticiones de acceso a memoria y bloques de datos en respuesta a dichas peticiones. Concretamente, la red conecta las cachés entre sí y con el controlador de memoria. Cada vez que una petición de un dato por parte del procesador falla en la caché, esta petición se envía al controlador de memoria a través de la red de interconexión. Por su parte, el controlador de memoria obtiene el dato pedido de memoria principal y lo envía de vuelta a la caché que lo pidió a través de la red de interconexión.

2.1.3 Sistema de memoria

El sistema de memoria RAM de los sistemas multiprocesador se divide en dos subsistemas principales. El primero se implementa mediante memoria caché mientras que el segundo es la memoria principal.

Memoria caché La memoria caché está ubicada en el interior del chip del procesador. Esta memoria tiene la característica de ser muy rápida pero pequeña. Estas características están relacionadas, ya que cuanto más grande es la memoria, más lenta es ésta. Por ello, esta memoria está distribuida en distintos niveles, siendo el primero de todos el de menor tamaño y mayor rapidez, y el último nivel de caché el que presenta un tamaño mayor.

Cuando el procesador realiza la petición de un dato a memoria y este no se encuentra en ninguno de los niveles de caché, desde el último nivel se realiza la petición del dato a la memoria principal, con una capacidad mucho mayor que la caché.

Memoria principal El subsistema de memoria principal ha llegado a ser la principal preocupación del diseño de procesadores multinúcleo debido a dos razones. Primera, éste representa el mayor cuello de botella, que se agrava al incrementar el número de núcleos, ya que éstos compiten por el acceso a un determinado controlador. Segundo, el coste de la memoria principal representa una fracción importante del coste total del sistema [HP12].

La memoria principal más común actualmente se implementa mediante tecnología DRAM fuera del chip, la cual se suelen presentar en el mercado en los denominados módulos DIMM. Cada módulo tiene varios dispositivos DRAM agrupados en filas. Un rango es un conjunto de dispositivos DRAM que operan al mismo tiempo leyendo o

escribiendo en el bus, normalmente de 64 bits de ancho [JED]. El número de dispositivos que componen el rango depende de la anchura del dispositivo específico, esto es, el número de pines de datos que tiene el dispositivo. Cada uno de los rangos está internamente organizado en bancos independientes. Un banco es la estructura de memoria más pequeña que puede operar en paralelo. Como los dispositivos de un rango actúan a la vez, cuando se accede a un banco, se accede a todos los dispositivos que componen un rango, con una transmisión de un total de 64 bits desde o hacia el bus.

Cada banco se implementa como una matriz organizada en filas (también llamadas páginas) y columnas de bloques. Cada banco tiene un RB, el cual almacena los datos de la última fila accedida y que se compone de una serie de amplificadores que actúan como registros, como se puede ver en la figura 2.1, que muestra el diagrama de bloques de esta organización de memoria. Por lo tanto, los accesos posteriores a la misma fila son más rápidos si ésta se encuentra almacenada en el row buffer. Actualmente, al final de un acceso, el controlador de memoria puede mantener la fila en el row buffer (política de página abierta) o cerrarlo si no se esperan más accesos a esta fila (política de página cerrada).

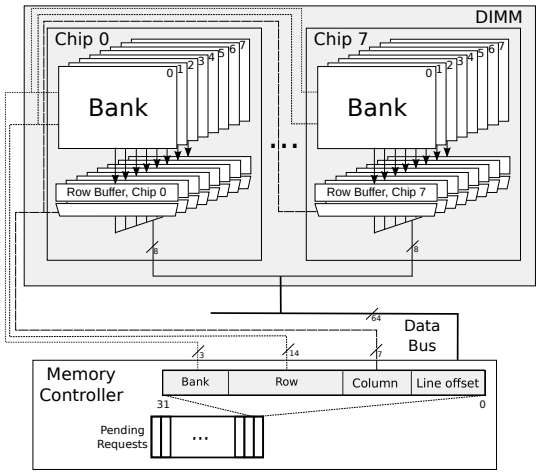


Figura 2.1: Arquitectura DRAM.

El controlador de memoria gestiona el flujo de datos hacia o desde los dispositivos DRAM. Concretamente, define la política de gestión del row buffer, el esquema de mapeo de direcciones y el esquema de ordenación de comandos para las transacciones de memoria. El controlador debe emitir tres comandos de forma secuencial para acceder a un banco DRAM: i) un comando de precarga para almacenar el row buffer en el banco, ii) una instrucción de activación para abrir la fila correspondiente a la dirección de fila del banco y llevarla al row buffer y finalmente, iii) un comando de lectura/escritura

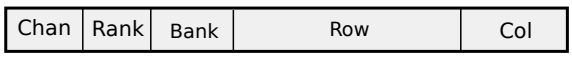
para acceder al row buffer en la posición indicada por la dirección de la columna. El tiempo de acceso a memoria se verá influenciado por la política de gestión del RB del controlador de memoria. Si la fila accedida está en el row buffer, la latencia es mucho menor y se necesita enviar solo un comando de lectura/escritura a la memoria. Si la fila no está en el row buffer, se necesita enviar los tres comandos, precarga, activación y lectura/escritura. Sin embargo, si el controlador utiliza una política de página cerrada, significa que nunca hay datos válidos en el row buffer, por lo que solo se necesita emitir los comandos de activación y lectura/escritura. En todos los casos, la latencia total es la suma de latencias de todos los comandos emitidos, ya que éstos se ejecutan de forma secuencial y en orden.

2.2 Funciones de correspondencia o mapeo

Las funciones de correspondencia o mapeo tienen por objeto determinar la posición del bloque buscado dentro del módulo de memoria. En otras palabras, definen como interpreta el controlador de memoria los distintos bits de la dirección física. Por supuesto, que la tasa de aciertos en el RB varía en función del esquema de mapeo adoptado.

En este trabajo se evalúa la propuesta bajo dos funciones de mapeo diferentes. La primera persigue explotar el paralelismo a nivel de banco dentro del módulo de memoria, esto es, bloques consecutivos se distribuyen de forma uniforme entre todos los bancos. Para ello, los campos de la dirección se interpretan siguiendo el orden *chan:rank:row:col:bank*, como puede apreciarse en la figura 2.2a. A este esquema le denominaremos *rcb* (siglas de los 3 campos de menor peso).

(a) Mapeo *rcb*



(b) Mapeo *brc*

Figura 2.2: Esquemas de mapeo de las direcciones de memoria.

Por otra parte, en la segunda función de mapeo se persigue explotar la localidad espacial del RB, mapeando bloques consecutivos en el mismo banco hasta consumir todas las direcciones de ese banco. Es decir, se diferencia de la primera función en la interpretación de los 3 campos de menor peso. Los campos de la dirección física siguen el orden *chan:rank:bank:row:col*, como puede apreciarse en la figura 2.2b. De manera

análoga a la anterior, a esta función la denominaremos *brc*. Como se puede apreciar se persigue explotar la localidad espacial de fila pero, por contra, presenta un bajo paralelismo a nivel de banco.

Capítulo 3

Trabajo relacionado

Las arquitecturas tradicionales de memoria DRAM son muy ineficientes para servidores de centros de datos y plataformas de HPC (High Performance Computing) y necesitan una revisión importante [UMC+10]. En consecuencia, una gran cantidad de trabajo de investigación se ha centrado en mejorar el rendimiento de la memoria principal.

Muchos de estas propuestas han trabajado en reordenar los accesos de memoria, es decir, se han centrado en definir políticas de planificación en el controlador de memoria. Trabajos recientes han abordado este problema para procesadores multinúcleo [LMNP11, RZ97, RDK+00, EMLP09], donde la localidad que tienen las aplicaciones en el row buffer depende de los co-runners. Estos enfoques permiten que las peticiones de memoria de múltiples aplicaciones puedan emitirse fuera de orden, permitiendo maximizar la tasa de aciertos del row buffer.

Uno de los trabajos es el propuesto en [RDK+00], que introduce el concepto de planificación de los accesos de memoria en el controlador. Propone ordenar las peticiones de memoria para aprovechar los tiempos muertos de los componentes de la memoria y realizar varios accesos de forma concurrente.

Otro de los trabajos [LMNP11] propone implementar un mecanismo hardware para maximizar el beneficio de los *prefetch* útiles minimizando el impacto negativo de los accesos inútiles. Para ello definen un sistema de prioridades adaptativas entre peticiones y prebúsquedas, y a su vez desecha peticiones inútiles del *prefetch* para liberar recursos de memoria basándose en su precisión.

Algunos trabajos [KPMHB10, MM08, MM07, ZLZZ08] proponen la planificación inteligente de la memoria para hacer frente tanto a la localidad de row buffer como al paralelismo a nivel de banco. Para llevar a cabo esta planificación, proponen llevar un

seguimiento de los patrones de acceso a memoria en el controlador de memoria, lo cual requiere una gran complejidad hardware.

Uno de estos trabajos [KPMHB10] propone una nueva política de planificación que mejora tanto la productividad como la igualdad de acceso entre diferentes hilos a los recursos de memoria. Para ello agrupa los hilos en *memory-intensive* y *non-memory-intensive*, y se aplica una técnica de prioridades entre estos grupos. Esto puede provocar que los hilos *non-memory-intensive* tengan largas esperas para el acceso a memoria, por lo que en el trabajo [MM08] plantea estrategias de planificación basadas en el seguimiento de patrones de acceso al controlador de memoria, aunque añadiendo una complejidad hardware adicional. El trabajo [MM07] propone el uso de heurísticas para estimar qué hilo es perjudicado en comparación con su ejecución en solitario, priorizándolo frente a los otros. Esto permite un acceso más justo (*fairness*) pero se produce una pérdida de productividad.

Por otra parte, centrándonos en el row buffer, la idea de usar row buffers más pequeños o buffers adicionales ha sido explotada en diversos trabajos recientes [GMMG12], [ZLZ+08, YJE11, ALSJ09, MLM12, LIMB09]. El objetivo principal es el ahorro de energía [GMMG12] mientras se mantiene el rendimiento mediante una adecuada gestión de pequeños row buffers. Por contra, nuestra propuesta consiste en asignar múltiples row buffers a las aplicaciones que lo necesiten y pocos a las que no se beneficien de ello.

Otro enfoque [YKK+13] trata de explotar el aprovechamiento del row buffer. Los autores proponen un esquema que reconoce peticiones en prebúsqueda en el controlador de memoria, y con hardware adicional intenta predecir si se necesitarán mas prebúsquedas de una fila antes de cerrarla. Si es así, los bloques que se predicen que serán accedidos por los núcleos se almacenan en un pequeño buffer en el propio controlador de memoria.

Un trabajo más cercano al propuesto en el trabajo es [HGCT13] que trata de resolver el problema de disponer de un solo row buffer con cargas multiprogramadas en un intento de aumentar el RBHR. En esta propuesta se propone asignar un row buffer por banco a cada aplicación, manteniendo la localidad espacial por hilo de ejecución. En este trabajo se implementan varios row buffers asociados a cada banco pero se asignan de manera estática, un banco a cada aplicación. Es por ello que nos referiremos a este esquema de row buffer como Static (STA). A diferencia de la gestion de STA, el esquema propuesto en esta tesina asigna dinámicamente un número de row buffers que no tiene porqué concordar con el número de aplicaciones sino con las necesidades éstas.

Capítulo 4

Arquitectura y gestión de memoria propuesta

Como se ha analizado en la sección 1.2 un gran conjunto de aplicaciones que se ejecutan en solitario mejoran notablemente su rendimiento con múltiples row buffers debido al reuso de página.

En este capítulo se presenta la propuesta *arquitectura de memoria dinámica* (DYN). Aunque esta propuesta puede aplicarse en cualquier entorno, este trabajo se ha centrado en CMPs con cargas multiprogramadas donde las interferencias entre aplicaciones en el uso del row buffer pueden ser más importantes debido a que las peticiones de las distintas aplicaciones se intercalan en el uso del RB. Además de esto también se describe la propuesta *arquitectura de memoria estática* (STA) [HGCT13] que se presentó en el capítulo 3.

Las propuestas presentadas introducen el diseño y la gestión de una *row table*, la cual consiste en sustituir el row buffer que tienen todos los bancos por un pequeño conjunto de *row buffers*, con el objetivo de incrementar la tasa de aciertos en el RB y acelerar el acceso a memoria. Esta sustitución implica una serie de cambios en la DRAM y en el controlador de memoria. Ligado a un mayor número de row buffers, son necesarios mecanismos que detecten posibles conflictos de concurrencia entre las peticiones, aseguren la independencia entre peticiones, corrupción de datos en los RB y eviten la inanición de peticiones, teniendo en cuenta que el objetivo es maximizar el paralelismo.

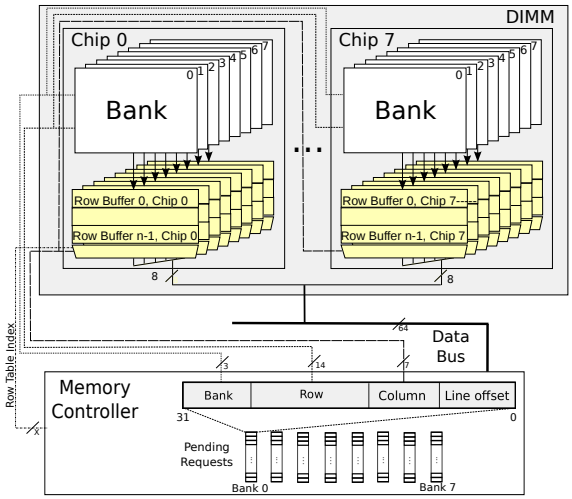


Figura 4.1: Arquitectura con varios RBs por banco.

4.1 Gestión estática de RBs

La arquitectura de agrupación estática de RBs fue propuesta en [HGCT13] y en esta tesina la denominamos como static o STA, y de manera análoga a la propuesta de este trabajo, persigue explotar la localidad de row buffer en cargas multiprogramadas. Para ello utiliza la arquitectura que muestra la figura 4.1, la cual representa un módulo DDR con estas propuestas. En lugar de disponer de un único row buffer y por lo tanto tener una sola fila abierta, se propone disponer de más RB para poder tener varias filas abiertas simultáneamente.

La diferencia con nuestra propuesta está en como se asignan los RBs a las aplicaciones. Esta propuesta asigna a cada hilo de ejecución un RB determinado. Nosotros la hemos extendido para poder aspirar a más de uno. Esta asignación se realiza de forma estática, es decir, a cada hilo de ejecución le corresponden el mismo número de row buffers.

4.2 Gestión dinámica de RBs

Esta propuesta usa la misma organización interna de memoria descrita anteriormente, la representada en la figura 4.1. Sin embargo, a diferencia de la anterior, el conjunto de row buffers está compartido entre todos los núcleos (y aplicaciones) del sistema, y se gestionan como un todo, sin tener en cuenta el núcleo que emite la petición de memoria, es decir, cualquier núcleo puede acceder a cada uno de los RBs de la row table. Para ello, se ha modificado el sistema de memoria, de tal forma que ahora la

estructura más pequeña de memoria que puede actuar en paralelo pasa de ser el banco a ser la row table junto con la matriz de celdas del banco. Por lo tanto, se puede estar realizando simultáneamente una lectura/escritura en el banco que necesite ser volcada a un RB (fallo de memoria) y una lectura/escritura que se realiza directamente sobre otro RB (acierto en memoria). Esto hace que en el tiempo que se tarda en manejar un fallo en memoria en una organización convencional puedan servirse varias peticiones en memoria que sean aciertos, con la consiguiente reducción de la latencia a memoria que tanto afecta al tiempo de ejecución de las aplicaciones.

Otra diferencia con la propuesta STA, reside en que los RBs se asignan dinámicamente a los núcleos en tiempo de ejecución de acuerdo con las necesidades de las aplicaciones. Se ha optado por una política LRU (Least Recently Used) para asignar entradas (filas) de la tabla a las aplicaciones. Esto significa que una tabla con n entradas mantiene, independientemente de los hilos que acceden a ella, los contenidos de las n filas de la matriz de datos del banco asociado más recientemente accedidas por las aplicaciones en ejecución en el CMP. Ésta es una de las mayores diferencias sobre otras propuestas, ya que la distribución de RBs entre aplicaciones no es equitativa, sino que se tiene en cuenta que las aplicaciones tienen necesidades distintas. Sin embargo, se ha realizado una modificación en la política LRU para row tables de gran tamaño (por ejemplo el doble de RB que de aplicaciones), mediante la cual se previene el reemplazo de la última página de una aplicación en el row table, es decir, se garantiza que a ninguna aplicación se le desaloje su último row buffer de la row table. Esta modificación de la política de reemplazo se ha realizado para garantizar *fairness* entre las aplicaciones en ejecución, y así prevenir un RBHR muy alto por parte de una de las aplicaciones de la mezcla porque mantiene todos los RBs mientras las demás aplicaciones se vean seriamente penalizadas.

Las row tables añaden una pequeña complejidad. Cada entrada de la tabla, además del contenido de una página (fila) de memoria (por ejemplo, 4KB), requiere algunos campos extra, tales como un campo para controlar la política de reemplazo LRU (2 bits para una tabla con 4 entradas), un campo extra para la modificación de la política LRU (2 bits para 4 aplicaciones) y un campo "válido"(1 bit) para controlar si hay alguna petición accediendo a esa entrada en ese instante.

DYN apenas necesita cambios en la lógica del planificador de memoria en el MC. El cambio a realizar consiste en que éste debe ser consciente de todas las filas almacenadas en la tabla para distinguir aciertos de fallos y proceder de acuerdo con una adecuada secuencia de comandos de memoria.

Para poder servir una petición es necesario garantizar que estén disponibles los

recursos necesarios en la memoria. Por ello, antes de extraer una petición de las colas del controlador de memoria se ha de garantizar que i) el bus de comandos de memoria esté libre, ii) en caso de acierto, que el bus de datos esté libre, iii) en caso de fallo, hay una entrada o RB libre o se tiene candidato para desalojar y el banco no debe estar sirviendo otro comando. Posteriormente, se deberá solicitar el bus de datos para leer/escribir un bloque, hasta que este recurso no esté garantizado la petición deberá esperar a que se libere el bus.

Observando las posibles combinaciones que pueden ocurrir cuando varias peticiones acceden a un mismo banco, dos peticiones pueden servir su bloque en paralelo si ambas son aciertos en las tablas fila y acceden a una entrada diferente (filas diferentes), a diferencia de los sistemas con solo un RB, ya que lo obtienen de entradas diferentes. Sin embargo, como ambas tienen que transferir su bloque por el mismo bus de datos, liberan las entradas cuando se le asigna el bus de datos y puede transferir el bloque. En caso de que se esté accediendo a la matriz de datos del banco por un fallo, es posible adelantar las peticiones que accedan a los bloques que estén el en row buffer. Dos fallos en el mismo banco no pueden servirse en paralelo ya que ambos necesitan acceder a la matriz de datos al mismo tiempo, es decir, que se procesen ambos comandos a la vez. Por lo tanto, tener más filas en la tabla incrementa la concurrencia, que crecerá cuantos más aciertos haya.

Teniendo esto en cuenta y los resultados mostrados en la Figura 1.3, DYN puede potencialmente conseguir importantes ganancias en el rendimiento.

Capítulo 5

Entorno de simulación

En este capítulo se describen las herramientas de simulación utilizadas durante este trabajo, el sistema modelado con todos sus parámetros, la metodología empleada y las métricas utilizadas. Además se proporciona una breve explicación de los benchmarks usados y cómo se han combinado para las pruebas de ejecuciones multinúcleo.

5.1 Herramientas de simulación

5.1.1 Multi2Sim

Multi2Sim [USPL07] es un simulador basado en C de un sistema completo capaz de modelar distintos tipos de hardware, incluyendo sistemas multiprocesador. Tiene la capacidad de modelar procesadores segmentados superescalares, arquitecturas multihilo y multinúcleo, unidades de procesamiento gráfico (GPUs), y soporte para los benchmarks más comunes en investigación. Se trata de un simulador de tipo *application-only* que permite ejecutar una o más aplicaciones sin tener que arrancar primero un sistema operativo. El paradigma de simulación usado en este simulador se puede dividir en dos módulos: la simulación funcional y la detallada. La funcional es solo una emulación del programa que recibe como entrada y tiene el mismo comportamiento que tendría dicho programa ejecutado nativamente en una máquina x86. El simulador detallado ofrece un modelo de estructuras de CPU como cachés, unidades funcionales, redes de interconexión, etc., y un procesador con seis etapas (fetch, decode, dispatch, issue, writeback y commit), ofreciendo soporte para ejecución especulativa. El número de hilos y el número de núcleos es configurable. Se pueden configurar la jerarquía de memoria con distintos niveles de caché, con cualquier número de cachés en cada nivel (se mantiene la coherencia mediante el protocolo MOESI). Las cachés pueden ser unificadas

o separadas para datos e instrucciones, privadas o compartidas por núcleo, por hilo, o por unidad de cómputo de GPU, y pueden servir rangos específicos de direcciones físicas. El modelo de red de interconexión entre diferentes niveles de la jerarquía de memoria también es ampliamente configurable. Incluye un conjunto de nodos finales, de conmutadores y de enlaces que permiten definir la topología de la red; y una tabla de encaminamiento bidimensional que permite definir cualquier algoritmo de encaminamiento. Por otra parte, se puede obtener un informe detallado de las estadísticas relacionadas con la segmentación del procesador, clasificadas por hilo y por núcleo de ejecución. Hay resultados de simulación genéricos y estadísticas tanto para las estructuras hardware (búfer de reordenación, cola de instrucciones, archivo de registros, búfer de destino de salto), como para cada etapa del pipeline. El informe de métricas de la jerarquía de memoria contiene una sección por cada caché, módulo de memoria principal y red de interconexión. Además, desde la versión 3.0, Multi2Sim ha aumentado sus capacidades de simulación con un modelo para las GPUs de AMD, que también cuenta con su propio depurador de pipeline. Dicho depurador es una herramienta para analizar la ejecución del código OpenCL en la unidad correspondiente del simulador. Mediante una interfaz gráfica actúa como una ayuda visual en la prueba del código y de la arquitectura del propio simulador. En sus últimas versiones, Multi2Sim usa el formato INI para todos sus archivos de entrada y salida.

5.1.2 DRAMSim2

DRAMSim2 [RCBJ11] es un simulador basado en un modelo preciso de ciclos que proporciona el controlador de memoria, los modulos DRAM que comprenden todo el almacenamiento del sistema y los buses que los comunican.

Provee una API C++ que puede ser utilizada para integrarlo en un simulador con un sistema completo, en este caso Multi2Sim.

DRAMSim2 modela todos los componentes del sistema de memoria así como los retardos que tienen lugar en cada uno de ellos, además de medir el consumo de energía de los módulos de memoria. Esto permite unos resultados similares a los que se obtendrían en un sistema de memoria real.

5.1.3 Sistema simulado

Para evaluar la propuesta se ha extendido el entorno de simulación multinúcleo Multi2sim para modelar el CMP, así como las jerarquía de memoria caché y la red de interconexión del sistema propuesto, mientras que para la simulación del controlador de

Procesador	
Número de cores	4
Frecuencia	3GHz
Política de lanzamiento	Fuera de orden
Predictor de salto	bimodal/gshare hybrid: gshare con historia global de 14 bits + 16K contadores de 2 bits, bimodal con 4K contadores de 2 bits y selección con 4K contadores de 2 bits
Ancho de Issue/Commit	3 instrucciones/ciclo
Tamaño ROB	256 entradas
Cola de load/store	64/48 entradas
Jerarquía de memoria	
L1 Instruccion cache	privada, 32KB, 8 vías, 64Bytes-line, 2 ciclos
L1 Data cache	privada, 32KB, 8 vías, 64Bytes-line, 2 ciclos
L2	privada, 256KB, 16 vías, 64Bytes-line, 11 ciclos
Red de interconexión	
Topología	Malla
Enrutamiento	X-Y
Tamaño del búfer de entrada/salida	144B
Ancho de banda de los enlaces	72B
Memoria principal	
Dispositivos DRAM	64 Meg x 8 x 8 bancos
Frecuencia del bus de memoria	1066MHz
Anchura del bus de memoria	8B/ciclo
Tamaño de la página/fila	4KB
Longitud del Burst (BL)	8
Canales	1
Controlador de memoria	prioridad FR-FCFS
Memoria total	4GB, 1 rango
Latencia	t_{RP}, t_{RCD}, t_{CL} 13.09ns cada uno

Tabla 5.1: Configuración del controlador de memoria y la memoria principal.

memoria y los módulos DRAM se ha realizado una extensión del entorno de simulación DRAMSim2.

La configuración de los parámetros principales de los diferentes componentes del sistema (la red, la jerarquía memoria y el procesador) se muestran en la Tabla 5.1. Los parámetros de memoria principal se han configurado según un dispositivo de memoria reciente MICRON DDR3 [mic11].

5.2 Métricas y metodología

La propuesta presentada en este trabajo se evalúa en el capítulo 6 en términos de prestaciones. En las pruebas realizadas, tanto en mezclas como en aplicaciones individuales, cada benchmark ejecuta 500M instrucciones para *warm-up* el sistema, luego cada aplicación ejecuta al menos 300M de instrucciones.

Para evaluar las prestaciones, se mide el número de ciclos totales que necesita la ejecución de cada aplicación así como las instrucciones y se obtiene el IPC de cada ejecución.

El IPC no es el único dato de estudio importante. Las mejoras en el sistema de memoria afectan de distinta manera en el IPC de las aplicaciones. Por tanto, es importante distinguir las mejoras a nivel de sistema y las mejoras a nivel de memoria principal.

Para estudiar las mejoras en memoria principal se usa el porcentaje de aciertos en el row buffer, es decir, del total de peticiones que acceden al row buffer, aquellos que encontraron el bloque en la fila actual y no tuvieron que acceder al banco a por ella.

Por otra parte, también es de interés el estudio de la energía consumida por el sistema, . Este consumo energético se mide internamente en DRAMSim2 de forma separada:

- Energía de activación: cuenta la energía consumida activando las filas del banco para las posteriores lecturas y escrituras, la cual está directamente relacionada con el porcentaje de aciertos en el RB.

- Energía de *background*, precarga y refresco: se refiere a la energía consumida en *background* para mantener los dispositivos activos, además de la energía consumida debido a la precarga de las líneas de bits, y la energía requerida para evitar que los condensadores pierdan el valor almacenado en el tiempo.

- Energía de lectura/escritura. Esta energía se consume cuando los datos están siendo transferidos por el bus de memoria en las operaciones de lectura y escritura.

- Energía de refresco: la energía que se mide es la consumida por la memoria al tener que recargar los valores en el banco de memoria al tratarse de memoria volátil.

5.3 Métricas de evaluación

En esta sección se describen las diferentes métricas que se han utilizado para calcular la eficiencia y prestaciones del sistema implementado. Estas métricas se utilizarán para evaluar el sistema completo y el sistema de memoria principal.

5.3.1 Métricas del sistema

Las métricas de prestaciones incluidas en esta sección se encargan de resumir el comportamiento percibido desde una visión externa del sistema, es decir, evalúan como se comporta todo el sistema trabajando conjuntamente: red, procesador y memoria.

Dentro de estas métricas de prestaciones, una de las más usadas es el IPC. Éste calcula las instrucciones que se han completado por ciclo de procesador, por lo tanto es interesante maximizar esta estadística.

Para calcular el IPC en cargas multiprogramadas, es decir, varias aplicaciones trabajando concurrentemente, se suele utilizar (entre otras) la media armónica de los IPC de las distintas aplicaciones. La media armónica de una lista de valores tiende a penalizar a los valores más pequeños de la lista. Por lo tanto, comparada con la media aritmética, ésta tiende a mitigar el impacto de los IPC muy altos y a agravar el impacto de los valores pequeños de IPC. Este hecho puede utilizarse para reflejar que algunos benchmarks de la mezcla están obteniendo malos resultados, aunque la suma de los IPC haya aumentado como resultado de haber incrementado mucho el IPC de alguna aplicación que se ha beneficiado a costa de perjudicar las prestaciones de otras aplicaciones. En otras palabras, la media armónica proporciona una ruda estimación del *fairness* en el acceso a los recursos en la evaluación del rendimiento.

$$WS_{hm} = \frac{n}{\sum_{i=1}^{n} \frac{1}{IS_i}} \tag{5.1}$$

5.3.2 Métricas de memoria principal

En este apartado se incluyen las métricas que evalúan el sistema desde un punto de vista interno, concretamente desde el punto de vista de la memoria principal, por lo tanto permiten evaluar la eficiencia de este subsistema.

Porcentaje de aciertos en el row buffer. Porcentaje de peticiones que encuentran su bloque en la fila que está almacenada en el row buffer, y por tanto, no tiene que

acceder internamente al banco:

$$RBHR = \frac{Num.\ Aciertos\ RB}{Num.\ AccesosaMemoria} \tag{5.2}$$

Energy Delay^2 Product. El ED^{2}P es una métrica que combina la energía consumida por el sistema con el tiempo de ejecución en el cuál se ha consumido. Esta fórmula ofrece 2/3 del peso de la ecuación al tiempo de ejecución y lo restante a la energía consumida, por lo que se prioriza un menor tiempo de ejecución frente a un pequeño aumento de la energía.

$$ED^2P = Energía * (TiempodeEjecución)^2 \tag{5.3}$$

5.4 Benchmarks

Para los experimentos se ha utilizado una amplia gama de aplicaciones científicas o benchmarks incluidas en el paquete SPEC CPU 2006 [**?**]. A continuación se describe el conjunto de aplicaciones lo componen.

Astar. Astar se deriva de una librería 2D de búsquedas de rutas que se utiliza en la IA del juego. Esta biblioteca implementa tres algoritmos de búsqueda de ruta diferentes: el primero es el bien conocido algoritmo A* para mapas con tipos de terrenos transitables y no transitables. El segundo, es una modificación del algoritmo de búsqueda de camino A* para la búsqueda de mapas con diferentes tipos de terreno y diferente velocidad de movimiento. El tercero es una implementación del algoritmo A* para los gráficos. Está formado por las regiones del mapa con relación de vecindad.

Bwaves. Bwaves simula numéricamente las ondas de choque en tres flujos viscosos de dimensiones transitorias transónicas.

La configuración inicial del problema de las ondas de choque consiste en una región de alta presión y densidad en el centro de una celda cúbica de una red periódica con baja presión y la densidad en otro lugar. Condiciones de contorno periódicas se aplican a la matriz de celdas cúbicas que forman una red infinita. Inicialmente, el volumen de alta presión comienza a expandirse en dirección radial como las ondas de choque clásicas. Al mismo tiempo, las ondas de expansión se mueven para llenar el vacío en el centro de la celda cúbica. Cuando el flujo alcanza la expansión de los límites, choca con las imágenes periódicas de otras células, creando así una estructura compleja de ondas

de interferencia no lineal. Estos procesos crean un sistema periódico amortiguado no lineal con energía que está siendo disipada en el tiempo. Por último, el sistema llegará a un equilibrio.

Bzip2. Bzip2 se basa en la versión 1.0.3 de Julian Seward. La única diferencia entre bzip2 1.0.3 y el benchmark bzip2 es que la versión SPEC de bzip2 no actúa sobre ningún archivo de E/S que no sea la lectura de la entrada. Toda compresión y descompresión ocurre completamente en la memoria con el fin de ayudar a aislar el trabajo realizado solo a la CPU y a la memoria.

CactusADM. CactusADM es una combinación de Cactus, un problema de código abierto, y BenchADM, un representante del núcleo computacional de muchas aplicaciones en la relatividad numérica. CactusADM resuelve las ecuaciones de evolución de Einstein, que describen cómo las curvas espacio-tiempo son la respuesta a su contenido de materia, y son un conjunto de diez ecuaciones diferenciales parciales no lineales acopladas.

DealII. DealII es un programa que utiliza deal.II, una biblioteca de programación en C++ cuyo objetivo son los elementos finitos adaptativos y la estimación de error. La biblioteca utiliza técnicas de programación del estado del arte del lenguaje de programación C++, incluyendo la biblioteca Boost.

El objetivo principal de deal.II es permitir el desarrollo de algoritmos de elementos finitos modernos, utilizando entre otros, sofisticados estimadores de error y mallas adaptativas. Escribir este tipo de programas es una tarea no trivial, y programas exitosos tienden a ser muy grandes y complejos.

Este benchmark resuelve una ecuación de tipo Helmholtz con coeficientes no constantes que está en el corazón de las soluciones para una amplia variedad de aplicaciones. El código utiliza métodos adaptativos modernos basados en la estimación de la dualidad de error ponderado para generar mallas óptimas.

Gamess. Una amplia gama de cálculos químicos cuánticos son posibles con GAMESS. Éste hace referencia a los siguientes cálculos: Cálculo del Campo autoconsistente (SCF) de la molécula de citosina mediante el método directo SCF. Cálculo SCF de agua y Cu2+ utilizando el método directo SCF. Cálculo SCF de triazolio iónico utilizando el método directo SCF.

Gcc. Gcc está basado en la versión 3.2 de gcc. Este benchmark genera código para un procesador AMD Opteron. El índice de referencia se ejecuta como un compilador con muchos de sus parámetros de optimización habilitados.

GemsFDTD. GemsFDTD resuelve las ecuaciones de Maxwell en 3D en el dominio del tiempo utilizando el método de dominio de tiempo en diferencias finitas (FDTD). Se calcula la sección transversal del radar (RCS) de un objeto perfectamente conductor (PEC). El núcleo del método FDTD son las aproximaciones precisas de segundo orden a las leyes de Faraday y de Ampere.

Gobmk. El programa juega al Go y ejecuta un conjunto de comandos para analizar las posiciones del Go.

Gromacs. Gromacs se deriva de GROMACS, un paquete versátil que realiza la dinámica molecular, es decir, la simulación de las ecuaciones de Newton del movimiento para sistemas con cientos de millones de partículas. Este benchmark lleva a cabo una simulación de la proteína lisozima en una solución de agua y iones. Mediante la simulación de los movimientos atómicos de estas estructuras se puede obtener una comprensión significativa de la dinámica de las proteínas y la función, y en algunos casos, incluso podría ser posible predecir la estructura de las nuevas proteínas.

Hmmer. Profile Hidden Markov Models son modelos estadísticos de múltiples alineamientos de secuencia, que se utilizan en la biología computacional para buscar patrones en las secuencias de ADN. La técnica se utiliza para hacer una búsqueda de bases de datos sensible, usando descripciones estadísticas de una secuencia de familia. Se utiliza para el análisis de secuencia de la proteína.

Lbm. Este programa implementa el denominado Método de Lattice Boltzmann (LBM) para simular los fluidos incompresibles en 3D. Es la parte computacionalmente más importante de un código que se utiliza en el campo de la ciencia de los materiales para simular el comportamiento de los fluidos con superficies libres, en particular la formación y el movimiento de burbujas de gas en el metal.

Leslie3d. Leslie3d se deriva de LESlie3d (Large-Eddy Simulations with Linear-Eddy Model in 3D). Es la solución primaria utilizada para investigar una amplia variedad de fenómenos de turbulencia tales como mezclado, la combustión, la acústica y la mecánica de fluidos generales.

Para CPU2006, el programa se ha creado una para resolver un problema de prueba que representa un subconjunto de estos flujos. Este tipo de flujo se produce en las regiones de mezcla de todas las cámaras de combustión que emplean inyección de combustible.

LESlie3d utiliza un algoritmo fuertemente conservador, de volúmenes finitos con el esquema de integración temporal Predictor-Corrector MacCormack.

Libquantum. Libquantum es una biblioteca para la simulación de un ordenador cuántico. Los ordenadores cuánticos se basan en los principios de la mecánica cuántica y pueden resolver ciertas tareas computacionalmente difíciles en tiempo polinomial.

Libquantum proporciona una estructura para la representación de un registro cuántico y algunas puertas elementales. Además, libquantum ofrece la simulación de decoherencia, el obstáculo más importante en la construcción de ordenadores cuánticos prácticos. Por lo tanto, no solo es posible simular cualquier algoritmo cuántico, sino también desarrollar algoritmos de corrección de errores cuánticos.

Mcf. Mcf es un benchmark que se deriva de MCF, un programa de programación de vehículos en el transporte público de masas. El programa está escrito en C. La versión de referencia utiliza casi exclusivamente aritmética de enteros.

El programa está diseñado para la solución de los problemas sobre un único depósito de programación de vehículos que se producen en el proceso de planificación de las empresas de transporte público.

Milc. El Código MILC es un conjunto de códigos escritos en C desarrollado por MILC para hacer simulaciones de cuatro dimensiones en máquinas paralelas MIMD. Milc en Spec CPU2006 utiliza la versión en serie del programa su3imp. La versión de un solo procesador de esta aplicación es importante y relevante, ya que el rendimiento paralelo depende de un buen rendimiento solo procesador.

Namd. Namd se deriva de la disposición de los datos y el bucle interno de NAMD, un programa paralelo para la simulación de sistemas biomoleculares grandes.

Omnetpp. Este benchmark realiza la simulación de eventos discretos de una red Ethernet de gran tamaño. La simulación se basa en la OMNeT++ sistema de simulación de eventos discretos, un marco genérico de simulación y abierto. El área de aplicación principal de OMNeT++ es la simulación de redes de comunicación, pero su arquitectura

genérica y flexible permite su uso en otras áreas tales como la simulación de sistemas de colas, redes, arquitecturas de hardware o procesos de negocios.

Perlbench. Perlbench es una versión reducida de Perl v5.8.7, el lenguaje de programación. La versión de SPEC de Perl ha omitido la mayor parte de las características específicas del sistema operativo.

Povray. POV-Ray es un trazador de rayos. El trazado de rayos es una técnica de representación que calcula una imagen de una escena simulando la forma en que los rayos de luz viajan en el mundo real, pero lo hace al revés. En el mundo real, los rayos de luz son emitidos por una fuente de luz e iluminan los objetos. La luz se refleja en los objetos o pasa a través de los objetos transparentes. Esta luz reflejada golpea el ojo humano o un lente de la cámara. Como la gran mayoría de los rayos nunca golpea a un observador, tomaría una eternidad trazar una escena. Por lo tanto, ray-trazadores como POV-Ray comienzan con su cámara simulada y trazan los rayos hacia atrás en la escena. El usuario especifica la ubicación de la cámara, fuentes de luz, y los objetos, así como las texturas de la superficie y de los interiores.

Para cada píxel del rayo es disparado desde la cámara a la escena para ver si se cruza con cualquiera de los objetos en la escena. Se calcula cada vez que un objeto es golpeado, el color de la superficie en ese punto. Con este fin los rayos se envían a cada fuente de luz para determinar la cantidad de luz que viene de él o si el objeto está en la sombra.

Soplex. Soplex se basa en SoPlex Versión 1.2.1., resuelve un programa lineal utilizando el algoritmo Simplex.

El LP se administra como una $m \times n$matriz A, junto con un vector b de dimensión m, y un vector coeficiente c que es la función objetivo, de dimensión n. En general, el problema es encontrar el vector x para: minimizar c'x sujeto a Ax <= b con x>= 0.

En la práctica, x puede tener límites superiores y las limitaciones de A (i,.) X <= b (i) podrían ser restricciones mayores-que o iguales a, o restricciones de igualdad.

SoPlex, como la mayoría de las implementaciones del algoritmo simplex, emplea algoritmos de álgebra lineal dispersa, en particular, una escasa LU-factorización y las rutinas adecuadas para resolver los sistemas de ecuaciones triangulares resultantes.

Sphinx3. Sphinx-3 es un sistema de reconocimiento de voz muy conocido en la Universidad Carnegie Mellon.

Wrf. WRF se basa en el modelo de Investigación del tiempo y Prospectiva, que es un sistema de predicción numérica del tiempo destinado a servir a la predicción operativa y las necesidades de investigación atmosférica. WRF es adecuado para un amplio espectro de aplicaciones a través de escalas que van de metros a miles de kilómetros.

Xalancbmk. Este programa es una versión modificada de Xalan-C++, un procesador XSLT escrito en un subconjunto portátil de C++. Xalan-C++ versión 1.8 es una aplicación robusta de las Recomendaciones W3C para XSL Transformations (XSLT) y el Lenguaje de rutas XML (XPath). Se utiliza el lenguaje XSLT para redactar hojas de estilo XSL. Una hoja de estilo XSL contiene instrucciones para la transformación de documentos XML a partir de un tipo de documento a otro tipo de documento (XML, HTML, u otros). En términos estructurales, una hoja de estilo XSL especifica la transformación de un árbol de nodos (la entrada XML) en otro árbol de nodos (el resultado de la salida o transformación).

Zeusmp. Zeusmp se basa en ZEUS-MP, un código de dinámico de fluidos computacionales desarrollado en el Laboratorio de Astrofísica Computacional (NCSA de la Universidad de Illinois) para la simulación de fenómenos astrofísicos. ZEUS-MP resuelve problemas en tres dimensiones espaciales con una amplia variedad de condiciones de contorno. El programa resuelve las ecuaciones de ideales (sin resistencia), no relativistas, hidrodinámica y magnetohidrodinámica, incluyendo campos gravitacionales aplicados externamente y auto-gravedad. El gas puede ser adiabático o isotérmico, y la presión térmica es isotrópica. Las condiciones de contorno se pueden especificar como un reflejo, periódico, entrada o salida. El problema resuelto en ESPEC CPU2006 es un 3-D Blastwave simulado con la presencia de un campo magnético uniforme a lo largo de la dirección x.

5.5 Mezclas

El diseño de las pruebas multinúcleo se ha basado en la caracterización presentada en la sección 1.2. Se ha diseñado un conjunto de mezclas de 4 núcleos usando una malla 2x2 para estudiar el comportamiento de las propuestas en diferentes escenarios. La Tabla 5.2 muestra las mezclas de los 4 núcleos. Cada mezcla contiene aplicaciones de diferentes categorías identificadas en el capítulo **??** para analizar las interferencias entre diferentes tipos en el acceso a memoria principal.

Mezcla	Benchmark			
	Core 3	Core 2	Core 1	Core0
m1	hmmer	mcf	soplex	sphinx3
m2	gobmk	sphinx3	wrf	zeusmp
m3	gromacs	hmmer	povray	soplex
m4	GemsFDTD	gromacs	sphinx3	wrf
m5	GemsFDTD	leslie3d	libquantum	zeusmp
m6	gromacs	povray	sphinx3	zeusmp
m7	astar	cactusADM	hmmer	povray
m8	cactusADM	libquantum	gobmk	wrf
m9	sphinx3	zeusmp	soplex	povray

Tabla 5.2: Mezclas de 4 núcleos.

Capítulo 6

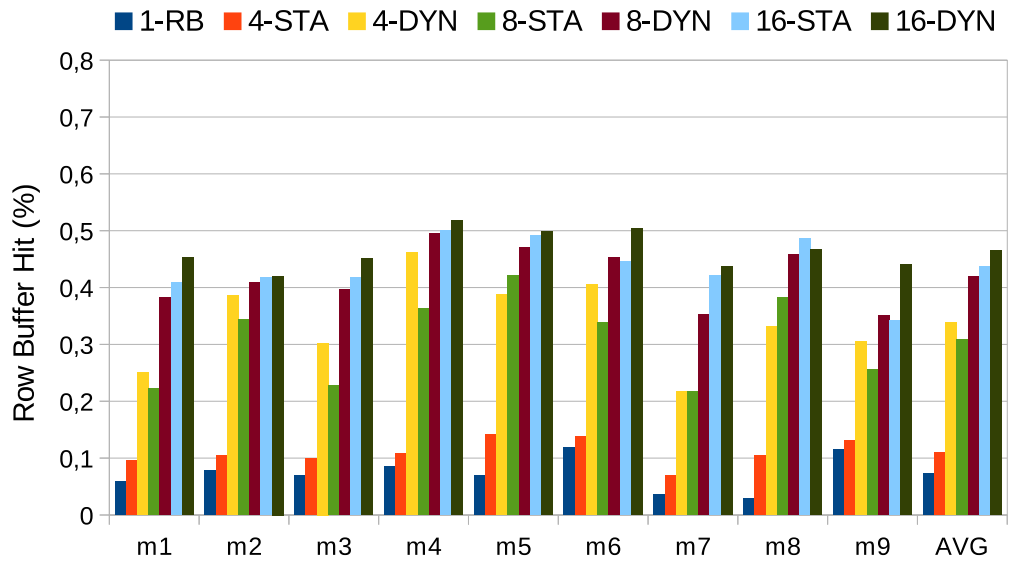
Evaluación experimental

En este capítulo se evalúa la propuesta que se realiza en esta tesina de máster. Para ello se han organizado los resultados obtenidos en 3 secciones. En la primera, se comparan las dos propuestas que se presentan en el capítulo 4. En la segunda sección se acota el número óptimo de entradas en la row table de la propuesta para obtener la mejor configuración del sistema. Por último, se evalúa la configuración escogida de forma dinámica a lo largo de una ejecución completa.

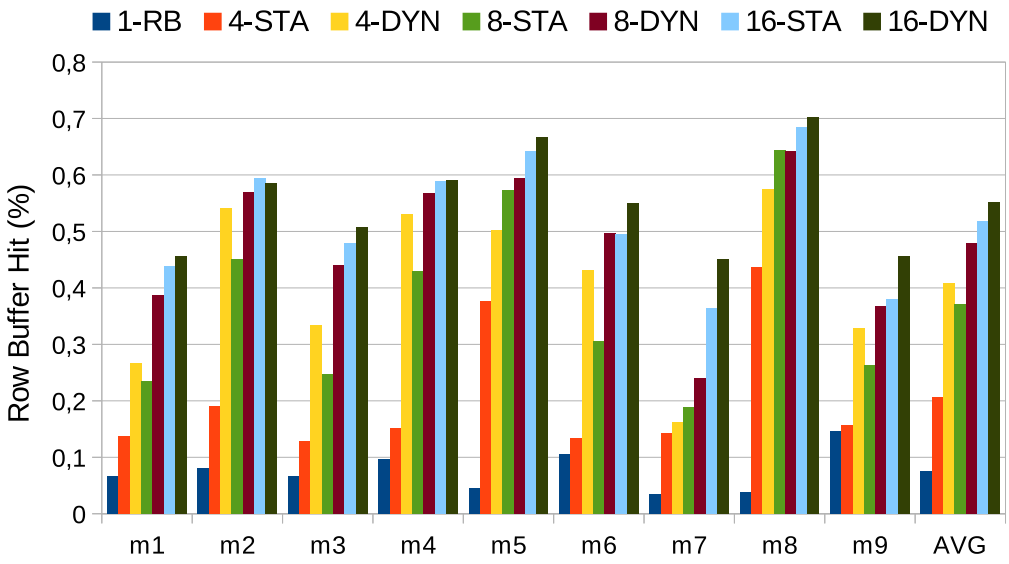
6.1 Evaluación del IPC y RBHR de la propuesta base

En esta sección se presentarán y analizarán los resultados obtenidos sobre las propuestas con las mezclas de la tabla 5.2.

Para poder comparar las propuestas se han implementado los modelos estudiados en el capítulo 3. El modelo STA asigna un número fijo de row buffers o entradas de la tabla del banco a cada hilo de ejecución o núcleo. Este número de filas se estableció a 1 en su trabajo original. Sin embargo, para realizar una comparación justa, este esquema ha sido modificado para asignar múltiples entradas de la tabla para cada núcleo. De esta forma, el número total de entradas en la tabla es igual al de los esquemas propuestos en este trabajo. Como se ha mencionado previamente, la propuesta presentada en esta tesina (DYN) se diferencia de STA en que no preasigna buffers a los núcleos, sino que las entradas se asignan dinámicamente de acuerdo con las necesidades de cada hilo en tiempo de ejecución. Esta política sigue la observación discutida en la sección 1.2 que afirma que cada benchmark requiere un número diferente de RBs para lograr su mejor rendimiento.



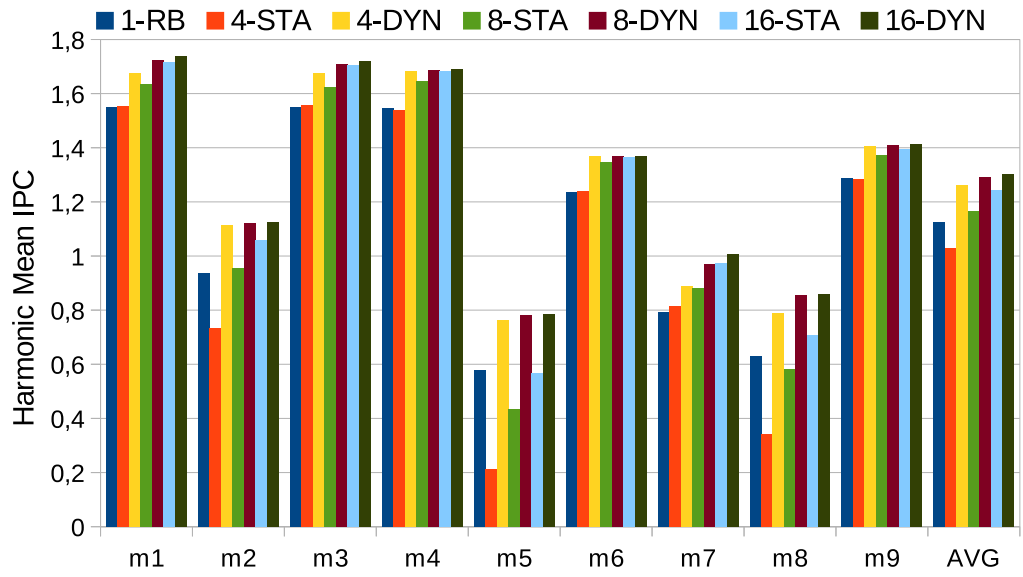
(a) Mapeo *rcb*



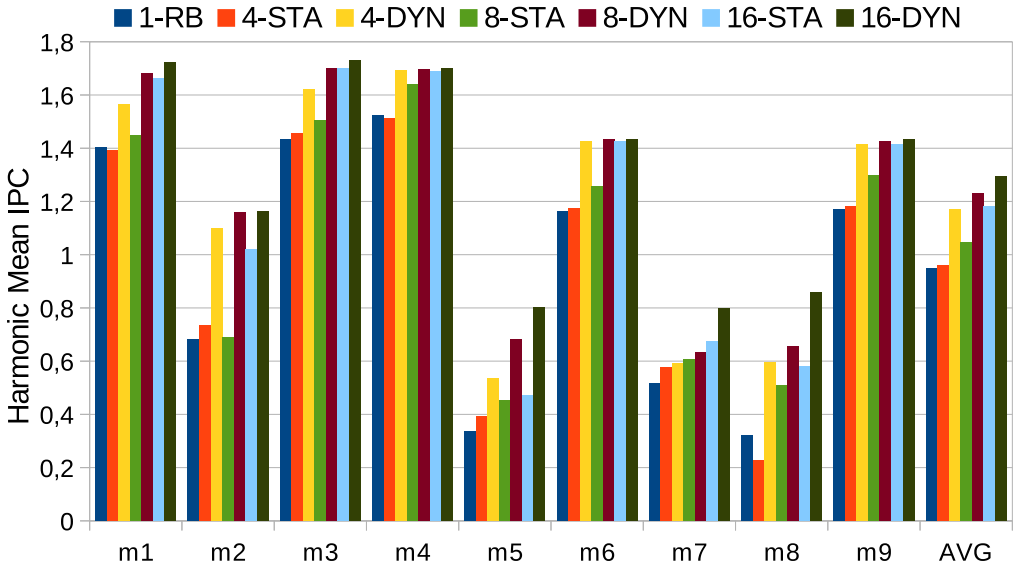
(b) Mapeo *brc*

Figura 6.1: Row buffer hit ratio de las mezclas.

Para analizar este hecho, la figura 6.1 evalúa el RBHR que obtienen estos dos esquemas con las dos funciones de mapeo presentados en esta tesina (rcb,brc) variando el número de RB en 4, 8 y 16. La primera observación que se puede extraer es que en ambos esquemas el aumento del número de RBs tiene un alto impacto en la tasa de aciertos obtenida. Se puede ver que debido a la asignación estática que realiza STA, la media de RBHR es mucho menor que con el mismo número de RBs con la configuración DYN, llegando a mejorar solo un 5 % respecto a la configuración habitual con 1 RB en el



(a) Mapeo *rcb*



(b) Mapeo *brc*

Figura 6.2: IPC de las mezclas.

esquema de mapeo *rcb*. También se puede ver como la configuración antes mencionada del esquema DYN con 4 RB presenta un mayor RBHR que la configuración STA con 8 RB en los dos esquemas de mapeo. La razón por la que esto es así radica en la mejora que proporciona en algunas aplicaciones disponer de más row buffers mientras que hay otras en las que no aporta beneficio alguno, lo que aprovecha eficientemente el mecanismo dinámico de asignación. Este razonamiento se verificará más adelante, en la sección 6.3, con el análisis detallado en tiempo de ejecución. Debido a esto, en las

mezclas con aplicaciones pertenecientes a la misma categoría o categorías de amplia mejora, no existe una gran variación en cuanto al RBHR (ej. mezcla 2), sin embargo, al estar presentes aplicaciones de amplia mejora y otras de escasa mejora, estas últimas penalizan a las primeras en el esquema STA, debido a que la mejora individual con el aumento de row buffers es muy distinta entre ambas.

También se puede observar como el porcentaje de aciertos que se producen en los RBs está estrechamente relacionados con el **mapeo de memoria** que se está utilizando. Se puede ver cómo con el cambio de mapeo de memoria varía el row buffer hit de un 70 % en algunas aplicaciones con el mapeo de memoria que explota la localidad de banco (brc) a un 51 % como máximo en el mapeo de memoria que explota el paralelismo de banco (rcb).

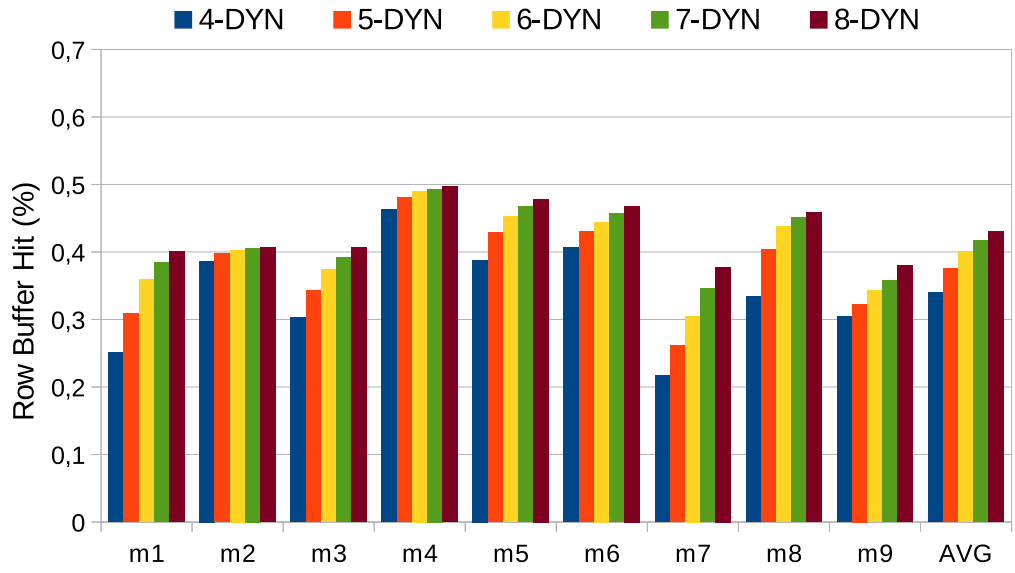
Por otro lado, si atendemos a la estadística que se utiliza para medir el rendimiento del sistema, el IPC, podemos ver en la figura 6.2 como el esquema DYN mejora el IPC del sistema base con el aumento de row buffers en todas las mezclas, e incluso con una configuración de 4 RBs tiene el mismo IPC que la configuración STA de 16 RBs. También se puede observar como el aumento en el número de row buffers a partir de la configuración de 4 RBs en la configuración DYN, no ofrece una mejora tan grande como la que se ve en el RBHR, llegando en promedio a un 8 % en la configuración *brc*.

Además, en cuanto a las funciones de correspondencia de memoria, el IPC del sistema que tiene una configuración de memoria que permite un mayor paralelismo a nivel de banco es similar al que ofrece una mayor localidad a nivel de banco.

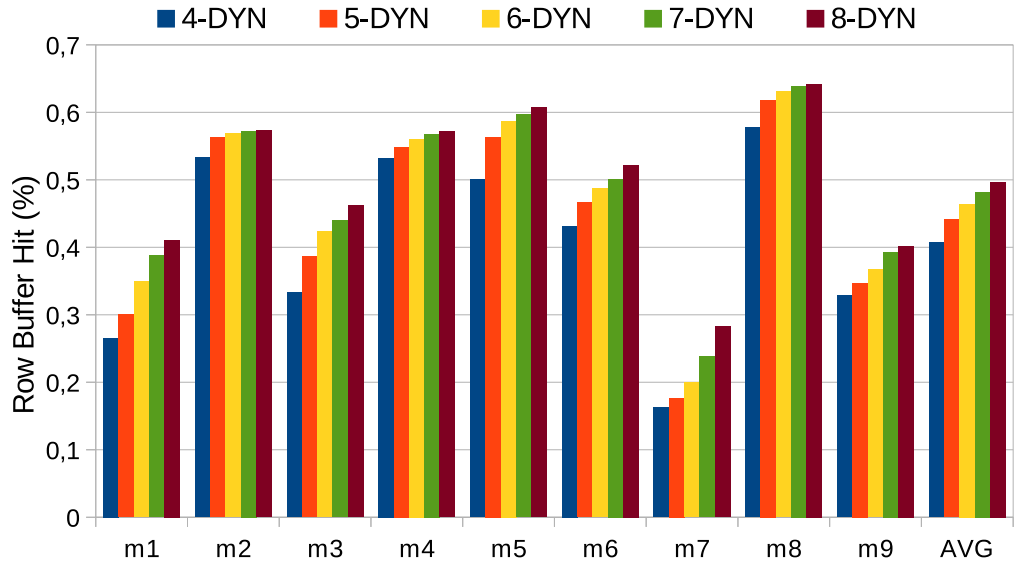
6.2 Ajuste de DYN

Una vez compradas las dos propuestas, se puede ver como la propuesta DYN mejora las métricas estudiadas respecto a la propuesta STA. Por otro lado, también se puede ver como la mejora que presenta el aumento de RBs no lleva una progresión lineal respecto al número de éstos, sino que se ve una amplia mejora al añadir RBs hasta la configuración de 8 RBs, a partir de la cual la mejora es mucho menor a pesar incluso de doblar el número de RBs. Por lo tanto, en esta sección se acotará el sistema a cinco configuraciones, de 4 a 8 row buffers, y se analizarán los resultados para seleccionar la mejor configuración del sistema en base a las mejoras obtenidas.

Al igual que en la sección anterior, también se estudiará el RBHR para ver cómo se comportan las mezclas. La figura 6.3 muestra los resultados, donde la media del RBHR de las mezclas muestra una progresión lineal en función del aumento de RBs en cualquiera de los dos esquemas de mapeo.



(a) Mapeo *rcb*



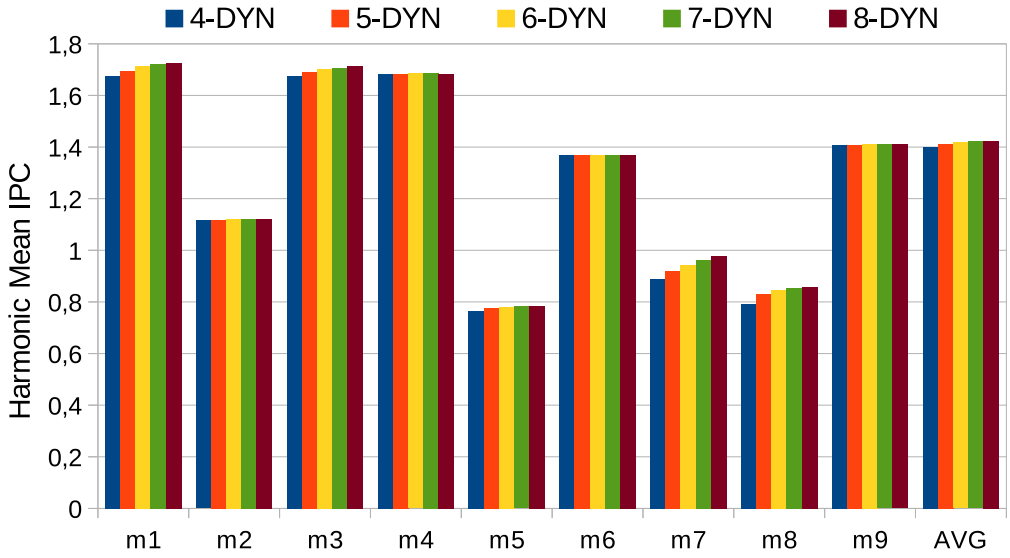
(b) Mapeo *brc*

Figura 6.3: Row buffer hit ratio de las mezclas.

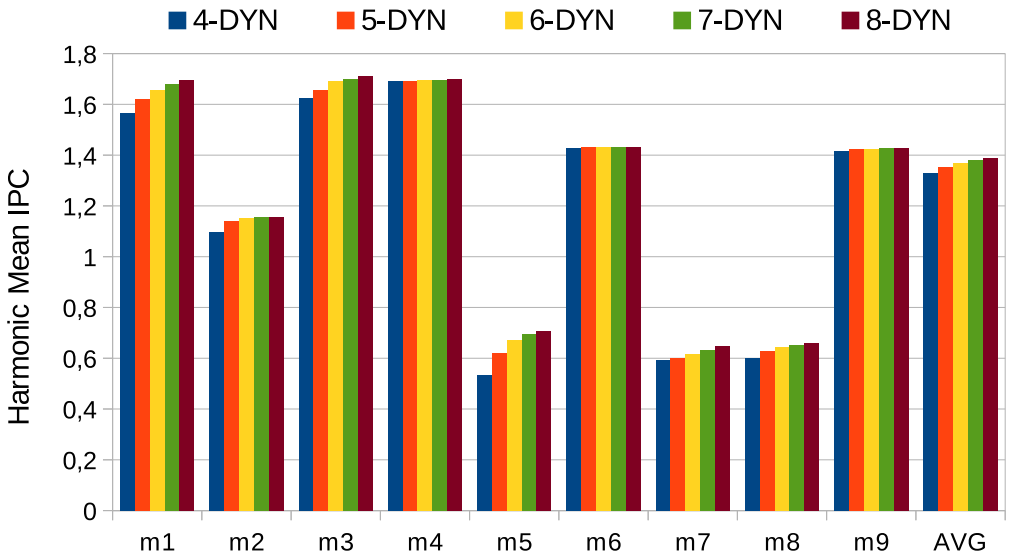
Si se compara el RBHR entre los dos esquemas de mapeo, se puede ver que aumenta un 10 % los dos, pero teniendo en cuenta que el esquema *brc* presenta un RBHR un 10 % mayor que *rcb* de media. También se puede ver como esta tendencia se invierte en algunas mezclas, como puede ser la mezcla 7.

La figura 6.4 muestra el IPC del sistema en la ejecución de las mezclas. Tal como muestra, y que como se estudió en la sección anterior, el IPC no aumenta con el aumento del número de RBs, sino que se mantiene prácticamente igual en los dos mapeos de

memoria.



(a) Mapeo *rcb*

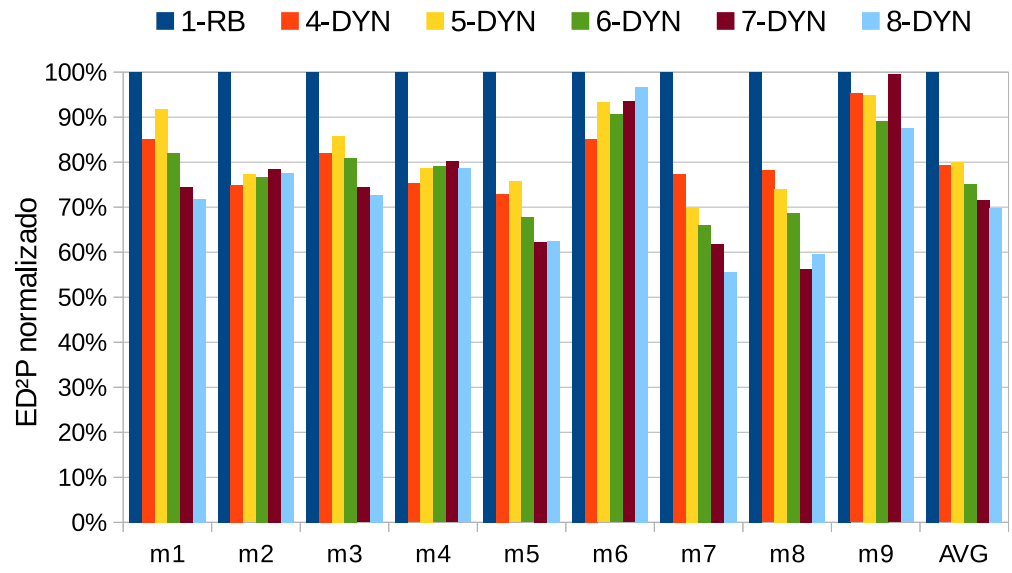


(b) Mapeo *brc*

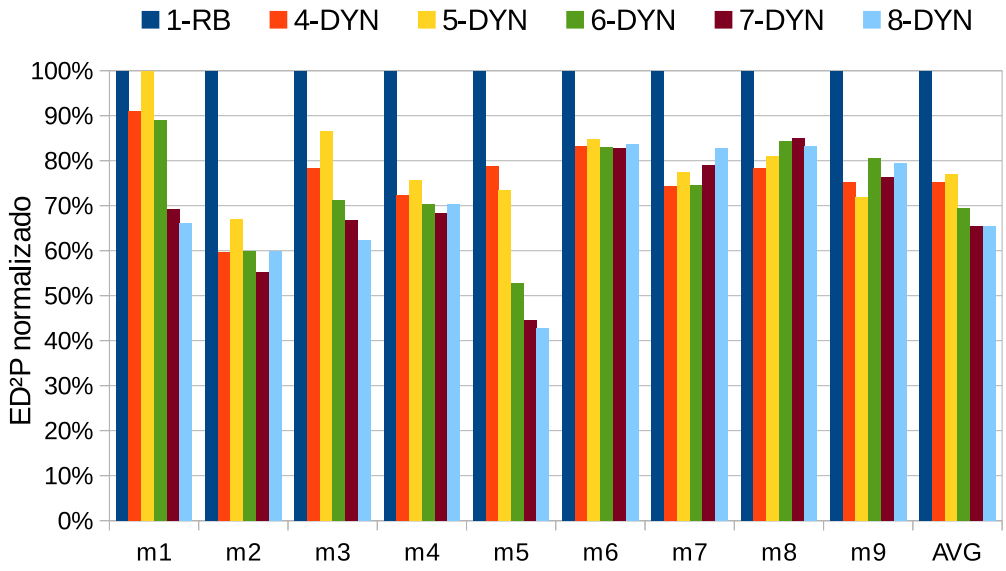
Figura 6.4: IPC de las mezclas.

Por último, para poder decidir sobre la mejor configuración, se ha obtenido el ED^2P (que evalúa tanto consumo energético como prestaciones), explicado en la sección 5.3.2. Las gráficas de la figura 6.5 muestran el ED^2P normalizado. Para ello, se ha obtenido el valor tomando como referencia el valor obtenido en la ejecución del sistema base para cada mezcla.

Tal como muestran las gráficas, a medida que se aumenta el número de RBs, el



(a) Mapeo *rcb*



(b) Mapeo *brc*

Figura 6.5: Energy Delay^2 Product normalizado sobre el sistema base.

ED^{2}P es menor. Esto es debido a la disminución del tiempo de ejecución de las mezclas con el aumento de row buffers, siendo éste más significativo que el aumento de consumo energético que éstos producen.

6.3 Análisis dinámico

En esta sección se realiza un análisis dinámico de una ejecución del sistema propuesto. La configuración para la que se muestran los resultados son las row tables con 8 RBs.

Para realizar el análisis se han tenido en cuenta los valores de IPC previamente obtenidos en la sección anterior. Debido a que estos valores son casi idénticos para las configuraciones de 7 y 8 RBs (que son los que mayor IPC presentan), se ha tenido en cuenta el RBHR junto con el ED^2P. Una vez analizados los resultados, se ha escogido la configuración que mayor RBHR ha obtenido y a su vez menor ED^2P presenta para realizar el análisis dinámico en tiempo de ejecución.

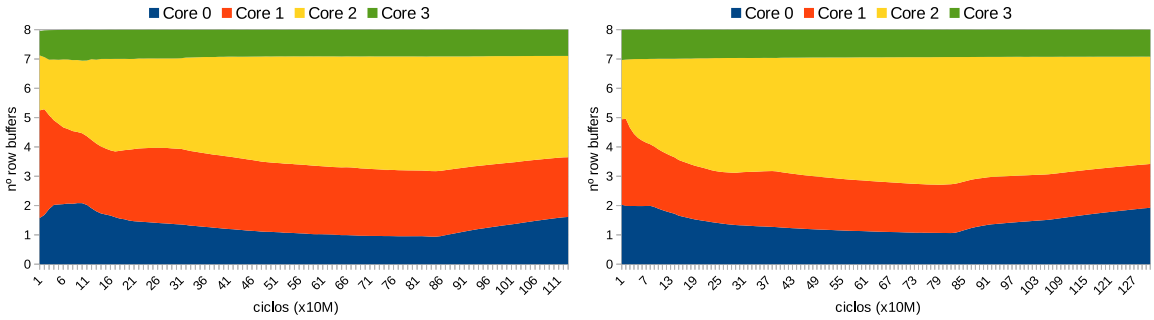
La figura 6.6 muestra la media de la distribución en la ocupación para 8 row buffers en la propuesta DYN a lo largo del tiempo de ejecución para cada una de las aplicaciones de la mezcla 5, para los dos esquemas de mapeo estudiados.

Se puede ver cómo se van distribuyendo los 8 row buffers en función de los accesos y demanda de las aplicaciones de forma dinámica. Como se puede apreciar, comparando esta figura con la figura 6.7, que muestra los fallos de caché de último nivel y por lo tanto los accesos a memoria, las aplicaciones que más acceden a memoria son las que tienen asignados un mayor número de RBs en el esquema DYN. El número de fallos de L2 que muestran las figuras 6.7a y 6.7b son distintos durante la ejecución de la misma mezcla; esto se explica por la política del controlador de memoria, que adelanta las peticiones a memoria que son aciertos en el RB, sin embargo, el total de accesos a memoria son los mismos.

Más en profundidad, se observa como la aplicación mapeada en el core 3 (`GemsFDTD`) tiene asignada una media de 1 row buffer. Esto es así debido a que esta aplicación realiza pocos accesos a memoria, como se puede ver en la figura 6.7. En la figura 6.8b se muestra el RBHR por aplicación de la mezcla con el mapeo `brc` y se observar como en el ciclo 127, el core 3 incrementa la tasa de aciertos en memoria cuando tiene una ráfaga de accesos a memoria, mostrado en la figura 6.7b.

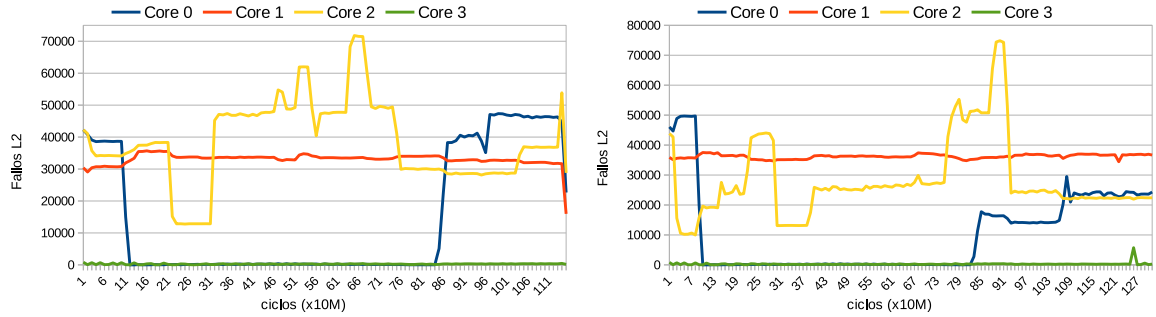
Si nos centramos en la función de mapeo *rcb*, la aplicación en el core 0 (`zeusmp`), la del core 1 (`libquantum`) y la del core 2 (`leslie3d`), al final de la ejecución presentan un número de accesos similar, por lo que el reparto de row buffers se realiza principalmente entre estas tres aplicaciones, y presentan un RBH en ese espacio de tiempo (figura 6.8a) similar al que presentan en su ejecución individual con 1 RB (figura 1.3).

Se ve como en el último tercio de ejecución, los fallos en L2 del core 0 aumentan, por lo que aumentan los accesos a memoria, y por lo tanto, aumenta el número de row



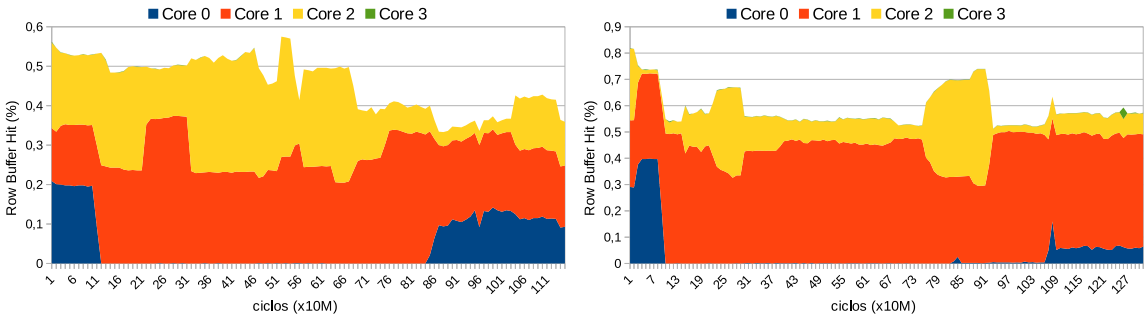
(a) Mapeo *rcb* (b) Mapeo *brc*

Figura 6.6: Nº de row buffers ocupados a lo largo del tiempo con 8 RBs en la mezcla 5.



(a) Mapeo *rcb* (b) Mapeo *brc*

Figura 6.7: Fallos de la caché L2 a lo largo del tiempo con 7 RB en la mezcla 5.



(a) Mapeo *rcb* (b) Mapeo *brc*

Figura 6.8: Row buffer hit a lo largo del tiempo con 7 RB en la mezcla 5.

buffers que se le asignan y también su RBHR.

Por otra parte, en cuanto a la función de mapeo de memoria *brc*, se puede observar como la ejecución se desarrolla de forma parecida a la del mapeo *rcb*, sin embargo tiene un RBHR un 20 % mayor, pero llegando a ser la ejecución un 16 % más lenta (véanse los valores del eje X) que el mapeo anterior. Esto se debe a que todas las peticiones de memoria se realizan en el mismo banco, y aunque aumenta el RBHR respecto a la otra, no se pueden realizar dos peticiones simultáneas y los tiempos de lectura y activación

del propio RB hacen que las peticiones no puedan aprovechar el bus de memoria todo el tiempo.

Capítulo 7

Conclusiones y trabajo futuro

7.1 Conclusiones

Este trabajo ha mostrado que el rendimiento del row buffer de un conjunto de aplicaciones individuales puede mejorar significativamente añadiendo buffers adicionales en los módulos de memoria. Esta observación tiene un especial interés en sistemas multinúcleo con cargas multiprogramadas porque los módulos de memoria actuales implementan un único row buffer que almacena la última página accedida por la última aplicación que accedió a memoria principal.

Basándose en esta observación y el impacto que tiene el RB en el rendimiento, en este trabajo se ha presentado el concepto de row table como el conjunto de row buffers compartidos por todas las aplicaciones que se ejecutan en el sistema y que compiten por el acceso a memoria principal. Los row buffers son asignadas dinámicamente a las aplicaciones en función de las necesidades que éstas tienen, de ahí que la denominemos DYN. Esta propuesta se ha evaluado sobre dos esquemas de mapeo de memoria distintos, uno que explota el paralelismo a nivel de banco (rcb) y otro que explota la localidad a nivel de banco (brc) y se ha comparado con una propuesta de agrupación de RBs estática (STA).

En la comparación de propuestas se puede ver como los dos diseños (STA y DYN) mejoran tanto el RBHR como el IPC en el sistema. Sin embargo, la mejora que proporciona la propuesta STA (propuesta del estado del arte) es mucho menor que la que ofrece nuestra propuesta DYN con un aumento pequeño del número de row buffers del sistema (4 u 8 RBs), llegando DYN con una configuración de 8 RBs casi a igualar el RBHR de STA con 16 e incluso alcanzar un IPC mayor. Esto es así debido a que aunque STA tenga 4 y 8 RBs, en realidad cada aplicación cuenta con 1 y 2 respectivamente, por lo que si una aplicación no tiene prácticamente accesos a memoria estos RBs están

desaprovechados, mientras que al asignarlos dinámicamente los RBs son utilizados por las aplicaciones que los necesitan. Sin embargo, también se debe tener en cuenta que las aplicaciones con pocos accesos no sufrirán inanición, por lo que se ha realizado un diseño que trata de no penalizar estas aplicaciones.

Una vez determinado el mejor rendimiento de esta propuesta, se ha realizado un análisis para determinar la mejor configuración. El estudio muestra como el RBHR crece de forma aproximadamente lineal en los dos mapeos de memoria explorados. El IPC muestra también esta tendencia aunque más suavizada. También se ha realizado un análisis del ED^2P en el que se evalúa la energía consumida por el sistema y el tiempo de ejecución de la mezcla de aplicaciones. Los resultados mostraron que mientras que las configuraciones con 4 y 5 RBs obtienen resultados un 20 % menor que el esquema base con 1 RB, los esquemas con 7 y 8 RBs obtienen valores un 30 % aproximadamente menores.

7.2 Trabajo futuro

Como trabajo futuro de esta propuesta se plantea la modificación de la política de asignación dinámica de row buffers para mejorar las prestaciones y el consumo. En concreto se actuará con más precisión en base a las necesidades de las aplicaciones, así como poder priorizar unas peticiones críticas a memoria sobre otras que no lo son.

Por otro lado, también se prevé la ampliación del sistema para que pueda alternar entre los distintos mapeos de memoria en función de las cargas del sistema.

7.3 Publicaciones relacionadas con el proyecto

J. Duro, J. Puche, V. Selfa, M. E. Gómez, S. Petit, J. Sahuquillo, “Explotación de la localidad de banco en cargas multiprogramadas”, en XXVI Jornadas de Paralelismo (JP2015), Córdoba, Spain. 2015.

J. Duro, V. Selfa, C. Gómez, M. E. Gómez, S. Petit, J. Sahuquillo, “Exploiting Bank Locaclity in Multiprogrammed Enviroments”, IEEE Transactions on Computers, to be submitted.

Bibliografía

[ALSJ09] Jung-Ho Ahn, J. Leverich, R.S. Schreiber, and N.P. Jouppi. Multicore dimm: an energy efficient memory module with independently controlled drams. *Comput. Archit. Letters*, 8(1):5–8, Jan 2009. 16

[EMLP09] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *Proc. of the 42Nd MICRO*, MICRO 42, pages 316–326, New York, NY, USA, 2009. ACM. 15

[GMMG12] Nagendra Dwarakanath Gulur, R. Manikantan, Mahesh Mehendale, and R. Govindarajan. Multiple sub-row buffers in dram: Unlocking performance and energy improvement opportunities. In *ICS*, pages 257–266, 2012. 16

[HGCT13] Enric Herrero, Jose Gonzalez, Ramon Canal, and Dean Tullsen. Thread row buffers: Improving memory performance isolation and throughput in multiprogrammed environments. *IEEE Trans. Comput.*, 62(9), 2013. 16, 17, 18

[HP12] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Plublisher Inc., 5 edition, Appendix E by T. Conte, 2012. 10

[JED] JEDEC website. 11

[KPMHB10] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceed. of the MICRO*, 2010. 15, 16

[LIMB09] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *ISCA*, pages 2–13, 2009. 16

[LMNP11] Chang Joo Lee, Onur Mutlu, Veynu Narasiman, and Yale N. Patt. Prefetch-aware memory controllers. *IEEE Trans. Comput.*, 60(10), 2011. 15

[mic11] Micron Technology, 4Gb: x4, x8, x16 DDR3 SDRAM, 2011. 23

[MLM12] Justin Meza, Jing Li, and Onur Mutlu. A case for small row buffers in non-volatile main memories. In *ICCD*, pages 484–485, 2012. 16

[MM07] Onur Mutlu and Thomas Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO*, pages 146–160, 2007. 15, 16

[MM08] Onur Mutlu and Thomas Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. In *ISCA*, pages 63–74, 2008. 15, 16

[RCBJ11] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Comput. Archit. Lett.*, 10(1):16–19, January 2011. 22

[RDK+00] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *ISCA*, pages 128–138, 2000. 15

[RZ97] T. Robinson and W.K. Zuravleff. Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order, 1997. US Patent 5,630,096. 15

[UMC+10] Aniruddha N. Udipi, Naveen Muralimanohar, Niladrish Chatterjee, Rajeev Balasubramonian, Al Davis, and Norman P. Jouppi. Rethinking DRAM Design and Organization for Energy-constrained Multi-cores. In *ISCA*, pages 175–186, 2010. 15

[USPL07] R. Ubal, J. Sahuquillo, S. Petit, and P. Lopez. Multi2sim: A simulation framework to evaluate multicore-multithreaded processors. In *SBAC-PAD*, pages 62–68, 2007. 21

[YJE11] Doe Hyun Yoon, Min Kyu Jeong, and Mattan Erez. Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput. *Comput. Archit. News*, 39(3):295–306, June 2011. 16

[YKK+13] P. Yedlapalli, J. Kotra, E. Kultursay, M. Kandemir, C.R. Das, and A. Sivasubramaniam. Meeting midway: Improving cmp performance with memory-side prefetching. In *International Conference on Parallel Architectures and Compilation Techniques*, 2013. 16

[ZLZ+08] Hongzhong Zheng, Jiang Lin, Zhao Zhang, E. Gorbatov, H. David, and Zhichun Zhu. Mini-rank: Adaptive DRAM architecture for improving memory power efficiency. In *MICRO*, pages 210–221, 2008. 16

[ZLZZ08] Hongzhong Zheng, Jiang Lin, Zhao Zhang, and Zhichun Zhu. Memory access scheduling schemes for systems with multi-core processors. In *ICPP*, pages 406–413, 2008. 15