Universidad de Cantabria

Departamento de Electrónica y Computadores

JERARQUÍA DE MEMORIA ESCALABLE PARA SISTEMAS MULTIPROCESADOR EN CHIP

Autor:

Pablo Prieto Torralbo

Directores:

Valentín Puente Varona

José Ángel Gregorio Monasterio

Santander, enero de 2014

University of Cantabria

Electronics and Computers Department

SCALABLE MEMORY HIERARCHY FOR CHIP MULTIPROCESSORS

Author:

Pablo Prieto Torralbo

Advisors:

Valentín Puente Varona

José Ángel Gregorio Monasterio

Santander, January 2014

Resumen

El número de transistores dentro del chip crece inexorablemente debido a la alta capacidad de integración posibilitada por el avance de la tecnología. Los arquitectos de computadores se esfuerzan por convertir estos avances en mejoras de rendimiento, mientras se enfrentan a los nuevos limitantes que llevan asociados. Así, en la última década, los avances en la arquitectura del procesador han sido reemplazados poco a poco por la integración de un mayor número de procesadores dentro de un mismo chip, siendo ya los multiprocesadores un estándar de los sistemas actuales.

Este tipo de sistemas suponen una solución eficiente a algunos de los limitantes tecnológicos encontrados, sin embargo requieren de un cambio en el modelo de programación, y trae sus propios problemas asociados. Es necesario que los arquitectos de computadores faciliten a los programadores de un entorno más intuitivo para la programación de aplicaciones paralelas que puedan ser ejecutadas en los distintos procesadores, con el fin de aprovechar al máximo las ventajas ofrecidas por este tipo de sistemas. Por otra parte, los sistemas multiprocesador no están exentos de condicionantes tecnológicos que limitan su efectividad. Así, aun cuando el incremento en el número de transistores integrados parece garantizar un aumento en el número de unidades de proceso y de memoria dentro del chip, las conexiones al exterior del chip son cada vez más escasas respecto al número de procesadores. Se hace necesario minimizar el número de accesos externos, incrementando la fracción del chip dedicada a la jerarquía de memoria y buscando mecanismos para una utilización más eficaz de los recursos disponibles. Las cada vez mayores estructuras de datos, y el aumento en el número de unidades de proceso hacen necesario encontrar soluciones escalables, siendo el ancho de banda a memoria fuera del chip, y a la jerarquía de memoria en el chip, factores clave para lograr un incremento en el rendimiento de los sistemas multiprocesadores actuales.

Durante el desarrollo de la tesis, hemos visto cómo han evolucionado los mecanismos para alcanzar mayores cotas de rendimiento, siendo cada vez más difícil obtener ventajas sustanciales mediante soluciones convencionales. En este tiempo se han abordado distintos componentes de la jerarquía de memoria, abarcando desde la jerarquía de cache

on-chip y la red de interconexión, hasta el controlador de memoria y el arbitraje de las peticiones fuera del chip.

A lo largo del presente documento se intentan exponer, de forma clara, los problemas y soluciones encontrados en los distintos componentes de la jerarquía de memoria, siempre buscando alternativas eficientes que aumenten la escalabilidad dentro de los requerimientos propios de este tipo de sistemas. Las propuestas presentadas tratan de ser correctas en forma y fondo, comparando sus efectos con propuestas semejantes en un entorno de simulación lo más completo posible.

Abstract

The number of transistors available on a chip grows inevitably due to the high integration capability enabled by technological advances. Computer Architects strive to convert these advances into performance improvements, while facing new associated constraints. Thus, in the last decade, advances in processor architecture have gradually been replaced by integration of a larger number of processors within a single chip, the multiprocessor being a standard in current systems.

Multiprocessor systems represent an efficient solution to some of the technological problems encountered; however, they require a change in the programming model, which brings its own associated issues. Computer Architects need to provide programmers with a more intuitive environment for programming parallel applications that can be executed on different processors, in order to maximize the advantages offered by these systems. Moreover, multiprocessor systems are not without technological constraints that limit their effectiveness. Thus, even if the increase in the number of integrated transistors seems to ensure an increment in the number of memory and processing units within the chip, the off-chip connections are becoming more and more scarce compared to the number of processors. It is necessary to minimize the number of external accesses, increasing the fraction of the chip devoted to the memory hierarchy and requiring mechanisms that provide effective use of available resources. The growing data structures and the increase in the number of processing units make it necessary to find scalable solutions, off-chip bandwidth and on-chip memory hierarchy being, key factors in order to achieve an improvement in the performance of current multiprocessor systems.

During the development of the thesis, we have seen how much the mechanisms used to achieve higher levels of performance have evolved, making it more difficult to obtain substantial benefits using conventional solutions. Over this period, we have addressed different components of the memory hierarchy, ranging from the on-chip cache hierarchy and interconnection network, to the memory controller and the arbitration of off-chip requests.

This document will attempt to clearly explain, problems and solutions found in various components of the memory hierarchy, always with the aim of finding efficient ways to

increase the scalability while bearing in mind the specific requirements of such systems. The proposals seek correctness by comparing their effects with similar solutions in a simulation environment that is as complete as possible.

Agradecimientos

La culminación de esta tesis doctoral ha sido un proceso largo y costoso, y no habría sido posible sin el apoyo y comprensión de las personas que me rodean. El camino no ha sido fácil y requiere de trabajo y resistencia, pero los momentos de satisfacción que se consiguen con la investigación compensan todo lo sufrido. Esto no siempre es fácil de explicar, y por eso quiero dedicar unas líneas a todos aquellos que han estado conmigo en estos años.

Las primeras líneas deben ir obligatoriamente a mi familia, y en primer lugar a mis padres, Juan Antonio y Gloria, a los que me debo. Sin ellos no sería la persona que soy, me han educado y se han preocupado de mi educación, y difícilmente habría llegado a donde estoy sin su ayuda y ejemplo. No siempre es fácil transmitir lo que haces cuando te dedicas a esto, pero ellos siempre me han apoyado sabiendo que era mi "vocación". Mis hermanos, Alfonso y Susana han sido también un gran apoyo en estos años, duros en algunos momentos. Uno ha sido un espejo al que mirarse toda mi vida, abriendo caminos y facilitando mis pasos, muchas gracias hermano. La otra ha sido una luz que ha iluminado la vida de los que la rodearon, y que ahora sigue acompañando a los que lamentamos su falta, nunca me olvidare de ti, Susi. Y aunque no puedo nombrarlos a todos pues tengo una familia amplia, me gustaría extender este agradecimiento a toda mi familia y a los que ya forman parte de ella, Jaione, Javier, Ana y Javier, porque sé que siempre puedo contar con vosotros.

Debo dar las gracias también a mis amigos que, en la cercanía o la distancia, han estado conmigo todo este tiempo, y han sido una vía de escape en los momentos más duros y me han sostenido con fuerza. No siempre ha sido fácil juntarnos, pero nunca me han faltado risas y palabras de ánimo cuando las he necesitado. Muchas gracias a todos, prometo dedicaros más tiempo y hacer esos viajes que tengo apalabrados desde hace tanto.

Y aunque ya son casi mi segunda familia, quiero dedicar unas líneas a parte a mis compañeros de trabajo durante todos estos años. Muchas horas al día juntos, cafés y viajes, no serían lo mismo con otras personas. Quiero agradecer especialmente a Jose,

nuestro system-manager, sin el cual mi trabajo difícilmente sería posible. Gracias también a Pablo, Javier, Lucía y Adrián, con los que he tenido la oportunidad de trabajar codo con codo, y demuestran con su esfuerzo diario que una universidad pequeña merece aparecer en el mapa, aunque no siempre sea fácil.

Gracias también por supuesto a mis directores de tesis, Valentín y José Ángel, responsables de las ideas y publicaciones que hay detrás de esta tesis. Sin su paciencia y apoyo habría sido imposible llevarla a cabo, y su exigencia y consejo me han cambiado como investigador y como persona. Espero que mi trabajo haya conseguido estar a la altura y cumpla sus expectativas.

Finalmente quiero dedicar esta tesis a una persona que ha estado junto a mí desde el comienzo y que ha vivido más de cerca todos estos años de trabajo. Muchas gracias Beatriz, porque esto no habría sido lo mismo sin tu apoyo, y por la paciencia que has demostrado aguantando los momentos más duros de la tesis, siendo el hombro en el que apoyarme y la palabra amable que todo lo sana. Has vivido conmigo los momentos más duros y más bonitos, y tu comprensión y cariño han sabido llevarme adelante. Te quiero, guapa.

Tabla de Contenidos

Lista de Figuras

Lista de Tablas

1 Introducción

1.1 Introducción

Siguiendo las pautas marcadas por Gordon Moore en 1965 cuando enunció su famosa predicción "el número de transistores en un chip se duplicará cada dieciocho meses aproximadamente", la industria ha creado su propio camino, buscando duplicar el rendimiento de los procesadores aproximadamente cada dos años. Así, hasta principios de siglo, el ritmo de crecimiento en el rendimiento de los procesadores se situaba cerca del 52% anual, obtenido principalmente gracias a dos ventajas. Por una parte, el aumento de la frecuencia de reloj, propiciada por el aumento de las etapas del pipeline y las mejoras tecnológicas. Por otra parte, la explotación del paralelismo a nivel de instrucción mediante arquitecturas más avanzadas (RISC, superescalares, ejecución fuera de orden, ejecución especulativa…), gracias al aumento del número de transistores que es posible integrar en un chip.

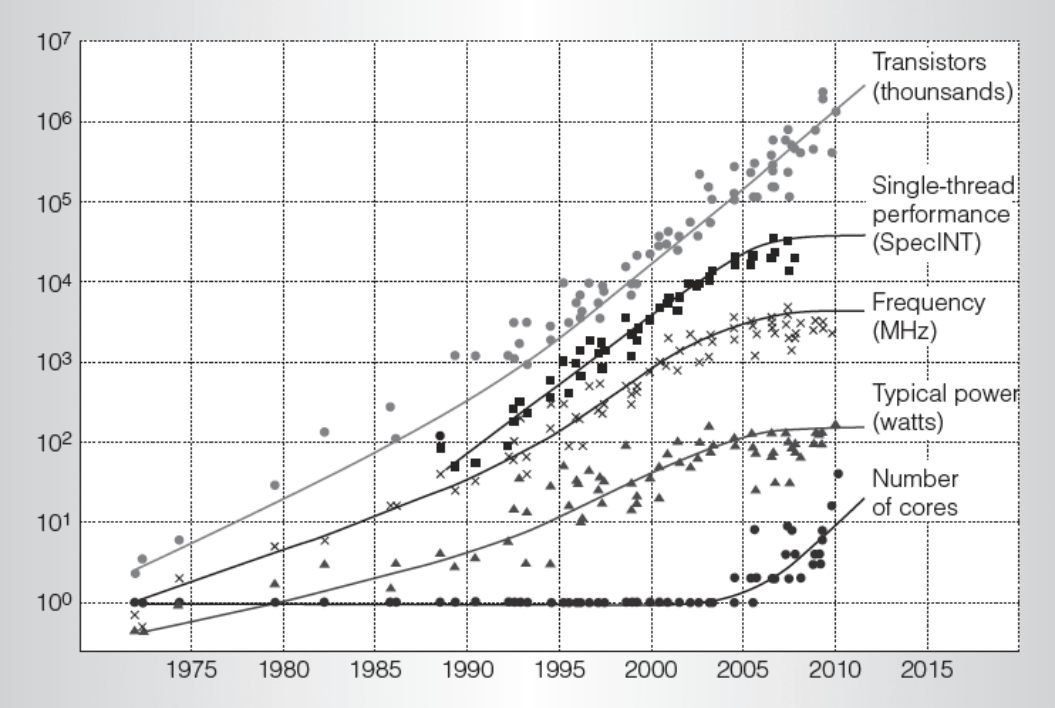

Figura 1-1. Tendencia del número de transistores, frecuencia de reloj, número de núcleos y rendimiento de los microporcesadores a lo largo del tiempo [1].

Desde mediados de los 80, la frecuencia de reloj ha crecido aproximadamente un 30% anual, gracias al aumento de velocidad de conmutación con la mejora de la tecnología (de 1µm a 90 nm) y la reducción en el número de puertas que deben atravesarse en cada ciclo. Así, teóricamente, el pipelining del procesador permite un incremento en la frecuencia de reloj equivalente al número de etapas del mismo, aunque en realidad este beneficio se ve atenuado por la subutilización de dichas etapas y la penalización de los latches necesarios entre cada una de ellas. Actualmente es difícil extraer beneficio del incremento del número de etapas del pipeline, y aunque la reducción de la tecnología permitiría seguir aumentando la frecuencia de trabajo, hacerlo supondría un incremento importante en el consumo de potencia, uno de los grandes limitadores en el diseño de arquitecturas de computadores a día de hoy. Las leyes de escalado de Dennard [2] constatan que al reducir a la mitad las dimensiones lineales de un transistor (escalando igualmente el voltaje y la corriente a la mitad), es posible reducir su consumo de potencia en un factor 4, manteniendo por tanto la densidad de potencia constante. Sin embargo, a medida que se reduce el voltaje umbral y las dimensiones del transistor, aumenta la corriente de fugas. Esto, además de aumentar el consumo de potencia del chip, incrementa su temperatura, lo que provoca un aumento de la corriente de fugas retroalimentándose, además de añadir problemas de fiabilidad. Como solución, se ha limitado la reducción en el voltaje umbral con el fin de mitigar los efectos de la corriente de fugas, lo que restringe el incremento en la velocidad del transistor. Así, podemos observar en la Figura 1-2 como la frecuencia de reloj apenas ha variado en la última década, después de haber seguido un crecimiento exponencial.

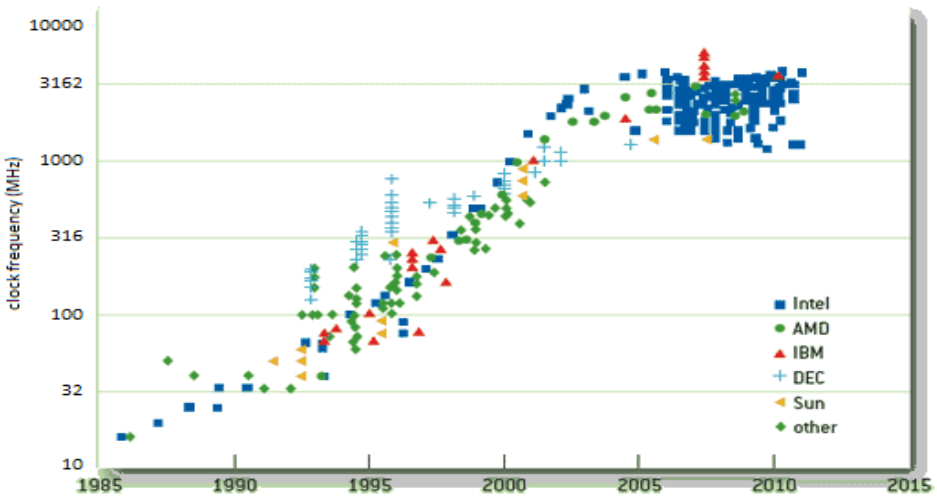

Figura 1-2. Escalado de la frecuencia del procesador con el tiempo [3].

Sin embargo, el número de transistores disponibles dentro del chip sigue en aumento y es necesario seguir buscando soluciones para incrementar el rendimiento de los procesadores. Con el gran número de transistores disponibles, es posible trasladar soluciones microarquitecturales más complejas que tratan de explotar el paralelismo a nivel de instrucción (ILP). Así, los procesadores superescalares, la ejecución fuera de orden, la ejecución especulativa y la predicción de saltos, permiten la ejecución simultánea de varias instrucciones, manteniendo el crecimiento en el rendimiento de los procesadores. Sin embargo, al igual que hemos visto con la profundidad del pipeline, estas soluciones tienen un límite, puesto que las aplicaciones tienen un paralelismo a nivel de instrucción limitado.

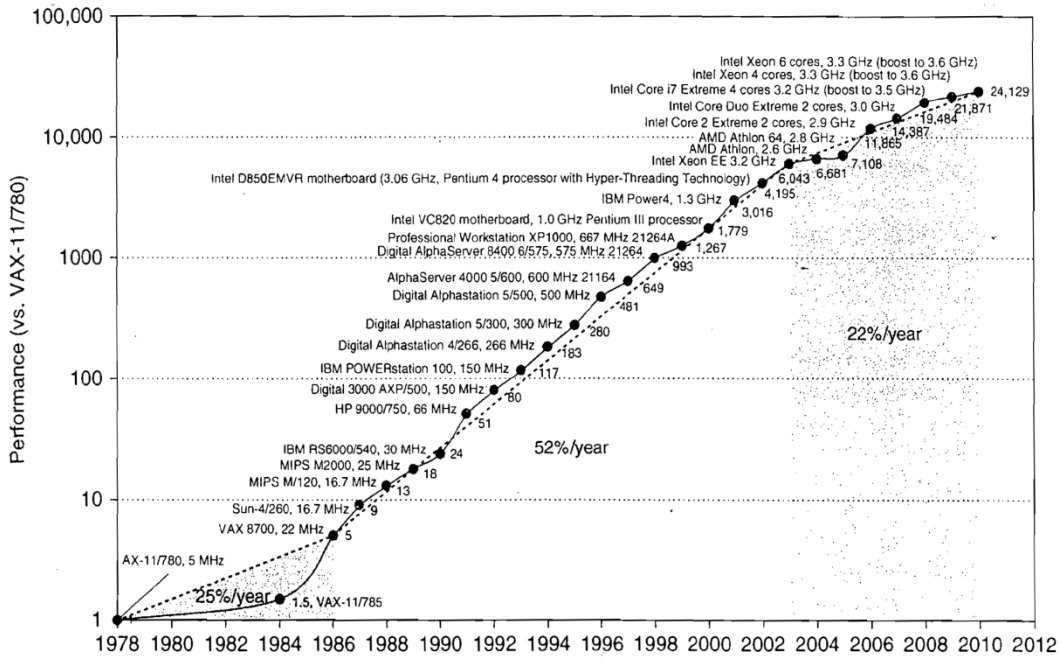

Figura 1-3. Crecimiento en el rendimiento de los procesadores desde finales de los 70 [4]**. Las medidas se corresponden con el comportamiento de un único núcleo ejecutando la suite de benchmarks SPEC.**

Tal y como se aprecia en la Figura 1-3, en 2003 el crecimiento en el rendimiento de los procesadores sufre una deceleración. La imposibilidad de aumentar la frecuencia de trabajo sin que la potencia se dispare, y el hecho de que cada vez sea más difícil conseguir un mayor paralelismo a nivel de instrucción sin complicar en exceso el diseño, hacen que sea necesario buscar alternativas dentro de la industria.

Así surge el paralelismo a nivel de aplicación, o *Thread Level Parallelism* (TLP), como alternativa de diseño. La idea es antigua, y se apoya en el hecho de que las aplicaciones que se ejecutan en un procesador pueden ser divididas, explícitamente por el

programador, en tareas más sencillas que pueden ser ejecutadas de forma simultánea en varios núcleos. Los multiprocesadores pueden aprovecharse de esta característica, y así los Multiprocesadores en Chip (CMP) pueden seguir incrementando el rendimiento sin hacer uso exclusivo de modificaciones como el aumento de frecuencia o la profundidad y el ensanchamiento del pipeline.

1.2 Multiprocesadores en Chip

Motivado por la disponibilidad de recursos, el hecho de que, con el aumento de la integración de transistores dentro del chip, sea posible incorporar múltiples procesadores en un solo chip donde antes sólo cabía uno, hacen de esta solución el paradigma central en el diseño de la arquitectura de computadores actual. Dividir el problema en porciones más pequeñas, que pueden ser resueltas simultáneamente, supone una excelente solución al problema de la escalabilidad en el rendimiento de los procesadores individuales. Idealmente, duplicar el número de procesadores podría reducir a la mitad el tiempo de ejecución de una aplicación sin aumentar la frecuencia de reloj. Dado que el aumento en el consumo de potencia debido a duplicar la frecuencia es mucho mayor que el asociado a duplicar el número de unidades de proceso, la ejecución paralela parece una solución apropiada al diseño de sistemas actuales.

Los primeros CMP surgen a mediados de la pasada década, y en su mayor parte consisten en replicar, dos o más veces, arquitecturas convencionales ya probadas, en un solo chip. Con el tiempo, el número de procesadores incluidos en cada chip ha aumentado, llegando a las decenas y centenares de núcleos [5]–[7].

Aun cuando las ventajas de escalabilidad de los CMP son claras, deben enfrentarse a una serie de problemas para alcanzar el rendimiento esperado. Hasta la llegada de los multiprocesadores, los incrementos en rendimiento de los procesadores eran transparentes a los programadores, a los que se les ocultaba como un único procesador que ejecuta instrucciones en orden cada vez más rápido. Sin embargo, este modelo de abstracción ya no es posible, y es necesario exponer el paralelismo a los programadores para poder mantener el crecimiento en rendimiento previsto.

Idealmente, duplicar el número de procesadores implica duplicar el rendimiento obtenido del sistema, sin embargo esto difícilmente es así en la práctica. La programación paralela no es una tarea sencilla, siendo necesario plantear mecanismos para mantener una coherencia secuencial, así como coordinar el acceso a los recursos compartidos.

Según cómo se afronten estos problemas podemos hablar de dos formas de programación paralela:

- Por paso de mensajes: en este caso la comunicación entre componentes es explícita, obligando a los programadores a manejar las comunicaciones de forma expresa.
- Memoria compartida: en cuyo caso la comunicación se produce al modificar la capa de memoria compartida, a través de la lectura y escritura de datos.

Mantener un espacio común de memoria facilita el trabajo al que se enfrentan los programadores, y lo hace un modelo más intuitivo para la programación de aplicaciones paralelas. Por eso, la arquitectura de memoria compartida es la más empleada en los CMP de propósito general, aunque ello requiere de sistemas capaces de mantener la coherencia y consistencia de los datos para los distintos procesadores dentro del chip.

Además de las limitaciones de uso asociadas a los CMP, existen una serie de limitantes tecnológicos asociados a los elementos compartidos, y en concreto uno de los más relevantes es el acceso a la memoria principal fuera del chip.

1.3 Memory Wall

El incremento de la complejidad del sistema dentro del chip, ha permitido un aumento en el flujo de instrucciones (IPC). Sin embargo, la velocidad de respuesta de la memoria principal no crece al mismo ritmo, provocando que su elevada latencia limite el rendimiento de los procesadores como se puede ver en la Figura 1-4. Este problema, conocido como *memory wall*, ha motivado la búsqueda de soluciones que mitiguen el impacto de los accesos fuera del chip.

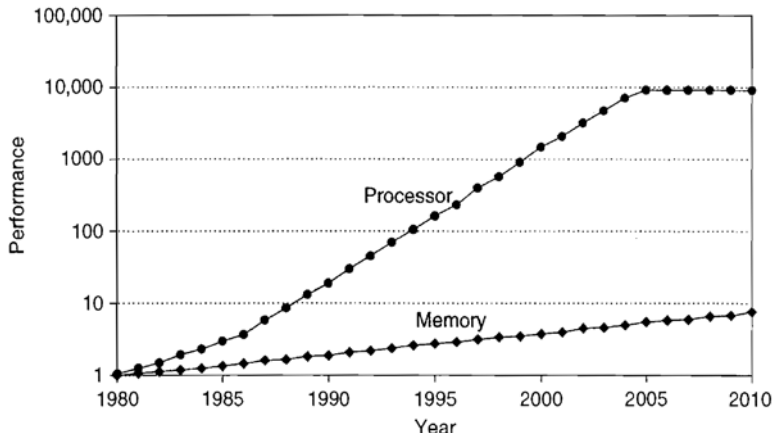

Figura 1-4. Diferencia en el rendimiento de la memoria respecto al procesador [4]**.**

Algunas de las formas más habituales de enmascarar la latencia de los accesos fuera del chip implican ejecución especulativa, *prefetching* y una eficiente jerarquía de cache *on-chip*. Sin embargo, las primeras soluciones suponen un aumento en el número de peticiones a la jerarquía por unidad de tiempo, en algunos casos con datos no útiles, lo que deriva en problemas de saturación del ancho de banda tanto dentro como fuera del chip. Por otra parte, es posible hacer un uso más eficiente de la cache *on-chip*, absorbiendo el impacto de las medidas anteriores a la vez que se amortigua la latencia a memoria.

El aumento de la capacidad de la cache en el chip es tanto una realidad como una necesidad. La alta capacidad de integración de las tecnologías en estudio, unido a técnicas como el apilamiento vertical (*3D stacking*), la DRAM *on-chip* o la memoria no volátil *on-chip* [8], [9], permitirían incluir grandes cantidades de memoria en el interior del chip. Sin embargo, un aumento de la capacidad de la cache implica generalmente un aumento en la latencia, es por esto que se hace necesario crear mecanismos que reduzcan la latencia de acceso a la cache o la mitiguen. Así, una solución frecuente es la fragmentación de la cache en distintos niveles que aumentan de capacidad y latencia progresivamente. Igualmente, surgen propuestas que aprovechan la alta capacidad de integración para aumentar la complejidad de la cache con alternativas más eficientes.

El problema del *memory wall* viene de lejos, como se puede ver en la Figura 1-4. Sin embargo, en la era de los multiprocesadores en chip, no solo el tiempo de acceso a la memoria principal supone un problema en el rendimiento, sino que la creciente presión por parte de un número más alto de CPU en el chip, convierten al ancho de banda fuera del chip en un cuello de botella.

1.4 Muro del Ancho de Banda a memoria

Con el tiempo, el número de transistores disponibles dentro de un chip sigue multiplicandose, y en general, incrementar el número de procesadores en un CMP y su complejidad, supone aumentar el tráfico a memoria principal. Esto implica que el ancho de banda a memoria *off-chip* debe incrementarse si queremos mantener el rendimiento global del sistema. De no ser así, la latencia adicional de los accesos fuera del chip, debido a la contención, reducirá el rendimiento de los procesadores del chip hasta que estos equiparen el del ancho de banda a memoria.

Desafortunadamente, la cuenta de pines fuera de chip está previsto que aumente en un 10% anual [10], mientras el número de procesadores dentro del chip puede llegar a duplicarse cada 18 meses. A día de hoy, esto supone que la tasa a la que crece el tráfico a memoria es mucho mayor que la capacidad de que sea servido, lo que implica un decremento en el rendimiento global del sistema y es lo que se conoce como el muro del ancho de banda a memoria o *Bandwidth Wall*. Es razonable pensar que este problema se acreciente en el futuro a medida que la capacidad de integración dentro del chip aumente sustancialmente, mientras las conexiones al exterior apenas mejoran, lo que se traduce en un aumento de la latencia fuera del chip y por tanto del problema del *Memory Wall*.

Aun cuando es posible aumentar la capacidad dentro del chip con el fin de mitigar estos efectos, y existiendo tecnologías que incrementan el ancho de banda fuera del chip [11], esto no soluciona el problema, sino que lo retrasa. El problema seguirá existiendo en el futuro según el número de procesadores dentro del chip siga creciendo, de manera que la contención en el acceso fuera del chip suponga el mayor cuello de botella en el rendimiento del sistema, aumentando considerablemente la ya de por si elevada latencia de acceso a memoria off-chip. Así, parte de las mejoras que se realicen en los futuros procesadores deben de hacer un uso eficiente del ancho de banda a memoria con el fin de que escalen.

1.5 Contribuciones de la Tesis

En esta tesis nos centraremos en los problemas derivados de la escalabilidad que afectan a la jerarquía de memoria, y en especial los que mitigan el impacto del *Memory Wall*, relacionado con la latencia de acceso, así como el *Bandwidth Wall*.

Las principales contribuciones de esta tesis son:

- Un análisis del impacto que tienen, en el comportamiento general del sistema, distintos parámetros de la jerarquía de cache en sistemas multiprocesador, como son el emplazamiento de los procesadores y la distribución de la capacidad de la cache.
- Un sencillo modelo analítico para evaluar el impacto en el rendimiento que puede tener la distribución de capacidad en los distintos niveles de cache, en sistemas multiprocesador con memoria compartida, y que permite filtrar el espacio de diseño inicial.

- Una arquitectura eficiente para el último nivel de cache que mitiga los efectos del *Memory Wall*, reduciendo la latencia media de acceso a la cache, a la vez que reduce la interferencia entre procesadores propia de los sistemas CMP. Esta arquitectura se apoya en la Arquitectura de Cache compartida de Acceso No-Uniforme (NUCA), pero garantiza, de forma dinámica, una porción de cache privada a cada procesador de acuerdo a sus necesidades de compartición de la aplicación en ejecución, a la vez que garantiza un uso eficiente de la capacidad disponible.
- Un sistema de directorio incompleto para almacenar grandes cantidades de bloques en poco espacio mediante *tags* parciales y control de saturación. Este sistema es utilizado para limitar los posibles destinatarios de las peticiones *broadcast* en un protocolo de coherencia, reduciendo el consumo de ancho de banda dentro del chip y el número de peticiones sobre los bancos de cache. El directorio incompleto, o filtro, puede dar falsos positivos, pero de no ofrece falsos negativos, y dispone de un sistema para evitar la saturación sin necesidad de reinicio.
- Un algoritmo de arbitraje para los controladores de memoria, que consigue minimizar el impacto en el rendimiento en caso de saturación del ancho de banda off-chip, a la vez que reduce los efectos de la interferencia entre procesadores. Este algoritmo propone el uso de la información disponible en los procesadores fuera de orden de una forma novedosa, para que, mediante mecanismos sencillos, sea posible ordenar las peticiones a memoria de forma eficiente, tanto en rendimiento como en uso equilibrado de los recursos.

Todas las contribuciones a esta tesis han sido publicadas en congresos y revistas internacionales con revisión por pares. Los artículos publicados abarcan distintos componentes de la jerarquía de memoria como la jerarquía de cache en el chip [12], [13], la red de interconexión en sistemas multiprocesador [14]–[16] o el ancho de banda a memoria fuera del chip [17]. Igualmente, se ha contribuido en el desarrollo de una herramienta de simulación de redes de interconexión de ejecución paralela con alto nivel de detalle y ejecución integrada en simuladores de sistema completo [18].

1.6 Contenido de la Tesis

La organización de los restantes capítulos de la tesis es la siguiente:

- El capítulo 2 sirve como punto de partida para describir las características de la jerarquía de memoria en los sistemas multiprocesador actuales, centrándonos en aquellas con las que vamos a trabajar a lo largo de la tesis. Se incluye además una descripción del entorno de simulación empleado durante el desarrollo de los distintos experimentos, así como las aplicaciones y métricas utilizadas en la evaluación de las propuestas de la tesis.
- El capítulo 3 se centra en el análisis de algunos parámetros importantes en el diseño de sistemas multiprocesador, que incluye la distribución de los procesadores y su efecto en el rendimiento global del sistema, y la distribución de la capacidad de cache disponible entre los distintos niveles de una jerarquía, presentando un modelo analítico sencillo.
- El capítulo 4 describe una arquitectura denominada SP-NUCA, que permite un uso eficiente de la gran capacidad disponible en el último nivel de la jerarquía de memoria *on-chip*, a la vez que reduce su latencia de acceso. Se presentan además soluciones a distintos problemas a los que se puede enfrentar una arquitectura de este tipo.
- En el capítulo 5 se presenta un algoritmo de control del ancho de banda a memoria (DROB), basado en la criticidad de las instrucciones en un procesador con ejecución fuera de orden. Este mecanismo, de fácil implementación, ayuda a mitigar uno de los grandes problemas que puede limitar la escalabilidad de los multiprocesadores *on-chip* de próxima generación (el *Bandwidth Wall*).
- Finalmente, el capítulo 6 resume las principales conclusiones obtenidas en este trabajo, y propone futuras líneas de investigación, como la aplicación de nuevas tecnologías de integración en la jerarquía de memoria de una forma viable.

2 Entorno y Motivación

La elevada latencia de los accesos a memoria fuera del chip supone un limitante considerable en el rendimiento de los sistemas multiprocesador en chip. La notable diferencia entre el rendimiento de la memoria y los procesadores, unido a las restricciones físicas impuestas por la tecnología, lo convierten en un problema difícil de abordar. Muchas son las propuestas que tratan de enmascarar la elevada latencia de acceso fuera del chip, bien desde el punto de vista del procesador (ejecución fuera de orden, prefetch, especulación…) como de la jerarquía de cache (reducción de la tasa de fallos, escalonado de la latencia…).

Sin embargo, al problema de la latencia hay que añadir el del ancho de banda a memoria. Con una proporción cada vez menor del ancho de banda al exterior del chip disponible por procesador y unos mayores requerimientos por parte de estos, el número de peticiones simultáneas se incrementa sin que la memoria principal pueda atenderlas al ritmo deseado. Este hecho provoca un incremento en la congestión en el acceso a la memoria principal, lo que, a su vez, agrava el problema de la latencia de los accesos fuera del chip. De nuevo existen soluciones que tratan de abordar el problema, bien reduciendo el número de fallos a memoria, aumentando la capacidad de la cache en el chip (alternativas a la SRAM, compresión de los datos en cache, apilamiento vertical…), bien disminuyendo el uso del ancho de banda (compresión de los datos a través del enlace, bloques de cache más pequeños…). Lamentablemente, estas soluciones no eliminan el problema, lo retrasan en el tiempo y es previsible que limite la escalabilidad y/o la agresividad de los procesadores en futuras generaciones.

El trabajo de esta tesis está centrado en la jerarquía de memoria para los CMP de las futuras generaciones, en particular, en sistemas de propósito general con procesadores homogéneos. Se pone el foco en el problema del *Memory Wall* y el ancho de banda fuera del chip, y las que serán las claves para afrontar este problema en el futuro de los CMP, tales como el manejo eficiente de los recursos disponibles dentro del chip, en constante crecimiento, así como el arbitraje en la competición por los recursos fuera de él.

A lo largo de este capítulo haremos una breve descripción de decisiones arquitecturales básicas de la jerarquía de memoria en un CMP, para familiarizarnos con los distintos términos y elementos que iremos utilizando a lo largo de la tesis. Haciendo especial hincapié en los que afectan a la porción compartida de la jerarquía de memoria en los CMP y su influencia en la escalabilidad. Haremos también una descripción detallada de las herramientas utilizadas durante el desarrollo de la tesis para validar y cuantificar las ventajas de las ideas propuestas mediante su implementación. Dado que tienen un alto nivel de detalle y su manejo y dominio entrañan una elevada dificultad hemos considerado oportuno incluir una breve descripción de sus elementos. De la misma forma, haremos una breve descripción de las cargas de trabajo empleadas durante los procesos de evaluación, que tratan de cubrir un amplio abanico de aplicaciones.

2.1 CMP de Memoria Compartida

A lo largo del capítulo anterior, se han introducido los avances tecnológicos en Arquitectura de Computadores y cómo estos apuntan a los CMP como la línea a seguir en la actualidad y el futuro cercano. De una forma más específica, los Multiprocesadores con memoria compartida son una solución frecuente, con la intención de simplificar algunos de los problemas que presenta la programación paralela [19], a la vez que permite un uso más eficiente de los recursos. Sin embargo, este tipo de arquitecturas se enfrenta a una serie de problemas específicos como son el mantenimiento de la coherencia entre los distintos niveles de la jerarquía de memoria, la interferencia entre los distintos procesadores y las altas latencias provocadas por las grandes estructuras comunes.

La creciente capacidad de integración en el chip, proporcionada por la tecnología, convierte a los multiprocesadores en sistemas extremadamente complejos. Las aplicaciones emergentes evolucionan hacía estructuras cada vez más demandantes en datos y la jerarquía de memoria debe ser capaz de absorber estas necesidades. Las mejoras proporcionadas por la tecnología y los avances arquitecturales, unido a las limitaciones expuestas en el capítulo anterior, hacen difícil definir con precisión cuál será el futuro de los microprocesadores. Sin embargo, es razonable esperar que el manejo eficiente de las aplicaciones paralelas, la reducción de la interferencia entre los distintos procesadores, y la escalabilidad de los recursos disponibles serán consideraciones a tener en cuenta.

Disponer de un sistema de memoria compartida implica mantener un modelo de consistencia y coherencia que permita al programador de aplicaciones ver el hardware lo más trasparente posible, asemejándose a un sistema de un solo núcleo.

2.2 Privado-Compartido

Una decisión fundamental de diseño en los sistemas multiprocesador es la forma de distribuir la capacidad de la cache y cómo los distintos núcleos pueden acceder a los datos de la misma. Así, podemos encontrar dos arquitecturas fundamentales: arquitecturas de cache privada y arquitecturas de cache compartida. En el primer caso, cada núcleo dispone de su propia porción de la cache, localizada físicamente próxima al procesador. Este tipo de arquitectura se beneficia de una baja latencia para los bloques de memoria que se encuentran en su porción de la cache y de uso privado, es decir, utilizados únicamente por un procesador, y reduce la interferencia entre procesadores. Por el contrario, tiene que lidiar con los problemas de los datos compartidos, bien sea mediante la réplica de datos (reduciendo la capacidad efectiva de la cache) o la migración, lo que suele incrementar la latencia, y además puede provocar un uso ineficiente de los recursos en el caso de que alguno de los procesadores no haga uso completo de su porción de cache. Por otra parte, en una arquitectura de cache compartida no es necesario replicar los datos que tienen múltiples propietarios, a costa de incrementar sustancialmente la latencia de acceso a los mismos. Además, los sistemas de memoria compartida permiten un uso más eficiente de la cache en caso de desbalanceo de carga, aunque incrementa la interferencia entre procesadores debido a los reemplazos.

Existen múltiples propuestas que abordan el problema de la distribución de la capacidad en los CMP, especialmente ante la perspectiva de un incremento sustancial en la capacidad de almacenamiento dentro del chip. Este tipo de soluciones se apoyan en una de las arquitecturas básicas, privada o compartida, y ajustan su funcionamiento para mitigar los problemas que llevan asociados. Así, encontramos arquitecturas de cache privada que intentan controlar el número de *réplicas* (copias de datos compartidos en distintas caches privadas), o aprovechar el espacio no utilizado por otros procesadores mediante el uso de *víctimas* (datos privados de un procesador que son almacenados fuera de su porción privada de cache). Y sistemas de cache compartida que tratan de reducir la latencia de acceso a los datos.

Una solución muy extendida para las arquitecturas de cache compartida consiste en trasladar la idea de los sistemas multiprocesador de memoria lógicamente compartida pero físicamente distribuida (NUMA [20]) a la cache dentro del chip, de forma que la cache se divide en bancos pequeños de rápido acceso unidos por una red de interconexión. Esto reduce el tiempo de acceso medio a los datos de la cache y aumenta su ancho de banda. La solución que se utiliza a lo largo de la tesis es la arquitectura de cache de acceso no uniforme (NUCA)[21], y como veremos a lo largo de la misma, de esta arquitectura básica han surgido distintas soluciones.

2.3 Arquitectura de Cache de Acceso No Uniforme

Las Arquitecturas de Cache de Acceso No Uniforme (NUCA), son una solución altamente escalable, lo que supone una gran ventaja ante las perspectivas de crecimiento de los recursos en el interior del chip. La intención de una arquitectura NUCA es distribuir la capacidad de la cache en bancos pequeños de rápido acceso conectados mediante una red punto a punto, independizando su distribución del número de *cores*. Gracias a esto se consigue una reducción significativa en la contención de los bancos de cache, dado que las peticiones se reparten entre un mayor número de ellos, aumentando la capacidad de respuesta a costa de un incremento en el área. Además, al distribuir los bancos en bancos más pequeños, estos tienen una menor latencia, habiendo bancos cercanos al procesador que tengan una latencia menor que una cache de acceso uniforme, y bancos lejanos con una latencia mayor debido al paso por la red de interconexión. Esto permite, explotando la localidad de las comunicaciones, obtener una latencia promedio más baja. Sin embargo, esta solución tiene un límite, de forma que no es posible incrementar el número de bancos indefinidamente reduciendo su capacidad, pues se aumenta el área requerida al elevar el número de controladores, así como el overhead de coste y latencia de la red de interconexión. En comparación con una cache de acceso uniforme, la NUCA consigue un mayor ancho de banda en el número de peticiones que pueden ser atendidas simultáneamente, a la vez que reduce la latencia media de acceso con una red de interconexión adecuada, pasando a ser no uniforme.

Hay que tener en cuenta que una cache de acceso uniforme tiene una red de interconexión, normalmente un árbol en forma de H, de manera que el tiempo de acceso a todos los bancos es uniforme (los bancos se encuentran en los extremos del árbol). Incluir una red con latencia no uniforme, como puede ser una red tipo malla o toro, requiere

aumentar la complejidad de los encaminadores, pero es el precio que hay que pagar para mejorar la escalabilidad del subsistema de memoria, existiendo además soluciones que minimizan este coste [22], [23].

En las distintas propuestas que se abordan a lo largo de la tesis suponemos que la arquitectura del último nivel es una cache compartida de acceso no uniforme (NUCA), aun cuando algunas de las propuestas son ortogonales a este tipo de diseño. En una arquitectura de este tipo, los datos que se encuentren en los bancos más próximos al procesador serán atendidos con menor latencia al tener que atravesar un menor número de encaminadores en la red. Por otra parte, aquellos datos que se encuentren en el extremo más alejado tienen una latencia considerablemente superior. Existen múltiples trabajos que abordan el problema del emplazamiento de los datos en arquitecturas de este tipo [12], [21], [24]–[26], y en el capítulo 4 proponemos una alternativa eficiente y de fácil implementación.

2.4 Inclusividad/Exclusividad del Último nivel de Cache on Chip

Centrándonos en un último nivel de cache (LLC) compartido, todavía quedan aspectos importantes a definir en su concepción. Una de las decisiones fundamentales, es si es necesario forzar la inclusividad en el último nivel de cache o no. Existen tres formas de entender la inclusividad en el último nivel de cache, teniendo cada una sus beneficios y contrapartidas.

En primer lugar, las caches inclusivas tienen una copia de todos los datos pertenecientes a los niveles superiores (más cercanos al procesador) de la jerarquía, lo que se garantiza mediante la invalidación de todos los bloques privados que sean expulsados del último nivel. Esto reduce la capacidad global dentro del chip, ya que los datos que se encuentran en las caches privadas de los procesadores tienen una copia innecesaria en el último nivel de cache, lo que supone una cache efectiva igual a la del último nivel de cache, pero por otra parte simplifica el protocolo de coherencia a la vez que mejora la búsqueda de datos en un sistema multiprocesador pudiendo reducir el consumo de ancho de banda y energía. Una versión opuesta, la cache exclusiva, implica que los datos existentes en los niveles privados no pueden tener una copia en la cache de último nivel, de forma que ésta se convierte en una cache de *víctimas* de las caches privadas. Esto implica un aprovechamiento máximo de la capacidad de la cache, puesto que la capacidad

efectiva es igual a la suma de la capacidad del último nivel de cache más la de las caches privadas. Sin embargo, este sistema complica la localización del dato, lo que podría limitar su escalabilidad. Finalmente, existe un sistema intermedio, no-inclusivo, en el que los datos son almacenados tanto en el nivel privado de cache como en el último nivel compartido, aunque una expulsión en este último no implica la invalidación de las copias privadas. La capacidad de este tipo de sistema oscila entre el de la cache inclusiva y la exclusiva, en función del criterio que se utilice en el último nivel de cache respecto a los datos replicados en los niveles privados. De la misma forma, la dificultad asociada a encontrar un dato viene determinada por el mismo criterio.

Tal y como demuestra Jaleel et. al. en su artículo [27], la desventaja asociada a la inclusividad no es debida tanto a la supuesta reducción de la capacidad efectiva como a los reemplazos de bloques privados que están siendo utilizados. Así, cuando lo capacidad de la cache del último nivel es muy superior a la de los niveles inferiores (4 veces o más), la cache inclusiva se comporta prácticamente igual que una no-inclusiva, como demuestra la decisión por la inclusividad en el Intel core-i7 “Haswell” [28] (con LLC 8 veces el agregado de las privadas), mientras que AMD opta por la no-inclusividad en su AMD FX Opteron, “Bulldozer” [29], (donde la LLC tiene la misma capacidad que el agregado de las privadas). El problema de las víctimas forzadas en los niveles privados es debido a que el algoritmo de reemplazo del último nivel de cache pierde información rápidamente sobre lo que ocurre en la cache privada, expulsando bloques que pueden estar siendo usados. El problema puede ser fácilmente solventado en el algoritmo de reemplazo como demuestra Jaleel en su trabajo, obteniendo resultados en rendimiento semejantes a los de una cache exclusiva sin perder las ventajas que aporta la inclusividad.

2.5 Modelo de Consistencia y Protocolo de Coherencia de la Cache

El hecho de poner a disposición de múltiples procesadores un espacio de memoria lógicamente compartida, aparte de las ventajas anteriormente descritas, genera una serie de problemas que es necesario afrontar. En un sistema de memoria compartida, varios procesadores pueden leer y escribir en la misma dirección de memoria. Esto supone una serie de ventajas desde el punto de vista de eficiencia y rendimiento, pero supone un reto a la hora de definir cuál es el comportamiento correcto de la jerarquía de memoria ante una serie de accesos sobre la misma posición de memoria. Especialmente importante

cuando tenemos en cuenta que el programador necesita un comportamiento razonable y determinista en la ejecución de su código. Desde el punto de vista de un sistema con un solo núcleo, el comportamiento correcto es sencillo, pues se espera que el resultado ofrecido por el procesador sea el equivalente a la ejecución secuencial del código descrito. Sin embargo, esto se complica cuando hablamos de sistemas multiprocesador, donde, aun cuando los procesadores pueden ejecutar el código que les corresponde con resultados equivalentes a una ejecución secuencial, es más difícil conseguir esto para todos los procesadores en su conjunto.

Para abordar el problema de la corrección en la ejecución más fácilmente se subdivide en dos problemas más concretos: la consistencia y la coherencia de la memoria.

2.5.1 Modelo de Consistencia de Memoria

La consistencia o modelo de consistencia de memoria define cual es el comportamiento correcto de una memoria compartida en términos de ordenación de lecturas y escrituras. Mantener la consistencia es fundamental a la hora de diseñar memorias compartidas, de forma que los programadores sepan qué esperar y los arquitectos los límites de lo que pueden implementar. Los modelos de consistencia establecen cuáles son los resultados posibles de una serie de lecturas y escrituras para una arquitectura de memoria compartida. Al contrario que en los sistemas de un solo núcleo donde normalmente sólo existe un resultado correcto, en los sistemas multiprocesador suele implicar varios resultados correctos y algunos incorrectos, lo que lo hace más complejo. A continuación definimos los tres modelos de consistencia fundamentales:

Consistencia Secuencial (SC) [30] es el modelo de consistencia más intuitivo y establece que una ejecución correcta de una aplicación, de múltiples hilos de ejecución, tiene que ser equivalente a la ejecución secuencial de los distintos códigos entrelazados de cualquier forma en un único procesador. Este modelo de consistencia es el más trasparente para el programador, pero limita las optimizaciones arquitecturales que pueden obtenerse de un sistema multiprocesador como veremos posteriormente.

El *Ordenamiento Total de Escrituras* (TSO) [31] mantiene las restricciones del modelo de consistencia secuencial, pero permitiendo el reordenamiento de las lecturas, que pueden adelantar a escrituras dentro del mismo núcleo que no vayan sobre el mismo bloque de cache. Con esto se permite el uso de los buffers de escritura en sistemas multiprocesador. Hay que tener en cuenta que en un sistema de un solo núcleo, las

escrituras pueden ser apartadas del camino crítico puesto que no contribuyen al estado del procesador y de esa forma no bloquean el flujo de instrucciones en caso de un fallo de cache. En ese caso, es posible mantener la consistencia mediante un buffer de escrituras que permite que las posteriores lecturas sobre el mismo bloque tengan acceso a la versión del dato más reciente. Sin embargo, en un sistema multiprocesador, la utilización de buffers de escritura hace que el modelo de consistencia secuencial no se cumpla, puesto que las lecturas pueden adelantar a escrituras anteriores de otros procesadores no ejecutándose por tanto secuencialmente. Es por esto que los sistemas que quieren utilizar buffers de escritura suelen optar por una versión relajada del modelo de consistencia secuencial denominado TSO (*Total Store Order*), que permite los resultados obtenidos debido a los adelantamientos producidos por el buffer de escrituras, siendo en el resto de casos igual que el modelo de consistencia secuencial.

Finalmente están los modelos de *consistencia relajada de memoria* [32], [33]. Éste es el que mejores resultados de rendimiento puede llegar a ofrecer, al ser el que menos limitaciones tiene en el ordenamiento de lecturas y escrituras. Sin embargo, se enfrenta a dos problemas fundamentales: por una parte es necesario establecer explícitamente cuándo se requiere ordenamiento y sincronización dentro del código, y debe suministrar a los programadores herramientas para definirlo y al software de bajo nivel para transmitirlo a la implementación hardware; además, no existe un modelo de consistencia relajada único y consensuado, lo que afecta a la portabilidad del software.

2.5.2 Protocolo de Coherencia

La coherencia simplifica la implementación del modelo de consistencia, permitiendo que el programador vea la cache de los distintos procesadores como un bloque monolítico más sencillo de manejar.

Un protocolo de coherencia proporciona una visión consistente de la memoria a cada procesador, y lo hace segmentando el espacio de memoria en bloques y controlando la forma en la que localmente se accede a dichos bloques. Los protocolos de coherencia controlan los permisos con los que los procesadores acceden a los bloques de memoria, de forma que un procesador no puede leer un dato mientras otro lo escribe, y todas las copias de lectura válida deben contener el mismo valor. Para forzar esta invariabilidad, los protocolos de coherencia codifican los permisos sobre los bloques usando una

selección de los estados de coherencia MOESI (*Modified*, *Owner*, *Exclusive*, *Shared*, *Invalid*) [34].

Aún cuando la mayoría de los protocolos de coherencia utilizados en sistemas actuales usan una selección de los estados básicos, todavía hay que decidir la forma de evolucionar por los distintos estados y cómo controlarlos. Vamos a considerar dos vertientes fundamentales relativas al protocolo de coherencia. Por una parte, el protocolo de coherencia basado en directorio, donde una estructura específica mantiene información sobre el estado de cada uno de los bloques de memoria contenidos dentro del chip, y un protocolo de coherencia *broadcast*, en el que la búsqueda de los datos dentro del chip se realiza mediante consultas generales a todos los posibles lugares donde puede encontrarse el bloque. Estos últimos se basan en pedir, de forma indiscriminada, a todas las localizaciones posibles dónde pueda encontrarse el bloque requerido por el procesador, incluidas las caches privadas de los otros procesadores. Este tipo de protocolo requiere del uso de una red de interconexión ordenada para su correcto funcionamiento y puede obtener una baja latencia pues la comunicación entre la fuente y el destinatario es directa.

Un ejemplo de protocolo *broadcast* es el Token B [35], diseñado para las arquitecturas de cache de acceso no-uniforme (NUCA) con el fin de crear un protocolo de coherencia en *broadcast* escalable, pudiendo ser usado con redes de interconexión punto a punto en lugar de un bus. Este protocolo MOESI establece que cada bloque de cache lleva asociado un número fijo de "*tokens*", más un *token* propietario. Cuando un procesador requiere la lectura de un dato, le basta con conseguir el bloque junto con un único *token*, que será servido por el procesador que sea el propietario (Owner) del dato. Para poder realizar un proceso de escritura sobre un bloque de la cache, el procesador requiere de todos los *tokens* asociados a ese bloque, incluido el propietario. En ambos casos, las peticiones se hacen en forma de *broadcast* sobre todos los dispositivos de la jerarquía de memoria que puedan tener una copia del dato, respondiendo o no según el tipo de petición y el estado en el que posean el dato (MOSI). Así, los bloques en estado compartido no-propietario (S), sólo responderán devolviendo el *token* en caso de una petición de escritura (*store*). Los bloques en estado propietario (MO), responderán con el bloque, y uno o todos los *tokens* según sea una lectura o una escritura. Además, la necesidad de tener todos los *tokens* para modificar un bloque, permite mantener la coherencia de la cache, mientras que el mecanismo de *tokens* permite réplica de datos, agilizando las lecturas. Típicamente, se desea que el número de *tokens* asociados a un

bloque de cache sea igual al número de procesadores en el sistema, permitiendo así que en un caso extremo, cada procesador pueda tener su copia de lectura. Al no disponer de un elemento ordenador, como puede ser el bus en otros protocolos *broadcast*, o el directorio como veremos posteriormente, el protocolo de coherencia basado en *tokens* evita los bloqueos mutuos mediante el uso de unas peticiones especiales denominadas *persistentes*, que utilizan una red de interconexión ordenada.

Por otra parte, los protocolos basados en directorio envían sus peticiones a un punto de la jerarquía de memoria, que será el encargado de o bien responder con el dato, o reenviar la petición al propietario del mismo. Este tipo de aproximación reduce el consumo de ancho de banda en la red de interconexión, siendo más escalable y eficiente en términos de energía, pero incrementa la latencia en las peticiones que sufran indirección. A rasgos generales, podemos encontrarnos Directorios centralizados [36], [37] o distribuidos [38], [39]. En el primer caso, el directorio es una estructura centralizada [40] que contiene la información sobre el estado de todos los bloques de la cache y sus compartidores. Cuando una petición falla, accede a la estructura de directorio para saber si el bloque se encuentra en la parte privada de la cache o no, y en el caso de que se encuentre en la cache, quién es el propietario del mismo. Después se encarga de enviar los mensajes correspondientes a los miembros de la jerarquía de memoria implicados para que atiendan la petición. Como alternativa, la estructura del directorio puede ser distribuida en porciones más pequeñas, lo que reduce el tiempo de acceso a la misma, aumenta el número de peticiones simultáneas resueltas por el directorio, e incluso mitiga los efectos de la indirección en arquitecturas como la NUCA. En este último caso, se puede aprovechar la distribución banqueada del último nivel de cache y el direccionamiento estático para asociar a cada banco la porción del directorio que corresponde a los datos que allí pueden ser almacenados. Así, se aprovecha la petición al último nivel de cache y al directorio en un único acceso, resolviendo ambas simultáneamente y evitando un paso en la indirección. Llevado al extremo, el directorio puede estar implementado en el propio banco de la cache (*in-cache*) [41], [42], de forma que al acceder al bloque en el último nivel de cache, se tiene acceso también al estado del mismo y a sus compartidores, lo que reduce el espacio necesario y la energía consumida. Una solución de este estilo obliga a que el último nivel de cache sea inclusivo respecto a todos los niveles inferiores de forma que pueda mantener el estado de todos los bloques dentro del chip, lo que como hemos visto obliga a una cache de último nivel suficientemente grande respecto a la porción privada.

Una de las principales ventajas del directorio sobre los protocolos basados en *broadcast* es un uso menos intensivo del ancho de banda tanto dentro como fuera del chip, lo que puede derivar en un menor consumo de energía, así como una mejor escalabilidad de los recursos.

Dado que buscamos escalabilidad, el sistema de referencia usado en esta tesis es un sistema multiprocesador con último nivel de cache compartido e inclusivo en arquitectura NUCA, protocolo de coherencia basado en directorio distribuido *in-cache* y consistencia TSO. En cualquier caso, hay que tener en cuenta que las soluciones propuestas a lo largo de la tesis, en su mayor parte, no dependen de ninguno de estos parámetros, y la variación de cualquiera de estas decisiones puede influir en la magnitud de los resultados obtenidos, pero no tanto en la solución en sí misma.

2.6 Jerarquía de Cache y Memoria Principal

A lo largo de la tesis vamos a trabajar sobre distintos niveles dentro de la jerarquía de memoria de un CMP de propósito general, desde las caches privadas de cada procesador a la memoria principal off-chip. Cada uno de los niveles de la jerarquía de memoria está diseñado para minimizar el tiempo de acceso del procesador a los datos e instrucciones que maneja. Los niveles más cercanos al procesador, más sencillos y de tamaño más reducido, tienen los tiempos de acceso más rápidos, enmascarando la latencia de acceso de los niveles subsiguientes de mayor capacidad y complejidad.

Determinar el número de niveles óptimo y su capacidad para un sistema multiprocesador no es trivial. Sin embargo, la limitación del ancho de banda a memoria debida a las conexiones *off-chip*, parece indicar que la porción de transistores dedicada a capacidad de memoria *on-chip* va en aumento, y es conocido que, dada una determinada capacidad de memoria, es conveniente distribuirla en distintos niveles gradualmente para crear una progresión en la latencia y la capacidad de almacenamiento. En el capítulo 3 exploraremos las implicaciones de esta decisión a la vez que buscamos un modelo sencillo para evaluar sus efectos y que parece confirmar este hecho.

Al otro lado de los pines de salida del chip y del problema de ancho de banda se encuentra la memoria principal, y para el desarrollo de esta tesis consideramos que se trata de una arquitectura *Double Data Rate Synchronous Dynamic Random-Access Memory* (DDR SDRAM), presente en la mayoría de los sistemas comerciales en las dos

últimas décadas, aunque podría ser extendido a otras arquitecturas de memoria principal. Un ejemplo sencillo de organización de este tipo de arquitectura podemos encontrarlo en la Figura 2-1, en el que cada controlador de memoria se hace cargo de un canal al que se encuentran conectados los módulos de memoria (DIMM).

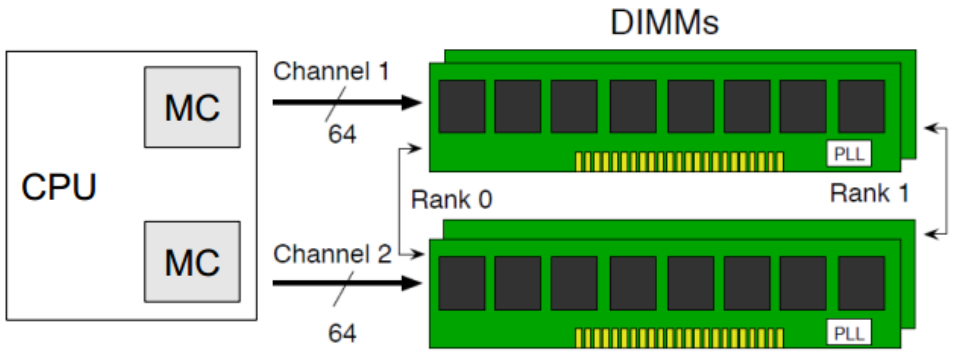

Figura 2-1. Ejemplo de arquitectura DRAM, con dos controladores de memoria conectados a dos módulos de memoria (DIMM), con 64 bits de ancho de canal.

Cada DIMM puede tener uno o más ranks, que a su vez se componen de varios chips de SDRAM (8 en el ejemplo) conectados para ser accedidos en paralelo. Un rank es el conjunto de dispositivos SDRAM necesarios para alimentar el bus de datos, de forma que cada chip SDRAM tiene un número escaso de conexiones de datos (8 en el ejemplo), que se combinan para crear un canal de datos mayor (64 bits en el ejemplo).

Cada chip SDRAM implementa un número de bancos independientes, cada uno de los cuales está creado como un array bidimensional de celdas DRAM, con varias filas y columnas, tal y como se aprecia en la Figura 2-2. El acceso a un dato dentro de la DRAM se hace por tanto mediante un direccionamiento que incluye el banco, fila y columna donde se encuentra. Físicamente, solo se puede acceder a una fila dentro del banco en un momento dado, y sus valores son amplificados y almacenados en el *row-buffer* del banco correspondiente. El proceso para mover una fila del array de memoria al *row-buffer* requiere de un comando que active la fila. Una vez se encuentra en el *row-buffer*, es posible leer o escribir en una porción de la fila, que viene determinada por la columna dentro del direccionamiento del dato.

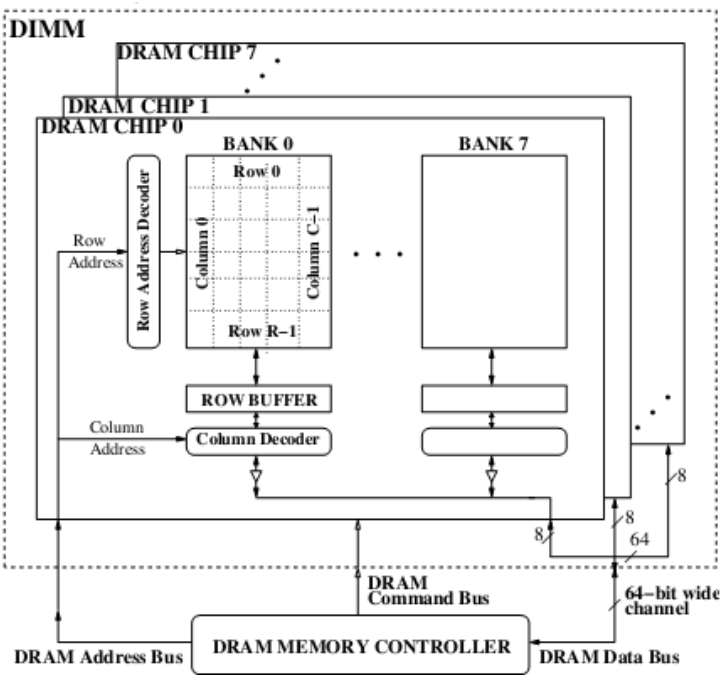

Figura 2-2. Organización de un módulo SDRAM

Dado que, cada vez que una fila es cargada en el row buffer, el contenido dentro del array es destruido, una petición puede ser de una de las siguientes categorías:

- Acierto en la misma fila: la petición accede a la fila que se encuentra cargada en el *row-buffer*, por lo que no es necesario volverla a cargar y puede ser accedida directamente con la única latencia asociada al tiempo de acceso a la columna. El tiempo de acceso es, por tanto, el más bajo.
- Conflicto en la fila: el acceso es a una fila que es distinta a la que se encuentra en el *row-buffer*. En este caso es necesario volver a escribir el contenido del *row-buffer* en el array (que había sido destruido) y reiniciar los amplificadores (precargar), antes de proceder a activar la siguiente fila, con la penalización en latencia que ello conlleva. El tiempo de acceso es por tanto el más alto.
- Fila cerrada: no existe ninguna fila cargada en el row-buffer, por tanto se trata del proceso “normal”, en el que hay que activar la fila buscada para cargarla en el *row-buffer*. El tiempo de acceso es medio.

Existen distintas formas de abordar el manejo de los datos en el *row-buffer*. Dada la alta localidad espacial de los datos en general, mantener la información en el *row-buffer* permite explotar esta característica y obtener latencias de acceso bajas a memoria principal (lo que se conoce como *open-page*). Sin embargo, aunque esto es beneficioso en

sistemas con un único procesador, que puede explotar la localidad espacial de sus datos, deja de ser relevante en sistemas multiprocesador, donde la competición por lo recursos y las peticiones simultáneas de varios procesadores reducen la probabilidad de acierto en el *row-buffer*, con la penalización que ello conlleva. Incluso, soluciones que tratan de incrementar el número de impactos en el *row-buffer* [43] pueden resultar perjudiciales en CMPs, provocando que *threads* con alta localidad en los datos ocupen agresivamente los recursos del sistema y ralentizando al resto de hilos [44]. Esto lleva a soluciones en los que la precarga se realiza automáticamente tras cada lectura/escritura (*close-page*).

Finalmente, el controlador de memoria es el interfaz entre los procesadores y la memoria principal, convirtiendo las peticiones de datos en comandos de DRAM (activar, leer, escribir, precargar…), intentando maximizar el uso de los recursos, respetando las limitaciones temporales de los bancos de DRAM y los buses de acceso. Dadas las ventajas que supone, consideramos que los controladores de memoria se encuentran en el interior del chip, facilitando la comunicación con los distintos componentes del CMP y cada controlador de memoria gobierna un único canal. De forma simplificada, podemos considerar el controlador de memoria como el que se representa en la Figura 2-3.

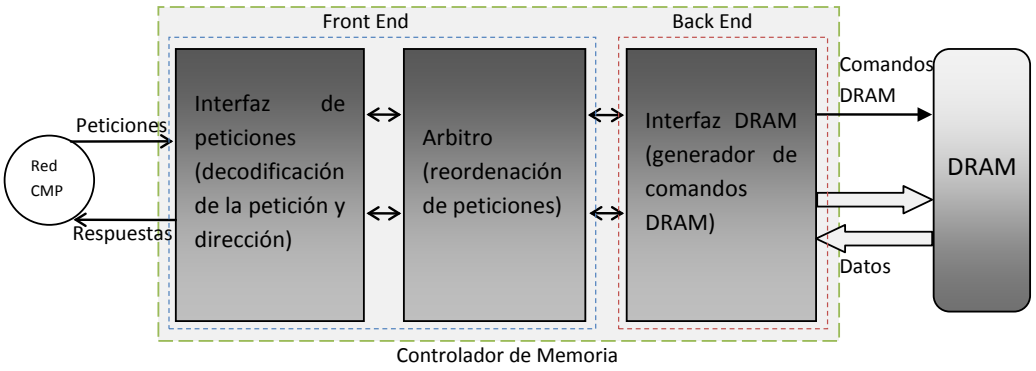

Figura 2-3. Esquema simplificado del controlador de memoria.

2.7 Aplicaciones y Cargas de Trabajo

Es difícil averiguar cómo serán las aplicaciones que se ejecutarán en los futuros sistemas CMP. Hasta ahora, el incremento en el rendimiento del hardware ha sido aprovechado por el software creando aplicaciones con requisitos mucho mayores que los disponibles en generaciones pasadas, y es de esperar que esta tendencia continúe. Adicionalmente, son más comunes las aplicaciones bajo los paradigmas de programación paralela y programación distribuida.

Bajo estas condiciones, es difícil seleccionar el conjunto de cargas de trabajo ideal que sea capaz de representar el conjunto de futuras aplicaciones, pero haremos el esfuerzo de maximizar la diversidad de aplicaciones en estudio a la vez que adaptamos los sistemas y las aplicaciones a los entornos de trabajo que se desean evaluar. Las cargas de trabajo elegidas abarcan aplicaciones numéricas pertenecientes a la suite de benchmarks paralelos NPB (*NAS Parallel Benchmarks*) [45], aplicaciones cliente-servidor pertenecientes a las suite Comercial de Wisconsin [46], y aplicaciones de las suites de rendimiento PARSEC [47] y SPEC2006 [48].

Aplicaciones cliente-servidor

Desarrollado por el grupo de Arquitectura de Computadores de la Universidad de Wisconsin-Madison, esta suite incluye cuatro cargas de trabajo que representan cuatro tipos distintos de servidores comerciales:

Tabla 2-1. Suite de cargas de trabajo cliente-servidor de la Universidad de Wisconsin-Madison.

Aplicación	Descripción
OLTP	Basado en el Benchmark TPC-C v3.0, modela 5 transacciones distintas sobre una base de datos de un vendedor al por mayor con 8 clientes por procesador. Usa el sistema de gestión de bases de datos DB2 v7.2 de IBM.
JBB	Basado en el Benchmark SPECjbb, donde cada thread simula un terminal de un almacén, de forma semejante al benchmark TPC-C. Se hace uso de la Máquina Virtual Java de Solaris HotSpot 1.4.0 Server.
APACHE	Servidor de contenido web estático. Se trata de una compilación del conocido servidor web Apache v1.3.19 sobre Solaris 10. Como cliente se usa *Scalable URL Request Generator* (SURGE) [49] con 10 clientes por procesador.
ZEUS	Servidor de contenido web dinámico basado en el benchmark SPECweb99. Hace uso del conocido servidor web Zeus con soporte para contenido dinámico y del generador de peticiones SURGE con 10 clientes por procesador.

Estas cargas de trabajo nos permiten evaluar una parte fundamental de las aplicaciones que se ejecutan actualmente, especialmente importante en la era del *cloud computing*. Son aplicaciones que hacen un uso intensivo de los recursos compartidos y se ven muy afectadas por la latencia asociada a este tipo de elementos, así como por la distribución de los mismos.

Cargas de Trabajo Paralelas NPB

Estas cargas de trabajo científicas están diseñadas para evaluar el rendimiento de los sistemas multiprocesador, siendo cargas intensivas y por tanto especialmente sensibles al rendimiento del procesador. La versión utilizada durante el desarrollo de esta tesis es la OpenMP 3.2.1, con distintos tamaños de problema. Las aplicaciones utilizadas pertenecientes a esta suite se describen en la Tabla 2-2.

Tabla 2-2. Cargas de trabajo de la suite paralela de la NAS.

Aplicación	Descripción
FT	Solución de una ecuación diferencial mediante Transformadas rápidas de Fourier.
IS	Ordenación de una lista de números enteros mediante *bucket sort*.
BT	Resolución de un sistema de ecuaciones en derivadas parciales mediante la factorización en tres operandos.
CG	Gradiente Conjugado para la resolución de un sistema de ecuaciones lineal.
LU	Resolución de un sistema de ecuaciones en derivadas parciales mediante el algoritmo de factorización LU.
SP	Resolución de un sistema de ecuaciones en derivadas parciales mediante el algoritmo *Beam and Warming*.
MG	Resolución de una ecuación discreta de Poisson tridimensional con el método *V-cycle Multi-grid*.

PARSEC

La suite de cargas de trabajo PARSEC en su versión 2.1 incluye una serie de aplicaciones multithread de áreas muy diversas y, de entre las disponibles, se han seleccionado las siguientes:

Tabla 2-3. Aplicaciones de la suite PARSEC versión 2.1

Aplicación	Descripción
Fluidanimate	Dinámica de fluidos para animación.
Streamcluster	Clustering online de un flujo de entrada.
Blackscholes	Conocido modelo para estimar el valor de una opción de mercado.
Swaptions	Precio de una cartera de este tipo de derivados financieros.
Canneal	Simulador para optimizar el coste de un diseño de chips.

SPEC CPU 2006

Dado que no todas las aplicaciones ejecutadas en los sistemas actuales son paralelas, es necesario incluir cargas de trabajo que cubran este apartado, de forma que podamos obtener resultados para aplicaciones sin ningún grado de compartición, e incluso combinaciones de aplicaciones distintas ejecutándose simultáneamente que permitan experimentar con distintos grados de interferencia. Para ello, hacemos uso de la conocida suite de benchmarks SPEC en su versión CPU2006, que incluye aplicaciones intensivas para ejecutar en un único núcleo. Las cargas de trabajo preparadas incluyen una o varias de estas aplicaciones con instancias repetidas para ocupar todos los procesadores disponibles en el chip. Así, simulando un sistema con 8 procesadores, una ejecución de *hmmer* implica ocho instancias distintas de la aplicación, una por cada procesador, mientras que la ejecución de *hmmer-omnetpp* implica 4 instancias de cada una. Es decir, el número de instancias de cada aplicación se corresponde con la división del número de procesadores entre el número de aplicaciones distintas en ejecución. Para garantizar una mejor distribución de las aplicaciones entre los distintos núcleos del sistema, haremos uso de herramientas que nos permiten vincular procesos a los distintos procesadores. De entre todas las aplicaciones pertenecientes a la suite SPEC CPU2006, hemos seleccionado las siguientes, buscando diversidad en IPC y tasa de fallos:

Tabla 2-4. Aplicaciones de la suite SPEC CPU2006

Aplicación	Tipo	Abrev.	Descripción
Bzip2	INT	bz	Compresión mediante bzip.
Gcc	INT	gc	Compilador de C.
Mcf	INT	mc	Optimización para horarios de transporte público.
Hmmer	INT	hm	Busqueda de secuencias genéticas mediante modelos ocultos de Markov.
Libquantum	INT	lq	Simulación de computación cuántica.
Omnetpp	INT	om	Simulación mediante OMNet++ de una red Ethernet
Astar	INT	as	Algoritmo A* para obtención de rutas en mapas 2D
Xalancbmk	INT	xa	Transformación de documentos XML mediante Xalan-C++
Milc	FP	ml	Cromodinámica Cuántica
Namd	FP	na	Simulación de sistemas bio-moleculares
Lbm	FP	lb	Simulación de mecánica de fluidos
Sphinx3	FP	sp	Reconocimiento de voz

Las características relevantes de cada una de las cargas de trabajo aquí presentadas serán desglosadas en los capítulos correspondientes donde esta información sea necesaria, así como la selección de cargas de trabajo utilizadas.

2.8 Herramientas de evaluación utilizadas

Las herramientas de simulación juegan un papel muy importante en la investigación en Arquitectura de Computadores. A la hora de validar propuestas, se busca que cada uno de los componentes del sistema objetivo sea simulado con un nivel de detalle suficiente, de forma que sea posible evaluar la interacción que estos componentes tienen durante la ejecución del sistema para las distintas propuestas. Además, es importante que las cargas de trabajo evaluadas sean realistas, como pueden ser las descritas en la sección anterior, lo que, en casos como las aplicaciones cliente-servidor, implica el uso de un sistema operativo completo [50].

Con el fin de evaluar las propuestas presentadas en el desarrollo de esta tesis, haremos uso del simulador de sistema completo "*Simics*" [51]. *Simics* es un simulador funcional que permite la ejecución de un sistema operativo comercial sin modificar, en nuestro caso *Solaris 10*, lo que nos permite ejecutar aplicaciones reales en un entorno lo más completo posible. Para la simulación en detalle de las distintas arquitecturas y como complemento de *Simics* se usa el entorno de evaluación GEMS [52], que se subdivide en dos partes: un simulador detallado del procesador, OPAL, capaz de simular procesadores con ejecución fuera de orden, actuando como módulo de tiempos del procesador en la simulación de *Simics*, y RUBY, que simula la jerarquía de memoria y la red de interconexión. A continuación se muestra un esquema del entorno de simulación y los diferentes módulos que se han usado durante la tesis.

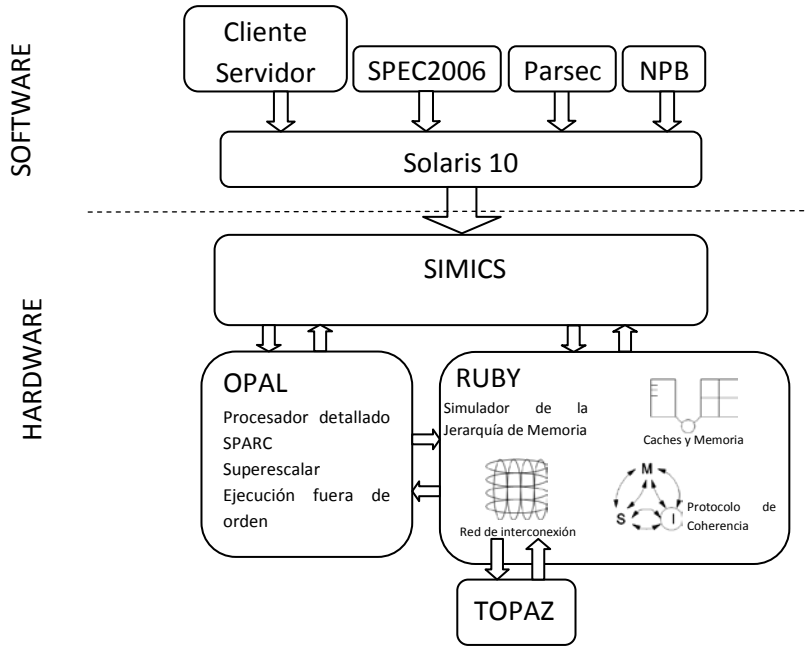

Figura 2-4. Esquema del entorno de simulación utilizado.

SIMICS

SIMICS es un simulador funcional de sistema completo perteneciente a *Wind River Systems*, que permite ejecutar un sistema operativo comercial sin modificación alguna, lo que permite la ejecución de aplicaciones reales sin hacer uso de trazas en un entorno lo más real posible. SIMICS es el encargado de mantener el estado de la ejecución y el control de las instrucciones que son ejecutadas en cada momento. Aun cuando el nivel de detalle que tienen los distintos componentes de la jerarquía es escaso, *Simics* soporta la carga dinámica de módulos de tiempo para los distintos componentes de la arquitectura, lo que permite simulaciones más detalladas sin perder funcionalidad.

GEMS

GEMS es un entorno de simulación para sistemas multiprocesador diseñado por la universidad de Wisconsin [52]. Este simulador, programado en C++, está preparado para funcionar como módulo de tiempos del simulador de sistema completo *Simics*. Esto permite aportar un mayor nivel de detalle en la descripción de la jerarquía de memoria y del procesador, y separar la simulación funcional de la simulación temporal, siguiendo la organización *timing first* propuesta por Mauer et. al. [53]. De esta forma, se delega la corrección de la ejecución a *Simics* y la ejecución de las distintas instrucciones del

procesador, mientras GEMS le informa del momento en el que debe ejecutarse cada instrucción. GEMS es una herramienta de simulación modular, lo que permite abordar las distintas partes de la arquitectura de manera separada y con alto nivel de detalle. GEMS se subdivide a su vez en dos componentes: RUBY y OPAL.

RUBY

RUBY es parte del entorno de simulación GEMS y se utiliza como módulo de tiempos de *Simics* para la jerarquía de memoria. RUBY permite la simulación, con suficiente detalle, de la jerarquía de cache, la memoria principal y la red de interconexión de sistemas multiprocesador. Se trata de un simulador conducido por eventos, que combina objetos programados en C, que simulan cada uno de los componentes hardware de la jerarquía de memoria, junto con un lenguaje de programación propio SLICC (*Specification Language for Implementing Cache Coherence*) que permite especificar las máquinas de estados propias del protocolo de coherencia.

En el desarrollo de esta tesis se ha realizado la mayor parte del trabajo en este componente del entorno de simulación, tanto en la especificación de objetos hardware como implementando máquinas de estados de los distintos protocolos de coherencia. Conviene señalar que RUBY es un simulador complejo con más de cincuenta mil líneas de código. Implementar un protocolo de coherencia no es una tarea sencilla, y validar su comportamiento no es trivial en protocolos especialmente complejos. Para ello, RUBY dispone de una herramienta, "*tester*"[54], que ejecuta una serie de operaciones a memoria pseudo-aleatorias con el fin de estresar el protocolo de coherencia y posteriormente comprueba que los resultados son correctos. Todas las implementaciones desarrolladas durante la tesis han sido extensamente verificadas mediante este *tester*, así como con aplicaciones reales, aun cuando, incluso con estas pruebas, es difícil determinar si el protocolo está libre de errores.

OPAL

OPAL, perteneciente al entorno de simulación GEMS, es un simulador del procesador que actúa como módulo de tiempos, sustituyendo al procesador secuencial de simics. Permite simular procesadores con ejecución fuera de orden, superescalares y con detalle sobre el pipeline y las unidades funcionales. Se trata también de un simulador conducido

por eventos y su complejidad es similar a la de RUBY. Igualmente, una parte del trabajo se ha apoyado en este simulador, especialmente en sus etapas finales.

TOPAZ

TOPAZ [18] es un simulador de redes de interconexión perteneciente al grupo de investigación de Arquitectura y Tecnología de Computadores de la Universidad de Cantabria. Se trata de un detallado simulador ciclo a ciclo que arroja resultados próximos a la implementación física con tiempos de simulación reducidos. Se trata de un simulador modular, altamente configurable, programado en C++, con descripción detallada de los distintos componentes, con capacidad para simular un gran número de encaminadores y topologías distintas. TOPAZ puede ejecutarse en solitario mediante el uso de cargas sintéticas, sin embargo en esta tesis se ha utilizado junto con el simulador GEMS, sustituyendo, cuando es relevante, al simulador de redes que lleva incorporado.

CACTI

CACTI [55] es una herramienta que modela los tiempos de acceso, área ocupada y potencia consumida para diferentes arquitecturas de cache y memoria. Toma como parámetros de entrada la capacidad de la cache, el tamaño de bloque, la asociatividad, el número de puertos de entrada y salida y el número de bancos, para devolver como resultados una configuración recomendada, así como datos sobre latencia, área y potencia. A lo largo de la tesis, CACTI ha sido utilizado para obtener los tiempos de acceso a los distintos componentes de la jerarquía de memoria, así como datos de área en los casos en los que ha sido necesario. Por defecto, se ha empleado una tecnología de 32nm, celdas SRAM, con un puerto de lectura y uno de escritura y acceso secuencial a tag y datos en el último nivel de cache.

2.9 Metodología y Métricas de Análisis

Las aplicaciones utilizadas durante los procesos de evaluación han sido compiladas usando la versión de gcc 4.3.1 con optimización –O3 en una máquina *Sun* real con una configuración similar a la de la máquina simulada. Para la creación de las cargas de trabajo de una única aplicación, ésta es ejecutada en la máquina simulada durante el tiempo necesario para que finalice el proceso de inicialización y se encuentre cerca de la mitad de la región de interés. Para las cargas de trabajo híbridas, todas las aplicaciones implicadas son ejecutadas en paralelo y se vinculan cada una a un procesador distinto

mediante la herramienta de Solaris: *processor_bind*. Cada aplicación es ejecutada hasta el comienzo de su región de interés, momento en el cual queda en espera. Cuando todas han alcanzado ese punto son despertadas a la vez y se ejecutan durante 500 millones de ciclos, momento en el que se realiza un *checkpoint* del estado de la máquina. En todos los casos, se realiza un proceso de calentamiento de las caches con una capacidad de cache igual al que se va a utilizar durante el proceso de evaluación, y se vuelve a realizar un *checkpoint* que se convierte en la carga de trabajo con el que realizar las pruebas.

Cada uno de los datos representados en las gráficas de resultados es obtenido mediante multiples ejecuciones de la simulación aplicando ligeras perturbaciones pseudo-aleatorias en las latencias. De esta forma, se estima la variabilidad de las distintas cargas de trabajo y se representa el intervalo de confianza del 95%, obteniendo resultados estadísticos válidos [56].

A la hora de evaluar los resultados obtenidos seguiremos el criterio “cuanto más bajo mejor”, de forma que, por ejemplo, al hablar de rendimiento, representaremos CPI o tiempo de ejecución de las aplicaciones. Hay que tener en cuenta que no todas las métricas son igual de válidas para todas las aplicaciones, de tal manera que, para aplicaciones multiprogramadas sin sincronización entre tareas, el IPC es una buena medida de rendimiento, ya que el código ejecutado en las diversas pruebas va a ser el mismo, así como las instrucciones ejecutadas. Por otra parte, cuando trabajemos con aplicaciones *multithread*, el uso del IPC puede dar lugar a conclusiones equivocadas. Es un error asumir que todas las ejecuciones de una aplicación *multithread* van a ejecutar el mismo número de instrucciones. La sincronización entre hilos añade una gran variabilidad al número de instrucciones ejecutadas. Esto puede dar lugar a errores como que una solución con un IPC mayor que otra haya tardado más en finalizar, dado que ha invertido más tiempo en la sección de sincronización [57]. Es por esto que, en aplicaciones *multithread*, usaremos el tiempo de ejecución, expresado en ciclos del procesador, como medida de rendimiento.

En la evaluación de las propuestas nos apoyaremos fundamentalmente en tres métricas distintas, que miden el comportamiento de nuestro sistema en términos de *throughput*, entendido como el rendimiento general de nuestro sistema, y *fairness*, que representa el reparto equilibrado de los recursos disponibles entre los distintos procesadores. Como medida de *throughput* usaremos la media Armónica del CPI, que es simplemente la

inversa de la media Aritmética del IPC, y que representa una medida directa del rendimiento, entendido como el número de instrucciones ejecutadas por ciclo en todo el sistema. Esta medida será sustituida por el tiempo de ejecución medido en ciclos del procesador en las aplicaciones multithread, tal y como se ha explicado anteriormente. Como medida de *fairness* usamos el *máximo slowdown del IPC* (o *máximo speedup del CPI*) de las aplicaciones respecto a su ejecución en solitario, lo que nos indica cuál es el máximo perjuicio que sufre una aplicación debido a su ejecución conjunta. Finalmente presentamos el frecuentemente utilizado *Weighted SpeedUp del CPI* [58], que mide un comportamiento equilibrado entre *throughput* del sistema y *fairness*. Hemos escogido estas medidas con el fin de mantener en todas las gráficas el criterio de cuanto más bajo, mejor.

$$Weighted\ Speedup\ CPI = \sum_i \frac{CPI_i^{competición}}{CPI_i^{solitario}}$$

$$Harmonic\ CPI = \frac{N}{\sum_i 1/CPI_i}$$

$$Maximum\ Slowdown = \max_i \frac{IPC_i^{solitario}}{IPC_i^{competición}} = \max_i \frac{CPI_i^{competición}}{CPI_i^{solitario}}$$

(2-1)

3 Diseño de la Jerarquía de Memoria

3.1 Introducción

Es previsible que el aumento del número de núcleos o el incremento de la capacidad de cache dentro del chip sigan mejorando el rendimiento de los multiprocesadores de alguna forma, reduciendo el impacto de los accesos fuera del chip y distribuyendo la carga de trabajo. Sin embargo, esta ganancia se verá reducida con el tiempo si no se toman decisiones arquitecturales que ayuden a la escalabilidad de las mismas, especialmente a medida que la integración se hace mucho mayor. Así por ejemplo, aun cuando con un número reducido de procesadores es posible establecer sistemas de comunicación directos, este tipo de sistemas es difícilmente escalable y requiere de soluciones distintas cuando imaginamos un futuro con decenas o cientos de núcleos. El problema puede ser mucho mayor cuando hablamos de arquitecturas con múltiples bancos de cache, o varios controladores de memoria. De la misma forma, es previsible que el futuro de la cache dentro del chip pase por aumentar con el tiempo de forma considerable, especialmente si tenemos en cuenta nuevas tecnologías de integración o soluciones como el apilamiento vertical.

A lo largo de este capítulo abordaremos algunas decisiones arquitecturales básicas que nos ayudaran a delimitar el espacio de diseño con el que vamos a trabajar, y apoyaremos estas decisiones con resultados, incluido un modelo analítico. El sistema de partida con el que elaboraremos los resultados obtenidos en la tesis es un sistema multiprocesador en chip, con último nivel de cache compartido de acceso no uniforme. El protocolo de coherencia sobre el que se trabaja es MESI, apoyado en un directorio distribuido *in-cache* en los distintos bancos de la NUCA, lo que fuerza a que el último nivel de cache sea inclusivo respecto a los niveles superiores. El punto de partida ha sido elegido basándonos en la sencillez con la que la arquitectura de memoria compartida soporta la programación paralela, siendo la arquitectura NUCA y la coherencia basada en directorio soluciones escalables como se ha descrito anteriormente.

3.2 Influencia de la Disposición de la Cache y los Procesadores en el Chip

Hasta ahora, la escasa concentración de procesadores en un chip hace que sea posible, e incluso más eficiente, el uso de un bus [59] o un anillo [28] como sistema de comunicaciones entre los distintos componentes de la arquitectura. Lamentablemente, este tipo de red de interconexión tiene baja escalabilidad, lo que hace difícil su uso en diseños de futuros multiprocesadores en chip. El incremento del número de elementos a conectar, tanto procesadores como bancos de cache, así como el apilado vertical, hace necesario el uso de redes de interconexión más escalables. Es por esto que las redes punto a punto se posicionan como la mejor alternativa en el diseño de este tipo de arquitecturas desde el punto de vista de la escalabilidad.

Pese a que las redes punto a punto están muy estudiadas en los multiprocesadores tradicionales y existen múltiples trabajos en su traslado a CMP, en este apartado vamos a centrarnos no solo en el diseño de la arquitectura de red, sino en la distribución de los distintos componentes de la topología, así como en el uso eficiente del ancho de banda disponible. Se trata de encontrar el sistema a emplear como referencia a lo largo de la tesis, y en este ámbito, se prueba que una correcta distribución de los procesadores en el chip afecta considerablemente al comportamiento de la red, de la misma forma que Abts et. al. [60] demostraron la relevancia del emplazamiento de los controladores de memoria. Se trata pues de buscar un compromiso que abarque un conexionado eficiente de los procesadores, evitando un uso ineficiente de la red que pueda provocar congestiones por un desequilibrio en el uso de los enlaces y, de la misma forma, ofrezca un buen compromiso en la latencia de acceso promedio. Para ello, realizaremos un estudio sobre un conjunto reducido de opciones y evaluaremos el impacto que éstas tienen sobre el rendimiento del sistema.

3.2.1 Conexionado en Malla

Partimos de una topología sencilla considerando una red en forma de malla con encaminamiento DOR (*Dimension Order Routing*). Evaluaremos los distintos emplazamientos de los procesadores sobre esta topología de red, considerando un CMP de 8 procesadores fuera de orden con 128 instrucciones en vuelo y dos niveles privados de cache, con una LLC NUCA de 256 bancos conectados en una malla 8×8 de 8 MB en total. El sistema utiliza un protocolo de coherencia basado en directorio, distribuido *in-*

cache en una NUCA de direccionamiento estático. La configuración completa puede verse al final del capítulo en la Tabla 3-2.

A la hora de definir los distintos esquemas de conexionado de los procesadores, hay que tener en cuenta que la práctica totalidad de los mensajes que recorren la red tienen un nodo con un procesador como origen o destino. En el protocolo de coherencia utilizado, los únicos mensajes que incumplen esta norma son los accesos fuera del chip, con origen en un banco de la NUCA y con destino un controlador de memoria, siendo los menos frecuentes. Es por esto que el emplazamiento de los procesadores tiene una gran importancia en el rendimiento global del sistema, tanto a nivel de latencia como de aprovechamiento del ancho de banda. Los esquemas que se proponen muestran el emplazamiento de los procesadores conectados a los distintos nodos de la red sin tener en cuenta las restricciones físicas asociadas a la ubicación de los procesadores en una tecnología con procesadores y bancos de cache en el mismo nivel, pero que podrían llegar a implementarse mediante técnicas de apilamiento vertical, donde comúnmente los procesadores se sitúan en un nivel distinto al último nivel de memoria *on-chip* [61].

Se consideran 3 emplazamientos básicos con diferentes propiedades:

- Distribución en BORDES (Figura 3-1): Es el emplazamiento habitual en las NUCAs, colocando los procesadores en los bordes de la malla debido a las restricciones tecnológicas. Este tipo de configuración es la que, en teoría, mayor distancia ofrece entre procesadores y bancos de la cache, y conlleva además un uso ligeramente desbalanceado de los recursos de la red, dado que los procesadores se concentran en dos puntos de la misma.
- Distribución en DAMERO (Figura 3-2): Distribuyendo los procesadores en la red de la forma más dispersa posible, obtenemos una distribución del tráfico mucho más balanceada. Se reduce la distancia media entre los procesadores y los bancos de cache, sin llegar a ser óptima.
- Distribución en BLOQUE (Figura 3-3): En este esquema los procesadores se conectan en el centro de la malla en encaminadores adyacentes. El objetivo es sacar ventaja al posicionar los procesadores uno al lado del otro de forma que se reduce la latencia de las comunicaciones entre los mismos. Así mismo, al estar en el centro de la malla, se reduce la latencia media de los procesadores a los distintos bancos de la cache de último nivel. La desventaja de este tipo de

emplazamiento es la desfavorable distribución del tráfico, que se concentra principalmente en un único punto.

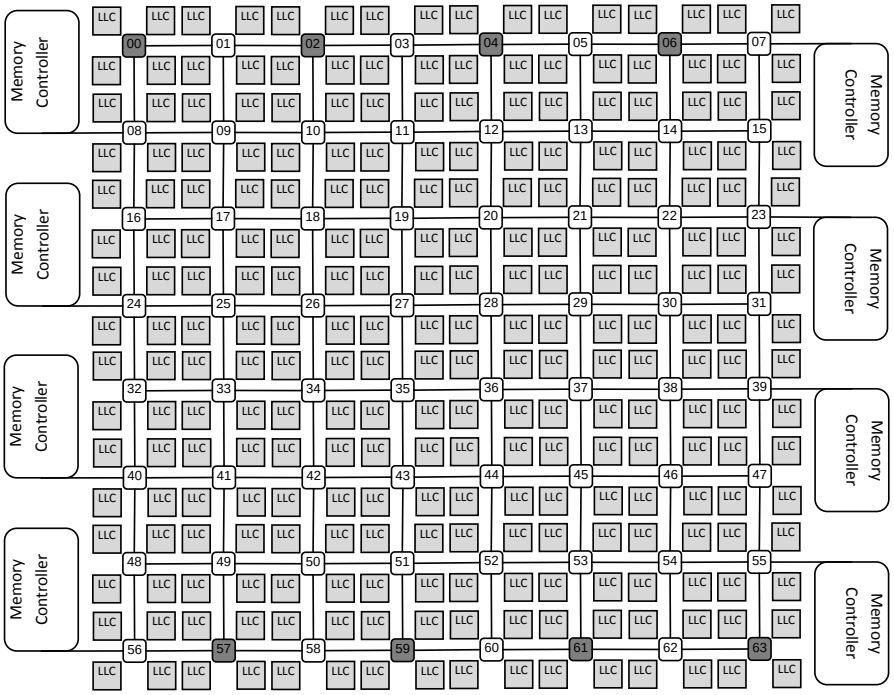

Figura 3-1. Distribución de los procesadores en los BORDES de una red malla 8×8, con 4 bancos de cache por encaminador. Los procesadores están conectados en los encaminadores sombreados de la figura.

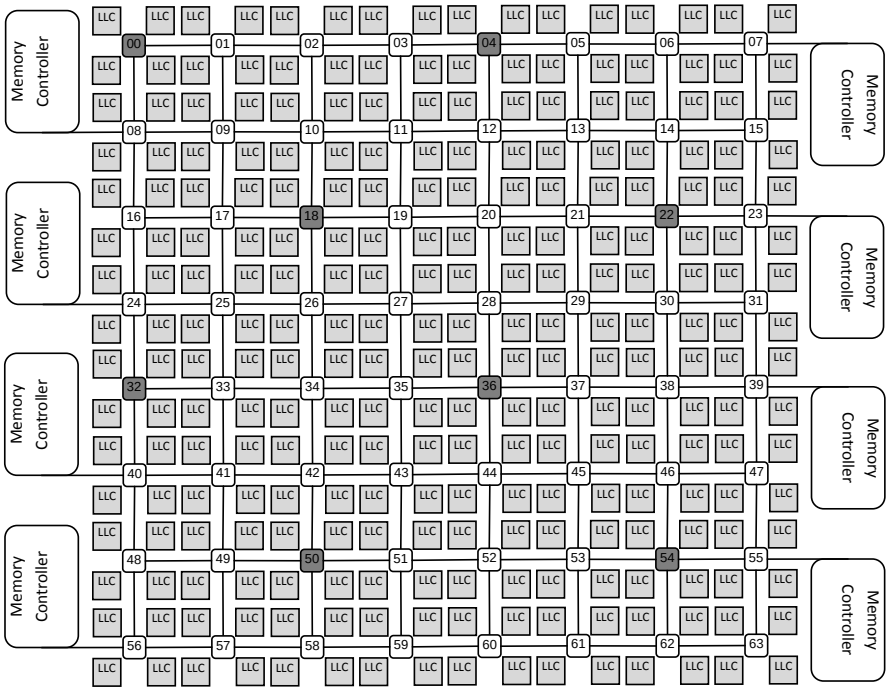

Figura 3-2. Distribución de los procesadores en DAMERO en una red malla 8×8, con 4 bancos de cache por encaminador. Los procesadores están conectados en los encaminadores sombreados de la figura.

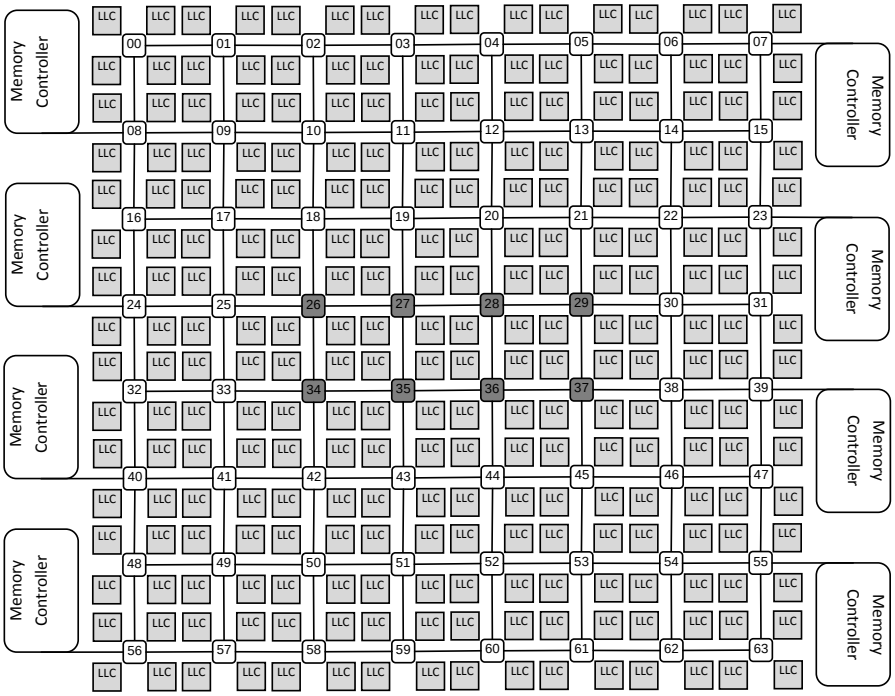

Figura 3-3. Distribución de los procesadores en BLOQUE en el centro de una red malla 8×8, con 4 bancos de cache por encaminador. Los procesadores están conectados en los encaminadores sombreados de la figura.

En todas las configuraciones, los controladores de memoria se conectan en los mismos encaminadores situados en los "laterales" de la red, para minimizar el impacto que su emplazamiento pueda tener en los resultados. Esto es en todo caso inevitable, ya que la distancia media con respecto a los procesadores varía (lo que finalmente influye en la latencia de las respuestas de memoria). Por simplicidad, el algoritmo de encamiento utilizado en la red es determinista con encaminamiento por dimensión (DOR), y el encaminador tiene un *pipeline* de tres etapas (*buffering*, arbitraje y *crossbar*). A continuación pasamos a evaluar el rendimiento obtenido para cada una de estas configuraciones bajo distintos supuestos.

En simulaciones con un modelado optimista de la contención en la red (tan solo la contención en los cables), usando el entorno de evaluación descrito en el apartado 2.8, los resultados obtenidos (Figura 3-4) muestran que la distribución en bloque es la que mejores resultados obtiene. Gracias a las ventajas obtenidas por la reducción de latencia y ante la ausencia de contención se obtienen mejoras de hasta un 24% en el rendimiento, en comparación con la distribución en el exterior (BORDES).

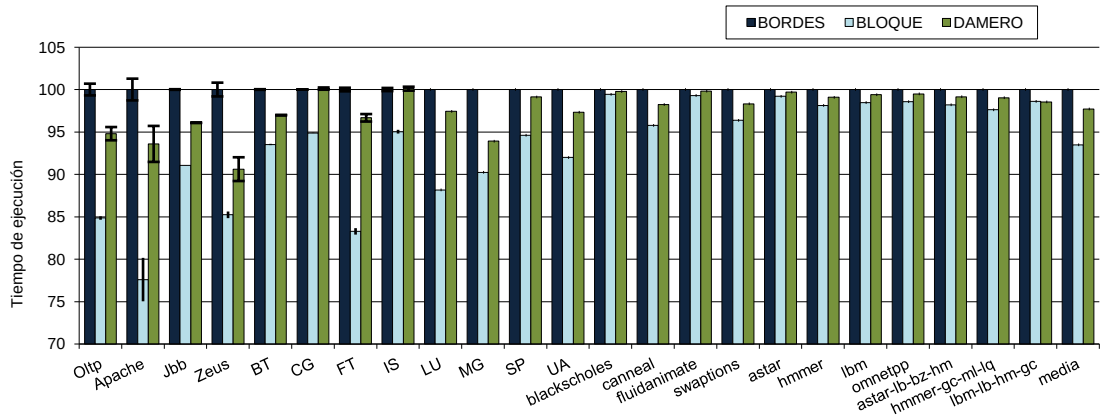

Figura 3-4. Tiempo de ejecución normalizado para una NUCA con 256 bancos y distintos emplazamientos de procesadores. Los resultados están normalizados a la distribución de los procesadores en los BORDES de una malla 8×8.

Los resultados son menos llamativos cuando modelamos correctamente la contención en la red, haciendo uso del simulador de red TOPAZ, que no solo simula la contención en los enlaces, sino también la contención en el *router*. En este caso, como se aprecia en la Figura 3-5, la alta densidad de tráfico provoca un descenso en el rendimiento de las soluciones BLOQUE y BORDES. Esto es debido a que el encaminamiento de paquetes se concentra en determinados puntos, creando puntos calientes de tráfico, especialmente relevantes en el caso de la distribución en BLOQUE como se puede ver en la Figura 3-6.

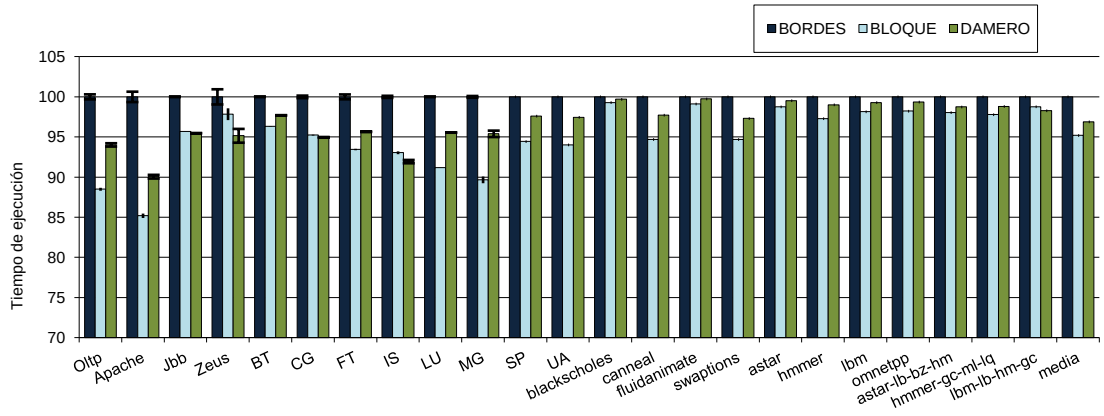

Figura 3-5. Tiempo de ejecución normalizado de los distintos emplazamientos de procesadores en presencia de contención en la red, normalizado al caso de emplazamiento en BORDES.

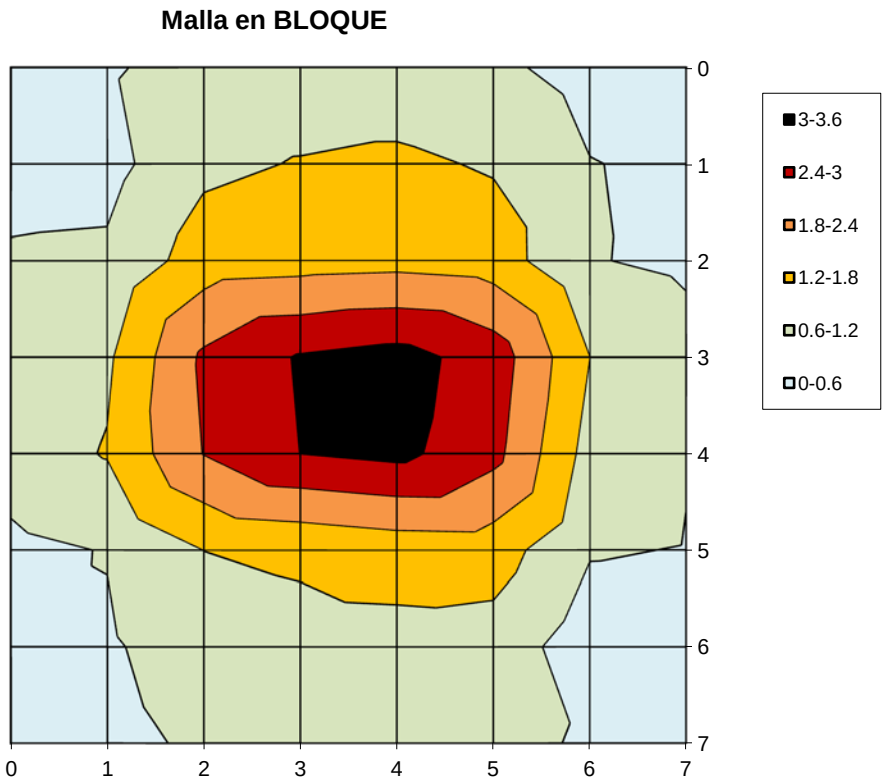

Figura 3-6. Distribución del tráfico en tanto por ciento, de la carga de trabajo cliente-servidor, Zeus, en una red malla 8×8 con los procesadores conectados en los nodos centrales en una distribución en BLOQUE.

Los datos presentados suponen una conexión de los procesadores a cualquier punto de la red de interconexión, lo que podría implementarse si consideramos soluciones como el apilamiento vertical. Sin embargo, teniendo en cuenta las limitaciones de área que supone implementar una arquitectura de este tipo en un chip bidimensional, al colocar los procesadores entre los bancos, provocamos un incremento de las distancias de la red o topologías irregulares. En la Figura 3-7, Figura 3-8 y Figura 3-9 se muestra una aproximación de un emplazamiento de los procesadores y los bancos de cache teniendo en cuenta el área que ocupan. Para las proporciones se ha tomado como referencia un chip comercial [62], aunque los resultados son similares en otros casos.

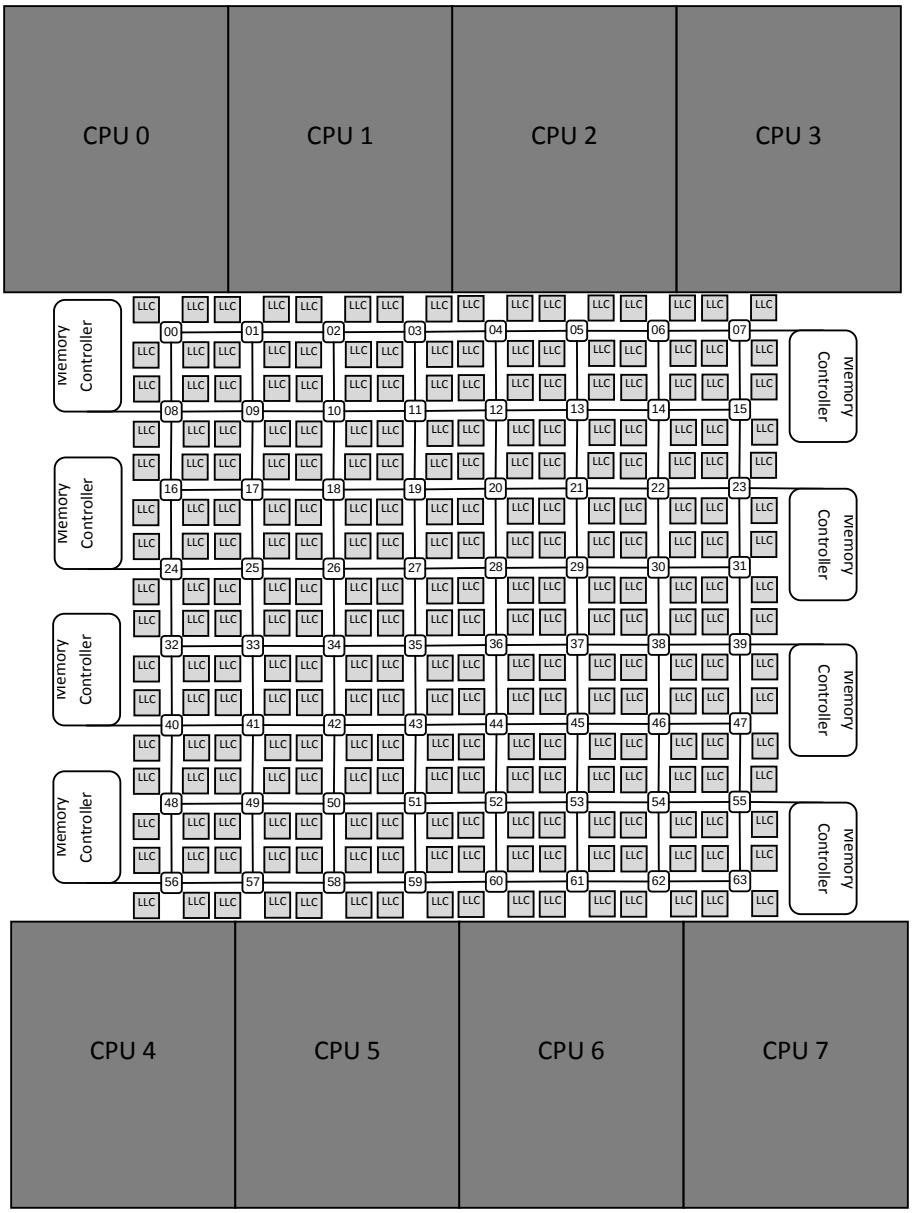

Figura 3-7. Emplazamiento de 8 procesadores en los BORDES de un sistema con 256 bancos de memoria teniendo en cuenta las limitaciones de área.

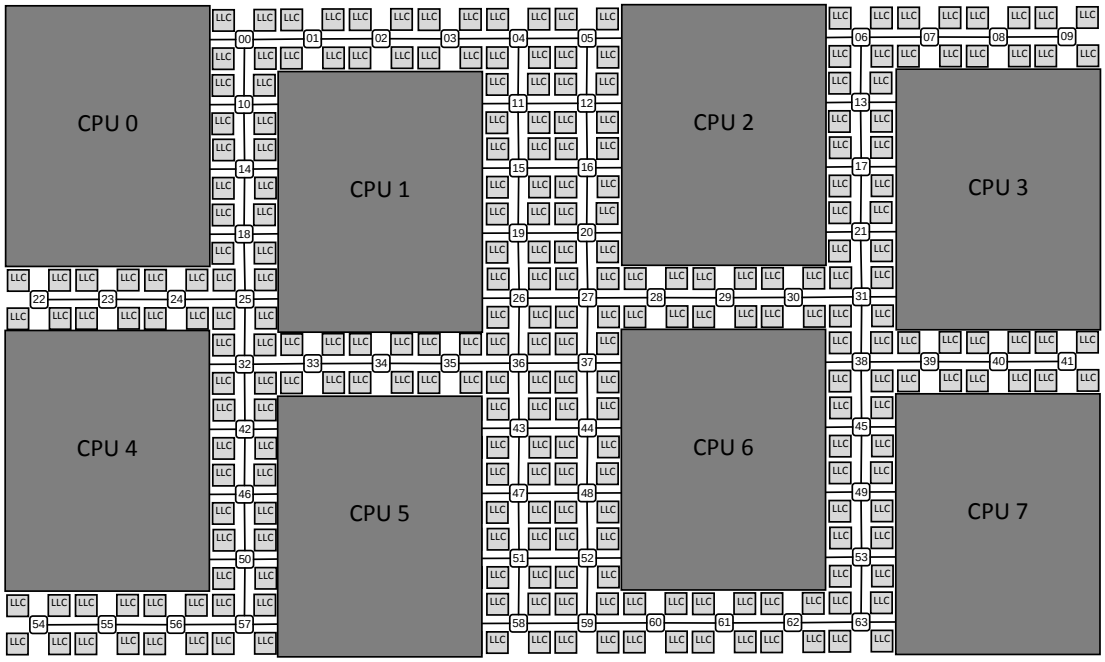

Figura 3-8. Emplazamiento de 8 procesadores en DAMERO en un sistema con 256 bancos de memoria teniendo en cuenta las limitaciones de área.

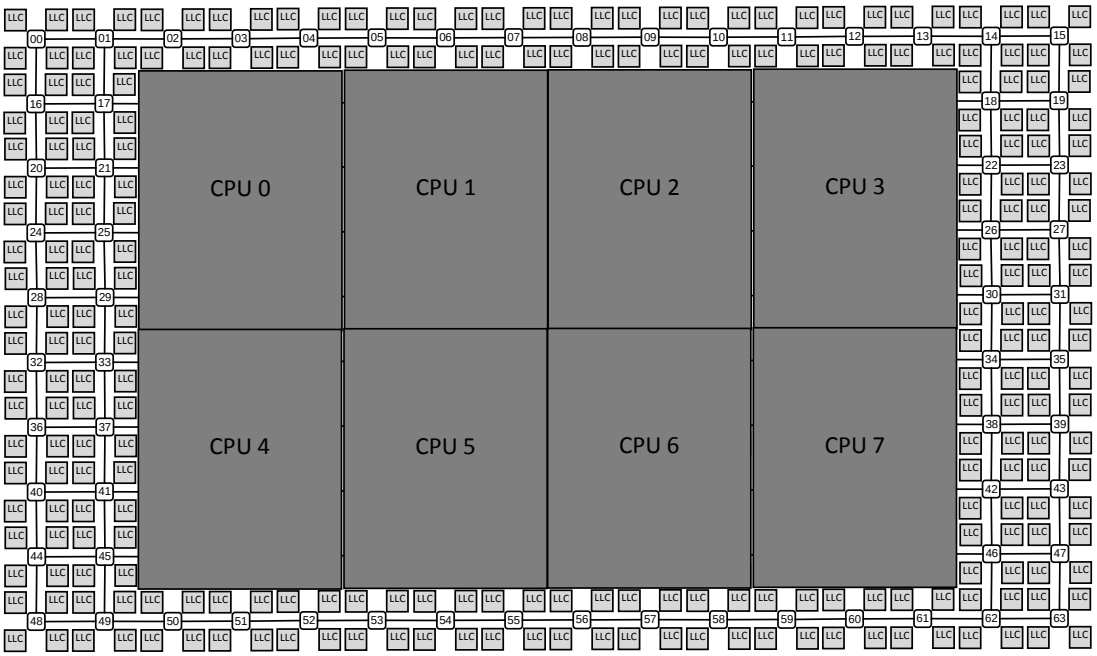

Figura 3-9. Emplazamiento de 8 procesadores en BLOQUE en un sistema con 256 bancos de memoria teniendo en cuenta las limitaciones de área.

Como se puede ver, en los dos últimos casos (DAMERO Y BLOQUE) no solo la malla ha perdido sus propiedades en cuanto a regularidad, sino que la distancia media ha aumentado considerablemente. Es sabido además, que organizaciones como BLOQUE pueden suponer un problema en la distribución de temperatura [63].

Dada la dificultad de implementar una topología irregular en el simulador de red TOPAZ, los resultados que se presentan en la Figura 3-10 no incluyen contención en los *routers*. En cualquier caso, son significativos, y se observa como la distribución en BLOQUE es la que peores resultados arroja. Esto es debido al incremento sustancial de la distancia media en la red, además de problemas de contención en los enlaces. Por otra parte, el emplazamiento en DAMERO reduce los problemas de contención gracias a una mejor distribución del tráfico, mejorando los resultados obtenidos por la distribución en BLOQUE. Sin embargo, se ve igualmente penalizado por el incremento sustancial en la distancia media de la red. De entre las distintas topologías, la que mejores resultados ofrece es pues la distribución en los BORDES, que aunque de partida y según lo visto anteriormente era la que peores resultados parecía ofrecer, una vez tenidas en cuenta las mínimas restricciones tecnológicas resulta ser la más eficiente.

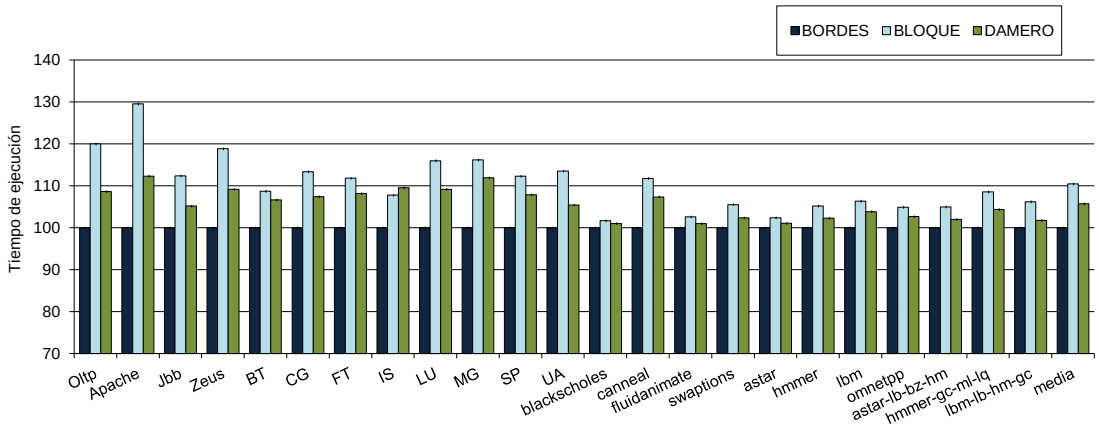

Figura 3-10. Tiempo de ejecución normalizado de los distintos emplazamientos de procesadores teniendo en cuenta limitaciones de área, normalizado al caso de emplazamiento en BORDES.

3.2.2 Conexionado en Toro

La pregunta que surge es si hay alguna forma de aprovechar las ventajas, en cuanto a tráfico y distancia, que ofrece un emplazamiento de los procesadores en DAMERO, evitando las restricciones de emplazamiento. La solución trivial es establecer enlaces similares a las *Transmision Line Caches* (TLC) propuestos por Beckmann et. al. [64], o hacer uso de arquitecturas de apilamiento vertical, tal y como se ha mencionado

anteriormente. Aquí vamos a hacer una propuesta de mejora distinta que consiste en sustituir la red de interconexión de una malla por la de un toro “foldeado” [65]. La ventaja del toro frente a la malla es una reducción de la distancia media y una mejor distribución del tráfico. Como contrapartida, los enlaces del toro “foldeado” son el doble de largos que los de la malla, aunque la complejidad de los encaminadores de ambas topologías es similar. Existe una ventaja adicional del toro “foldeado” cuando hablamos de distribución de los procesadores, y es su capacidad para conseguir una mejor distribución de los procesadores en el mismo plano que los bancos de cache, manteniendo las restricciones tecnológicas. Así, una determinada distribución de los procesadores en los bordes de un toro foldeado (Figura 3-11) se corresponde con una distribución muy similar a un DAMERO en su contrapartida extendida (Figura 3-12), con la ventaja añadida de conseguir mejores distancias medias en el toro que en la malla.

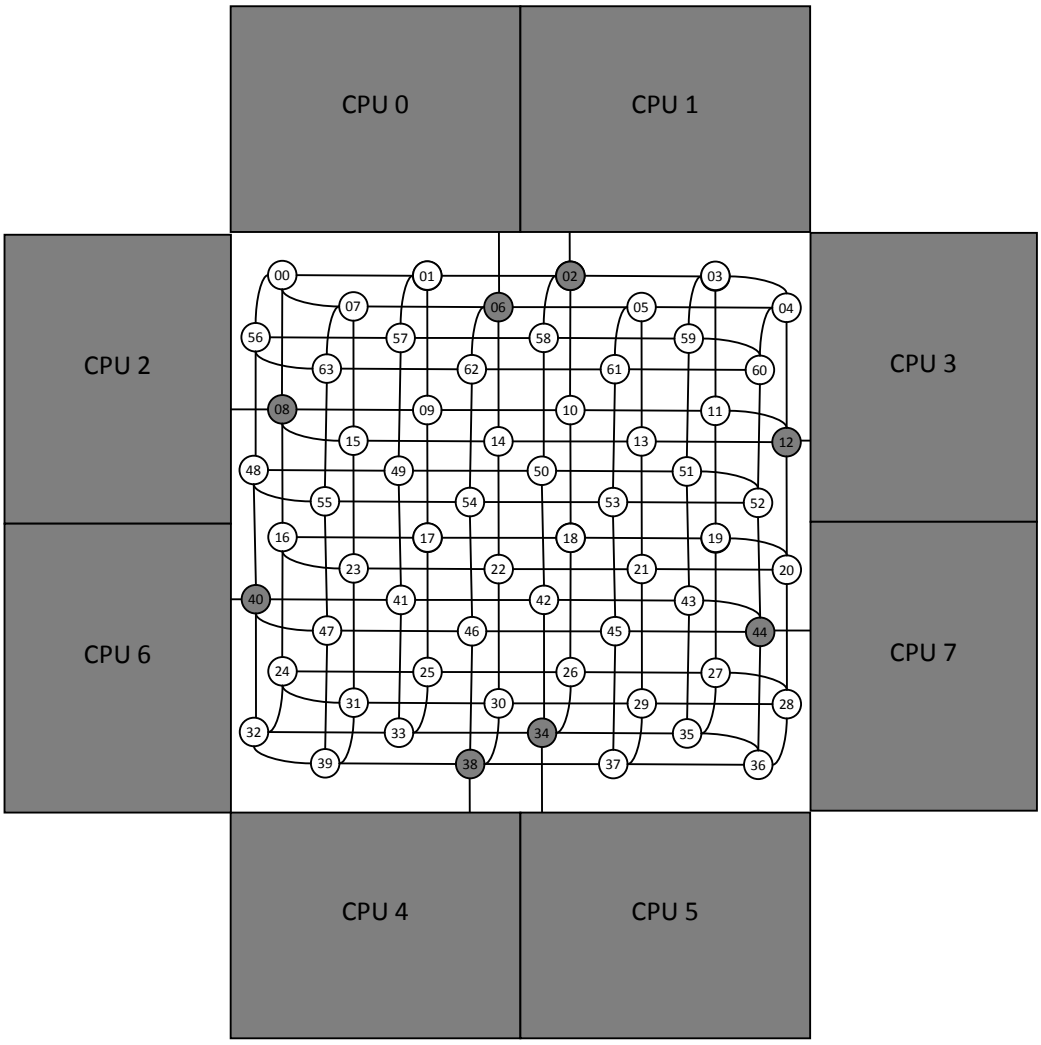

Figura 3-11. Conexiones de los procesadores en los bordes de un toro foldeado 8×8.

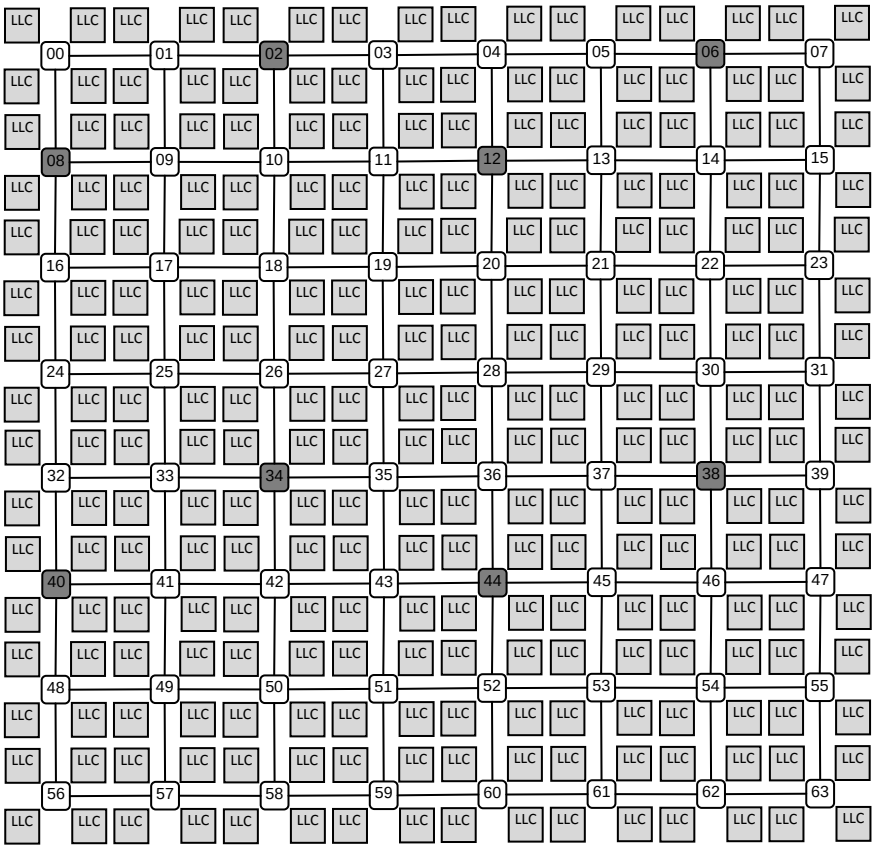

Figura 3-12. Correspondencia de las conexiones de un toro foldeado en su versión desplegada, muy similar a la vista anteriormente en la malla.

Otra de las ventajas del toro foldeado, aunque no perfecta debido al encaminamiento utilizado, es que se consigue una mejor distribución del tráfico en la red, como se puede comprobar en la Figura 3-13, donde se muestra la distribución del tráfico de la aplicación Zeus en una topología realista en toro foldeado en DAMERO frente a la implementación realista de BORDES en una malla. Cada fila y columna de la gráfica se corresponde con una fila o columna de la red de interconexión, siendo los puntos de intersección los encaminadores. Como se puede observar, la distribución en DAMERO, con una topología de tipo toroidal, distribuye mejor el tráfico entre los distintos nodos de la red, mientras que la implementación en BORDES tiende a acumular el tráfico en los extremos, especialmente con encaminamiento determinista.

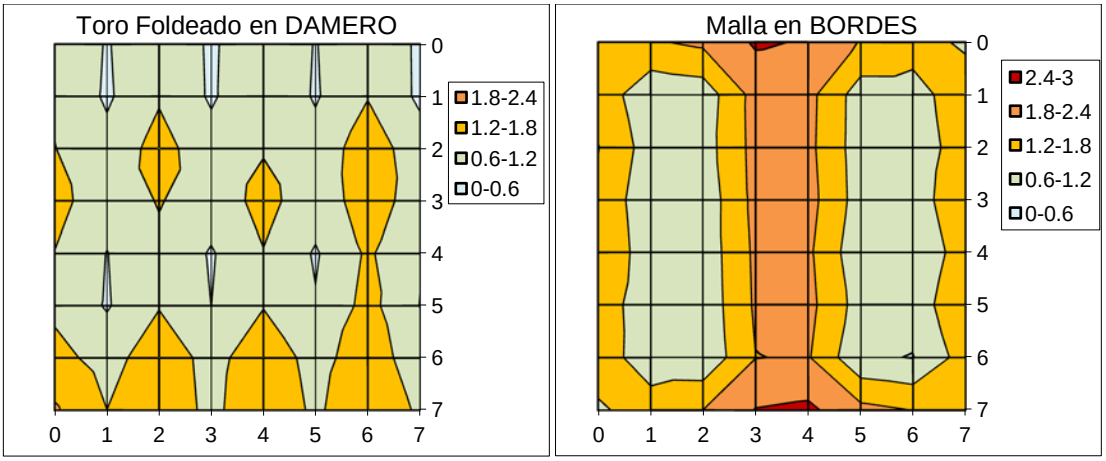

Figura 3-13. Distribución del tráfico en tanto por ciento para distintas configuraciones durante la ejecución de la aplicación Zeus. La figura de la izquierda se corresponde a la distribución del tráfico en un toro foldeado con emplazamiento de procesadores que se asemeja a un DAMERO, mientras que la figura de la derecha representa una implementación con topología malla y procesadores conectados en los BORDES.

Como resultado de las dos ventajas asociadas al toro foldeado, una menor contención y por tanto una distribución del tráfico más homogénea y la reducción de la distancia media, el rendimiento obtenido por esta configuración en DAMERO es superior al emplazamiento en BORDES en una malla. En la Figura 3-14 se muestran los resultados obtenidos en una simulación con contención, y se observa cómo incluso considerando que es necesario dos ciclos para atravesar los enlaces del toro foldeado, los resultados siguen siendo favorables al toro. Si suponemos que las distancias en el chip son lo suficientemente pequeñas como para tener enlaces en el toro de un ciclo, los resultados son evidentemente aun más favorables.

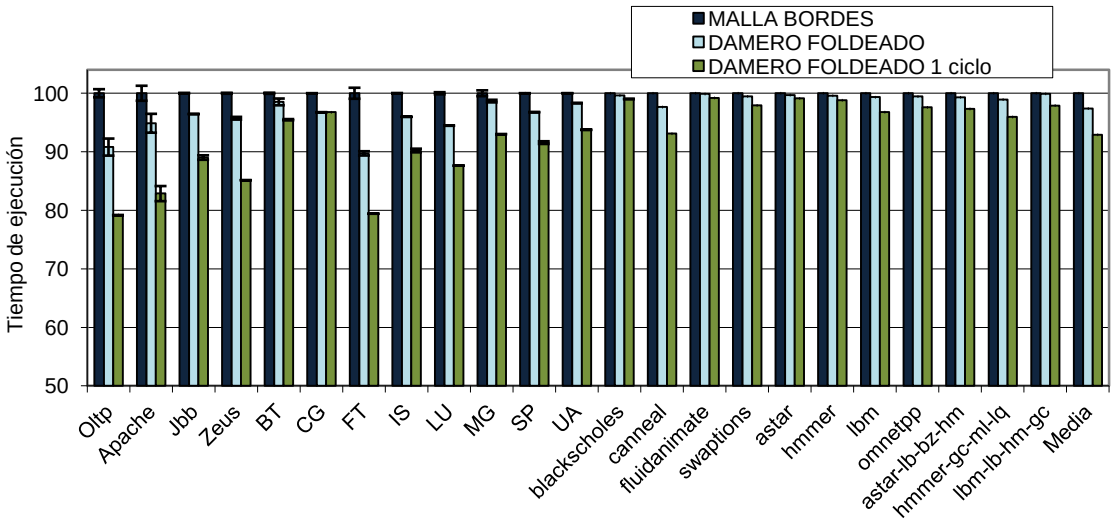

Figura 3-14. Tiempo de ejecución normalizado para distintos tipos de topología de red, con emplazamiento de los procesadores en los bordes. Los resultados están normalizados a la distribución de los procesadores en los BORDES de una malla 8×8.

Los resultados presentados hasta ahora muestran los efectos que tienen tanto la topología de red como el emplazamiento de los procesadores en un sistema con 8 procesadores y 256 bancos de cache. Una configuración que podría justificarse si consideramos que, en el futuro, es posible que el área dedicada a la cache sea mucho mayor que la dedicada a procesadores [66], y que los resultados no difieren demasiado si tomamos un único banco de cache por *router* para un total de 64. Sin embargo, es una configuración sobredimensionada actualmente, y es más probable que el número de nodos de la red se aproxime al número de procesadores. A continuación se muestra un ejemplo de cómo un toro foldeado 4×4 se corresponde con un emplazamiento en DAMERO cuando conectamos los ocho procesadores en sus bordes de la manera adecuada, tal y como hemos visto anteriormente, siendo esta configuración la que utilizaremos de aquí en adelante.

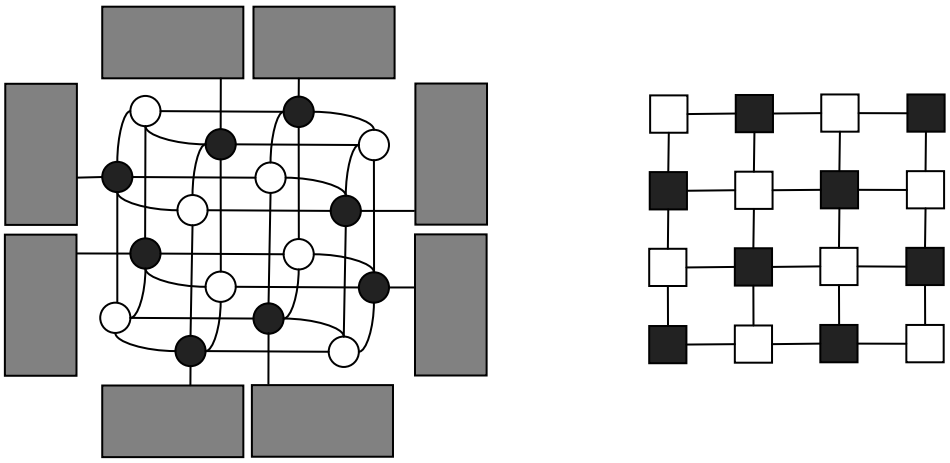

Figura 3-15. Conexión de ocho procesadores en los bordes de un toro foldeado 4×4 y su equivalencia en su versión extendida como un damero perfecto.

3.3 Distribución de la Capacidad en distintos Niveles

Partiendo de esta configuración, vamos a presentar un modelo analítico que ayuda a estrechar el espacio de diseño de la jerarquía de cache en el chip. Este modelo permite estimar, de forma aproximada, la proporción de silicio que debemos dedicar a cada nivel de cache para minimizar el tiempo de acceso promedio percibido por el procesador. Obviamente, no existe una distribución perfecta de la capacidad de la cache entre los distintos niveles, pues depende de las aplicaciones que van a ejecutarse en el CMP, pero buscaremos cuál es la distribución más eficiente para un rango amplio de escenarios de

trabajo. Por supuesto, se podrían obtener resultados más precisos con simulaciones específicas de las distintas arquitecturas, pero el modelo permite llevar a cabo un análisis preliminar que reduce las posibilidades sin usar la enorme cantidad de tiempo y recursos que implican ese tipo de simulaciones para llegar a un resultado similar.

El trabajo se apoya principalmente en los modelos propuestos por Chow [67][68], Przybyski et. al. [69][70] y Hartstein et. al. [71]. El primero propone la hipótesis, verificada experimentalmente, de que la tasa de fallos de la cache es una función potencial de la capacidad de la cache. Przybyski usa esa relación entre los fallos de cache y la capacidad en jerarquías de cache de multiples niveles, y Hartstein hace una aproximación teórica unida a simulaciones que concluyen que el exponente de la función potencial está directamente relacionado con el tiempo de re-referencia de las peticiones a la cache, y varía entre -0.3 y -0.7 (regla de la raíz de 2).

Todos estos trabajos previos se han llevado a cabo sobre sistemas con un único procesador y hasta dos niveles de cache. En este apartado vamos a extender este trabajo a sistemas multiprocesador y aplicaciones con compartición de datos, lo que implica múltiples factores que pueden afectar a los patrones descritos para la tasa de fallos de la cache. El trabajo emplea estos datos para determinar la idoneidad de añadir un nuevo nivel de cache o no en una arquitectura dada. Los resultados que se obtienen muestran que el patrón de re-referencia todavía domina en la tasa de fallos de la cache, por encima de otros efectos más complejos introducidos por las aplicaciones multithread corriendo sobre arquitecturas similares al estado del arte actual.

En este apartado vamos a estudiar el comportamiento de la tasa de fallos de una arquitectura de cache multinivel y obtener una función que nos permita determinar la idoneidad de añadir o no un nivel extra a la jerarquía y obtener la distribución óptima de la capacidad entre los distintos niveles. Comenzaremos con un modelo genérico de la latencia de la cache, después nos centraremos en una arquitectura de cache específica, y finalmente validaremos el modelo mediante simulación. Fruto de este análisis, elegiremos la distribución empleada en el resto de la tesis, siendo la alternativa que ofrezca los mejores resultados.

3.3.1 Modelo de Cache Genérico

Partimos de una aproximación conocida [67], [70]–[72] en sistemas de procesador único, por la que los fallos de cache pueden aproximarse empíricamente por una función potencial de la capacidad de la cache, C :

$$M = m_0 C^{\propto}$$

(3-1)

Donde M es la tasa de fallos, y m_0 y α son parámetros que dependen de la aplicación que se esté ejecutando en ese momento. De acuerdo con Hartstein et. al. [71], m_0 puede expresarse en función de su patrón de re-referencia de acuerdo a la fórmula:

$$m_o = R_0 \Gamma(-\alpha)(A_l M_{ave})^{-\alpha}$$

(3-2)

Donde R_0y $\alpha = (1 - \beta)$ son constantes del patrón de re-referencia de la aplicación, A_l es el tamaño de la línea de cache y M_{ave} es la tasa media de fallos. Por simplificar, nosotros representaremos m_o como función de dos constantes R y K_0:

$$m_o = R(K_0)^{-\alpha}$$

(3-3)

Ambos parámetros dependerán por tanto del patrón de re-referencia de la aplicación [71] y pueden ser obtenidos empíricamente. Podemos ver un ejemplo de esto en la Figura 3-16, donde se representa la tasa de fallos de una aplicación (Transformada Rápida de Fourier, FT tamaño A de los benchmarks paralelos NPB), y la función de regresión potencial que se ajusta a su región lineal.

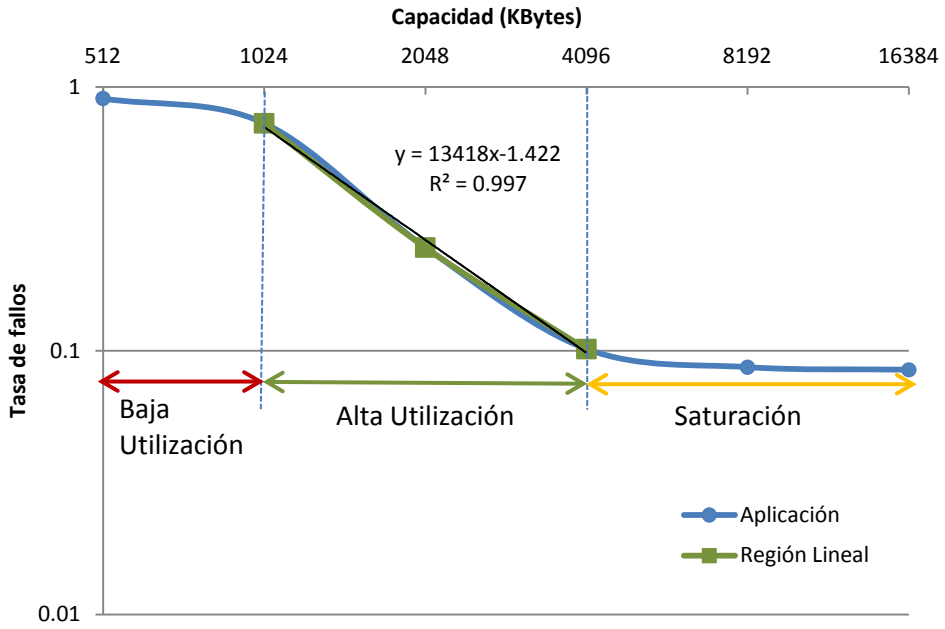

Figura 3-16. Comportamiento de la tasa de fallos de una aplicación y su región lineal. Ambos ejes están en escala logarítmica, con base 2 y base 10 respectivamente.

Como se puede observar en la figura, para una aplicación existen tres comportamientos diferentes de la tasa de fallos en función de la capacidad de la cache. Así podemos identificar tres intervalos:

- *Baja Utilización*, o "No hay suficiente Cache". La aplicación remplaza casi todos los datos antes de ser re-referenciados mientras la cache tenga una capacidad inferior a un valor dado. Podemos decir que en este intervalo, la aplicación no hace uso alguno de ese nivel de cache y es independiente de la capacidad que dediquemos. Su tasa de fallos es elevada y prácticamente plana (porción superior izquierda de la figura).
- *Alta Utilización*, o "Región Lineal". Esta es la porción que interesa a la hora de estudiar el comportamiento de la cache. El rendimiento de la aplicación que se ejecuta mejora a medida que aumenta el tamaño del nivel de cache. Este patrón se corresponde con la región central de la Figura 3-16, representada con puntos cuadrados, y que puede ser fácilmente aproximada por una función potencial.
- *Saturación* o "Demasiada Cache". A partir de este nivel es imposible conseguir más rendimiento de la cache incrementando su tamaño. La tasa de aciertos ha alcanzado el máximo y añadir más capacidad a este nivel de cache no va a mejorar significativamente el rendimiento del sistema. De nuevo en este caso el comportamiento de la aplicación deja de depender del tamaño de la cache, y la

tasa de fallos es baja y prácticamente constante (porción inferior derecha de la figura).

Dado que en las regiones extremas, la aplicación tiene un comportamiento que no depende de la cantidad de cache que dediquemos a un determinado nivel, centraremos nuestro interés en el segundo intervalo, la región lineal, donde la tasa de fallos depende del tamaño de cache y se ajusta a la función descrita por la ecuación (3-1). Hay que tener en cuenta esta función potencial es válida siempre y cuando dominen en la cache los fallos de capacidad. Si los fallos de conflicto son abundantes, debido a una baja asociatividad por ejemplo, la ecuación (3-1) deja de ser válida, ya que los fallos de conflicto no están relacionados con el patrón de re-referencia de la aplicación.

Trabajando en la región lineal, m_0 y α son independientes del tamaño de la cache, y pueden utilizarse en multiples niveles de la jerarquía de memoria, siempre y cuando la relación de capacidad entre niveles sea lo suficientemente alta [71]. Sin embargo, en un sistema con una proporción más ajustada entre los distintos niveles y multiples procesadores es necesario que cada aplicación sea caracterizada en los diferentes niveles y cada nivel tenga su pareja de parámetros.

3.3.2 Latencia de acceso a la Cache

En general, la latencia de acceso de una petición a la jerarquía de memoria viene dada por la latencia del nivel de la jerarquía que responde a la petición. Por tanto, la latencia media de una jerarquía se puede aproximar por la siguiente función:

$$L_{globl} = H_1 L_1 + M_1 H_2 L_2 + M_1 M_2 H_3 L_3 \ldots$$

(3-4)

Donde H_i y M_i son la tasa de aciertos y fallos del nivel i de la jerarquía de memoria respectivamente, y L_i es la latencia media de acceso a ese nivel. El primer término por tanto se corresponde con los accesos a la cache de primer nivel, mientras que el último término de la función se debiera corresponder con la contribución de la memoria principal.

Hemos visto anteriormente que la tasa de fallos de un nivel de cache puede ser aproximada empíricamente por la ecuación (3-1). Hay que tener en consideración que esta ecuación simplificada no tiene en cuenta las interdependencias que se producen en el patrón de re-referencia entre los distintos niveles, de forma que grandes variaciones en el

tamaño de un nivel de cache, afectan al patrón de re-referencia de los niveles siguientes [68]. Es por esto que vamos a considerar dos parámetros diferentes para cada uno de los niveles de la jerarquía. Sustituyendo la ecuación (3-1) en la ecuación (3-4), y expresando la tasa de aciertos en función de la tasa de fallos, obtenemos:

$$L_{globl} = (1 - M_1)L_1 + M_1(1 - m_{02}C_2^{\alpha_2})L_2 + M_1 m_{02}C_2^{\alpha_2}(1 - m_{03}C_3^{\alpha_3})L_3 + \cdots + M_1 M_2 M_3 \ldots M_n L_{mem}$$

(3-5)

Siendo L_{mem} la latencia de acceso a memoria principal, la cual asumimos que siempre acierta y no tiene contención.

Sabemos además, que la latencia de acceso a un banco de cache depende de la capacidad del mismo, de forma que podemos expresar la latencia de los distintos niveles en función de la capacidad. En una cache de mapeo directo, el tiempo de acceso es proporcional al área que ocupa la cache, linealmente proporcional a la raíz cuadrada de la capacidad. En un diseño más complejo, tiene sentido suponer que, para una asociatividad determinada, el tiempo de acceso puede ser representado por una función potencial de la capacidad, similar a la tasa de fallos:

$$L_i = l_0 C_i^{\gamma}$$

(3-6)

La ecuación (3-6) define la latencia de acceso a un banco de cache, L_i, en función de su capacidad C_i, un factor de latencia l_0 y un exponente γ. Usando Cacti 6.0 [55] en un amplio rango de configuraciones de cache, se comprueba que la aproximación es suficientemente precisa para nuestros propositos. En el ejemplo de la Figura 3-17 se observa una cache de 8 vías, con bloques de 64 bytes, acceso en serie a tag y datos, tecnología de 32nm y diferentes valores de capacidad. Suponemos una frecuencia de reloj de 3GHz.

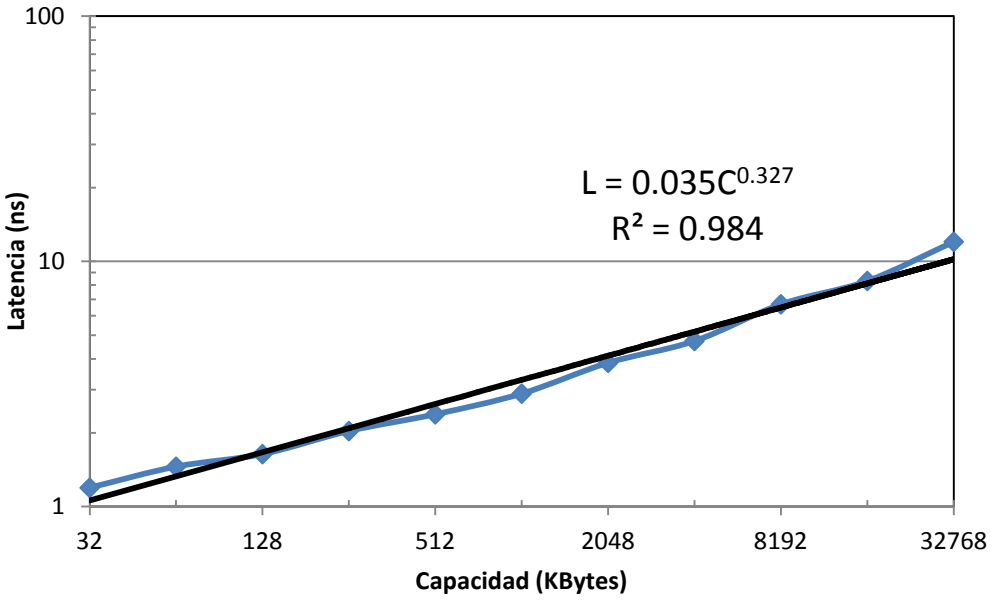

Figura 3-17 Latencia de acceso a un banco de cache como función potencial de la capacidad, usando tecnología de 32nm.

Hay que tener en cuenta que la latencia de acceso de cada nivel no se corresponde tan solo con la latencia de acceso del banco, sino que incluye una penalización por fallar en los niveles anteriores. Esta penalización es función de la latencia de fallo de los niveles previos y por tanto podemos representar la latencia de acceso a un nivel de jerarquía como:

$$L_i = l_0 C_i^{\gamma} + \sigma L_{i-1}$$

(3-7)

Donde σL_{i-1} es la latencia de fallo del nivel previo y σ es un factor de proporcionalidad, y dependerá de la forma en la que se accede a la cache. Sustituyendo la latencia de acceso de cada nivel (3-7) en la expresión de la latencia global (3-5), y considerando el caso concreto de una arquitectura de tres niveles de cache, obtenemos la siguiente expresión:

$$\begin{aligned} L_{globl} = {} & (1 - M_1)L_1 + M_1(1 - m_{02}{C_2}^{\alpha_2})(l_{02}{C_2}^{\gamma_2} + \sigma L_1) \\ & + M_1 m_{02}{C_2}^{\alpha_2}(1 - m_{03}{C_3}^{\alpha_3})(l_{03}{C_3}^{\gamma_3} + \sigma L_2) \\ & + M_1 m_{02}{C_2}^{\alpha_2} m_{03}{C_3}^{\alpha_3}(L_{mem} + \sigma L_3) \end{aligned}$$

(3-8)

Dado que el tamaño del primer nivel de cache vendrá determinado, entre otras cosas, por el tiempo de ciclo del procesador, suponemos que será constante para todos los casos

que vamos a estudiar. De la misma forma, dado que nos centraremos en la jerarquía de caches, consideramos la memoria principal igual para todos los casos, y por tanto L_{mem} será considerada constante.

3.3.3 Aplicación del Modelo a un sistema CMP-NUCA

Para realizar el estudio y validación del modelo, vamos a centrarnos en una arquitectura específica. Partimos de un sistema compuesto por un CMP con P procesadores, que imitan las características de procesadores comerciales como el Intel Haswell [28]. Cada procesador incluye una cache de primer nivel (L1) y otra de segundo nivel (L2), ambas privadas, y un tercer nivel (L3) compartido en arquitectura NUCA [73], distribuida en B bancos conectados mediante una red punto a punto. La coherencia de cache es mantenida mediante un directorio distribuido *in-cache*, embebido en los bancos de L3, con inclusividad forzada de los niveles inferiores, tal y como hemos comentado anteriormente.

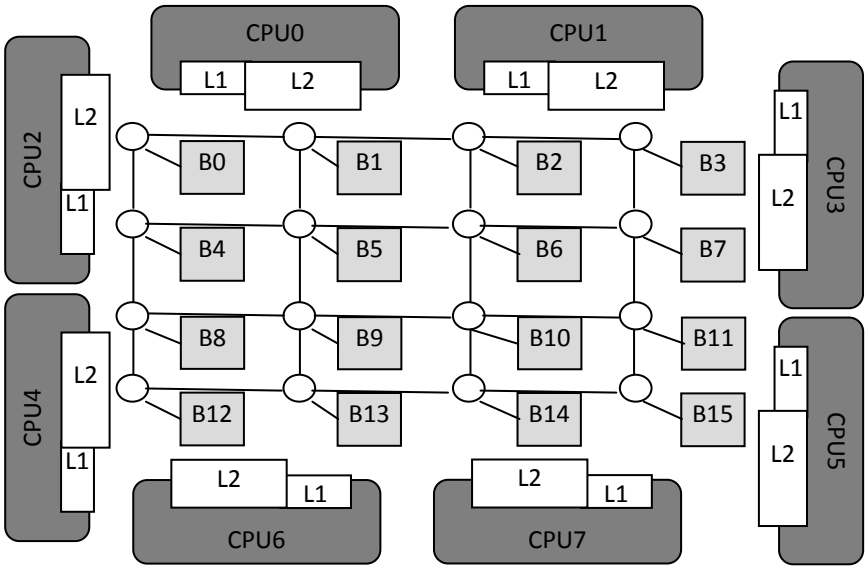

Figura 3-18. Esquema de CMP con 8 procesadores y cache de nivel 3 NUCA con 16 bancos conectados a un toro foldeado 4×4.

Para simplificar, vamos a ignorar la contención en la red o las indirecciones de coherencia a la hora de estimar la latencia de acceso a L3. En cualquier caso, debemos considerar la latencia de la red de interconexión, para lo que usamos la distancia media de la red, la longitud del pipeline del *router* y el tamaño de paquete para peticiones y respuestas con datos. Con estos datos podemos estimar fácilmente la latencia media base de la red en ausencia de contención (N_0), y podemos ser más precisos en la latencia base

si, conociendo la posición en la que se encuentran los procesadores, tenemos en cuenta que los mensajes tienen como origen y destino un procesador, y como punto intermedio un banco de cache. De esta forma, la latencia de L3 en un sistema como el mostrado en la Figura 3-18, puede representarse como:

$$L_3 = l_0 \left(\frac{C_3}{B}\right)^{\gamma} + N_0$$

(3-9)

Donde B es el número de bancos de la NUCA, y N_0 es la latencia media de un paquete que atraviesa la red de interconexión en ausencia de contención.

Vamos a aplicar el modelo de la ecuación (3-8) para la evaluación de un caso concreto. Queremos determinar la proporción optima de cache que debemos dedicar a los dos últimos niveles (L2 y L3) en un sistema como el que se muestra en la Figura 3-18. Para ello hay que tener en cuenta que en la ecuación (3-8), C_2 se refiere a la capacidad de la cache de segundo nivel de cada procesador (C_{2p}), y de aquí en adelante nos referiremos al agregado de las caches privadas de segundo nivel como C_2. Dado que intentamos determinar la cantidad de cache que dedicamos a los dos últimos niveles, suponemos un área fija C que debemos repartir entre ambos. De esta forma, podemos expresar la capacidad de ambos niveles como una fracción de C:

$$C_{2p} = xC$$

$$C_3 = C - C_{2p}P = (1 - xP)C$$

(3-10)

Donde x es la proporción de cache que dedicamos a una L2. Sustituyendo (3-9) y (3-10) en (3-8) queda:

$$L_{globl} = (1 - M_1)L_1 + M_1(1 - m_{02}(xC)^{\alpha_2})(l_{02}(xC)^{\gamma_2} + \sigma L_1)$$
$$+ M_1 m_{02}(xC)^{\alpha_2}\left(1 - m_{03}\left((1 - xP)C\right)^{\alpha_3}\right)\left(l_{03}\left(\left(\frac{1 - xP}{B}\right)C\right)^{\gamma_3} + N_0 + \sigma L_2\right)$$
$$+ M_1 m_{02}(xCP)^{\alpha_2} m_{03}\left((1 - xP)C\right)^{\alpha_3}(L_{mem} + \sigma L_3)$$

(3-11)

Con esta expresión, podemos determinar el tiempo de acceso medio en función del área que dedicamos a los niveles L2 y L3, o lo que es lo mismo, x. Para ello, fijamos los parámetros del sistema, como la capacidad total dedicada a los dos últimos niveles ($C = 16MB$), el número de procesadores ($P = 8$) y el número de bancos de la NUCA ($B = 16$). En la Figura 3-19 se representa la latencia de acceso promedio en función de la relación de capacidades, $x = \mathrm{C}_{2\mathrm{p}}/\mathrm{C}$, y la relación de los exponentes entre los patrones de re-referencia de L2, α_2 y L3 α_3.

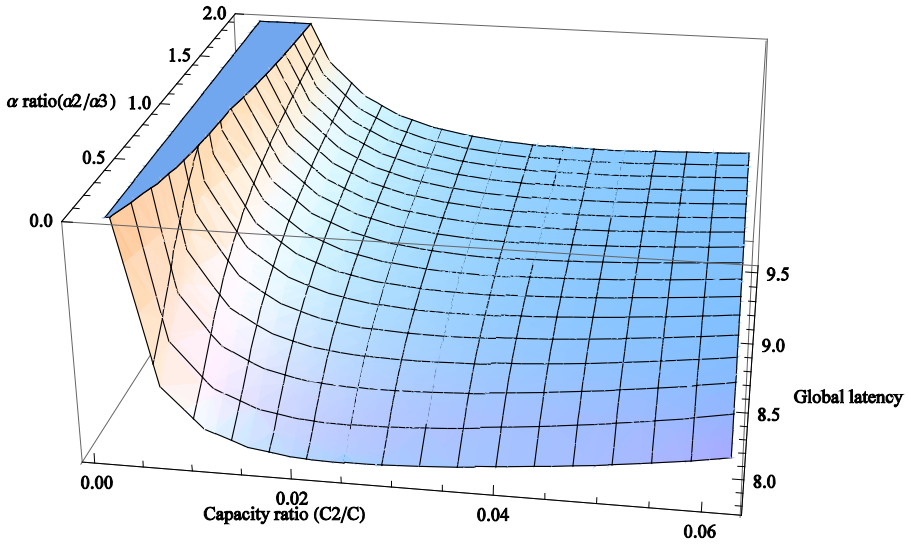

Figura 3-19. Comportamiento de la latencia de acceso media frente a C_{2p}/C y α_2/α_3.

Como se puede observar, la forma de la función presenta un mínimo, de forma que conocidos los parámetros de nuestra aplicación (α_2 y α_3), el comportamiento de la misma se corresponderá con una de las curvas de la figura, de forma que es posible determinar la proporción de capacidades óptima para la cual la latencia media que percibe el procesador es mínima.

3.3.4 Caracterización de las Aplicaciones

Como se ha visto previamente, la tasa de fallos puede expresarse en función de la capacidad de la cache gracias al patrón de re-referencia siempre que trabajemos en la región lineal de funcionamiento de la aplicación respecto a la cache. Ambos parámetros m_0 y α de la ecuación (3-1) son dependientes de la aplicación que se está ejecutando. De hecho, estos parámetros pueden ser distintos para cada uno de los niveles de cache, debido a que los niveles inferiores afectan al patrón de re-referencia, y por tanto deben obtenerse por separado para cada nivel de la jerarquía de memoria. Igualmente, una

aplicación puede tener varios *working set* dentro de la misma ejecución [74], de forma que tenga un comportamiento lineal en el nivel 2 de cache (L2) (comportamiento de la porción más pequeña y utilizada del *working set*), mientras que la L3 esté en un estado de baja utilización (por no tener suficiente espacio para la porción grande del *working set*) o estar en un estado de saturación.

Vamos a obtener estos parámetros mediante la experimentación, empleando el simulador de sistema completo basado en *Simics*, extendido tan solo con el simulador de memoria, *ruby*. Se han realizado mediciones del comportamiento de la tasa de fallos de los distintos niveles de cache variando su capacidad, centrándonos en los niveles de interés, L2 y L3. Caracterizamos el segundo nivel de cache (L2) variando su tamaño y midiendo la tasa de fallos mientras mantenemos el tamaño de L3 lo suficientemente grande (64MB) para evitar su interferencia. De la misma forma, evaluamos el patrón de re-referencia de L3 variando su tamaño mientras mantenemos la capacidad de L2 constante en 64KB. En todos los casos, la L1 se ha mantenido a un tamaño constante de 32KB dividido en instrucciones y datos. Dado que solo era necesario caracterizar el patrón de re-referencia, las medidas se han realizado sin usar el modelo de tiempos del procesador (opal). Tan solo la memoria ha sido simulada en detalle para capturar la actividad del protocolo de coherencia, así como un modelo básico de la contención en la red.

Escogemos tres tipos distintos de cargas de trabajo, con diferentes grados de compartición y se intenta que todas las aplicaciones escogidas se encuentren en la región lineal para ambos niveles de cache. Así, tenemos, por un lado, aplicaciones multiprogramadas (gcc y bzip2), consistentes en ocho copias de un mismo programa ejecutándose simultáneamente en el sistema y que carecen de datos compartidos. Por otro lado las aplicaciones multithread, elegidas entre las aplicaciones paralelas de la NAS (en su implementación OpenMP, versión 3.5.1) [45], y las cargas de trabajo transaccionales de la *Wisconsin Comercial Workload Suit* [46] tienen distinto grado de compartición, siendo estas últimas aquellas en las que es más relevante.

Tabla 3-1. Principales Parámetros de las aplicaciones en estudio.

	L2			L3			
Workload	$\mathbf{m_{02}}$	$\boldsymbol{\alpha_2}$	$\mathbf{R^2}$	$\mathbf{m_{03}}$	$\boldsymbol{\alpha_3}$	$\mathbf{R^2}$	$\boldsymbol{\alpha_2 / \alpha_3}$
BZIP2	38.014	**-0.394**	0.95	22605	**-0.745**	0.98	0.533
GCC	553.79	**-0.715**	0.99	2.3E5	**-0.994**	0.98	0.719
OLTP	43.858	**-0.395**	0.97	7022.3	**-0.744**	0.98	0.53
APACHE	33.506	**-0.356**	0.96	3550.5	**-0.678**	0.99	0.525
JBB	47.212	**-0.418**	0.99	228.64	**-0.467**	0.99	0.895
IS	5.5005	**-0.191**	0.91	4E7	**-1.181**	0.99	0.162
FT	947.35	**-0.683**	0.98	3.2E4	**-0.814**	0.98	0.839

Se obtiene la tasa de fallos para diferentes tamaños de cache de cada nivel y se aproxima mediante regresión por una función potencial, determinando así los parámetros m_0 y $\propto$. En la Figura 3-20 se puede ver un ejemplo de cómo ajusta la regresión en la aplicación FT. En la Tabla 3-1 hay un resumen de los parámetros característicos de cada aplicación, tal y como se expresan en la ecuación (3-1) de cada nivel. Además se incluye el índice de correlación de cada regresión (R^2), donde se observa que, en la mayoría de los casos, es cercano a 1.0 (lo que implica que la función potencial se ajusta perfectamente al comportamiento de las aplicaciones en estudio).

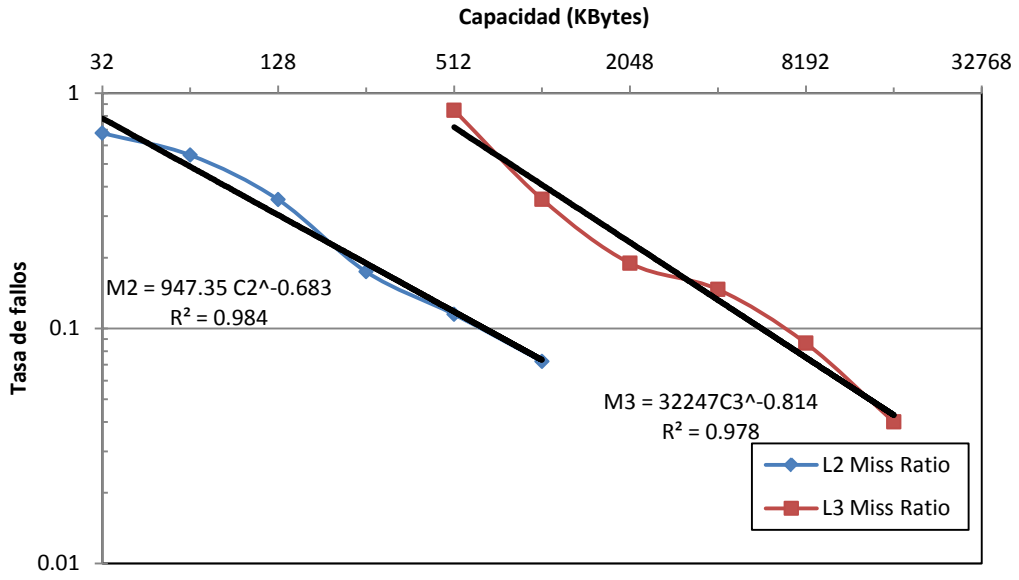

Figura 3-20. Tasa de fallos de la cache de nivel 2 (L2) y de nivel 3 (L3) para la aplicación FT tamaño W.

3.3.5 Aplicación del Modelo: Número óptimo de niveles y Distribucion Óptima de niveles 2 y 3

Una vez conocidos los parámetros de las diferentes aplicaciones, es posible utilizar el modelo de latencia media de la ecuación (3-11) para analizar la respuesta del sistema a una variación amplia de tamaños de cache, lo que dado el esfuerzo computacional de las simulaciones de sistema completo, podría ser prohibitivo.

Sustituyendo los valores de la Tabla 3-1 en la ecuación (3-11), podemos predecir el comportamiento del sistema en estudio para cada aplicación. Vamos a tener en cuenta las limitaciones de área disponible, es decir, los niveles de cache L2 y L3 comparten una misma cantidad global de área que deben distribuirse C. Como ejemplo, la Figura 3-21 muestra la latencia media para la aplicación OLTP, donde existe un mínimo en la latencia para una relación de capacidades aproximada $C_{2p}/C = 0.03$. Lo que se traduce en una relación óptima entre el tamaño agregado de las L2 privadas y la L3 de aproximadamente 1/3.

$$x = 0.03 \rightarrow C_3 = C(1 - xP) = C(1 - 0.03 \cdot 8) = 0.76$$
$$\frac{C_2}{C_3} = \frac{0.03 \cdot 8}{0.76} \approx \frac{1}{3}$$

(3-12)

Es interesante hacer notar, que este resultado coincide con la Figura 3-19, para la curva del caso particular $\propto_2/\propto_3 = 0.53$.

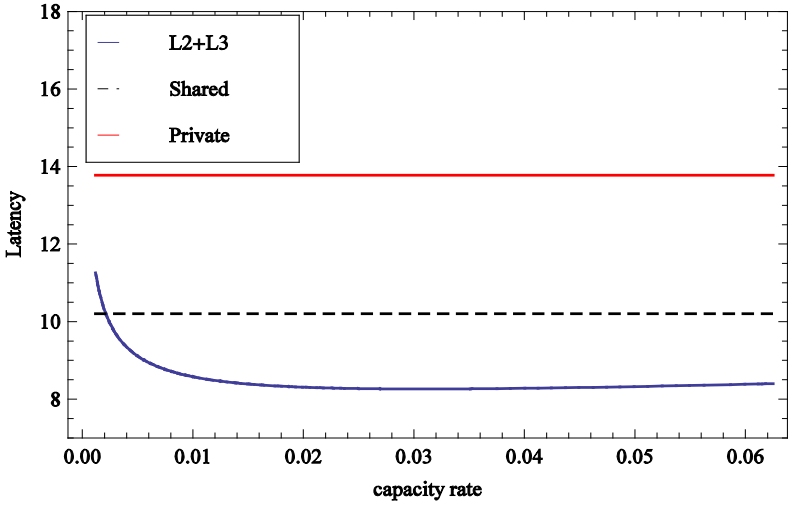

Figura 3-21. Latencia media de la apliación OLTP en función de la relación de capacidades C_{2p}/C, comparado con arquitecturas con dos niveles de cache completamente privado o completamente compartido.

Este modelo también puede ser usado para predecir el comportamiento de un sistema con solo dos niveles de cache, con cache privadas y sin cache L3 compartida ignorando el tercer término en la ecuación (3-11), o con una cache completamente compartida y sin L2 privadas ignorando el segundo término en (3-11). Se puede ver un ejemplo en la Figura 3-21 del comportamiento teórico de la latencia media de acceso en estos casos para la aplicación OLTP (las dos líneas horizontales). Para este caso concreto, se observa que es preferible una configuración completamente compartida (sin L2 privadas) frente a la versión de caches privadas (sin L3 compartida).

Esto no se cumple en todas las aplicaciones, como se puede ver en la Figura 3-22, donde se muestran todas las aplicaciones bajo estudio. Tal y como se puede observar, según el modelo teórico, existen aplicaciones donde en dos niveles de cache, es preferible el uso de una cache completamente privada (principalmente las SPEC), y otras donde es preferible una cache compartida (la mayoría de las transaccionales y NAS). En cualquier caso, tener tres niveles de cache es la solución más equilibrada, obteniendo en promedio una reducción en la latencia de acceso del 30% con respecto a la cache privada, y del 15% respeto a la compartida.

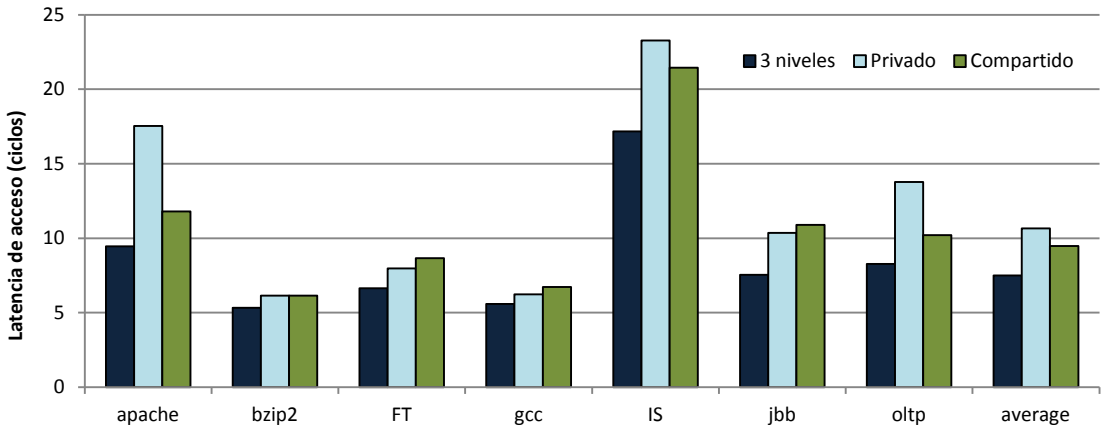

Figura 3-22. Comparación de la latencia de acceso teórica para una arquitectura de tres niveles y una de dos niveles (completamente privado o completamente compartido).

3.3.6 Validación del Modelo

Para validar la precisión del modelo, vamos a realizar simulaciones con alto nivel de detalle para cada aplicación y relación de capacidades de cache. Para ello, simulamos un sistema con 8 procesadores de 4 vías y ejecución fuera de orden, con una ventana de instrucciones de 128 entradas. Cada procesador tiene asociada una cache L1 privada de 32KB de Instrucciones y otra de Datos, una cache L2 privada de 4 vías estrictamente exclusiva con las L1, y entre todos los procesadores comparten una L3 inclusiva de 32

vías y distribuida en una SNUCA de 16 bancos y un protocolo de coherencia basado en directorio. La L2 y L3 comparten el área dedicada en el chip, de forma que incrementar el tamaño de L2 implica reducir el área dedicada a L3. Ambas caches disponen de un tamaño total de 16MB para compartir. La red de interconexión es un toro 4×4 con un banco de L3 conectado a cada switch, los procesadores conectados en damero, tal y como se ha visto anteriormente, y los controladores de memoria conectados en los encaminadores que no tienen procesador.

Desde la Figura 3-23 a la Figura 3-27, podemos ver como los resultados arrojados por el modelo teórico de latencia se aproximan a aquellos obtenidos mediante simulación de un sistema real. Sin embargo lo que buscamos no es tanto el valor absoluto, sino la localización del mínimo de la función y por tanto la relación de capacidad óptima entre L2 y L3. En las gráficas se muestran los resultados obtenidos por las distintas aplicaciones, así como una última gráfica (Figura 3-28) en la que cada punto ha sido obtenido promediando los resultados de las siete aplicaciones. Es importante tener en cuenta que el modelo analítico no tiene en cuenta efectos importantes como la inclusividad de L3, lo que provoca reemplazos indeseados en L1 y L2 cuando el tamaño de L3 disminuye, ni tiene en cuenta el incremento en latencia que se produce en L3 cuando existen indirecciones o el provocado por la congestión en la red, el acceso a los bancos o en los controladores de memoria.

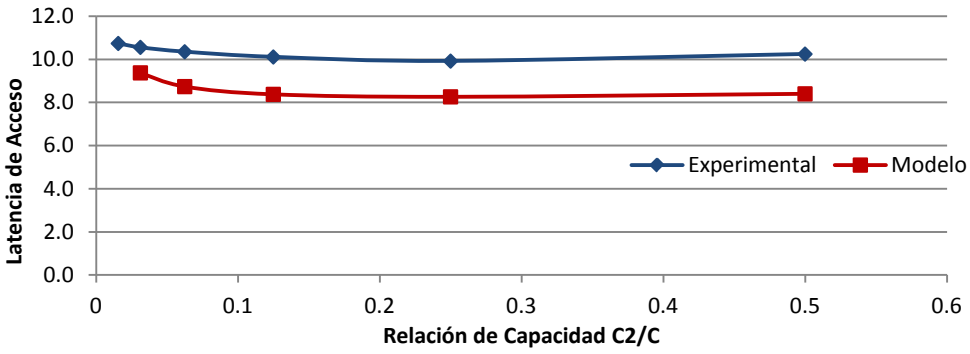

Figura 3-23. Comportamiento del modelo teórico de latencia frente a la simulación de la aplicación OLTP. Latencia de acceso global frente a la relación de capacidad (C_2/C).

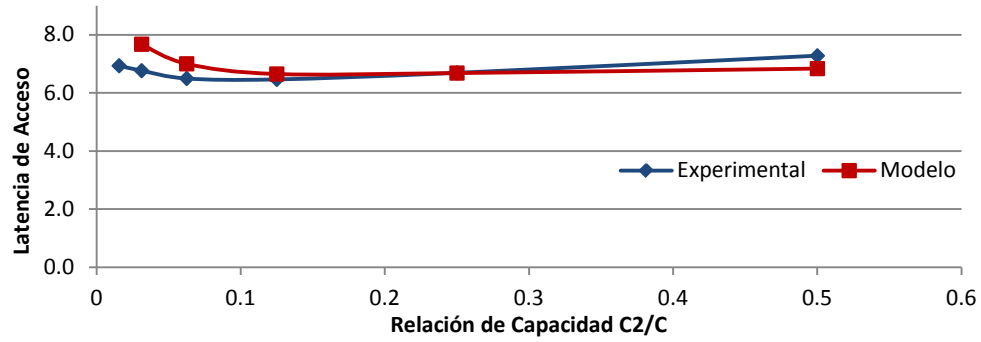

Figura 3-24. Comportamiento del modelo teórico de latencia frente a la simulación de la aplicación FT. Latencia de acceso global frente a la relación de capacidad (C_2/C).

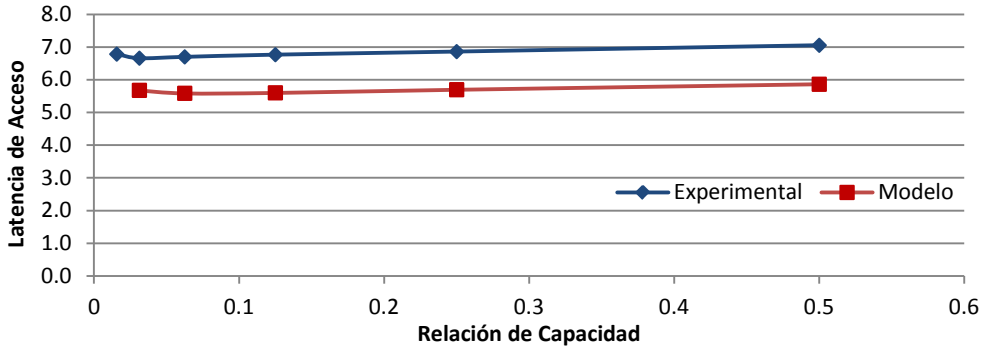

Figura 3-25. Comportamiento del modelo teórico de latencia frente a la simulación de la aplicación GCC. Latencia de acceso global frente a la relación de capacidad (C_2/C).

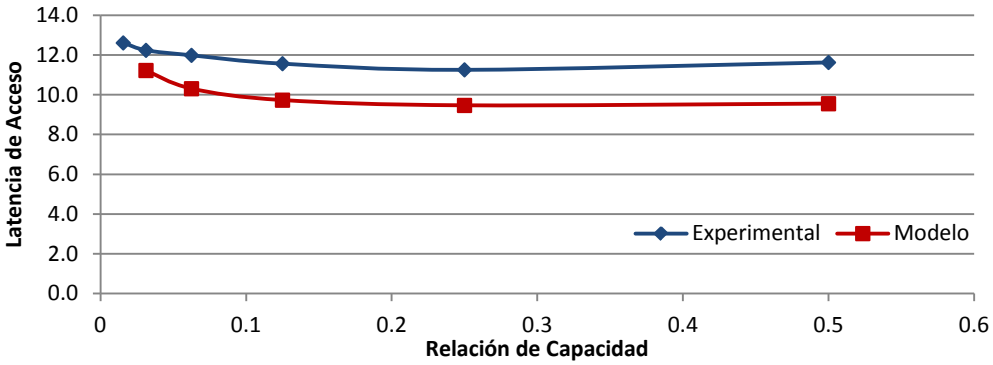

Figura 3-26. Comportamiento del modelo teórico de latencia frente a la simulación de la aplicación APACHE. Latencia de acceso global frente a la relación de capacidad (C_2/C).

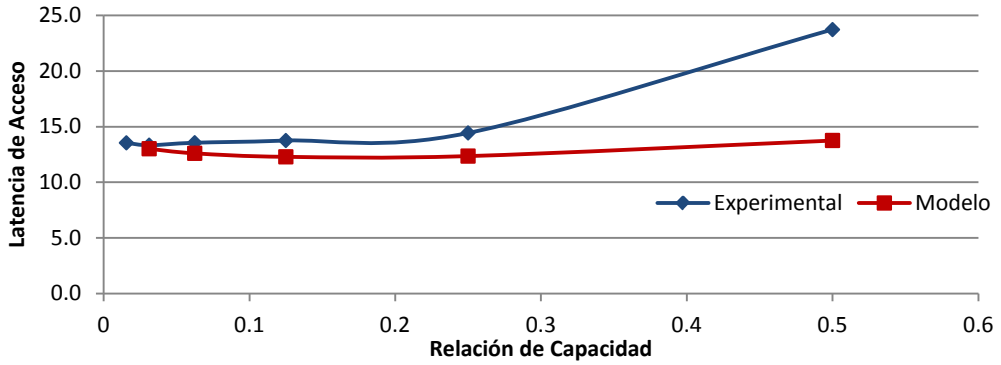

Figura 3-27. Comportamiento del modelo teórico de latencia frente a la simulación de la aplicación IS. Latencia de acceso global frente a la relación de capacidad (C_2/C).

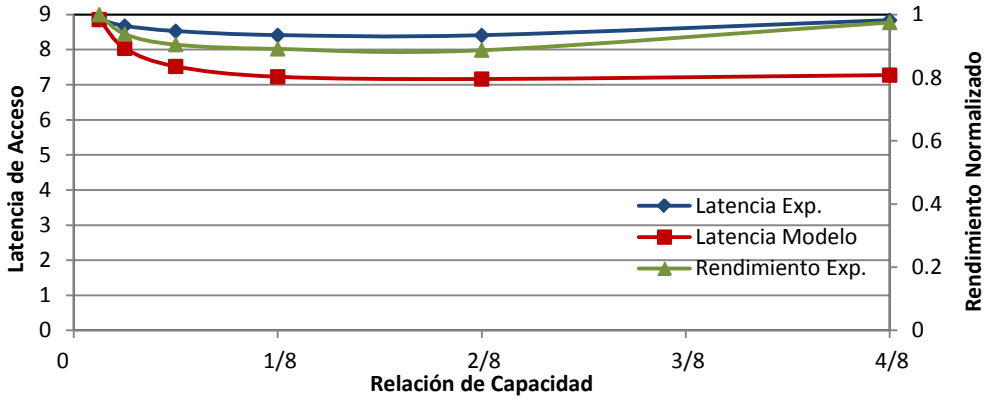

Figura 3-28. Comportamiento del modelo teórico de latencia frente a la simulación del promedio de todas las aplicaciones. Latencia de acceso global frente a la relación de capacidad (C_2/C).

Se aprecian dos diferencias notables entre la simulación y el modelo analítico en prácticamente todas las aplicaciones. Por una parte, el modelo teórico tiende a mostrar una progresión más brusca al inicio que lo que se observa en el comportamiento simulado. Esto es debido a que la ecuación (3-11) tiende a infinito cuando la capacidad que se dedica a uno de los niveles tiende a cero. Por otro lado, la gráfica de las simulaciones presenta mayor pendiente cuando se dedica 1MB a la L2 de cada procesador. Esto es debido a la inclusividad de L3 (cuya capacidad en ese caso es de 8MB, igual que la suma total de las L2), de forma que comienzan a producirse un gran número de invalidaciones externas indeseadas debido a esto. Este comportamiento no es recogido por el modelo analítico y de ahí la divergencia.

Parece claro que en el rango en el que la relación entre L2 y L3 es razonable, la simulación y los resultados analíticos difieren en menos de un 20%. El modelo tampoco pretende obtener un resultado preciso en latencia, pero sí en la búsqueda de la distribución óptima de capacidades. Podemos observar como los resultados

experimentales para las distintas aplicaciones presentan un mínimo en la latencia media de acceso. Este mínimo depende de cada aplicación, pero se observa cómo, tanto el modelo teórico como la simulación, sitúan este valor aproximadamente en $C_2/C = 1/8$. Teniendo en cuenta que cuando hablamos de C_2, nos referimos a la suma de la capacidad de las L2 de cada procesador. Así, $C_2/C = 1/8$ se corresponde con caches de nivel 2 de 256KB y L3 de 14MB. Es importante además, que aunque las diferencias en latencia experimentales no parecen significativas, el caso óptimo es en la media (Figura 3-28) un 12% más rápido que la configuración de peor caso, lo cual indudablemente tendrá un impacto significativo en el rendimiento del CMP.

Como vimos en la Figura 3-22, una arquitectura de dos niveles de cache obtiene peores resultados en rendimiento. En la Figura 3-29 presentamos los resultados obtenidos con las configuraciones expuestas anteriormente. Se observa cómo en todas las aplicaciones, salvo en IS, la decisión sobre privado o compartido es modelada correctamente. La discrepancia en IS es debida a que, para caches privadas de 2MB, IS alcanza el nivel de saturación, lo que no puede ser capturado por el modelo (algo que se aprecia en la Figura 3-27).

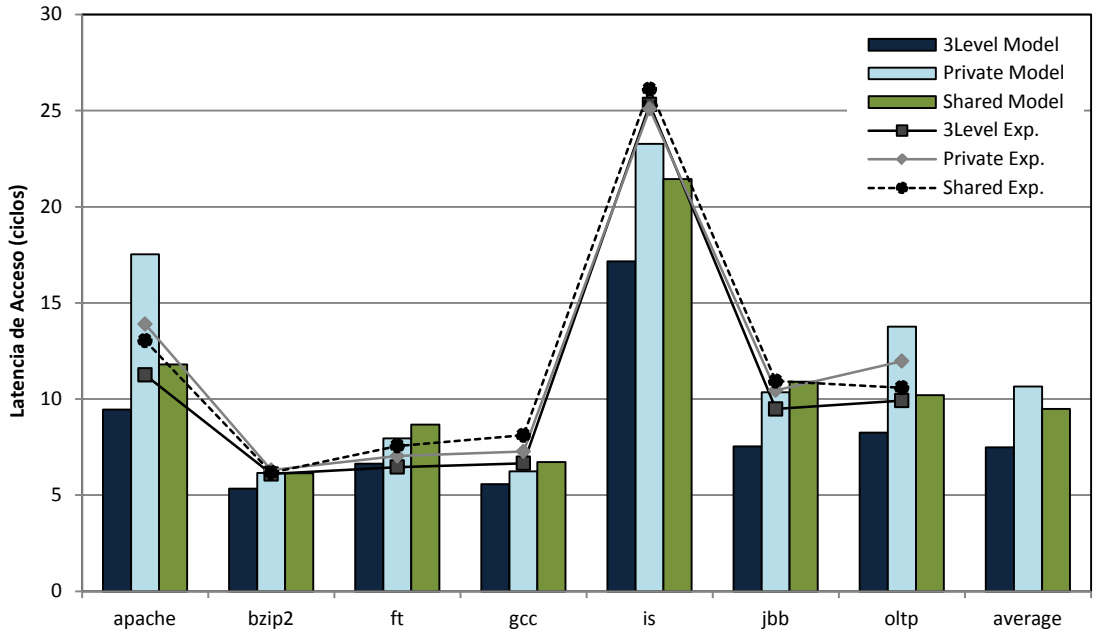

Figura 3-29. Comparación de la latencia de acceso media del modelo teórico y de las simulaciones para distintas configuraciones de cache.

3.3.7 Extendiendo el modelo: Sistemas y Aplicaciones de próxima generación

Hasta ahora hemos creado un modelo sencillo de latencia de acceso a la jerarquía de memoria y hemos caracterizado los parámetros de funcionamiento de acuerdo a una serie de aplicaciones, probando por último su adecuación a un sistema simulado realista. Para comprobar la extensión del modelo, vamos a probarlo con un sistema con un mayor número de procesadores (16) y más capacidad de cache (128MB dedicados a L2 y L3).

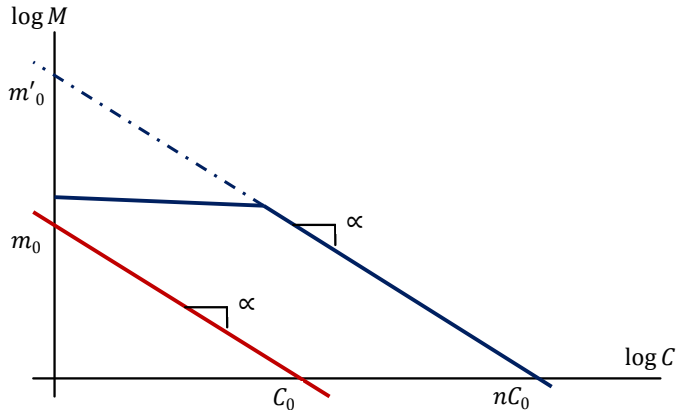

Figura 3-30. Dependencia de m_0 con el desplazamiento de la región lineal de la aplicación

No es posible modelar las aplicaciones que nos vamos a encontrar en el futuro, pero es posible escalar la caracterización del comportamiento en cache de aquellas que conocemos. Es de esperar, que las aplicaciones que se ejecuten en nuestro sistema escalen en requerimientos (proporcionalmente), de forma que el sistema siga trabajando en la región de dependencia de la cache (región lineal). Tal y como se observa en la Figura 3-30, si desplazamos los requerimientos de capacidad de nuestra aplicación en un factor n, m_0 se desplaza por un factor $n^{-\alpha}$. De acuerdo con la ecuación (3-3), equivale a decir que K_0 es multiplicado por un factor n (en nuestra propuesta, $n = \frac{C_0'}{C_0} = \frac{128}{16} = 8$). Para modelar los parámetros por tanto, tomamos $\propto_2$ y $\propto_3$ como los promedios de los valores representados en la Tabla 3-1 (0.45 y 0.80 respectivamente), y escalamos m_{02} y m_{03} de los valores obtenidos para 16MB, al tamaño de 128MB (n=8).

Para validar el modelo, se ejecutan 5 aplicaciones que se encuentran en la región lineal en un sistema de 128MB. Las aplicaciones son: IS en su tamaño superior y LU de la suite NPB, Zeus y Apache de las transaccionales y Omnetpp de las SPEC. Los resultados se comparan con aquellos obtenidos de sustituir los resultados escalados en la ecuación (3-11). En la Figura 3-31 se muestra la comparación entre las dos latencias medias,

incluyendo además la gráfica del rendimiento normalizado. De nuevo el modelo obtiene conclusiones similares a la experimentación que consume mucho más tiempo. Los resultados experimentales, obtenidos promediando los resultados de las distintas aplicaciones, muestran un mínimo en latencia difícil de definir, que se mueve entre $C_2/C = 1/8$ y $C_2/C = 1/16$ (1MB y 512KB por cada cache L2 privada respectivamente). Por su parte, el modelo teórico y el rendimiento experimental sitúan el mínimo, de forma más definida, cerca de $C_2/C = 1/8$ (1MB por cada cache privada). Hay que tener en cuenta que los parámetros del modelo no están obtenidos de la evaluación de aplicaciones reales, sino que han sido extrapolados de los datos obtenidos para aplicaciones que tenían un comportamiento lineal en el rango de los 16MB de cache.

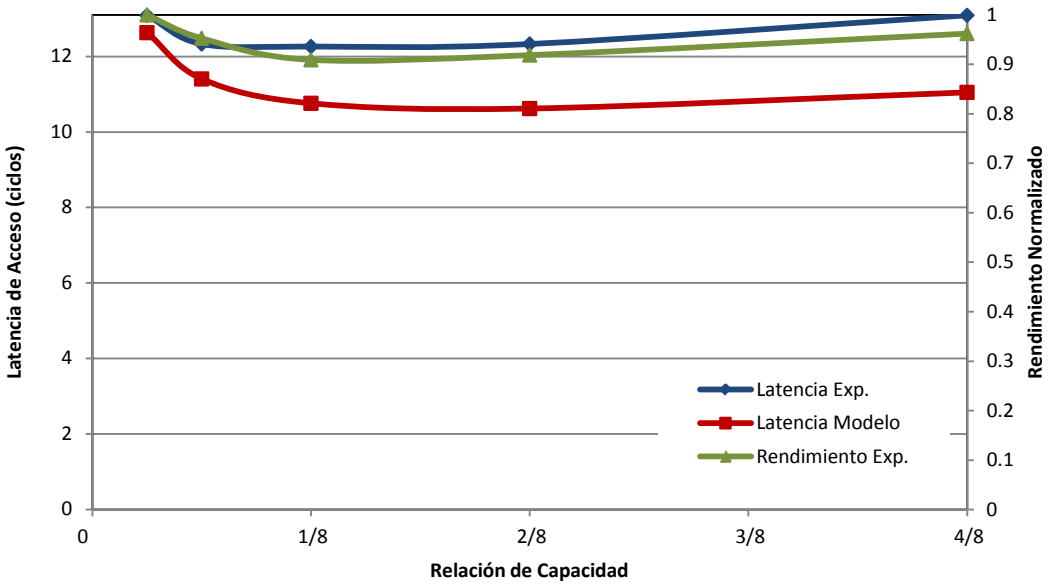

Figura 3-31. Resultados Teóricos y simulaciones sobre cinco aplicaciones en un sistema de 16 procesadores y 128MB de cache repartidos entre L2 y L3.

3.4 Conclusiones

En este capítulo hemos presentado la importancia que tienen ciertos parámetros en el diseño de la cache a la hora de distribuir la capacidad que tenemos disponible. Por una parte, se ha visto la relevancia que tiene el punto en el que se conectan los procesadores en la red de interconexión, tanto desde el punto de vista de congestión de la red, como de latencia de acceso media. Aun cuando los resultados obtenidos están ligados a una configuración determinada con una red de grandes dimensiones, la conclusión es semejante siempre que existan más nodos en la red que procesadores a conectar. La red juega un papel importante en el rendimiento de sistemas multiprocesador, especialmente

en una situación de decenas de núcleos [75], [76], y no solo es importante el diseño de los componentes de la misma, sino que es necesario planificar su topología y la forma en la que irán conectados los distintos componentes del sistema, tal y como también demuestran Abts et. al. en su artículo sobre el emplazamiento de los controladores de memoria. Los resultados presentados se refieren a sistemas en los que el número de procesadores sea menor que el número de nodos en la red, que, si bien no es una situación común en la actualidad, puede ser relevante si se cumplen las previsiones que apuntan a un crecimiento de la capacidad de la cache en el chip mucho mayor que el área dedicada a procesadores [66].

Por otra parte, hemos presentado un modelo sencillo que nos permite evaluar el comportamiento que tendrá nuestra jerarquía de memoria en distintas configuraciones, así como tomar decisiones sobre el número óptimo de niveles dentro de la jerarquía y la cantidad de área que dedicamos a cada uno de ellos. El modelo, aunque sencillo, ha sido comprobado con distintas configuraciones del sistema, siendo sus resultados relevantes cuantitativamente, e incluso con valores próximos a los obtenidos mediante costosas simulaciones. Los resultados además constatan, de una forma analítica, la conocida relación entre distintos niveles de cache como un factor $\times 4 - \times 8$ como óptimos en rendimiento.

Los resultados obtenidos en este capítulo son utilizados para la definición del sistema de referencia que usaremos durante el resto de la tesis. Las características fundamentales se presentan en la Tabla 3-2. Se trata de un sistema multiprocesador con tres niveles de cache, siendo los niveles más próximos al procesador, privados y exclusivos entre sí, y el último nivel de cache compartido e inclusivo respecto a los privados. Los procesadores con ejecución fuera de orden tienen 128 instrucciones en vuelo y un pipeline de 4 instrucciones de ancho. En general, simularemos sistemas de 8 procesadores, salvo cuando se indique lo contrario. Hay que tener en cuenta que las características se presentan por cada procesador, aunque la L3 sea compartida, de forma que la capacidad del último nivel de cache es de 8MB cuando hablamos de un CMP de 8 procesadores.

Tabla 3-2. Características principales del sistema.

Core Parameters	
Issue/Retire Width	4/4
ROB Size	128 entries
Functional Units	4 ALU, 2 LD-ST, 2 FP, 2 BR
Min. Latency Fetch-to-Dispatch	7 cycles
Branch Predictor	YAGS [77] 8K PHT, 4K Exception Table, 4k BTB, 32-entry RAS
Cache Hierarchy	
L1 Instruction	Private, 32KB, 4way, 2cycles, Pipelined, 64B block
L1 Data	Private, 32KB, 4way, 2cycles, Pipelined, 64B block
L2 Private	128KB, 4-way, 8cycles, 64B block, Exclusive with L1
L3 Shared S-NUCA	2 Banks per core, each one: 512KB, 16-way, 10-cycles, 64B block, Inclusive with Private Caches
Coherence Protocol	In-cache MESI Directory
Interconection Network	Folded Torus
Memory Scheduling	Unlimited Store-sets

4 NUCA Privada/Compartida

4.1 Introducción

Dada la gran cantidad de cache dentro del chip que es previsible que dispongan y necesiten los sistemas multiprocesador en el futuro [66], es necesario una organización de la arquitectura de cache que minimice la latencia de acceso a la misma mientras reduce el uso del ancho de banda a memoria.

Es difícil determinar la mejor política para la arquitectura de cache dentro del chip, ya que ésta depende de las aplicaciones que están siendo ejecutadas y su grado de compartición. Así, la arquitectura de cache debe ser lo suficientemente flexible como para adaptarse al comportamiento de las mismas, aumentando la tasa de aciertos y la utilización de los recursos disponibles dentro del chip con el fin de minimizar los accesos fuera del chip, a la vez que se reduce la interferencia entre procesos y la latencia de acceso *on-chip*.

Una forma eficiente y conocida de organizar una gran capacidad de cache reduciendo el tiempo medio de acceso es la arquitectura de Cache de Acceso No-Uniforme (NUCA) tal y como se ha introducido en la sección 2.3. Sin embargo, incluso una solución de este tipo requiere mejoras en el algoritmo de distribución de los datos con el aumento del número de procesadores y la capacidad de cache *on-chip*, pues la latencia de acceso puede ser importante y no se garantiza el aislamiento de los datos de los distintos procesos corriendo en el sistema. La literatura en torno a las NUCA es amplia e incluso existen implementaciones comerciales [28], y muchas son las soluciones que abordan este problema. A lo largo del capítulo iremos desgranando las distintas alternativas planteadas y explicaremos nuestra propuesta.

4.2 Arquitecturas de Cache de Acceso no Uniforme de Direccionamiento Estático (S-NUCA)

Muchos de los CMP actuales están basados en sistemas de memoria compartida. Esta solución tiene el problema de tener un tiempo de acceso elevado respecto a su alternativa privada, pues el tamaño del banco es mucho mayor, aunque tiene la ventaja de

proporcionar un uso más eficiente del espacio de cache y simplifica la implementación de la coherencia y consistencia de los datos, reduciendo además el número de réplicas en el último nivel de cache. El objetivo de la arquitectura NUCA es distribuir la cache en pequeños bancos de rápido acceso conectados mediante una red punto a punto. Con esto se consiguen dos cosas principalmente, por un lado, al tener un mayor número de bancos, se reduce la contención en los bancos, aumentando el número de peticiones simultáneas que pueden ser atendidas concurrentemente y mejorando por tanto el tiempo de acceso. Por otra parte, el distribuir los bancos estos pueden ser más pequeños y con menor latencia, habiendo bancos cercanos al procesador, y por tanto con una latencia de acceso menor que la de una cache de acceso uniforme, y bancos situados más lejos y con una latencia mayor debido a la red de interconexión. En comparación con una cache de acceso uniforme, conseguimos un mayor ancho de banda en número de peticiones en vuelo, a la vez que típicamente reducimos la latencia media de acceso, que pasa a ser no uniforme. Los principales inconvenientes son el incremento en el número de controladores de cache y la mayor complejidad de los *routers*.

Aunque existen diferentes formas de distribuir los datos entre los distintos bancos de la NUCA, nos vamos a centrar en la opción de menor coste de implementación: la distribución de datos estática o SNUCA, de forma que cada dato puede estar solamente en un único banco en función de su dirección de memoria. Con este algoritmo de direccionamiento, los datos se almacenan en el banco de cache determinado por la dirección del mismo, de forma que se eligen una serie de bits de la dirección del bloque para determinar el banco al que corresponde (típicamente los menos significativos para homogeneizar la distribución de los datos entre los distintos bancos, maximizando la utilización de los recursos).

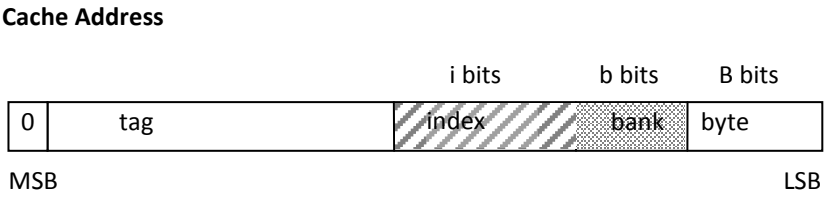

Figura 4-1. Interpretación típica de los distintos bits de la dirección de memoria en el direccionamiento en una SNUCA, usando los bits menos significativos para el direccionamiento del banco.

Tal y como vemos en la Figura 4-1, este tipo de direccionamiento estático tiene la ventaja de requerir muy poco tiempo para identificar el banco donde se aloja un bloque en la LLC, ya que el banco en el que se encuentra un dato en concreto viene determinado directamente de la interpretación de los *b* bits dentro de la dirección de memoria. Podemos ver un ejemplo concreto para la S-NUCA de la Figura 4-2.

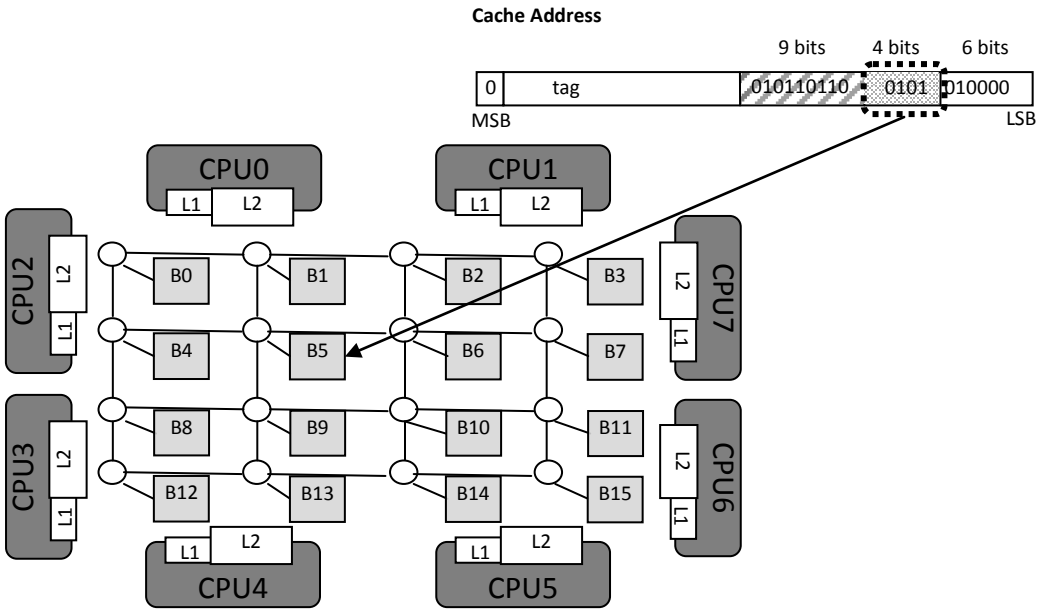

Figura 4-2. Ejemplo concreto de direccionamiento estático para un sistema con 8 MB de cache repartidos en 16 bancos NUCA (b=4), bloques de cache de 64bytes (B=6) y 16 vías en cada banco (i=9).

Se puede observar que en una arquitectura de este tipo, el tiempo de respuesta de la cache depende de la posición del procesador que realiza la petición y la dirección del bloque, puesto que el banco en el que se encuentra alojado depende únicamente de su dirección de memoria y no hay otro criterio que lo rija.

4.3 Mejorando la SNUCA

Investigaciones posteriores tratan de sacar partido a la distribución de la NUCA para sistemas multiprocesador, de forma que los datos estén más cerca de los procesadores que los utilizan. Así, sus mismos creadores desarrollaron la DNUCA [21] (Dinamic NUCA). Sirviéndose de las ventajas de la NUCA, la DNUCA, mediante una política de migraciones sucesivas, trata de conseguir que los datos más frecuentemente utilizados por un procesador en particular, estén en los bancos que le queden más próximos. En este algoritmo, los distintos bancos son agrupados en sectores, de forma que un dato solo puede estar en un único banco de cada sector, y la búsqueda se realiza de forma secuencial o paralela por los distintos sectores. Aun cuando la solución fue propuesta

inicialmente para sistemas de un único núcleo, es posible aplicar la misma solución a sistemas CMP. Sin embargo, hay que tener en cuenta que habrá al menos tantos sectores como procesadores, y normalmente uno más que se corresponde a la zona central o compartida. Cuando un procesador accede a cierta posición de memoria, el sistema evalúa si es necesario migrar el dato hacia un sector más cercano al procesador que lo ha solicitado (normalmente mediante un umbral de peticiones que se debe superar). Con este movimiento de bloques de cache entre bancos, conocer la localización de un bloque concreto no es trivial. Las pruebas que se presentarán para la comparación con nuestra propuesta se realizaron con una implementación del algoritmo de búsqueda perfecto en CMP propuesto por Beckmann [78], esto es, el procesador tiene conocimiento del banco en el que se encuentra el dato antes de realizar la petición. Esto aumenta considerablemente la complejidad de la solución respecto a la arquitectura base, la SNUCA. Esta solución ha demostrado ser eficiente en sistemas con un único procesador, sin embargo presenta algunos problemas en su aplicación a CMPs. Esto se debe a que, en aplicaciones paralelas, se produce un exceso de migraciones de datos compartidos solicitados por más de un procesador, que terminan en la posición de los bancos compartidos localizados en los bancos centrales. Lo que en la práctica es equidistante a un direccionamiento estático, pero con un exceso de tráfico y un consumo mucho mayor de energía por los procesos extra de lectura y escritura en los bancos, así como reemplazos adicionales. Posteriores trabajos han tratado de paliar estos efectos mediante cambios en la arquitectura que favorezca el acceso a los bancos compartidos [64] [79].

A ésta solución han sucedido alternativas que mejoran del rendimiento de la cache basadas en la NUCA, tratando de aproximar los datos privados de los distintos procesadores, mientras mantienen los datos compartidos equidistantes [80]–[82]. Estos trabajos atacan los problemas de compartición, migración o réplica de datos para mejorar la latencia en chip mediante mejoras en el emplazamiento de los datos, réplica de bloques o uso de los *tags*.

De forma semejante a la propuesta que aquí se presenta, el trabajo de Chang et. al. [81] sobre caches cooperativas trata de reducir la latencia de acceso a los datos partiendo de caches privadas, mientras crea una capa, virtualmente compartida, para aumentar la capacidad efectiva de la cache. Sin embargo, necesita un mecanismo para conseguir que las caches privadas sean conscientes del comportamiento global del sistema, bien mediante *snooping* o el uso de un mecanismo centralizado que propague información

constante sobre el estado de las replicas y bloques compartidos, lo que limita su capacidad de adaptación.

Huh et. al. [73] analiza diferentes grados de compartición de particionados estáticos y dinámicos para NUCA, llegando a la conclusión de que un mapeado estático con grado de compartición 2 o 4 puede proporcionar buenos resultados de latencia, mientras que el mapeo dinámico puede mejorar el rendimiento a costa de una mayor complejidad y consumo de energía.

Dybdahl y Stenström [24] proponen una estrategia de cache cooperativa basada en NUCA que elimina la interferencia entre procesos y explota los beneficios de las caches privadas mediante el uso de "shadow tags" para determinar cuándo añadir una vía extra a la porción privada de la cache. Esta solución requiere del uso de un mecanismo centralizado que decide el mejor particionado posible y controla el reemplazo de todas las caches compartidas, lo que hace que sea una alternativa poco escalable para caches de gran tamaño o CMPs con un gran número de procesadores. Además, el mecanismo de particionado toma sus decisiones después de un gran número de fallos, lo que supone una menor capacidad de reacción a los cambios de comportamiento de las aplicaciones.

4.4 SP-NUCA

En este apartado introduciremos la Arquitectura de Cache de Acceso No Uniforme Privada/Compartida (SP-NUCA), una arquitectura eficiente para el último nivel de cache (LLC). Siguiendo el camino de un sistema híbrido, capaz de aunar los beneficios en el uso eficiente de la memoria de un sistema compartido, pero con una latencia reducida propia de los sistemas distribuidos, hemos ideado la SP-NUCA. La SP-NUCA se basa en la idea original de una cache compartida con arquitectura NUCA de direccionamiento estático (SNUCA). Al igual que la DNUCA, el objetivo es que los bloques de datos privados de cada procesador estén situadas lo más cerca posible del mismo, manteniendo los datos compartidos equidistantes a todos los procesadores. A igualdad de espacio en la cache, la SP-NUCA podría proveer de una mayor cache privada que una arquitectura con un nivel extra de cache, o ir aumentando el espacio compartido en la medida que una aplicación paralela lo necesite.

La SP-NUCA es una arquitectura de cache compartida de último nivel, que emula un nivel de caches privadas mediante un doble direccionamiento estático (lo que reduce su complejidad de implementación).

Los bancos almacenan datos tanto públicos como privados, de forma que en un mismo *set* se pueden tener vías con bloques usados por un solo procesador como compartidos entre varios. Dado que cierto número de bloques de la cache pueden compartir el mismo direccionamiento estático y privado, se hace necesario incluir un bit que permita discernir si un bloque pertenece a una categoría u otra. Todos los bancos actúan como bancos compartidos de todos los procesadores y como privados de alguno de ellos. El grupo de bancos que se consideran privados para un procesador se corresponden con los más próximos al mismo, y sus datos privados pueden ser accedidos únicamente por él. El direccionamiento para estos bancos privados es el correspondiente a una SNUCA, pero con un número más reducido de bancos (tantos como correspondan a cada procesador). Todos los bancos del último nivel de cache actúan como bancos compartidos, y su direccionamiento es el mismo que el que tendría una SNUCA convencional. De esta forma, un banco de cache puede almacenar datos privados del procesador al que está ligado, y/o datos compartidos por cualquiera de los procesadores del chip (y no necesariamente por el procesador más próximo).

En la Figura 4-3 vemos el ejemplo de una SP-NUCA, donde los dos bancos resaltados crean la porción privada del CPU0, y junto con el resto, forman la región compartida.

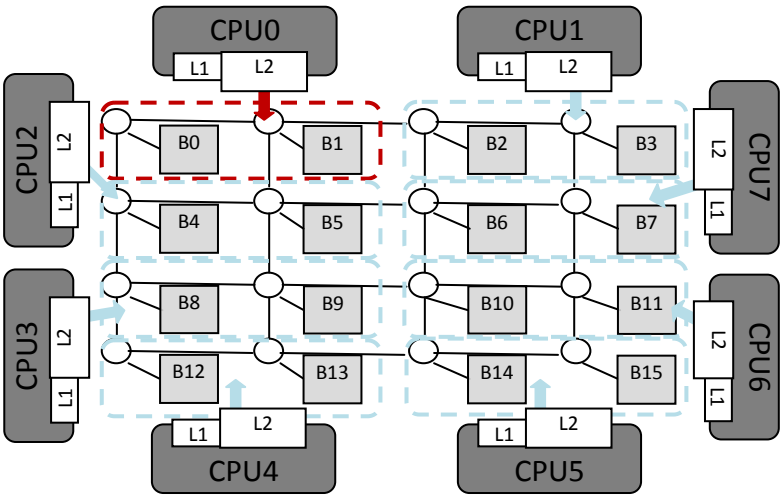

Figura 4-3. Esquema de una SP-NUCA de 8 procesadores y 16 bancos de cache. Los dos bancos resaltados forman la región privada del procesador 0. Los 16 bancos en su conjunto forman la región compartida.

Para que la SP-NUCA pueda funcionar, es necesario un direccionamiento distinto para los bancos privados de cada procesador, y los bancos compartidos. Con el fin de simplificar los controladores de cache y la inclusión de futuros mecanismos de búsqueda, se hace coincidir parte de la dirección del banco privado y público usando los mismos bits de la dirección de memoria como se observa en la Figura 4-4, en la que los bancos son mapeados con los bits menos significativos para aumentar la dispersión de los datos en la cache. Aún cuando los tamaños de los *tags* son distintos para ambos tipos de dato, hay que tener en cuenta que el banco tiene que tener capacidad para almacenar el de mayor tamaño (bloque privado), lo que aumenta ligeramente el tamaño del tag respecto a una S-NUCA convencional.

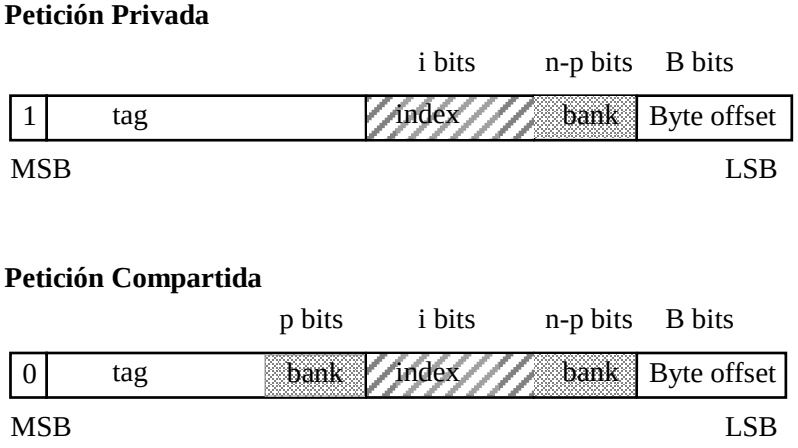

Figura 4-4. Interpretación de la dirección de memoria para el direccionamiento a un banco privado o compartido.

En la Figura 4-4 se puede observar cómo se interpreta la dirección de memoria en función de si la petición debe ir a un banco público o privado, donde "*B*" es el tamaño en bytes del bloque de cache, "*i*" es el índice del set, "*n*" es el logaritmo en base dos del número total de bancos de la NUCA y "*p*" el logaritmo en base dos del número de procesadores del sistema. Como se puede observar con este esquema, el mapeo de los bancos comparte parte de los bits, tanto para la parte compartida como la privada, y el índice del *set* es el mismo en ambos casos. La SP-NUCA requiere, como hemos dicho antes, espacio para almacenar el *tag* de mayor tamaño en cada uno de sus bloques, pues todos pueden comportarse tanto como datos privados como compartidos, lo que implica un *tag* "*p*" bits mayor que el de una S-NUCA convencional.

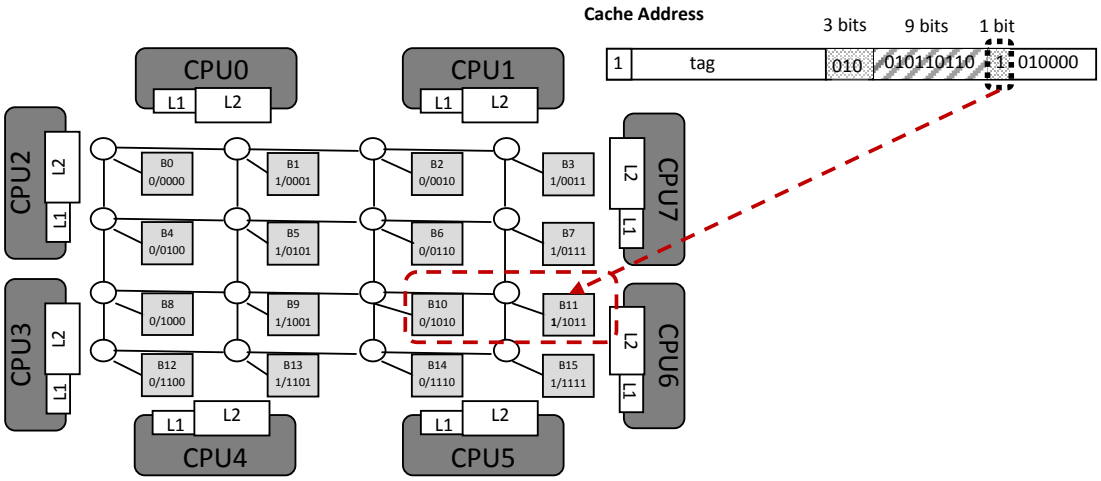

Figura 4-5. Ejemplo de direccionamiento privado en SP-NUCA para un sistema concreto con 8 MB de cache repartidos en 16 bancos NUCA (b=4), bloques de cache de 64bytes (B=6) y 16 vías en cada banco (i=9). Petición de una dirección concreta asociada al núcleo *CPU6*.

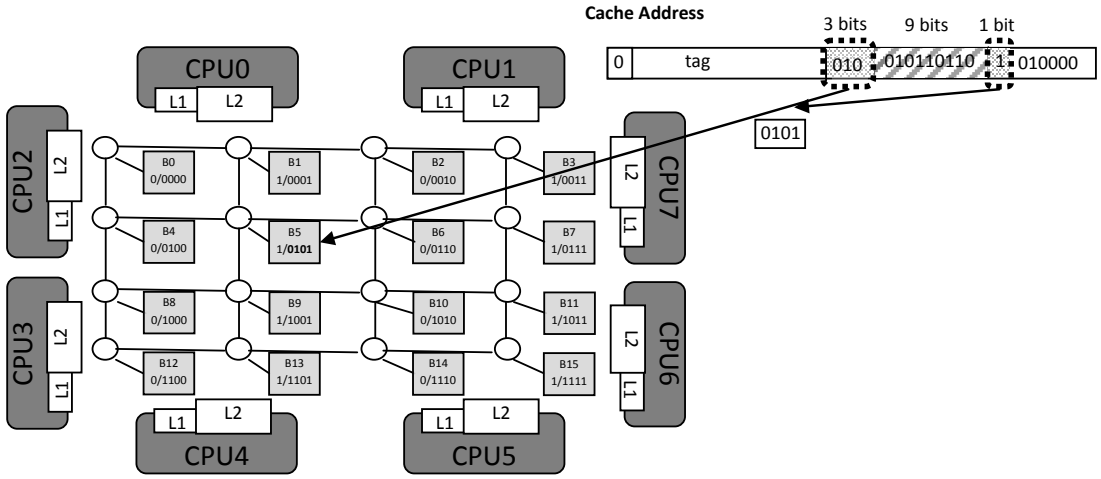

Figura 4-6. Ejemplo de direccionamiento compartido en SP-NUCA para un sistema concreto con 8 MB de cache repartidos en 16 bancos NUCA (b=4), bloques de cache de 64bytes (B=6) y 16 vías en cada banco (i=9). Petición de una dirección concreta asociada al núcleo *CPU6*.

En las Figuras 4-5 y 4-6 se puede ver un ejemplo para un caso concreto, utilizando la dirección de memoria del ejemplo de la Figura 4-2. Se observa como, la dirección del banco privado (Figura 4-5) se representa con los bits menos significativos de la dirección de memoria (después del bloque), en este caso concreto, dado que a cada procesador le corresponden dos bancos privados, es suficiente con 1 bit. La dirección del banco compartido (Figura 4-6) se fracciona dentro de la dirección de memoria, haciendo coincidir el índice del *set* en ambos tipos de direccionamiento, simplificando las migraciones en aquellos casos en los que coincida el banco privado y compartido (cambiando un único bit), y facilitando alguna de las mejoras propuestas posteriormente.

Para determinar si un dato debe ser privado o compartido, se determina que todos los datos llegan a la cache provenientes de la memoria en forma de dato privado del procesador que realiza la petición, y son alojados en el banco privado correspondiente. A no ser que otro procesador requiera la línea, ésta permanece en el banco privado hasta que es reemplazada, consiguiendo así una latencia baja mientras permanezca en el banco privado. Tan pronto como otro procesador reclama la misma línea, el dato es marcado como compartido, migrado al banco correspondiente de la parte compartida y no se mueve de allí hasta que, tras un reemplazo, abandona la cache. Los motivos que llevan a esta decisión son los siguientes:

- Los datos en los bancos privados son los que tienen una menor latencia de acceso, por lo que todo dato, por defecto, será considerado como tal hasta que se demuestre lo contrario.
- Los datos privados son accedidos por un único procesador, en el momento en el que dos procesadores piden el mismo dato, ese bloque pertenece a un dato compartido, y es susceptible de ser accedido múltiples veces por distintos procesadores. Aunque es posible ralentizar este proceso, o plantear alternativas (como las réplicas en el caso de múltiples lectores), en una primera aproximación las migraciones se hacen tan pronto como se detecta que el dato es compartido.
- Es conveniente evitar el movimiento continuo de datos entre las partes privadas y compartidas, tanto por los problemas que comportan en el consumo de energía y gestión de reemplazos debido a las múltiples escrituras del mismo bloque en distintos bancos, como porque los bloques en tránsito reducen temporalmente la capacidad efectiva de la cache de una forma o de otra.

Una vez definido la forma en la que los bloques se convierten en privados o compartidos, es necesario definir el algoritmo de búsqueda de datos. La arquitectura que se presenta, SP-NUCA, ha sido estudiada con protocolos de coherencia basados tanto en token [35] como en directorio, siendo este último caso implementado mediante un directorio *in-cache* distribuido, para aprovechar la naturaleza “banqueada” de la NUCA. En ambos casos el procedimiento para resolver una petición a la jerarquía de memoria es el mismo. En caso de que se produzca un fallo en los niveles superiores de la cache (más cercanos al procesador), la petición es lanzada al banco del último nivel de cache correspondiente a la parte privada del procesador demandante. Si el dato tampoco se

encuentra en ese banco, la petición es trasladada al banco compartido correspondiente. Tanto si el bloque se encuentra en la parte privada como en la compartida, se envía una copia del mismo a la cache de nivel 1 del procesador que hace la petición. Si el dato no se encontrase en la parte compartida, la petición es reenviada a todos los demás bancos privados del último nivel de cache de los demás procesadores y a memoria. Si es otro banco el que responde, el bloque migra hacia el banco correspondiente de la porción compartida. Como consecuencia, la próxima vez que un procesador requiera ese mismo dato, lo encontrará en la zona compartida y la latencia de acceso será menor al evitar la indirección a bancos privados remotos. En el caso de que la respuesta proceda de memoria, el dato es almacenado en el banco privado correspondiente del último nivel de cache, tal y como se ha explicado anteriormente.

Hay que notar que, con el fin de que las caches de los niveles próximos al procesador puedan hacer sus reemplazos en el banco adecuado del último nivel, el bloque debe mantener información de su estado privado/compartido. El protocolo con el que vamos a trabajar en esta tesis es un protocolo de coherencia MESI basado en directorio distribuido, aunque debe entenderse que el protocolo de coherencia de la SP-NUCA es más complejo que la versión directorio de una arquitectura tradicional.

4.5 Asignación de Capacidad Privado-Compartido

Uno de los principales retos en el diseño de una arquitectura híbrida privada/compartida es decidir cuántas vías de cada *set* se dedican a datos privados y cuántos a compartidos. Así se consideran diversas opciones:

4.5.1 Particionado Estático

Por una parte, la solución más directa es crear un particionado estático de las vías de los bancos de cache, de forma que un porcentaje de vías se reserva automáticamente para almacenamiento privado, y el resto se dedica a datos compartidos. Aun cuando esta decisión se justifica por el bajo coste de implementación que supone, es descartada inmediatamente ya que un particionado de este tipo puede provocar un uso ineficiente de la cache en caso de que una de las dos particiones no use completamente el espacio que se le concede mientras otra tiene requisitos mayores. En un caso semejante, es preferible optar por una arquitectura con un nivel más de cache, consiguiendo mejores resultados.

4.5.2 Shadow Tags

Una alternativa más versátil es partir de un particionado estático y desplazar la frontera según sean los requisitos del sistema y de cada uno de los procesadores. Una forma de implementar esto es mediante "*shadow tags*" [83]. Este sistema fue descrito por Suh et. al. para un particionado dinámico de las caches, y fue extendido por Dybdahl y Stenstrom [24] para su uso en NUCAs. La idea es establecer dinámicamente el número de vías para cada una de las partes privadas y compartidas, haciendo un seguimiento de las necesidades de cada una de las partes. Nuestra propuesta de *shadow tags* tiene algunas diferencias con respecto a las planteadas por estos autores. Para empezar, no disponemos de un sistema centralizado de monitorización, sino que la evaluación se realiza en cada set. Esto permite una mayor flexibilidad al determinar la porción privada y compartida, favoreciendo un uso más eficiente de la cache. Por otra parte, la evaluación no se hace tras una determinada cantidad de fallos, sino que se integra en el propio algoritmo de reemplazo, lo que simplifica el algoritmo y agiliza la toma de decisiones.

El algoritmo de *shadow tags* evalúa el impacto que supone que una porción de la cache, privado o compartido, pierda su bloque más antiguo con respecto a que la porción opuesta gane ese bloque. Para saber los efectos que supone perder el bloque más antiguo, se registran los aciertos en el bloque menos recientemente usado (LRU) de cada una de las porciones mediante una pareja de contadores por *set*. De la misma forma, para evaluar los efectos de añadir un bloque, cada *set* tiene unos *shadow tags* que almacenan las últimas direcciones reemplazadas en cada porción. Como en el caso anterior, se registran los aciertos que se dan en estos *tags*, lo que equivale a haber acertado en la cache si se dispusiese de un mayor número de bloques en esa porción. Cuando se hace un reemplazo en un determinado set, el algoritmo compara el contador asociado al *shadow tag* de la porción correspondiente frente al contador asociado al bloque LRU de la otra porción. Si la diferencia entre ambos contadores supera un cierto umbral, se toma la decisión de reemplazar el bloque LRU de la porción opuesta al dato entrante (privado o compartido). En caso contrario, el algoritmo de reemplazo escoge el bloque LRU correspondiente a la porción del nuevo dato.

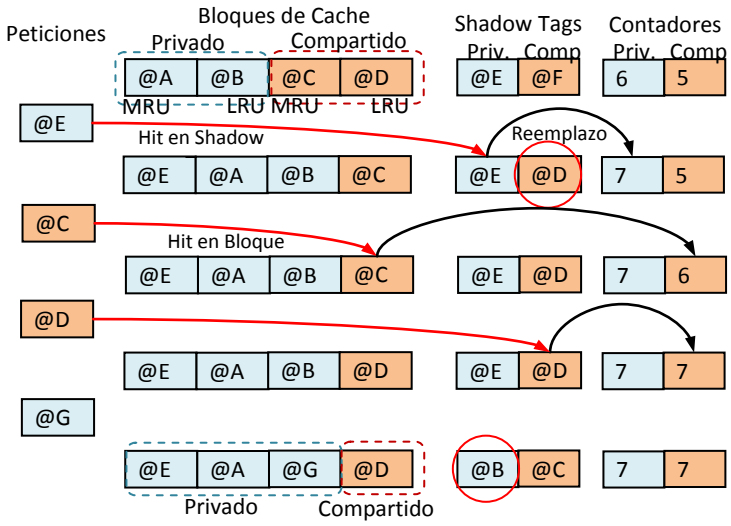

Figura 4-7. Ejemplo de Reemplazo usando Shadow Tags, para un set de cache con 4 vías.

En la Figura 4-7 podemos observar un ejemplo del proceso que sigue el algoritmo de *shadow tags* tras una serie de peticiones sobre un set de cache de 4 vías. Partimos de un estado inicial en el que hay dos vías dedicadas a datos privados (a la izquierda) y dos a datos compartidos (a la derecha), con direcciones almacenadas en los registros de *shadow tags*, uno por cada tipo, debido a reemplazos previos. Los contadores comienzan con valores arbitrarios. Cuando llega la primera petición se trata de una petición de un dato privado de dirección @E que no se encuentra en la cache, y que se corresponde con la dirección guardada en el *shadow tag* de la porción privada. Por esto, se actualiza el valor del contador de la porción privada, pues de haber tenido una vía más, la cache no habría fallado. Además, dado que el contador de la porción privada es mayor que el de la parte compartida, y suponiendo un umbral de crecimiento de 0, el algoritmo de reemplazo decide expulsar el bloque LRU de la porción compartida, actualizando el *shadow tag* correspondiente. La siguiente petición se corresponde con un bloque presente en la porción compartida de la cache, @C, dado que es el bloque LRU, se actualiza nuevamente el contador correspondiente, pues en caso de que la porción compartida perdiese un bloque, la petición habría sido un fallo. En este caso no hay reemplazo, y los *shadow tag* quedan como estaban. Con el bloque @D ocurre algo semejante al primer caso, @E, salvo que al estar igualados los contadores, el reemplazo se realiza sobre la porción compartida a la que pertenece @D. Finalmente, la petición de @G no se corresponde con ningún bloque de la cache ni de los *shadow tags*, de forma que entra en

la parte privada, y dado que los contadores siguen igualados, reemplaza al bloque LRU de esa misma porción.

En el análisis de este mecanismo de reemplazo se han probado distinto número de *shadow tags* por *set*, así como distintos umbrales para decidir si se produce un cambio en la categoría de un determinado bloque. En las simulaciones se ha comprobado que, a medida que aumenta el número de *shadow tags* almacenados, el funcionamiento es mejor, como parece razonable. Por otra parte, el mejor umbral de crecimiento es el mínimo, esto es cero, con el que se logra un ajuste mucho más dinámico, ya que umbrales más altos provocan un comportamiento conservador que tarda en adaptarse al comportamiento dinámico de las aplicaciones en ejecución. Hay que tener en cuenta que el coste de implementación de este algoritmo es relativamente elevado, ya que requiere la presencia de tags adicionales, contadores y mantener el bloque LRU para cada una de las porciones de cada "set", lo que duplica el coste del algoritmo de referencia.

4.5.3 Siempre Roba

Este algoritmo es una versión muy simplificada del caso anterior: cuando hay que hacer un reemplazo, el algoritmo siempre escoge como víctima el bloque LRU de la porción contraria al dato que entra en el banco. Esto es, si nos llega un dato privado de memoria y no hay bloques libres en la línea, el algoritmo de reemplazo escoge el bloque LRU de la parte compartida, si existe. El coste de implementación de este sistema es el doble de un algoritmo LRU normal, ya que hay que llevar cuenta de los bloques más antiguos para ambas porciones de cache, privada y compartida.

4.5.4 LRU Global

Finalmente se opta por la decisión más simple y a la vez más efectiva de las probadas. Dado que no se requiere ningún tipo de orden entre los datos privados o compartidos, el cambio de una vía dentro de un set de privada a compartida o viceversa debiera tener un coste bajo (simplemente cambiando un bit). Así, la decisión se delega en el algoritmo de reemplazo utilizado, considerando el banco de cache como un todo a la hora de hacer un reemplazo. Partiendo del caso concreto LRU, no se tiene en cuenta ninguna información adicional a la hora de elegir la víctima en caso de necesitar eliminar un bloque de la cache, sino que se expulsa el bloque dentro del set que fue accedido con mayor antigüedad. Esto es, si una línea llega a un banco proveniente de memoria, por tanto es un dato privado, el algoritmo de reemplazo debe decidir qué línea debe ser eliminada para

hacer hueco a la que acaba de llegar. Si los datos privados almacenados en el banco son usados con frecuencia, la porción dedicada a los mismos tenderá a crecer, pues el algoritmo de reemplazo elegirá con mayor probabilidad un dato compartido para ser reemplazado, cambiando la línea de un dato compartido a un dato privado. En caso contrario, no se produce ningún cambio en la distribución privado/compartido en ese reemplazo. A diferencia de los casos anteriores, este mecanismo de particionado no supone ningún incremento en complejidad sobre la arquitectura base, pues usa el algoritmo de reemplazo existente sin modificar.

4.5.5 Análisis Comparativo

Como se puede ver en la Figura 4-8, tras estudiar distintos algoritmos de reemplazo, algunos basados en trabajos del ámbito [24], el sistema que obtuvo el mejor rendimiento, con un menor coste, fue el que evaluaba de forma global el set (en nuestro caso LRU) de manera que no se hace distinción entre vías privadas o compartidas. Este algoritmo consigue los mejores resultados en media, y parece adaptarse adecuadamente a los distintos comportamientos de las cargas de trabajo aplicadas. Por otra parte, observamos cómo en la solución con *shadow tags*, usando 8 *tags* por bloque, los resultados de rendimiento no son mejores que los obtenidos por LRU con una complejidad mucho mayor. Por el contrario, el algoritmo “siempre roba” no funciona mal en cargas multiprogramadas, pero sufre en exceso por la movilidad de los bloques cuando se trata de aplicaciones *multithread*, obteniendo resultados pobres. Finalmente el caso estático se muestra ineficiente en muchas de las cargas, especialmente multiprogramadas, donde las 12 vías privadas de 16 posibles, se quedan escasas. Así pues optamos por emplear el algoritmo privado/compartido basado en LRU Global, que no solo da excelentes resultados, sino que además implica un incremento nulo en la complejidad e implementación del algoritmo de reemplazo respecto a un banco de NUCA estática convencional.

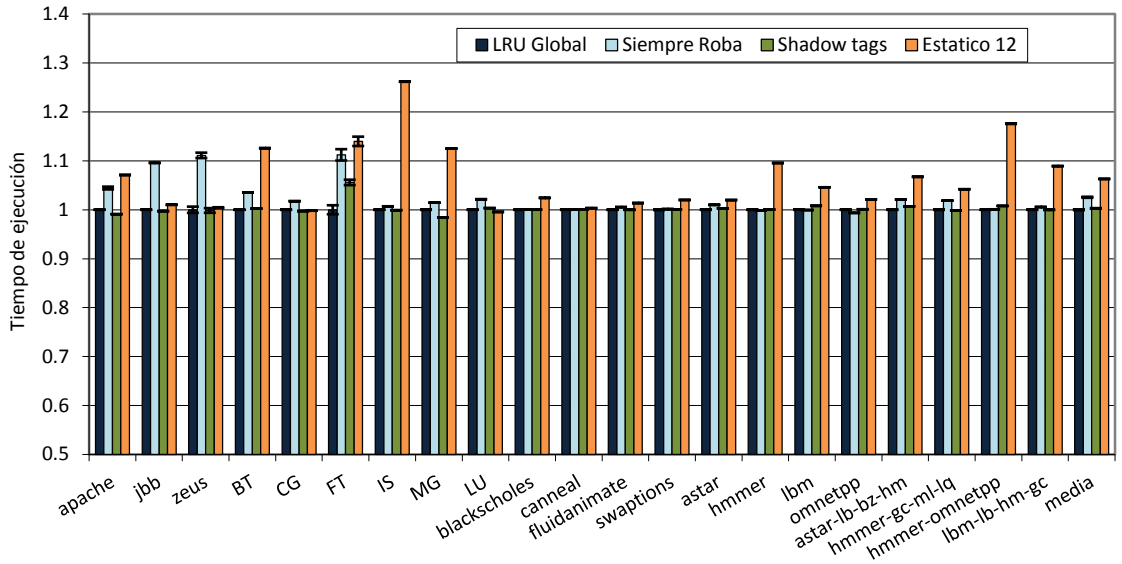

Figura 4-8. Resultados de rendimiento de los distintos algoritmos de reemplazo para distintas aplicaciones, normalizados respecto a la SP-NUCA usando LRU.

4.6 SP-NUCA Basada en Directorio

Nos vamos a centrar en la implementación de la SP-NUCA con un protocolo de coherencia MESI basado en directorio. El directorio propiamente dicho queda distribuido entre los distintos bancos de la SP-NUCA, y la coherencia es mantenida por la "parte compartida" del último nivel de cache. Para poder hacer esto, el último nivel de cache, la SP-NUCA, debe ser inclusivo con los niveles inferiores, aun cuando las partes compartidas y privadas de la SP-NUCA son estrictamente exclusivas entre sí (esto es, un dato del último nivel de cache no puede estar a la vez en la parte compartida y en la parte privada de un procesador). Gracias a la inclusividad, los bloques privados tienen información de todos los bloques que contiene la L1 del procesador al que corresponden, de forma que no es necesario transmitir los fallos de la parte compartida hacia los niveles inferiores. Por otra parte, los bloques compartidos mantienen el estado de aquellos datos que tienen o han tenido más de un propietario, y aseguran la coherencia de los mismos al ser un punto de paso obligado de todas las peticiones entre procesadores. De esta forma, y tal y como se explico anteriormente, cuando un banco privado no dispone del dato requerido por el procesador, la petición se reenvía al banco compartido correspondiente. Es el banco compartido, en el caso de que no pueda responder a la petición, el que reenvía la misma a los distintos bancos privados correspondientes a los demás procesadores. Para mantener la integridad del directorio, la petición queda pendiente en un pequeño buffer

que almacena la dirección de la petición en vuelo, junto con un contador del número de contestaciones recibidas. Los bancos privados deben contestar a la petición tanto si tienen el bloque como si no, para simplificar el protocolo de coherencia y evitar peticiones innecesarias off-chip. Si el banco compartido recibe un número de contestaciones negativas igual al número de procesadores de los que espera respuesta, entonces reenvía la petición a memoria, sabiendo ya que el dato no se encuentra en el chip. Si uno de los bancos privados estuviese en posesión del dato solicitado, o lo estuviese alguno de los niveles de cache privados asociados, su contestación sería entonces afirmativa, enviando con ella el bloque de cache, lo que denominamos como una “migración a compartido” de un bloque. En este caso, el bloque se reenviaría al procesador que lo solicitó como dato compartido, y el banco compartido queda en estado de bloqueo en espera de recibir contestación de los demás bancos privados y del procesador demandante, para evitar carreras en las peticiones sobre el mismo bloque. Este proceso implica que las peticiones a memoria deben esperar al último procesador en enviar la respuesta, lo que ralentiza considerablemente el tiempo de acceso a memoria en el caso de un fallo de cache. La petición a memoria podría realizarse en paralelo, con la de los bancos privados, pero eso complicaría el mantenimiento de la coherencia dentro del chip e incrementaría el tráfico al exterior del chip de manera significativa.

4.7 Filtrado

Pese a que, como se ha visto anteriormente, la petición a memoria podría ser hecha en paralelo con la petición al resto de bancos privados, dado que, como se hace hincapié en esta tesis, el ancho de banda a memoria es un recurso limitado, usaremos la alternativa más conservadora. Es decir, el banco compartido debe esperar a la respuesta de todos los bancos privados antes de lanzar una petición a memoria principal, lo que además facilita el mantenimiento de la coherencia dentro del chip y evita duplicidad de datos sin complicar el controlador de memoria. Sin embargo, es conveniente encontrar una solución al problema de la latencia de los accesos a memoria, que, además, aumentan el tiempo de bloqueo del banco compartido y el tiempo en el que permanecen los datos almacenados en el buffer de peticiones pendientes, lo que supone aumentar su tamaño, al tener que esperar un mayor número de respuestas simultáneamente.

La solución adoptada es la inclusión de un directorio auxiliar, denominado filtro, dentro de la porción compartida que tenga información sobre el contenido de los bloques

privados repartidos por toda la cache. De esta forma se puede eliminar la porción de *broadcast* de la solución directorio, a la vez que se reduce el tiempo de acceso a la memoria principal sin generar peticiones extra. El problema es que la cantidad de información que debe almacenar un filtro de este tipo no es práctica, especialmente si queremos tener una copia válida en cada banco para evitar indirecciones, de forma que su implementación se hace impracticable. Así, es necesario convertir el filtro en un directorio incompleto, pero que sea capaz de reducir de forma considerable el número de peticiones a bancos privados que no tienen el bloque, minimizando así el tiempo de acceso a memoria, a la vez que rebajamos la presión del tráfico en la red de interconexión y los accesos fuera del chip, siendo en ambos casos más eficiente que la solución *broadcast* y los protocolos basados en token.

El directorio incompleto actúa pues como elemento de filtrado que reduce el número de posibles bancos candidatos a contener el bloque requerido. Este filtro tiene una restricción importante: se permiten los falsos positivos, pero de ninguna forma se puede dar un falso negativo. Es decir, el filtro puede indicarnos posibles poseedores del dato que no lo fuesen realmente, ya que eso derivaría, en el peor de los casos, en una petición en broadcast tal y como se producen en la arquitectura de base sin filtro. Sin embargo, no podemos permitirnos un falso negativo, es decir, el filtro no puede indicar que el bloque no se encuentra en una posición en la que realmente se halla, puesto que eso implicaría una petición a memoria innecesaria que ya se encuentra en la cache, y de no disponer de un directorio en el controlador de memoria, podría llevarnos a una incoherencia en los datos.

Esta restricción descarta como solución de diseño el filtro de Bloom básico o cualquier otro tipo de filtro con saturación, ya que sería complejo reiniciarlo (provocaría falsos negativos), y de no hacerlo, su degeneración terminaría por no evitar los broadcast. La implementación final de este filtro consiste en un conjunto de *tags* incompleto con los datos contenidos en los bancos privados. Gracias a la coincidencia parcial de parte de los bits de direccionamiento de banco privado con los de banco compartido (como vimos en la Figura 4-4), podemos asegurarnos que en un único banco compartido se mapean un número limitado de bancos privados (2^p, tantos como procesadores), y a su vez las direcciones de un banco privado se dividen entre varios bancos compartidos (2^p), como se puede ver en la Figura 4-8. En esta figura se observa cómo por el doble direccionamiento estático, la porción compartida del banco B6 (0110) puede almacenar

datos de los bancos cuyo direccionamiento privado sea un 0 (bancos pares). Por otra parte, los datos privados de B6 (dirección privada 0 del procesador CPU7), pueden almacenarse en cualquiera de los bancos compartidos cuya dirección termina en 0 (los bancos pares). Aprovechándonos de esta forma de la coincidencia en el índice de la línea de cache, no es necesario almacenar en el filtro toda la dirección de los datos privados ni el banco al que pertenecen, sino que nos basta con el *tag* (que puede ser el de la porción compartida, algo menor, pues ya conocemos el resto de bits). Sin embargo, si dejamos espacio para almacenar los *tags* de todas las posibles vías de los bancos privados que pueden mapear en un banco compartido, no hablamos de un filtro, sino que hemos reducido el tamaño de nuestro directorio distribuido gracias a una serie de características intrínsecas al direccionamiento de la SP-NUCA. Así, el tamaño que tendría nuestro filtro en proporción a la cache sería:

$$TamañoFilt = NumBancos \cdot (NumSets \cdot NumVias \cdot NumProcesadores \cdot BitsTag)$$
$$TamañoCache = NumBancos \cdot NumSets \cdot NumVias \cdot TamañoBloque$$
$$TamañoFilt = TamañoCache \cdot \left(\frac{NumProcesadores \cdot BitsTag}{TamañoBloque}\right)$$

(4-1)

Para el caso concreto de un sistema como el de la Figura 4-9, con una cache de 8MB con 16 vías y bloques de 64bytes, y una memoria principal de 4GB, hablamos de un directorio que es aproximadamente el 20% del tamaño de la cache.

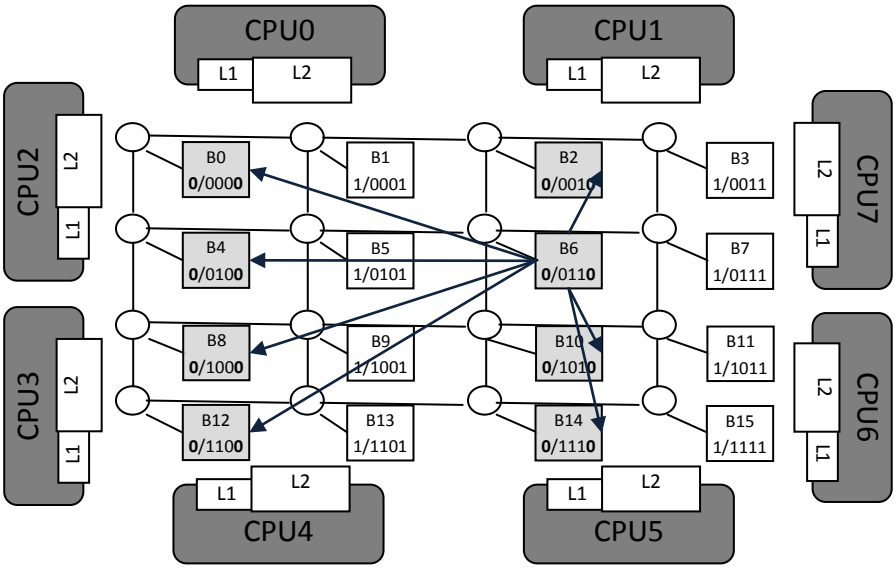

Figura 4-9. Correspondencia de un banco privado y 8 bancos compartidos, y viceversa.

Como vemos, aun con estas ventajas, almacenar todas las vías de los N bancos privados sobre los que se mapea el banco compartido supone un filtro de tamaño considerable. Es por esto que optamos por reducir el número de vías que se pueden almacenar dentro del filtro y añadir un contador de saturación para las vías restantes. Esto es así porque el filtro no es un directorio completo y no tiene reemplazos, y la única forma de que un dato desaparezca del filtro es que sea expulsado de la cache. Cuando un set del filtro tiene todas las vías asociadas a un procesador ocupadas, se incrementa el contador de saturación en uno. Si llega una petición no resuelta al banco compartido, el filtro debe ponerse en el peor caso, y suponer que cualquier procesador puede tener ese dato, de forma que si un set tiene su contador de saturación mayor que cero se realiza un *broadcast* de la petición a todos los procesadores. Sin embargo, como hemos dicho anteriormente, es necesario un mecanismo para decrementar el filtro sin reiniciarlo. Si nos llega un reemplazo de un banco privado cuya dirección no coincide con ninguno de los *tags* parciales almacenados en el filtro, podemos deducir que el bloque se trate de uno de los no capturados en el filtro y representados en el contador de saturación. De forma que reducimos el contador de saturación de ese set en uno, permitiendo de esta forma la "desaturación" del filtro sin necesidad de reiniciarlo. Esto hace al filtro robusto frente a falsos negativos, y que no degenere en una acumulación de falsos positivos, reduciendo su tamaño, a costa de disminuir su precisión.

A continuación se muestran los resultados de rendimiento obtenidos para un filtro con distinto número de vías privadas almacenadas por procesador, comparado con la SP-NUCA sin filtro, y la SP-NUCA en el caso de tener un directorio/filtro ideal. Los datos han sido sacados con un sistema semejante al de la Figura 4-9, con procesadores fuera de orden, 8MB de cache en 16 vías y 64bytes por bloque. El filtro ideal es capaz de almacenar las 16 vías de los bancos privados que se mapean sobre un mismo banco compartido (lo que hace un total de 128 vías en nuestro caso). En la Figura 4-10 se observa como con solo 4 vías por procesador, la tasa de fallos del filtro es lo suficientemente baja como para mantener un rendimiento cercano al del filtro ideal. Esto es así porque en este sistema, cada banco privado tiene 8 posibles bancos compartidos en los que mapearse, que son aquellos con los que coincide en sus bits menos significativos (ver Figura 4-9). Teniendo en cuenta que cada banco privado puede tener un máximo de 16 vías por cada set, en una distribución perfecta de las direcciones de memoria, estas 16 vías se reparten entre los 8 bancos compartidos correspondientes, bastando solo 2 vías en

el filtro por procesador. Dado que en la realidad esto no se cumple, nos hace falta el doble de vías para mantener un nivel aceptable en la fiabilidad del filtro sin que haya exceso de saturaciones.

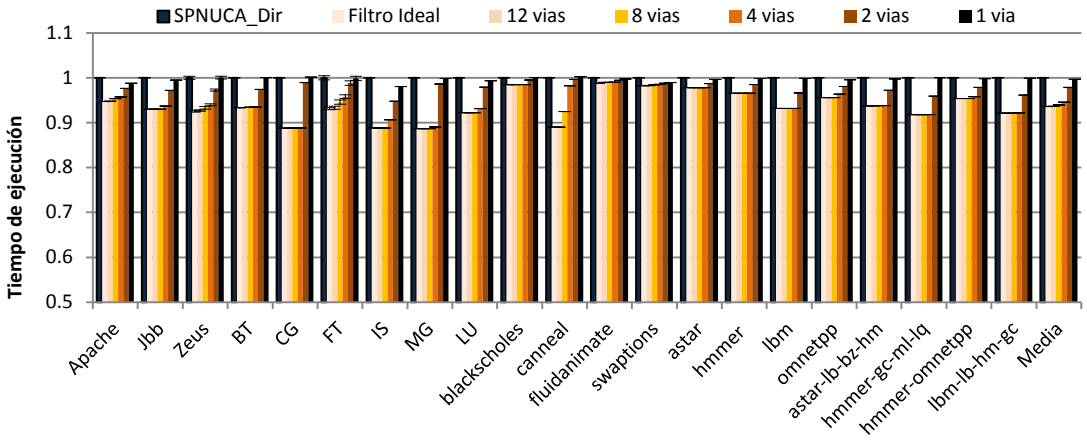

Figura 4-10. Rendimiento de la SP-NUCA con un filtro con distinto número de vías en distintas aplicaciones, normalizado a la SP-NUCA sin filtro.

Con esta solución, conseguimos reducir el tamaño del filtro en un factor $\frac{2}{P}$, donde P es el número de procesadores, y dotando al filtro el doble de las vías teóricamente necesarias para aumentar su fiabilidad. Para nuestro ejemplo concreto, y añadiendo el espacio que ocupan los contadores de cada set, el filtro es aproximadamente un 5% del tamaño de la cache. Aún así, buscamos reducir su tamaño aún más, rebajando la cantidad de bits que almacenamos en un *tag*. Ya sabemos que podemos eliminar del *tag* privado los bits que ya están tenidos en cuenta en el mapeado compartido, pero vamos a intentar reducir el tamaño del *tag* aun más, lo que supondrá que al hacer comparaciones incompletas, se produzca un mayor número de falsos positivos. Hay que tener en cuenta que estos falsos positivos no provocan un *broadcast*, sino que, en el peor caso, crean peticiones *multicast* a aquellos procesadores que pueden contener el dato.

Hacemos una evaluación para nuestro caso concreto y ya fijado el número de vías por procesador a 4. Tal y como se observa en la Figura 4-11, para un sistema con una memoria 2^9 veces mayor que la cache, el tamaño del *tag* que mantiene un funcionamiento adecuado del filtro es 10 bits. Hay que tener en cuenta que en este ejemplo, el tamaño ideal de *tag* es el equivalente a 13 bits, de forma que se observa como a diferencia del número de vías, el número de bits afecta muy negativamente al comportamiento del filtro. De hecho, los resultados con 4 vías y tag completo muestran

un rendimiento muy semejante al caso del filtro ideal, pero a medida que reducimos el número de bits en el tag, la precisión del filtro disminuye rápidamente.

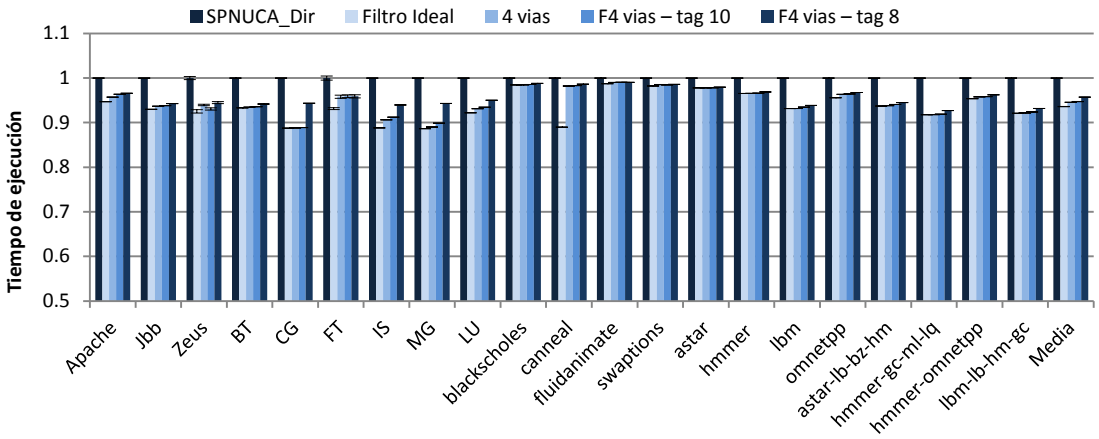

Figura 4-11. Resultados de rendimiento para distintos tamaños del tag almacenado en el filtro en distintas aplicaciones, normalizado al caso de la SP-NUCA sin filtro.

Quedándonos con una configuración del filtro en la que cada set del mismo contiene 4 vías por procesador y 10 bits de tag por entrada, más 7 bits para el contador de saturación. Con estos datos, el tamaño final del mismo es de 327KB, apenas el 4% del tamaño total de la cache. Hay que tener en cuenta que el tamaño de tag óptimo podría variar con el incremento del tamaño de la memoria principal, sin embargo el algoritmo de compresión utilizado es trivial (eliminar los bits más significativos del tag), y un *hash* más eficiente podría reducir incluso más el tamaño del mismo. En cualquier caso, se trata de un parámetro poco sensible para el tamaño del filtro, siendo posible trabajar con una memoria de 4TB en algo menos del doble del tamaño mostrado.

4.8 Cache de Víctimas

Con las soluciones presentadas hasta ahora, solventamos los problemas de latencia de acceso al último nivel de cache y reducción de la interferencia entre los distintos procesos. Sin embargo, la SP-NUCA no consigue un uso eficiente de los recursos dentro del chip cuando no se ejecutan aplicaciones en todos los procesadores, la utilización de la cache no es homogénea o no existen aplicaciones con datos compartidos. En ese caso, la capacidad de los bancos privados del procesador no utilizado no puede ser accedida, suponiendo una infrautilización de la cache *on-chip*, tal y como ocurre en el caso de caches privadas. Es por esto que se hace necesario implementar el uso de víctimas, de forma que la porción compartida de la SP-NUCA actúe como cache de víctimas de las porciones privadas, tal y como hace el Power 7 de IBM [59]. Con esto permitimos un uso

de los recursos en el chip semejante a una cache compartida pura, mientras mantenemos las ventajas de latencia, minimizando la interferencia entre *threads* de la SP-NUCA.

Al permitir a los procesadores hacer uso de la porción compartida como cache de víctimas, hay que tener en cuenta que, un comportamiento destructivo por parte de alguno de los procesadores puede provocar interferencias en el resto del sistema. En un caso extremo, las víctimas generadas por un procesador que haga *streaming* de datos, polucionarían las porciones privadas de los demás procesadores, provocando reemplazos prematuros de datos relevantes de la cache.

Para solventar el problema, añadimos un tercer tipo de bloque a la arquitectura que denominaremos “víctima”. Una víctima es un bloque expulsado de un banco privado y que es alojado en la porción compartida de la SP-NUCA. Este tipo de bloque se une por tanto a los ya conocidos (privados y compartidos), aunque estos últimos deben ser más importantes de mantener en cache. Determinaremos el número de víctimas que son admitidas en un banco usando el algoritmo de reemplazo, tal y como hemos hecho con el particionado dinámico de las porciones compartida y privada. Dado que las víctimas son bloques menos relevantes, estás serán admitidas únicamente cuando ello no implique un aumento en la tasa de fallos de las porciones privada y compartida. Esto es, un banco de L2 sólo aceptará una víctima, si eso no afecta a su propio rendimiento.

Aunque hasta ahora hemos probado que el algoritmo de reemplazo LRU era la solución más eficiente en el particionado de la SP-NUCA, esto era así porque la importancia de ambas regiones era similar. Sin embargo, es necesario aplicar algún tipo de política distinta para que los bloques menos relevantes (víctimas) no interfieran con los bloques más importantes (privados y compartidos). Es por eso que estableceremos una política de reemplazo que limite el máximo número de réplicas que garanticen un funcionamiento correcto de la cache, a la vez que permitimos su expansión en caso de infrautilización de los recursos. Es importante además que esta política se adapte dinámicamente al estado del sistema, siguiendo la filosofía de la SP-NUCA vista hasta ahora.

El algoritmo utilizado se apoya en el “*Dynamic Set Sampling*” propuesto por Qureshi et al [84], reduciendo su complejidad para adaptarse al problema concreto. Cada línea de los distintos bancos de cache acepta un número máximo de víctimas, que vendrá definido por los resultados del algoritmo. El algoritmo explora los efectos de aumentar o reducir el

número de víctimas permitidas en cada banco individualmente, de forma que se adapta dinámicamente al comportamiento de las distintas aplicaciones que se ejecutan en el sistema. Para ello, suponemos que la tasa de errores se distribuye por las distintas líneas de cache de forma uniforme, y utilizamos unas pocas líneas dentro del banco para que nos sirvan de referencia a la hora de tomar decisiones [84]. Así, dentro del banco dividimos las líneas en tres categorías:

- "Líneas convencionales", que aceptan una cantidad máxima de víctimas dada por el algoritmo.
- "Líneas de referencia", que no aceptan ninguna víctima y sirven como referencia de la tasa de fallos de la SP-NUCA convencional, de forma que si la tasa de fallos de una línea convencional es semejante a la de referencia, podemos concluir que las víctimas no afectan al rendimiento.
- "Líneas de exploración", que aceptan una víctima más que las líneas convencionales, de forma que podamos predecir la posible variación en la tasa de fallos en caso de incrementar el número de víctimas admitidas.

En cada línea convencional estará permitido un número máximo de víctimas $n_{máx}$ determinado por el algoritmo. Cada línea calcula la tasa de fallos existente en los bloques prioritarios (privados y compartidos) que posee. Si la tasa de fallos de las líneas convencionales es mayor que la de las líneas de referencia, se están aceptando demasiadas víctimas y es necesario reducir $n_{máx}$, tal y como se aprecia en la Figura 4-12. Si por el contrario, los valores son semejantes, y lo mismo sucede con la tasa de fallos de las líneas de exploración, entonces es posible aumentar el valor de $n_{máx}$ sin que ello afecte al rendimiento, como se puede observar en la Figura 4-13. Hay que tener en cuenta que este tipo de movimientos puede ser producido por cambios de fase dentro de la ejecución de una misma aplicación.

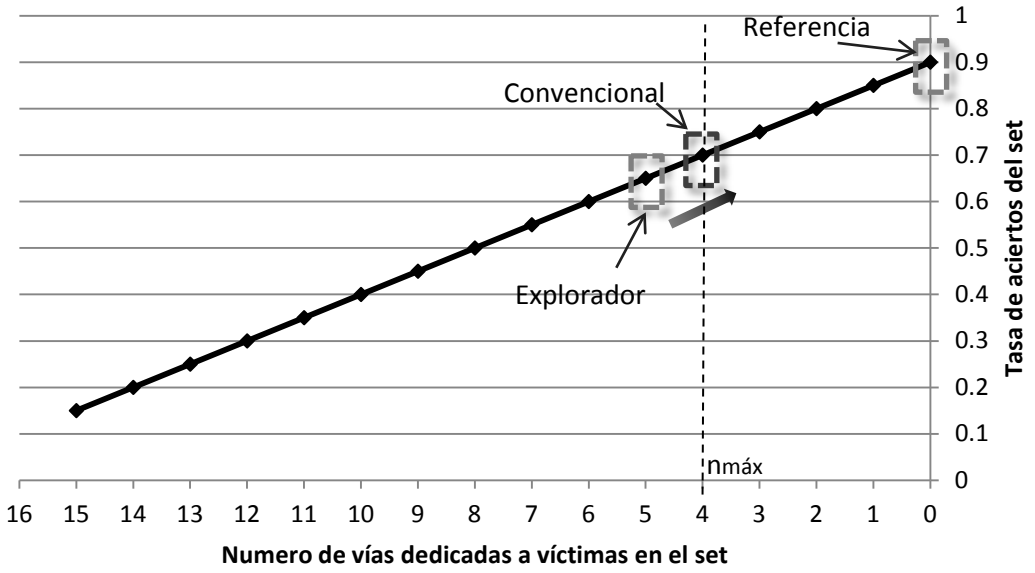

Figura 4-12. Comportamiento del número máximo de víctimas en una aplicación que se encuentra en fase de Alta Utilización. El número de víctimas se reduce para aproximar el comportamiento a los set de referencia.

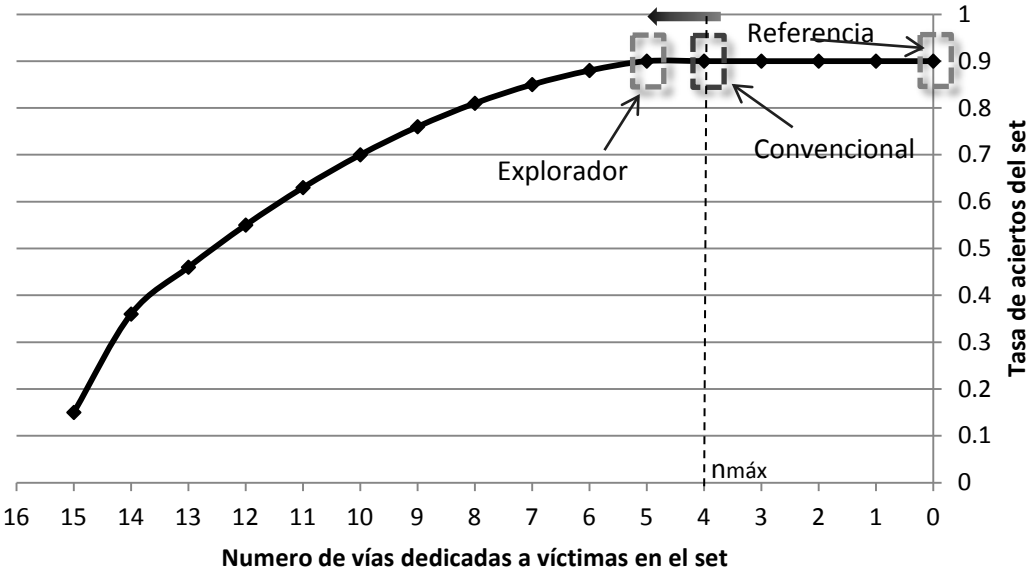

Figura 4-13. Comportamiento del número máximo de víctimas en una aplicación que se encuentra en fase de Saturación. El número de víctimas se aumenta dado que se espera un comportamiento semejante a referencia.

Cada línea de cache debe guardar un contador con el número de víctimas que tiene almacenadas. Cuando se trate de una línea convencional, si el número de víctimas presentes en la línea es menor que $n_{máx}$, se reemplaza el bloque LRU de toda la línea. Si fuese mayor o igual a $n_{máx}$, el bloque a reemplazar sería escogido únicamente entre aquellos marcados como víctimas. En las líneas de referencia todas las víctimas son

rechazadas, mientras que en las líneas de exploración se admite una víctima más de las establecidas para el banco.

Dado que la tasa de fallos en cache varía durante la ejecución de la aplicación, haremos uso de una media móvil para mantener la información de las tasas de fallos en cada uno de los tipos de línea (convencional, exploración y referencia). Para ello hacemos uso de la Media Móvil Exponencial (EMA), que permite actualizar el valor de la tasa de fallos en cada acceso de la siguiente forma:

$$EMA_n = EMA_{n-1}(1-\propto) + h \propto$$

(4-2)

Donde $\propto = \frac{2}{N+1}$, siendo N el número de muestras que son consideradas representativas en el EMA. Si medimos el EMA en un tanto por uno, h es el valor con el que actualizaremos el EMA en cada acceso, y en nuestro caso tomará valor 1 si hay un acierto en la cache y valor 0 si es un fallo.

Para implementar este mecanismo de decisión es necesario mantener tres registros en cada banco (uno por cada tipo de línea a considerar) de b bits cada uno. Para simplificar los cálculos a realizar en cada acceso, tomamos un $\propto$ que sea potencia de 2 en la forma $\propto = 2^{-a}$ de manera que las operaciones arriba descritas puedan ser implementadas mediante desplazamientos:

En caso de acierto: $EMA_n = EMA_{n-1} - (EMA_{n-1} \cdot 2^{-a}) + (2^b \cdot 2^{-a})$

(4-3)

En caso de fallo: $EMA_n = EMA_{n-1} - (EMA_{n-1} \cdot 2^{-a})$

(4-4)

Siendo 2^b la representación binaria de un 100% de aciertos.

Una vez calculada la tasa de fallos para cada tipo de línea, es necesario tomar una decisión sobre el valor que debe tomar $n_{máx}$ en función de los resultados obtenidos. Cada cierto número de accesos al banco de cache se realiza una comparación entre las distintas tasas de fallos para decidir si el comportamiento de las líneas convencionales, que son la mayoría del banco, no tiene degradación sobre las líneas de referencia (y por tanto las víctimas no están degradando el rendimiento), y si es posible incrementar el

número de víctimas en función de la tasa de fallos del explorador. Para ello, definimos un valor de desviación umbral que permitimos como degradación en la tasa de fallos respecto al bloque de referencia, lo que nos permite cierta flexibilidad al algoritmo. Siguiendo con la representación binaria, la desviación umbral tomará valor en la forma 2^{-d}.

Si llamamos TF_{REF} a la tasa de fallos de referencia, TF_{CON} a la tasa de fallos de las líneas convencionales, y TF_{EXP} a la tasa de fallos del explorador. Podemos representar la decisión tras un determinado número de accesos como:

$$n_{máx} = \begin{cases} n_{máx} - 1 & si \quad TF_{REF} - (TF_{REF} \cdot 2^{-d}) \geq TF_{CON} \\ n_{máx} + 1 & si \quad TF_{REF} - (TF_{REF} \cdot 2^{-d}) \leq TF_{EXP} \\ n_{máx} & en\ cualquier\ otro\ caso \end{cases}$$

(4-5)

El siguiente paso es decidir el número de líneas de cada tipo que vamos a dedicar a medir los resultados. Hay que tener en cuenta que cuantas más líneas dediquemos a exploración y referencia, menos líneas convencionales tiene el banco, y por tanto menos podemos beneficiarnos de sus ventajas. Por otra parte, cuantas más líneas dediquemos a las medidas, más precisos serán los resultados obtenidos, pero más potencia se consume en la actualización del EMA. Dado que no necesitamos una información precisa del comportamiento del procesador, y nos basta con un conocimiento aproximado de la influencia que nuestras decisiones implican, y siendo la potencia un factor limitante en el diseño de nuevas alternativas, se trata de mantener un número muy reducido de líneas medidoras. Así, tomamos una línea de referencia y una exploradora por banco para minimizar el número de líneas no convencionales, mientras dedicamos dos convencionales a medida.

A continuación evaluamos la sensibilidad del sistema a los distintos parámetros de las ecuaciones, en concreto la tasa de actualización del EMA (a) y la sensibilidad al rendimiento (d). Fijamos como parámetro $b = 8$ de forma que usamos 8 bits para almacenar la información para cada una de las distintas tasas de fallos.

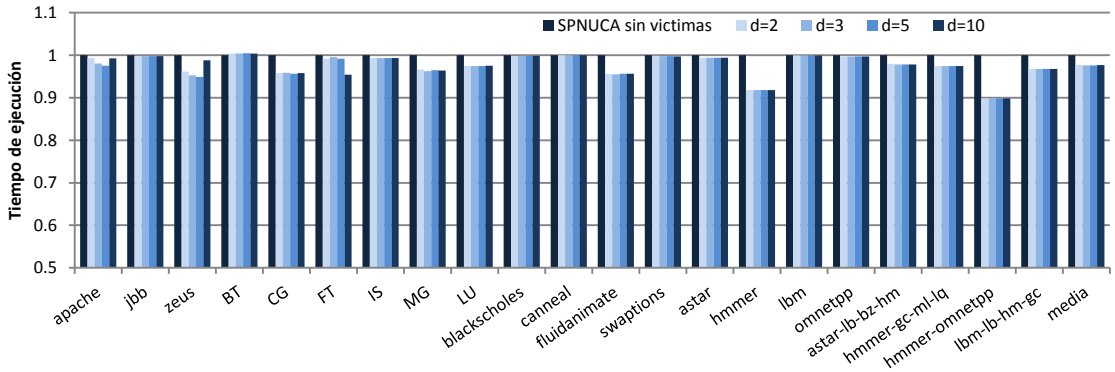

Figura 4-14. Análisis de la sensibilidad a la variación en el rendimiento. Resultados normalizados a la SP-NUCA sin víctimas.

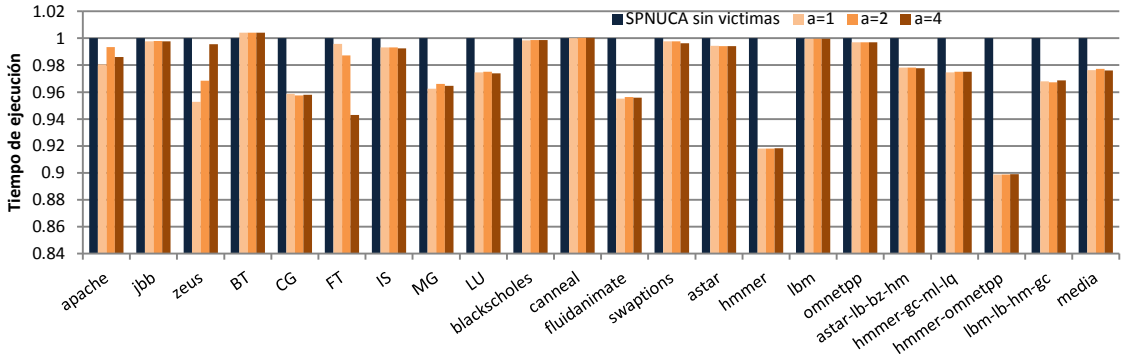

Figura 4-15. Análisis de la sensibilidad al historial. Resultados normalizados a la SP-NUCA sin víctimas.

Como se puede observar (Figura 4-10 y Figura 4-11), y salvo casos aislados, la sensibilidad a los parámetros no es muy elevada. Tomaremos los valores d=3 por obtener los mejores resultados promedio, y a=1, por ser más estable y posibilitar una reacción más rápida del algoritmo a posibles cambios del comportamiento de la aplicación.

4.9 Resultados

Finalmente comparamos la propuesta con dos arquitecturas NUCA, una estática que usaremos como referencia y una D-NUCA. La D-NUCA se caracteriza por permitir la migración de los datos de forma que se aproximen a los procesadores que las utilizan progresivamente. El principal problema que tiene una arquitectura de este tipo es localizar el bloque una vez ha comenzado a migrar. En la versión de la D-NUCA utilizada para establecer la comparación se ha utilizado un algoritmo de búsqueda perfecto, de forma que el procesador sabe de antemano el banco en el que se encuentra el bloque cuando se realiza la petición, aunque eso no evita la posibilidad de que el bloque se encuentre en proceso de migración y ésta tenga que culminar antes de poder acceder.

Debido al área adicional requerida para el almacenamiento del filtro de la SP-NUCA así como el pequeño espacio adicional requerido para el manejo de las víctimas, se presentan dos resultados de la propuesta. Por una parte una versión de la SP-NUCA básica sin mejoras, y por otro la SP-NUCA con filtrado de peticiones y manejo de víctimas, pero con menos capacidad de cache, en concreto una vía menos por banco, lo que supone algo más de un 6% de capacidad, suficiente para almacenar ambas estructuras con los parámetros especificados anteriormente.

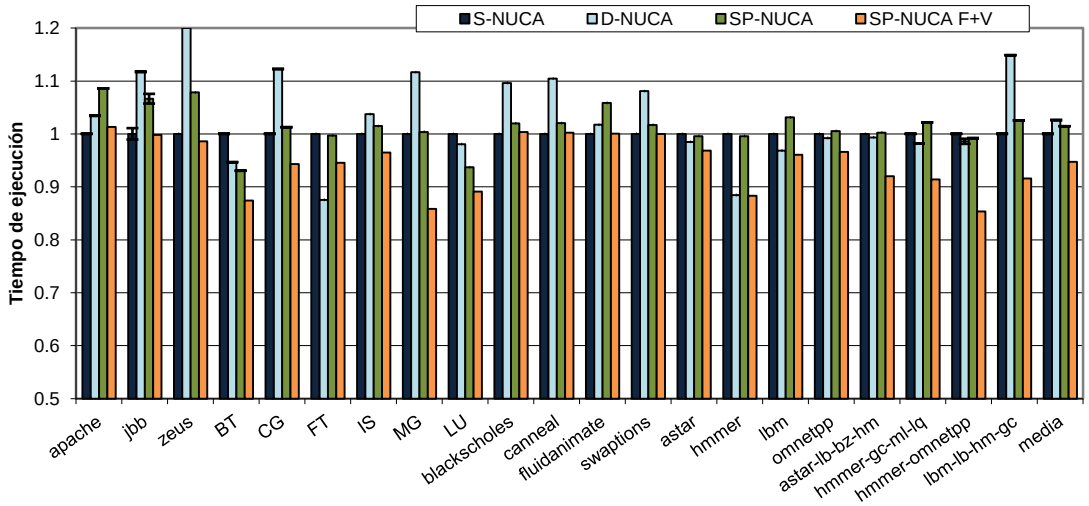

Figura 4-16. Comparativa de rendimiento entre SP-NUCA, D-NUCA y S-NUCA normalizado contra esta última.

En la Figura 4-16 se observa como la S-NUCA obtiene buenos resultados en aquellas aplicaciones con mayoría de datos compartidos, como las transaccionales y las PARSEC. La D-NUCA por su parte obtiene ventaja de aquellas aplicaciones con datos privados, aun cuando sus resultados se resienten en las aplicaciones con abundancia de datos compartidos a pesar de contar con un predictor ideal, o en las mezclas de aplicaciones, donde unas parecen interferir con las otras. Por otra parte, la SP-NUCA sencilla, aunque consigue explotar sus ventajas, sufre en aquellas aplicaciones con alta tasa de fallos a memoria debido a la altísima penalización que sufren sus accesos fuera del chip, hasta un 50% más lentos que los de la S-NUCA. En cambio, la SP-NUCA con filtro y víctimas obtiene los mejores resultados en prácticamente todas las aplicaciones, a pesar de la pequeña reducción en la capacidad de cache, y la ventaja que tiene la S-NUCA en las aplicaciones con alta tasa de datos compartidos, ya que evita una indirección (el paso por el banco privado). En total, la SP-NUCA completa consigue una mejora del 5.3% de media, llegando al 15% en casos puntuales.

4.10 Conclusiones

En este capítulo hemos presentado una arquitectura de cache de último nivel eficiente y capaz de obtener excelentes resultados en rendimiento. La arquitectura presentada, SP-NUCA, es de fácil implementación, pudiendo, mediante dos direccionamientos estáticos distintos, acercar los datos privados a los bancos más próximos al procesador, y con el mismo algoritmo de remplazo, controlar dinámicamente el uso que los distintos procesadores hacen de la cache. Durante sucesivas iteraciones de mejoras hemos depurado la arquitectura hasta llegar a una implementación enfocada hacia el rendimiento, pero sin comprometer la sencillez, reduciendo la latencia de acceso a memoria e incrementando la utilización de los recursos dentro del chip sin comprometer el aislamiento de la porción privada de los distintos procesadores. La arquitectura SP-NUCA tiene la capacidad de adaptarse dinámicamente al comportamiento de la aplicación sin depender de parámetros fijos, lo que la hace fácilmente escalable, y extrapolable a cualquier entorno, aunque su versatilidad la hace idónea para sistemas de propósito general.

Dentro de las mejoras a la SP-NUCA se presenta un directorio parcial o filtro, capaz de almacenar una porción importante de datos en un espacio reducido. Esta estructura eficiente reduce el espacio necesario mediante la tolerancia de falsos positivo, pero restringiendo los falsos negativos, de manera que se mantiene la simplicidad del protocolo de coherencia y se reduce el número de accesos a memoria. Al contrario que un filtro contador de Bloom, esta estructura no puede incurrir en un overflow y no debe ser reiniciada, ocupando un espacio semejante y manteniendo una tasa de acierto aproximadamente constante. Sería posible implementar una solución del estilo si se acepta la invalidación de los bloques de cache implicados en la saturación de una línea del filtro, con el coste que ello conlleva.

5 Reparto del Ancho de Banda a Memoria

5.1 Introducción

Como se ha mencionado en el capítulo 1, el *bandwidth Wall*, o el incremento de la diferencia de rendimiento entre el procesador y la memoria principal, es un problema ampliamente conocido. Aun considerando la cantidad de memoria que podremos integrar dentro del chip y mejorando la efectividad de las caches, tanto en latencia como en uso de los datos, es previsible que también crezca el numero de procesadores integrados dentro del Chip y las necesidades de las aplicaciones ejecutadas en los CMP, mientras que el ancho de banda a memoria principal parece limitado debido a las restricciones físicas de las conexiones off-chip. Existiendo soluciones al problema de acceso fuera del chip (fotónica con multiplexado ultra denso de longitudes de onda…), es previsible que esto sea una solución temporal, y que la administración del ancho de banda off-chip en un CMP siga siendo un problema de futuro [66], dado que el número de transistores en el chip por pin de salida continua en aumento.

Este problema es mayor si consideramos el previsible aumento en el número de threads corriendo en un sistema, ejecutando una amplia variedad de tareas distintas. En un caso extremo, una tarea con un comportamiento agresivo en memoria, podría afectar al rendimiento de todo el CMP. Es por esto que hay que trabajar en un uso eficiente del ancho de banda a memoria, a la vez que se busca una reducción de la interferencia entre las aplicaciones.

A la hora de mejorar el uso del ancho de banda a memoria se buscan dos objetivos distintos. Por una parte, se pretende mejorar el rendimiento global, buscando que cada petición atendida en memoria tenga el mayor efecto posible en los procesos que se están ejecutando dentro del chip. Por otra parte, es conveniente reducir la interferencia entre los accesos simultáneos a memoria de dos o más threads, garantizando que todas las peticiones son atendidas en un tiempo razonable, a la vez que se reduce el impacto que,

aplicaciones agresivas, pueden tener en el resto del sistema, y evitando la posibilidad de inanición por falta de respuesta a un thread concreto.

El trabajo previo que trata este problema es abundante y será descrito en la sección 5.3. Posteriormente, presentaremos la aproximación de este trabajo, la cual se centra en la visión del problema por parte del procesador. Aunque todavía existen CMP basados en procesadores sencillos, las necesidades desde el punto de vista de procesadores de propósito general apuntan a procesadores que sean capaces de explotar el paralelismo a nivel de instrucción. Lo que, hoy en día, se traduce en el uso de procesadores con ejecución fuera de orden. Este capítulo se centra en cómo explotar la naturaleza de las operaciones a memoria en esta clase de sistemas, de forma que se haga un uso eficiente del ancho de banda a memoria.

Vamos a estudiar cómo es posible mejorar el comportamiento del sistema si las peticiones son atendidas en memoria teniendo en cuenta la criticidad de la instrucción asociada en el procesador, medida como la capacidad que tiene esa instrucción de bloquear el ritmo de retirada de instrucciones en el *Reorder Buffer* (ROB). En particular, se usará la distancia de esa instrucción a la cabeza del ROB como principal medida de criticidad. El algoritmo de selección propuesto actúa en el controlador de memoria y usa esta distancia para reordenar las peticiones que serán atendidas en memoria de acuerdo a una serie de criterios de prioridad y envejecimiento. La propuesta presentada, a la que llamaremos DROB, mejora los resultados obtenidos por algoritmos de particionado del ancho de banda a memoria de similar complejidad, tanto en términos de rendimiento como de equidad o *fairness*. Además, DROB permite al usuario buscar un mejor resultado en rendimiento o *fairness*, a costa del otro, modificando un simple parámetro. Siendo posible ajustar este parámetro de forma adaptativa en el controlador de memoria para alcanzar un resultado equilibrado entre rendimiento y *fairness* mediante un mecanismo de bajo coste.

5.2 Motivación

Se toma un sistema de referencia como el de la Figura 5-1, con N procesadores con ejecución fuera de orden, cada uno con uno o más niveles de cache privada y un nivel de cache compartida, y M controladores de memoria administrando B bancos de memoria. Siempre que una petición a la jerarquía de memoria no sea atendida por los recursos existentes dentro del chip, un controlador de memoria se debe hacer cargo de la petición,

aplicando el correspondiente criterio de ordenación según prioridades y enviando la petición al banco de DRAM correspondiente. Hay que tener en cuenta que, en este sistema de referencia, aquellas peticiones de un procesador que coincidan sobre el mismo bloque de memoria serán atendidas en el *Miss-Status Handling Registers* (MSHR) de la cache local correspondiente. Las peticiones a memoria de diferentes procesadores al mismo bloque de memoria serán atendidas en el MSHR del último nivel de cache compartido. Hay que tener en cuenta también que las operaciones de escritura sobre la memoria off-chip únicamente estarán asociadas a *write-backs* en el último nivel de cache, cuando un fallo en el chip produzca un reemplazo que expulse un bloque sucio, y por tanto están fuera del camino crítico a memoria.

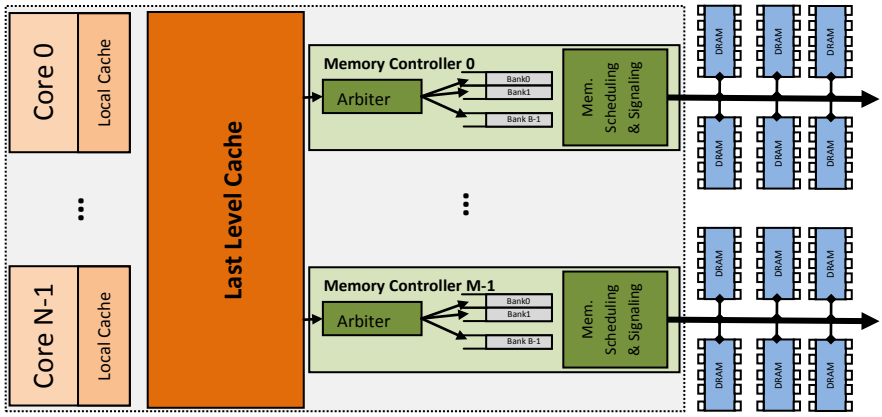

Figura 5-1. Esquema del Sistema de referencia, con detalle sobre el controlador de memoria

El controlador de memoria es el responsable de aplicar la prioridad adecuada a las peticiones que van llegando y encolarlas, según la dirección de memoria, en la cola del banco correspondiente. Desde la cola del banco, la lógica de *scheduling* elige una petición y la ejecuta en el banco correspondiente. Como punto de partida, vamos a tomar una lógica de *scheduling First Ready-First Come First Serve* (FR-FCFS) de Rixner et. al. [43]. Aunque hay algoritmos más avanzados en función de la implementación física de la DRAM, los interfaces de señalización, etc… En la mayoría de los casos, éste punto de partida es complementario a lo presentado aquí.

Muchas de las propuestas de gestión del ancho de banda a memoria, trabajan sobre el árbitro del controlador de memoria, tal y como se muestra en la Figura 5-1. El objetivo es crear un algoritmo que reordene las peticiones a memoria entrantes, basándose en un criterio de ordenación determinado. Este algoritmo de ordenación modifica el criterio FR-

FCFS en la cola de cada banco según la política utilizada. De acuerdo a la prioridad determinada por el árbitro, las operaciones de DRAM asociadas son insertadas en las correspondientes colas y abandonarán el chip en el orden adecuado. Dado que la limitación del ancho de banda fuera del chip es un problema que atrae una atención considerable, hay muchos trabajos recientes centrados en el ordenamiento de las peticiones a memoria [85]–[90]. En la mayoría de estos trabajos, la clave está en inferir el comportamiento del procesador con los datos disponibles en el controlador de memoria y actuar adecuadamente, coordinando esta decisión entre los distintos bancos [88], [91]. En algunos casos, el mecanismo involucrado en la toma de decisiones requiere de complejos algoritmos y/o información proveniente de la capa de software, o limita la flexibilidad de uso del CMP [92].

Intuitivamente, la idea subyacente en muchas de las propuestas es limitar la interferencia de las aplicaciones altamente demandantes en memoria con las aplicaciones menos demandantes. El objetivo es, con limitado impacto en su rendimiento, limitar el consumo de ancho de banda de las primeras, garantizando el servicio y aumentando el rendimiento de las segundas. En cierta medida, la aproximación aquí presentada busca un objetivo semejante, aunque toma en consideración la criticidad de las instrucciones que acceden a memoria. Desde el punto de vista de un procesador fuera de orden, la criticidad de una instrucción representa la pérdida de rendimiento asociada al retrasar su ejecución.

5.3 Trabajo Relacionado

Entre los trabajos existentes que abordan el problema de la compartición del ancho de banda a memoria, existen algunos que merece la pena resaltar especialmente por su relevancia o su impacto en el desarrollo de esta línea de la tesis.

El primer trabajo es el llevado a cabo por Liu et. al. [93], en el que se explica la relevancia del particionado del ancho de banda a memoria en el rendimiento global en sistemas CMP. Liu demuestra, analíticamente, el esquema de particionado óptimo, que en teoría obtiene los mejores resultados de rendimiento y *fairness*, y cuya filosofía coincide con el resto de trabajos sobre la materia: controlar y mitigar el impacto negativo que producen las aplicaciones más demandantes a memoria sobre las menos demandantes, sin provocar inanición. Este trabajo además establece una relación entre el CPI del procesador y la frecuencia de fallos en memoria principal, algo que hemos podido comprobar experimentalmente.

El algoritmo de arbitraje de memoria de Liu consiste en repartir "crédito" a memoria entre los distintos procesadores en función de sus necesidades, pero mitigando las diferencias mediante una fórmula obtenida analíticamente (5-1), de manera que la porción de crédito por procesador no se aleja en exceso de la media.

$$\beta_i = \frac{M_i \cdot A_i}{\sum_{j=1}^{P} M_j \cdot A_j}$$

$$\beta'_i = \frac{(M_i \cdot A_i)^{2/3}}{\sum_{j=1}^{P} (M_j \cdot A_j)^{2/3}}$$

(5-1)

Donde β_i es la porción de ancho de banda que usaría el procesador i en caso de no estar limitado, β'_i es la porción de ancho de banda garantizada por el algoritmo de Liu y M_i es la tasa de fallos de cache del procesador i y A_i su frecuencia de acceso a la cache, siendo por tanto $M_i \cdot A_i$ la frecuencia de accesos a memoria.

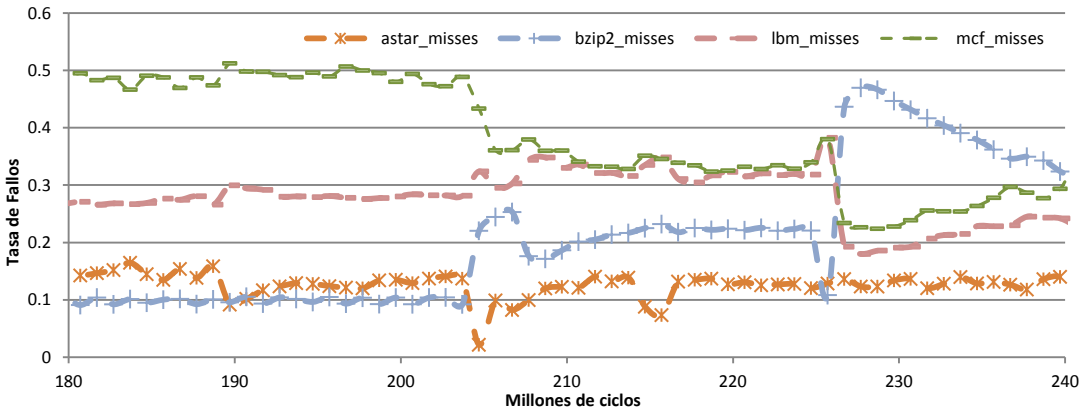

Figura 5-2. Frecuencia de accesos a memoria normalizada de distintas aplicaciones corriendo simultáneamente.

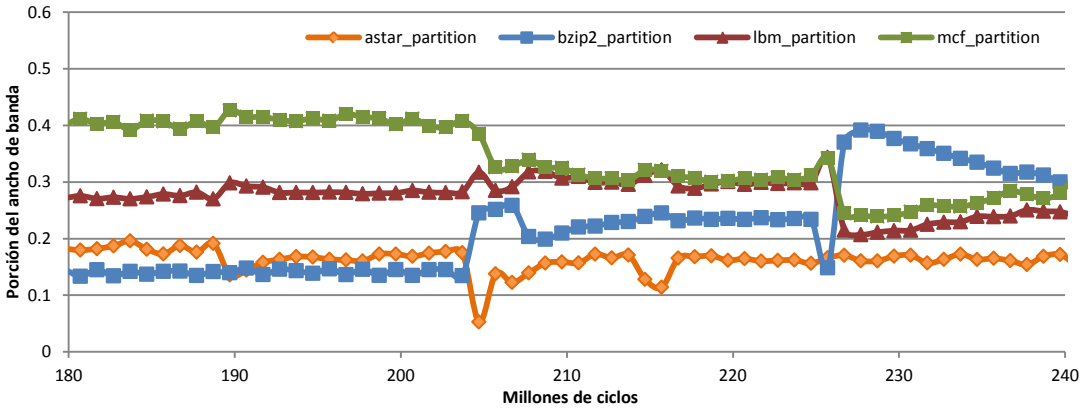

Figura 5-3. Porción de ancho de banda garantizado mediante el algoritmo de Liu a distintas aplicaciones que se ejecutan simultáneamente.

En la Figura 5-3 se aprecia el efecto que tiene el algoritmo de Liu sobre el ancho de banda garantizado a cada procesador, respecto a su frecuencia de acceso (Figura 5-2). Como se puede ver, el ancho de banda garantizado tiende a “igualar” el ancho de banda dedicado a cada procesador, de forma que los procesadores más exigentes en memoria no interfieran en los procesadores menos demandantes, que tienen garantizado una porción de ancho de banda mayor en comparación.

Aunque, al evaluar el algoritmo, los resultados fueron discretos, como se puede apreciar en la Figura 5-4, queda patente que administrar correctamente el ancho de banda a memoria entre los distintos threads que se lo disputan tiene un impacto apreciable en el rendimiento y especialmente en el fairness.

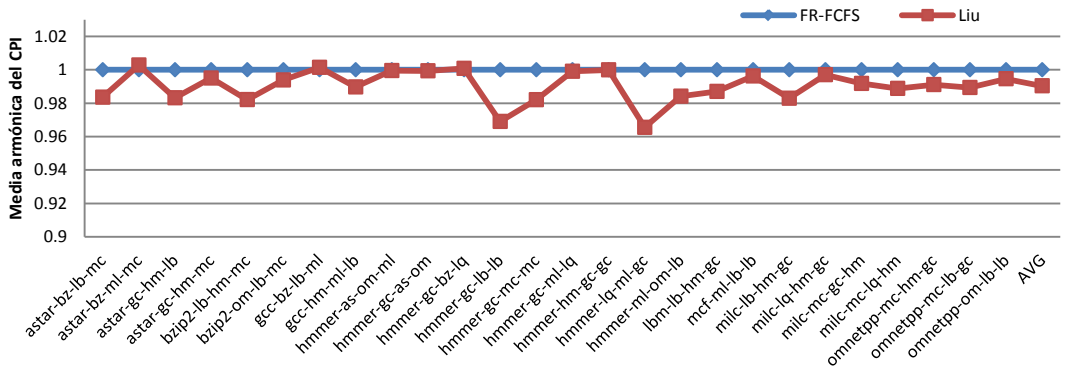

Figura 5-4. Media armónica del CPI de la solución de Liu normalizada al caso base FR-FCFS en diferentes cargas de trabajo.

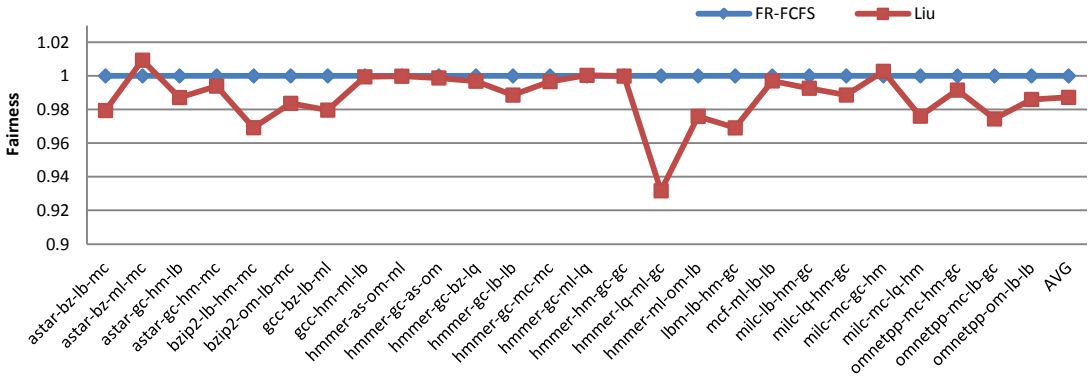

Figura 5-5. Máximo slowdown del CPI entre las distintas aplicaciones para la solución de Liu normalizado al caso base FR-FCFS.

Se puede observar como las decisiones que establece Liu a la hora de priorizar las peticiones que llegan a memoria tienen poco impacto, dado que en general se trata de un FR-FCFS en el que algunas peticiones son congeladas durante un breve intervalo. La

solución es por tanto conservadora y se aleja poco del punto de partida (Figura 5-4), pero aun se puede observar cómo, en los casos de mayores diferencias entre las aplicaciones, se consiguen ganancias del 4% en rendimiento y del 7% en *fairness*.

Existen otros trabajos que se centran en el reordenamiento de las peticiones en el controlador de memoria, al igual que la idea propuesta en esta tesis. Así, Mutlu et.al. [89] proponen un sistema basado en ráfagas de peticiones ordenadas bajo la premisa "el que menos tarda, primero", una solución usada entre otras cosas en el *scheduler* de tareas del sistema operativo. Esta solución, denominada PAR-BS (*Parallel-Aware Batch Scheduling*), reordena las peticiones que llegan a memoria de forma que el procesador que tiene menos peticiones pendientes en el controlador tiene mayor prioridad. Al igual que la solución propuesta por Liu y otras semejantes, se trata de inferir el estado del procesador a través de la información que se tiene en el controlador de memoria, y de nuevo la solución consiste en acelerar a los procesadores con un IPC más alto y que consumen poco ancho de banda en detrimento de aquellos más agresivos en memoria, salvo que en este caso se produce un reordenamiento real de las peticiones. Para evitar la inanición de los procesadores más agresivos, el algoritmo añade el concepto de "ráfagas", de forma que las peticiones se agrupan en conjuntos con un número fijo de peticiones por procesador. De esta manera, todos los procesadores aseguran un mínimo de peticiones atendidas en un intervalo de tiempo (una idea semejante al concepto de los cubos propuesto por Liu). Los resultados obtenidos por este algoritmo de arbitraje son bastante positivos en rendimiento y especialmente en *fairness*, sin embargo toma sus decisiones basándose únicamente en la información obtenida en memoria y sin tener en cuenta la relevancia de cada petición individualmente, dependiendo además de un parámetro definido estáticamente, lo que en ocasiones le lleva a tomar decisiones no óptimas.

Mutlu propone otra aproximación a la filosofía "el que menos tarda, primero" en su artículo sobre ATLAS [90] (*Adaptive per-Thread Least Attained Service*). Basándose en un principio semejante al PAR-BS, predice el comportamiento del procesador usando información de intervalos más largos (en lugar de información instantánea como PAR-BS), y soluciona el problema de inanición de los procesos más lentos mediante un contador de envejecimiento. Además, al usar información obtenida en periodos largos, reduce la comunicación necesaria entre controladores y toma decisiones más estables en el tiempo. Este sistema demuestra ser superior en rendimiento respecto a su solución anterior (PAR-BS), a costa de un empeoramiento notable en el *fairness*. Usaremos los dos

algoritmos propuestos por Mutlu como alternativas de diseño a nuestra propuesta como puntos de referencia de rendimiento y *fairness*, siendo además propuestas de complejidad similar a la que se presenta en esta tesis.

Kim et. al. [88] propone un algoritmo de arbitraje en el controlador de memoria infiriendo el comportamiento del procesador según los datos recogidos en el controlador de memoria, tal y como hacen ATLAS y PAR-BS, siendo sus resultados muy similares a éste último y su complejidad algo mayor.

Ebrahimi et. al. [85] mejora el algoritmo propuesto por Kim para su uso en aplicaciones *multithread*, pero requiere de una colaboración por parte de la capa de software para su correcto funcionamiento, y aumenta considerablemente la complejidad del sistema.

Finalmente, existen otros algoritmos para el arbitraje en el controlador de memoria que, en lugar de centrarse en la reordenación de peticiones, trabajan sobre los comandos enviados a los bancos de DRAM (activación de fila, precarga, lectura del buffer, escritura…). Así, por mencionar algunos, Ipek et. al. [86] propone el uso del la técnica de aprendizaje por refuerzo [94] para desarrollar un controlador de memoria que aprende de diversas variables del entorno para escoger la acción (comandos de DRAM) adecuada en cada momento con el fin de mejorar el rendimiento global del sistema a largo plazo. Siguiendo esta línea de trabajo, Mukundan et. al. [91], amplía el esquema propuesto por Ipek permitiendo recompensas a múltiples objetivos (no sólo rendimiento, también energía, *fairness*...) y calibrando su importancia mediante el uso de algoritmos genéticos. Estas soluciones obtienen excelentes resultados, sin embargo, en la mayoría de los casos, son ortogonales a nuestra propuesta.

5.4 Métricas de Priorización de Peticiones desde el Punto de Vista del Procesador

Como alternativa, se presenta un algoritmo de ordenación de las peticiones a memoria desde un enfoque distinto a lo visto anteriormente. Se traslada el foco al extremo opuesto del problema, el procesador, intentando que la memoria sea consciente, en la medida de lo posible, de la situación en el chip, de forma semejante a lo propuesto por Ghose et. al. [95]. Para ello, se hace uso de las características propias de los procesadores actuales. Trabajamos con procesadores con ejecución fuera de orden, intentando extraer

información que nos pueda ser útil a la hora de establecer un criterio en el controlador de memoria. Este proceso no es trivial, y pasaremos por una fase de análisis previa a la discusión de un algoritmo de ordenación y su posible implementación.

Existen abundantes trabajos centrados en hacer frente a la escasez de ancho de banda fuera del chip desde la perspectiva de la DRAM. Tiene sentido aprovecharse de las particularidades de este tipo de sistemas, combinándolas con técnicas centradas en la otra parte de problema, el procesador. Aunque, como hemos visto, hay numerosos trabajos que tratan de deducir el comportamiento del procesador mediante la información disponible en el controlador de memoria, aquí se opta por mirar directamente a lo que está haciendo el procesador. No todas las operaciones de memoria son igualmente importantes en un procesador fuera de orden, algunas peticiones tardarán más que otras en bloquear la retirada de instrucciones y otras pueden proceder de instrucciones especulativas que quizá deban ser descartadas. Por tanto, parece pertinente usar esa relevancia a la hora de ordenar las peticiones en el controlador de memoria. En un momento dado, cada controlador de memoria debe hacerse cargo de peticiones a memoria procedentes de los distintos procesadores del sistema. Estas operaciones pueden ordenarse antes de ser enviadas al banco correspondiente para ser atendidas. Este ordenamiento no debe subestimarse, pues puede tener un impacto importante en el rendimiento, *fairness* y consumo energético del sistema. Vamos a explorar el uso de distintos criterios desde el punto de vista del procesador previo a la propuesta de un algoritmo óptimo de ordenación en memoria. Nos centraremos en aquellas peticiones de memoria que están en el camino crítico del procesador, que se corresponden con las lecturas (*read*) en memoria principal. Hay que tener en cuenta que las escrituras (*write*) en memoria principal, como se ha mencionado anteriormente, se corresponden con reemplazos de un bloque sucio en el último nivel de cache, por lo que no forman parte del camino crítico de una petición del procesador.

Aunque existen trabajos centrados en optimizar los algoritmos de reemplazo y su manejo en el controlador de memoria [96], minimizando así la latencia read-after-write, no se va a tomar en consideración este problema. Este tipo de soluciones son complementarias al análisis que aquí se presenta. Así pues, al igual que en sistemas como el Power 5 [97], las escrituras en memoria siempre tendrán la prioridad mínima.

A continuación, realizamos un análisis de posibles parámetros dentro de un procesador con ejecución fuera de orden que podemos tener en cuenta a la hora de determinar la criticidad de una instrucción en el controlador de memoria. Nos quedaremos con aquellas soluciones que permitan discernir suficientemente las peticiones y cuyo coste de implementación respecto al beneficio obtenido no sea elevado.

5.4.1 Distancia a la Cabeza del ROB

Reducir el ritmo de retirada de instrucciones en el procesador tiene un impacto importante en el rendimiento, ya que está directamente relacionado con una disminución del IPC. El caso más habitual de disminución del ritmo de instrucciones retiradas es debido a instrucciones de ejecución lenta que bloquean la cabeza del ROB, o la presencia abundante de *rollbacks* debido a los fallos de especulación. En los sistemas actuales, una de las causas más probables de que una instrucción tarde mucho en ejecutarse es que se deba a una petición que no es atendida en el chip y por tanto tenga que acceder a memoria principal. Por tanto, cuando una petición llega al controlador de memoria, su criticidad es mayor cuanto menor sea la distancia a la cabeza del ROB de la instrucción que la ha provocado, ya que tiene más probabilidades de bloquear la retirada de instrucciones y degradar el IPC del procesador.

Para evaluar la relevancia de este parámetro se ha llevado a cabo un experimento sencillo con 36 cargas de trabajo formadas por la mezcla de aplicaciones de las SPEC2006 (usando el entorno de trabajo descrito en el apartado 2.8) simulando 4 procesadores fuera de orden con un ROB de 128 entradas. En este experimento, en el momento en que una petición llega al controlador de memoria, se determina el número de instrucciones que hay entre la instrucción que ha provocado la petición y la cabeza del ROB.

Como se puede ver en la Figura 5-6, esta distancia es fuertemente dependiente de la aplicación y la carga de trabajo en la que se encuentre. Por ejemplo, en el caso *hmmer-gcc-lbm-lbm*, las operaciones de memoria de los procesadores 0 (*hmmer*) y 1 (*gcc*) parece mucho más críticas que aquellas pertenecientes a los otros dos. Si pudiésemos utilizar esa información de forma apropiada en el controlador de memoria, podríamos acelerar unos valiosos ciclos la resolución de aquellas operaciones más críticas, con un impacto moderado en el resto. En este ejemplo, si de alguna forma pudiésemos saber la situación del ROB en cualquier momento, parece razonable favorecer las peticiones pertenecientes

a *hmmer* y *gcc* sobre las peticiones de *lbm*. Hay que tener en cuenta que los datos presentados en la Figura 5-6 son una media a lo largo de una ejecución de un gran número de ciclos, y que los valores cambian constantemente durante la ejecución de las aplicaciones.

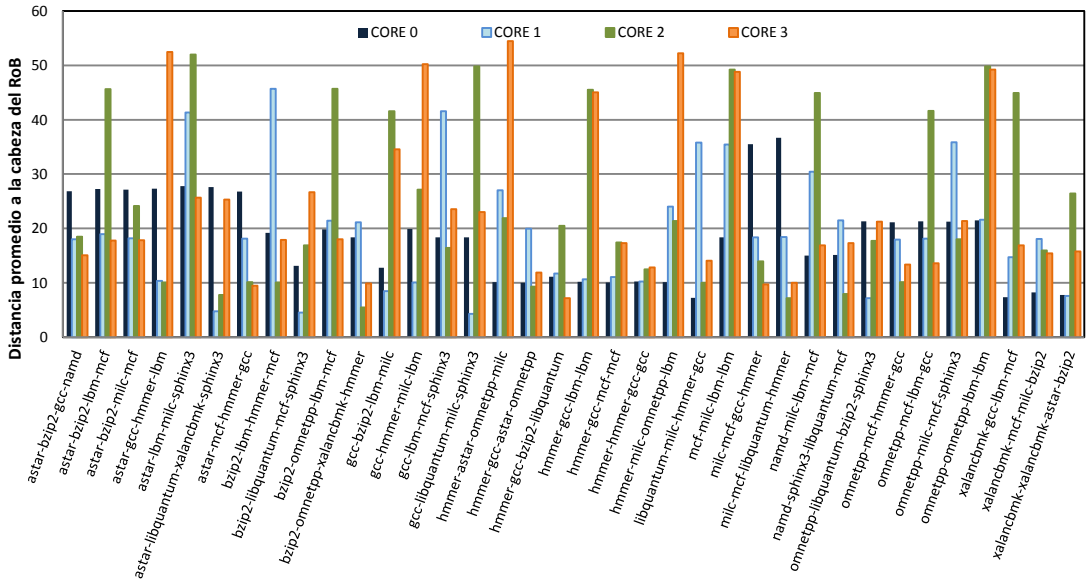

Figura 5-6. Distancia media de una instrucción a la cabeza del Reorder Buffer cuando la petición que provoca llega al controlador de memoria.

5.4.2 Optimizando el fetch de instrucciones

Aunque menos habitual, un fallo en el chip de una petición de un bloque de instrucciones (*instruction fetch*), tiene un impacto crítico en el rendimiento de un procesador fuera de orden. En esta circunstancia, dado que esta operación se ejecuta en orden, el pipeline del procesador podría llegar a pararse. Aunque en muchas aplicaciones este tipo de fallos no son frecuentes, en otras, como las aplicaciones comerciales, puede ser bastante habitual, debido a la gran cantidad de código de este tipo de aplicaciones [50]. Por tanto, en un entorno multiprogramado, el número de peticiones a memoria provocadas por un *fetch* puede ser muy distinto para cada uno de los distintos procesadores según las características de lo que estén ejecutando.

En estas circunstancias, parece adecuado que el controlador de memoria asigne la máxima prioridad a este tipo de peticiones, de forma que su resolución se acelere lo máximo posible y por tanto desbloquee el pipeline del procesador rápidamente. Esto tendrá impacto no solo con procesadores compitiendo por el ancho de banda a memoria, sino individualmente para cada procesador. Permitir que los *fetches* adelanten a otras

peticiones del mismo procesador que están esperando en la cola de un banco puede ser beneficioso, especialmente si esas peticiones están lejos de la cabeza del ROB. En las pruebas se han observado mejoras significativas en rendimiento cuando se priorizan los *fetches* sobre otras peticiones a la memoria, y en general, no existe un impacto negativo con esta medida.

5.4.3 Grado de especulación

Cuando un fallo en el chip alcanza el controlador de memoria, la instrucción asociada puede haber lanzado la petición de forma especulativa. Vamos a denotar como "grado de especulación de una instrucción" el número de operaciones entre la instrucción y la cabeza del ROB que han sido ejecutadas sin todos sus parámetros conocidos, como por ejemplo: saltos no resueltos, loads especulativos, etc. Intuitivamente, las operaciones a memoria serán menos críticas cuanto mayor sea el grado de especulación de la instrucción que las lanza, ya que aumenta las posibilidades de que la instrucción tenga que ser descartada por un roll-back. Siguiendo un procedimiento semejante al utilizado en el caso de la distancia a la cabeza del ROB y centrándonos en el control especulativo, se muestra el número medio de saltos no resueltos cuando una petición llega al controlador de memoria para distintas mezclas de aplicaciones de las SPEC2006. Como se puede ver en la Figura 5-7, para un ROB de 128 entradas, el número medio de saltos no resueltos por delante de una instrucción para las distintas aplicaciones es bastante similar, variando entre 1 y 3. Esto parece indicar que este criterio puede no ser especialmente útil a la hora de discriminar entre las distintas operaciones a memoria, especialmente si consideramos que esta es una información ideal, y que debemos trabajar con la información disponible cuando la petición sale del procesador.

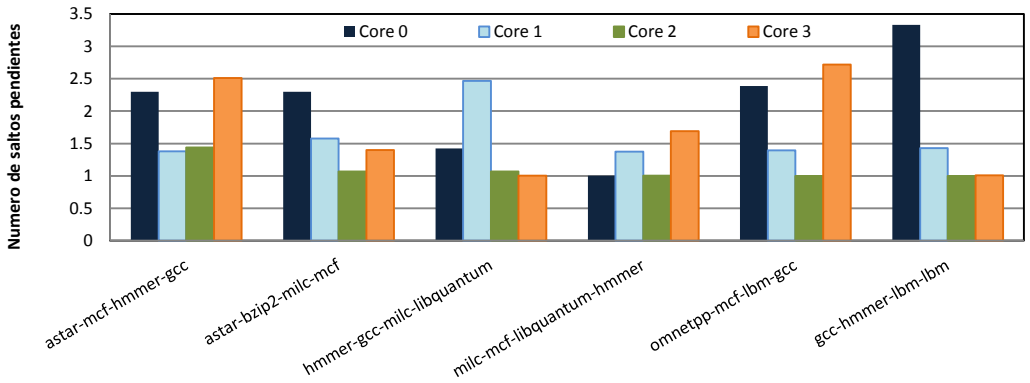

Figura 5-7. Grado de especulación de una instrucción cuando llega al controlador de memoria medido como el número de saltos pendientes existentes entre la instrucción y la cabeza del RoB.

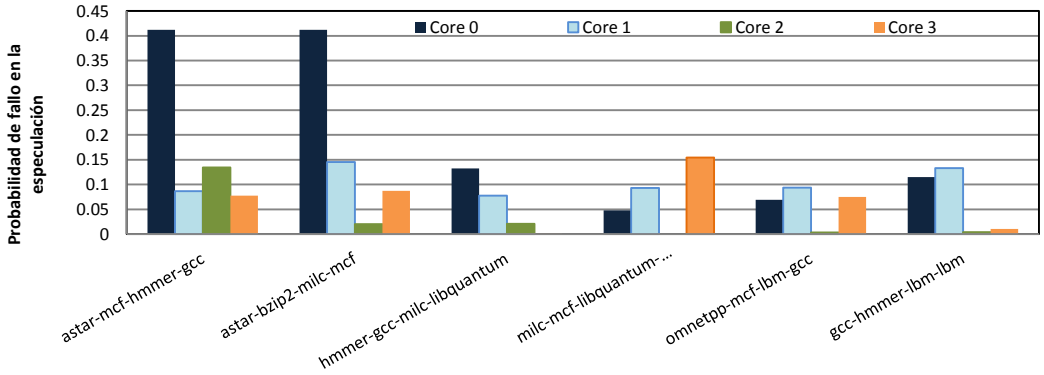

Figura 5-8. Probabilidad de que una instrucción especulativa que llega al controlador de memoria sea descartada posteriormente.

En cualquier caso, esta información está incompleta, ya que la posibilidad de fallo en una instrucción de salto especulativa depende también de la precisión del predictor de saltos. En particular, esto es especialmente importante si se une una gran tasa de fallo del predictor con un alto grado de especulación. Para dar una mayor perspectiva al problema, en la Figura 5-8 se muestra la probabilidad de que una petición que llega al controlador de memoria sea desechada posteriormente (debido a un roll-back), calculada mediante la siguiente fórmula:

$$P_{RB} = 1 - (1 - M)^N$$

(5-2)

Donde M es la tasa de fallo promedio del predictor de saltos, y N el número de saltos especulativos por delante de la petición:

Los resultados muestran que el comportamiento es bastante similar para las distintas aplicaciones, salvo por la presencia de algunos casos patológicos, siendo únicamente apreciable el caso de “astar”, donde un porcentaje muy alto de fallos a memoria son desechados debido a fallos en la especulación. En las pruebas realizadas, esta métrica no ha resultado relevante para la complejidad que lleva asociada y fue descartada de la solución final.

5.4.4 Instrucciones dependientes

Así como mirar hacia delante en el ROB proporciona información relevante a la hora de catalogar la criticidad de una determinada operación en memoria, mirar hacia atrás también puede aportar pistas sobre su relevancia atendiendo al número de instrucciones

que dependen de ella. Intuitivamente, una petición con mayor número de instrucciones dependientes, esperando que el registro origen esté disponible para ser ejecutadas en el ROB, debe ser considerada más importante que otra que tiene menos. La clave vuelve a ser si hay suficientes diferencias entre las distintas peticiones que llegan a memoria de los distintos procesadores como para establecer un criterio fiable. De nuevo se evalúa la relevancia de este parámetro llevando a cabo el mismo experimento que en los casos anteriores y buscando en el ROB el número de instrucciones dependientes (directa o indirectamente) del registro destino de la instrucción que genera una petición a memoria, cuando ésta llega al controlador.

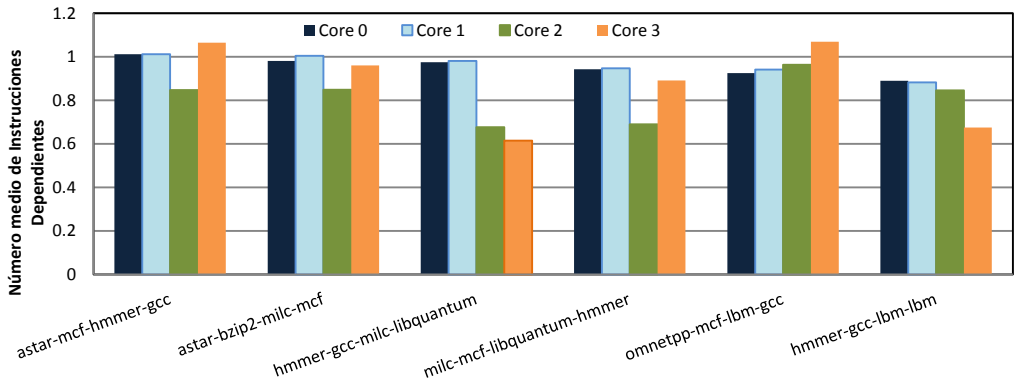

Figura 5-9. Numero medio de instrucciones dependientes en el ROB cuando la petición llega al controlador de memoria.

Como se puede ver en la Figura 5-9, a pesar de tener hasta 128 instrucciones en vuelo, las diferencias entre los distintos procesos no son suficientes como para hacer una discriminación clara. Es más, en la mayoría de los casos, hay una correlación clara entre el número de instrucciones dependientes y la distancia a la cabeza del ROB. Intuitivamente, si suponemos un grado de dependencia semejante para todos los bloques que se traen de memoria, cuanto menor sea la distancia a la cabeza del ROB de la instrucción demandante, mayor será su distancia a la cola, y por tanto más posibilidades de encontrar instrucciones dependientes detrás. Parece, por tanto, que tener la distancia a la cabeza del ROB y esta métrica es redundante. La primera permite establecer mayores diferencias, por lo que ésta métrica no se utiliza en el algoritmo de ordenamiento presentado aquí.

En los análisis previos no se ha tenido en cuenta la naturaleza de las operaciones a memoria. Técnicamente, en un *store* los procesadores no necesitan del resultado de la operación para progresar, ya que no hay dependencias de datos asociadas, y la *Store*

Queue o el *Write Buffer* van a proveer el dato actualizado a los subsiguientes *loads* hasta que el *store* sea guardado en memoria. Parece por tanto que, desde el punto de vista del controlador de memoria, lo razonable es clasificar las operaciones de lectura en memoria de acuerdo a la naturaleza de la instrucción que las provoca, priorizando los *loads* sobre los *stores* [86]. Está decisión debe, en cualquier caso, ser meditada cuidadosamente, pues los *loads* sobre un determinado bloque de memoria pueden solaparse con *stores* ejecutados anteriormente, coincidentes en el mismo bloque y que han fallado en el chip. Este tipo de *loads* quedarán esperando a la resolución del *store* previo en el MSHR correspondiente, de forma que retrasar el *store* asociado tendría un impacto directo en el rendimiento. A continuación se presenta cómo de frecuente es esta situación para distintas aplicaciones de las SPEC2006. En la Figura 5-10 se muestra el número de *loads* que coinciden en el MSHR con un *store* que ha fallado anteriormente, y el valor es obtenido cuando el *store* finaliza (se completa). Las barras indican el máximo y el mínimo de *loads* coincidentes con un *store* en distintas ejecuciones donde la aplicación se ejecuta en diferentes mezclas de aplicaciones.

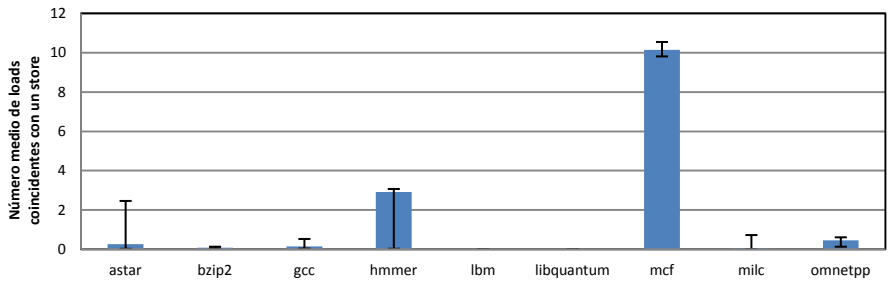

Figura 5-10. Número promedio de loads coincidentes en el mismo bloque que un store que ha fallado a memoria.

Como se puede observar, existe una fuerte disparidad entre las distintas aplicaciones. Así, para *mcf* o *hmmer*, el número de *loads* que piden el mismo bloque que un *store* anterior es relativamente grande, mientras que en otras aplicaciones como *lbm* o *libquantum*, las posibilidades de que esa situación ocurra es despreciable. Las diferencias en la localidad espacial de las distintas aplicaciones es la que justifica esta diferencia de comportamiento, aun más, las aplicaciones pueden variar significativamente su comportamiento en función del instante que se observa y de las aplicaciones con las que entran en conflicto (un mayor retardo en la resolución de los *stores* aumenta la posibilidad de aparición de *loads* coincidentes). Por tanto, es necesario un análisis más cuidadoso antes de retrasar un *store* en el controlador de memoria, puesto que podría afectar a un gran número de *loads* coincidentes. Además, esta información es difícil de

obtener en el controlador de memoria, puesto que, incluso en el supuesto de un conocimiento perfecto del estado del MSHR por parte del controlador, en el momento de tomar la decisión, es probable que una parte significativa de los *loads* coincidentes no hayan sido ejecutados todavía. En la práctica, esta solución no ofrece beneficios notables frente a su complejidad, llegando incluso a observar perdidas de rendimiento. Por tanto, el bajar la prioridad a los *stores* no se implantará en el algoritmo de ordenación final presentado en este trabajo.

5.5 Algoritmo de Planificación Basado en la Criticidad de las Peticiones a Memoria

En el apartado anterior hemos discutido el potencial de distintos criterios para ordenar las peticiones a memoria, suponiendo que el controlador de memoria dispone de acceso instantáneo al estado del procesador cuando la petición llega a él. Aunque útil como ejercicio teórico, su implementación es inviable si queremos mantener la escalabilidad del CMP. Es por tanto que tendremos que hacer llegar el estado del procesador al controlador de memoria, de una forma realista, enviando la información relevante a través de la petición a la jerarquía de memoria y trasladándola en cada fallo de cache hacia el siguiente nivel hasta alcanzar el controlador de memoria. Para que la información sea lo más precisa posible, debemos extraerla en el instante en el que la petición abandona el procesador, es decir, cuando la instrucción que la provoca sea ejecutada en el caso de un *load*, o *committed* en el caso de un *store*. Sin embargo, el estado del procesador puede haber cambiado cuando la petición llegue al controlador de memoria, y por tanto deberemos reajustar esa información usando los datos disponibles en el controlador.

5.5.1 Distancia a la cabeza del ROB

Como se explicó anteriormente, aunque es posible enviar la distancia a la cabeza del ROB de la instrucción acoplada a la petición que genera, esta información llega al controlador de memoria tras recorrer el resto de la jerarquía de memoria, habiendo pasado un número variable de ciclos (dependiente de la contención) y pudiendo, por tanto, haber cambiado. Lo que se busca es estimar, razonablemente, la posición de la instrucción en el ROB cuando un fallo de cache llega al controlador de memoria. Sabiendo que obtener con precisión el valor absoluto será difícil, nos vale con obtener un valor relativo con el que comparar la criticidad de las operaciones de memoria que se encuentren en el controlador en un momento dado. En general, cuando un procesador genera una petición

a memoria, la instrucción que la provoca alcanzará la cabeza del *Reorder Buffer* aproximadamente después del número de ciclos dado por la siguiente fórmula:

$$NumCycles = \frac{DistToRoBHead}{IPC} = DistToRoBHead \cdot CPI$$

(5-3)

Teniendo en cuenta que la distancia a la cabeza del ROB se mide en número de instrucciones. Esta información es suficiente para ordenar las peticiones a memoria de forma adecuada, pues es posible saber cuál es la primera que llegará a bloquear la retirada de instrucciones. Sin embargo, no es necesario un conocimiento absoluto del momento en el que esto ocurrirá, sino que es suficiente conocer la relación existente entre las distintas peticiones para establecer un criterio de prioridad. Así, podemos expresar el CPI de los distintos procesadores normalizado con respecto al más lento, de manera que obtendríamos un criterio de prioridad para las peticiones de la forma:

$$\begin{aligned}\overline{PrioLevel_i} &= DistToRoBHead_i \cdot CPI\_normalizado_p \\ &= DistToRoBHead_i \cdot \frac{CPI_p}{\max\{CPI_0 .. CPI_N\}}\end{aligned}$$

(5-4)

Siendo $\overline{PrioLevel_i}$ el nivel de prioridad asociado a la petición i y CPI_p los ciclos por instrucción del procesador p. El CPI normalizado se calcula respecto al procesador más lento para que su valor oscile entre 0 y 1. En cualquier caso, no es fácil mantener la información del CPI de cada procesador actualizada en el controlador de memoria. Por otra parte, cada controlador de memoria conoce el número de fallos por unidad de tiempo en el chip de cada procesador, para la región de memoria a la que está asociado. Desde esta métrica se puede extraer directamente la tasa de fallos a memoria de cada procesador, la cuál es una aproximación suficiente del CPI [93], y podemos estimar la relación entre los CPI de los distintos procesadores como la relación de sus tasas de fallos.

$$APROX\_CPI\_normalizado_p = \frac{Miss\_freq_p}{\max\{Miss\_freq_0 .. Miss\_freq_N\}}$$

(5-5)

Siendo éste parámetro un factor que nos permite ajustar la distancia a la cabeza del ROB, podemos considerar esta aproximación lo suficientemente precisa como para

discriminar en la criticidad de las distintas operaciones de memoria. Sustituyendo (5-5) en la ecuación (5-4), la distancia corregida a la cabeza del ROB o nivel de prioridad de una petición i, puede ser calculada como:

$$\overline{PrioLevel_i} = DROB_i \cdot \frac{Miss\_freq_p}{\max\{Miss\_freq_0..Miss\_freq_N\}}$$

(5-6)

Donde DROB es la distancia a la cabeza del ROB en el momento en el que la petición i abandona el procesador p, para un CMP con N núcleos. Hay que tener en cuenta que, según lo expresado en la ecuación (5-6), una petición a memoria tiene más prioridad cuando su distancia corregida a la cabeza del ROB, *PrioLevel*, es menor.

La tasa de fallos se calcula cada millón de ciclos, y para evitar errores debidos a comportamientos anómalos puntuales de las aplicaciones, la tasa de fallos de cada procesador se actualiza mediante una media móvil exponencial, siguiendo un proceso similar al expuesto en la sección 4.8. Dado que la tasa de fallos se puede calcular independientemente en cada controlador y se realiza en intervalos lo suficientemente grandes, no es necesaria una comunicación entre los distintos controladores de memoria, lo que supone una ventaja frente a otras soluciones recientes [90].

5.5.2 Fetches y Escrituras

Como se explicó anteriormente, priorizar los fetch de instrucciones sobre otras peticiones a memoria puede tener un impacto positivo en el rendimiento a un coste despreciable, y por tanto vamos a implementarlo. Los *fetches* deben llegar marcados como tales a memoria, y su prioridad es considerada máxima según llegan al controlador, y no necesitan usar la fórmula descrita en el apartado anterior. En lugar de eso, su nivel de prioridad es automáticamente puesto a 0 (máxima prioridad). Por otra parte, sabemos que las escrituras en memoria principal son debidas a *write-backs* del último nivel de cache, y no están en el camino crítico del procesador. Para mantener un algoritmo de ordenación uniforme para todas las peticiones, lo que hacemos es establecer el nivel de prioridad de las escrituras al tamaño del *Reorder Buffer*, lo que supone una distancia a la cabeza máxima (prioridad mínima).

5.5.3 Algoritmo de ordenación en memoria

Una vez que una petición llega al controlador de memoria, se le asigna una prioridad de acuerdo a la ecuación (5-6) y se inserta en la cola del banco correspondiente, la cual debe ser una cola prioritaria (ordenada por orden de prioridad). Con el paso del tiempo, las peticiones van acumulándose y es necesario algún mecanismo de envejecimiento que permita a las peticiones que llevan esperando mucho tiempo en la cola adelantar a aquellas que llegan nuevas pero con mayor prioridad, o corremos el riesgo de provocar retrasos severos e incluso inanición. Para ello, hemos ideado un mecanismo sencillo de envejecimiento de las peticiones y que además permite, como veremos posteriormente, un control sobre los beneficios obtenidos en nuestro sistema.

Partiendo de una situación en la que no existen peticiones de alta prioridad en la cola del banco, como por ejemplo la situación (a) de la Figura 5-11, comienza el proceso de búsqueda de peticiones de alta prioridad. En el ejemplo, cada bloque en la cola representa una petición, donde el número mayor se corresponde con el nivel de prioridad, y el subíndice representa el procesador del que proviene la petición. El algoritmo de selección de peticiones de alta prioridad necesita un parámetro de entrada que denominamos "Distancia Umbral". Al comienzo del algoritmo, se recorre la cola de forma que todas las peticiones cuya prioridad está por debajo de esta "Distancia Umbral" son marcadas como de alta prioridad (mediante un bit). Hay que tener en cuenta que la cola está ordenada en función del nivel de prioridad, luego no hace falta recorrerla entera. Por otra parte, todas las peticiones que no han cumplido el requisito, reducen su nivel de prioridad en la misma cantidad que la "Distancia Umbral", creando así el proceso de envejecimiento de las peticiones. Supongamos que en el ejemplo de la Figura 5-11, la "Distancia Umbral" utilizada es 16. En ese caso, tras el proceso de selección (b), podemos ver que cinco peticiones han sido marcadas como de alta prioridad, mientras que el resto de peticiones han visto ajustado su nivel de prioridad. Por tanto, el algoritmo de envejecimiento usado es:

$$For\ (i\ in\ peticiones\_pendientes)$$
$$if\ \left(\overline{PrioLevel_i} < ThresholdDistance\right) \rightarrow La\ petición\ i\ es\ de\ Alta\ Prioridad$$
$$else \rightarrow \overline{PrioLevel_i} = \overline{PrioLevel_i} - ThresholdDistance \quad \textbf{(5-7)}$$

Cuando todas las peticiones prioritarias abandonan la cola, situación (c) de la Figura 5-11, comienza de nuevo el algoritmo de búsqueda de peticiones de alta prioridad y envejecimiento. Hay que tener en cuenta que hasta ese momento nuevas peticiones habrán llegado al controlador de memoria, se corresponden con los bloques blancos del estado (c). Después de volver a aplicar el criterio de selección y envejecimiento, siete peticiones han sido marcadas como de alta prioridad.

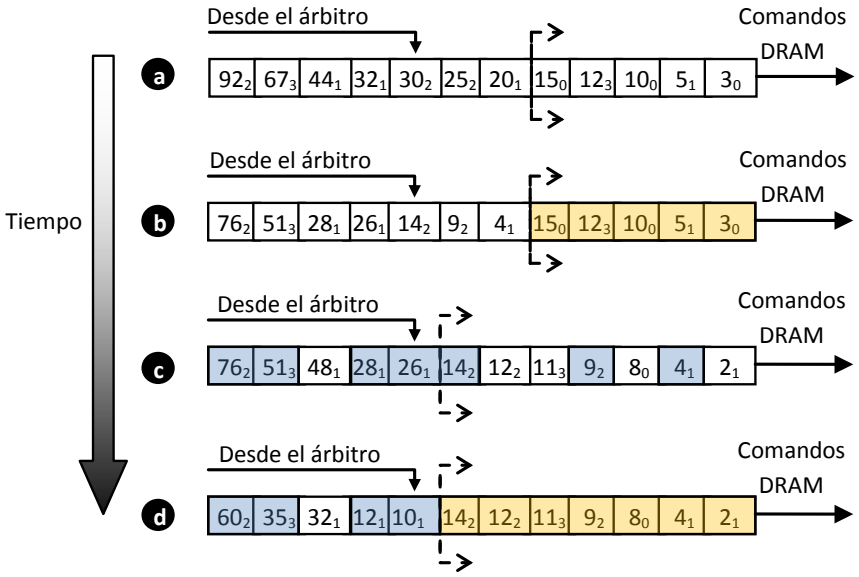

Figura 5-11. Ejemplo de arbitraje de las peticiones en el controlador de memoria con distancia umbral 16.

En contraste con otros algoritmos de ordenación en ráfagas, como PAR-BS [89], la longitud de la ráfaga entre decisiones es variable y no se exige la presencia de un número mínimo de peticiones por procesador. De hecho, en el primer caso de ordenación del ejemplo, no se marca como prioritaria ninguna petición del procesador 2. Además, una vez que una petición ha sido marcada como prioritaria, no se vuelve a hacer uso de la información sobre el procesador del que procede para la ordenación de las operaciones de memoria, mientras que en PAR-BS el orden de las operaciones pendientes es determinado por el número de peticiones pendientes de cada procesador, y las peticiones son agrupadas por procesador. Lo que es equivalente a priorizar el procesador con el menor CPI en cada intervalo. ATLAS por otra parte, con el fin de capturar un comportamiento más amplio, utiliza la media móvil exponencial de la tasa de fallos coordinada entre controladores para reordenar las peticiones de los distintos procesadores. En nuestro caso, se utiliza una combinación de la tasa de fallos local promediada del procesador y la distancia a la cabeza del ROB de cada petición para la ordenación. Aunque la tasa de fallos es una buena aproximación al CPI, la distancia a la cabeza del ROB es una

herramienta muy eficaz para corregir efectos locales, y permite capturar de una forma más especifica la criticidad de las distintas operaciones a memoria.

A continuación se presenta el algoritmo final para determinar el nivel de prioridad de las peticiones que llegan a memoria, dada una petición con distancia a la cabeza del ROB "$DROB_i$", originada por el procesador p, con tasa de fallos $Miss\_freq_p$:

$$\overline{PrioLevel_i} = DROB_i \cdot \frac{Miss_{freq_p}}{\max\left\{Miss_{freq_0}..Miss_{freq_N}\right\}}$$

$$if(request\_type_i = fetch) \rightarrow \overline{PrioLevel_i} = 0$$

$$else\ if\ (request\_type_i = writeback) \rightarrow \overline{PrioLevel_i} = \text{ROB\_size}$$

(5-8)

Posteriormente, cuando no existan peticiones prioritarias en la cola del controlador, se ejecuta el algoritmo de detección de peticiones prioritarias y envejecimiento visto anteriormente:

$$For\ (i\ in\ peticiones\_pendientes)$$
$$if\ \left(\overline{PrioLevel_i} < ThresholdDistance\right) \rightarrow La\ petición\ i\ es\ de\ Alta\ Prioridad$$
$$else \rightarrow \overline{PrioLevel_i} = \overline{PrioLevel_i} - ThresholdDistance$$

(5-9)

Como veremos posteriormente, el parámetro de entrada de este algoritmo, "*Distancia Umbral*" (*ThresholdDistance*), puede ser utilizado para mejorar el rendimiento global a costa del *fairness* o viceversa. Además, este parámetro puede ser ajustado dinámicamente en ejecución con el fin de obtener los mejores resultados en las dos figuras de mérito, simultáneamente.

5.6 Evaluación

Para evaluar la propuesta nos compararemos con tres arbitrajes distintos: FR-FCFS [43], PAR-BS [89] y ATLAS [90]. Como se explica en el apartado 5.3, hemos escogido estos algoritmos por su complejidad similar en el controlador de memoria a la propuesta aquí presentada, por sus buenos resultados en rendimiento y *fairness*, y su capacidad de actuar en tiempo de ejecución. Aunque se han probado otras alternativas de arbitraje para el particionado del ancho de banda, no se muestran sus resultados por no ser competitivos [93], u obtener resultados muy semejantes a los ya representados [88].

Usaremos como referencia el algoritmo FR-FCFS por ser el más común. Este algoritmo no tiene ningún parámetro a considerar. Para PAR-BS, que mejora a FR-FCFS en rendimiento y *fairness*, el *BatchCap* usado, o límite de peticiones por procesador en cada ráfaga, es 5, aunque se muestran resultados variando este parámetro posteriormente. Finalmente, para ATLAS, que mejora los resultados de rendimiento de PAR-BS, tomamos un *HistoryWeight* de 0.875 para la EMA (Media Exponencial Móvil) y un intervalo de medición (*quantum length*) de 1 millón de ciclos. Finalmente, para nuestra propuesta a la que denominaremos DROB, el *HistoryWeight* para los EMA será igualmente de 0.875 y usaremos una *Distancia Umbral* de 16.

Compararemos estos algoritmos tanto en rendimiento como en *fairness* y para ello vamos a usar las tres métricas vistas en el apartado 2.9. Ambos resultados, rendimiento y *fairness*, son importantes y nuestro objetivo es conseguir los mejores resultados en las dos, aunque en algunos casos solo sea relevante una de ellas. Es por eso que, en el apartado 5.7, discutiremos como nuestra propuesta puede mejorar los resultados obtenidos en cualquiera de las métricas simplemente variando un parámetro.

Para la evaluación de la propuesta hemos utilizado el entorno de trabajo descrito en el capítulo 2. Usamos además un simulador de DDR2/3 basado en el controlador detallado de memoria existente en la última versión de GEMS. En éste se modela la contención en los bancos, las distintas latencias y refrescos, así como la contención en los buses de acceso.

El objetivo de este estudio es evaluar las propuestas en un entorno de alta contención en memoria. Debido a los requerimientos computacionales del entorno de trabajo y la gran cantidad de simulaciones a realizar, se ha escalado el sistema de forma que podamos reproducir las características buscadas en una arquitectura con la que poder trabajar reduciendo el número de procesadores. Simulamos un sistema de 4 procesadores de ejecución fuera de orden con 128 instrucciones en vuelo y un ancho de *issue* de 4 instrucciones. Las principales características del sistema se encuentran en la Tabla 3-2. A continuación se presentan las características fundamentales de la memoria principal, Tabla 5-1. Hay que tener en cuenta que el ancho de banda a memoria ha sido escalado de acuerdo a las aplicaciones de las que disponemos, manteniendo la premisa inicial de que el ancho de banda a memoria será escaso en el futuro [10]. Siguiendo este razonamiento,

el ancho de banda para este sistema de cuatro procesadores es de 1.6GB/s, con 8 bancos de memoria.

Tabla 5-1. Características principales del subsistema de memoria

Sistema DRAM			
Controlador DRAM	En-chip 1,6 GB/s ancho de banda pico DRAM	*Parámetros DRAM*	DDR2-200 8 bancos tCL=15ns tRCD=15ns, tRP=15ns
Configuración DIMM	Un solo rank 8 chips RAM en un DIMM Ancho de canal 64-bit		

5.6.1 Características de las Cargas de Trabajo

Para la evaluación de la propuesta vamos a usar cargas de trabajo compuestas por mezclas de aplicaciones de las SPEC2006. Las aplicaciones están compiladas usando la versión 4.3.1 de gcc con la optimización –O3. Todas las aplicaciones se ejecutan en su versión de referencia. Los procesos se duermen al comienzo de la región de interés y son despertados simultáneamente, ejecutando al menos 500 millones de instrucciones. Cada proceso está ligado a un procesador mediante la utilidad de Solaris *processor_bind*, de forma que luego sea fácil identificar las aplicaciones para evaluar los resultados.

Dado que el número de combinaciones posibles entre las aplicaciones de las SPEC es inmanejable, hemos caracterizado cómo es de demandante en memoria cada una de ellas obteniendo el número de fallos por cada mil instrucciones (MPKI). De los resultados se han escogido las 12 aplicaciones que se muestran en la Tabla 5-2

Tabla 5-2. Caracterización de las aplicaciones en función de sus necesidades de memoria.

Altos requerimientos de memoria				**Bajos requerimientos de memoria**			
Aplicación	***MPKI***	***Dist. media a Cabeza ROB***	***IPC***	***Aplicación***	***MPKI***	***Dist. media a Cabeza ROB***	***IPC***
Mcf	97.38	16.90	2.0942	*Astar*	9.26	24.35	0.5765
Libquantum	50.00	6.28	1.1102	*Hmmer*	5.66	9.97	0.3718
Lbm	43.52	44.95	2.3235	*Bzip2*	3.98	19.76	0.5343
Milc	27.90	33.41	1.7102	*Gcc*	0.34	14.71	1.0868
Sphinx3	24.94	21.48	1.0370	*Namd*	0.19	15.05	0.3010
Xalancbmk	22.95	7.36	0.9831				
Omnetpp	21.63	20.21	1.4316				

Se han dejado de lado muchas de las aplicaciones menos agresivas en memoria (nueve), ya que sus resultados eran irrelevantes (no es posible mejorar el arbitraje en memoria si no existe saturación en el uso de los recursos), además de otras aplicaciones (cinco) difíciles de trasladar por limitaciones del entorno de trabajo. La mayoría de las aplicaciones escogidas (siete) pertenecen al grupo de aplicaciones muy demandantes en memoria (un gran número de fallos por cada mil instrucciones), y cinco de ellas tienen bajos requerimientos de memoria (un menor número de fallos cada mil instrucciones). En la Tabla 5-2 se han representado además la distancia media a la cabeza del ROB que tienen esas aplicaciones, que, como puede observarse, no tiene una correlación directa con sus necesidades en memoria. Esto es debido a que en algunos casos las peticiones a memoria tienden a ir agrupadas en el tiempo, mientras que en otros la distribución de fallos es más homogénea a lo largo de la ejecución. Hay que tener en cuenta que los números representados en la tabla han sido obtenidos mediante la ejecución en solitario de la aplicación en el sistema y promediando al final su valor.

Tomando esa división como referencia (según el requerimiento a memoria), hemos creado 40 cargas de trabajo distintas combinando un número diferente de aplicaciones demandantes y no demandantes. La Tabla 5-3 muestra un resumen de las cargas de trabajo agrupadas según el porcentaje de aplicaciones demandantes presentes en la misma. En cada grupo, se desglosan las distintas combinaciones de aplicaciones.

Tabla 5-3. Cargas de trabajo evaluadas en ordenadas en función del número de aplicaciones demandantes en memoria presentes.

Porcentaje de aplicaciones demandantes	Combinación de aplicaciones
0%	astar-bzip2-gcc-namd hmmer-astar-bzip2-bzip2 hmmer-hmmer-gcc-gcc
25%	astar-gcc-hmmer-lbm astar-mcf-hmmer-gcc astar-namd-namd-xalancbmk hmmer-gcc-astar-omnetpp hmmer-gcc-bzip2-libquantum
50%	astar-bzip2-lbm-mcf astar-bzip2-milc-mcf bzip2-lbm-hmmer-mcf bzip2-omnetpp-xalancbmk-hmmer gcc-bzip2-lbm-milc Gcc-hmmer-milc-lbm hmmer-astar-omnetpp-milc hmmer-gcc-lbm-lbm Hmmer-gcc-mcf-mcf Libquantum-milc-hmmer-gcc milc-mcf-gcc-hmmer omnetpp-mcf-hmmer-gcc xalancbmk-xalancbmk-astar-bzip2
75%	astar-lbm-milc-sphinx3 astar-libquantum-xalancbmk-sphinx3 bzip2-libquantum-mcf-sphinx3 Bzip2-omnetpp-lbm-mcf gcc-lbm-mcf-sphinx3 gcc-libquantum-milc-sphinx3 Hmmer-milc-omnetpp-lbm milc-libquantum-sphinx3-namd Milc-mcf-libquantum-hmmer namd-milc-lbm-mcf namd-sphinx3-libquantum-mcf omnetpp-libq-bzip2-sphinx3 Omnetpp-mcf-lbm-gcc xalancbmk-gcc-lbm-mcf xalancbmk-mcf-milc-bzip2
100%	mcf-milc-lbm-lbm omnetpp-omnetpp-lbm-lbm omnetpp-milc-mcf-sphinx3 omnetpp-xalancbmk-milc-lbm

5.6.2 Rendimiento

Vamos a medir la productividad de los distintos algoritmos de arbitraje en el controlador de memoria utilizando dos métricas distintas. Como hemos explicado anteriormente, usaremos la media armónica del CPI como medida de rendimiento para mantener la métrica de cuanto menos mejor en todas las gráficas. Igualmente usaremos también el *Weighted SpeedUp del CPI* [58], utilizado comúnmente como medida que balancea rendimiento y *fairness*, y que representa la suma de los *slowdown* que sufre el CPI de cada aplicación al ejecutarse en conjunto con otras respecto a su ejecución en solitario.

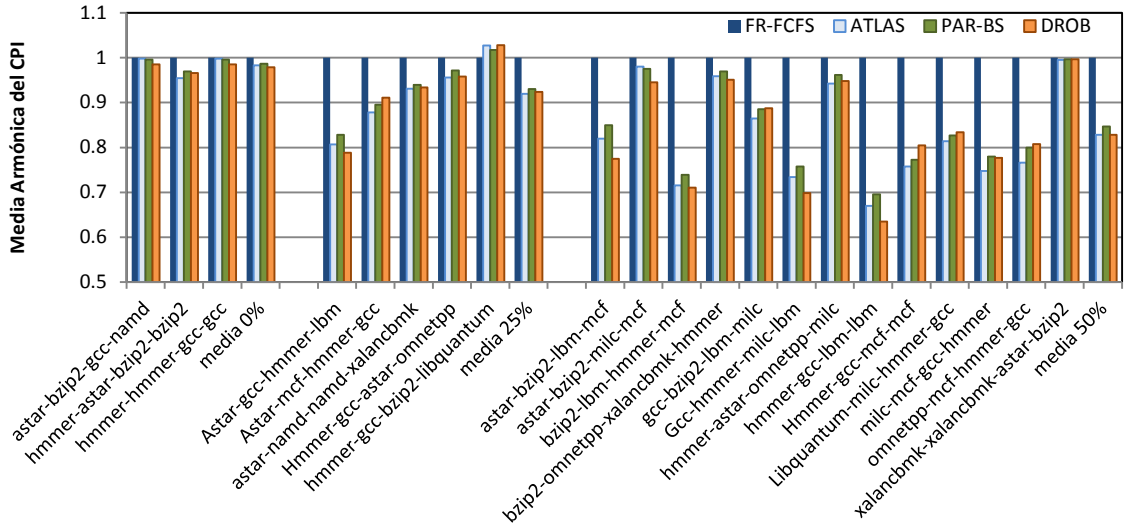

Figura 5-12. Media Armónica del CPI normalizado con FR-FCFS para distintas combinaciones de cargas de trabajo, con un porcentaje de aplicaciones demandantes menor o igual al 50%.

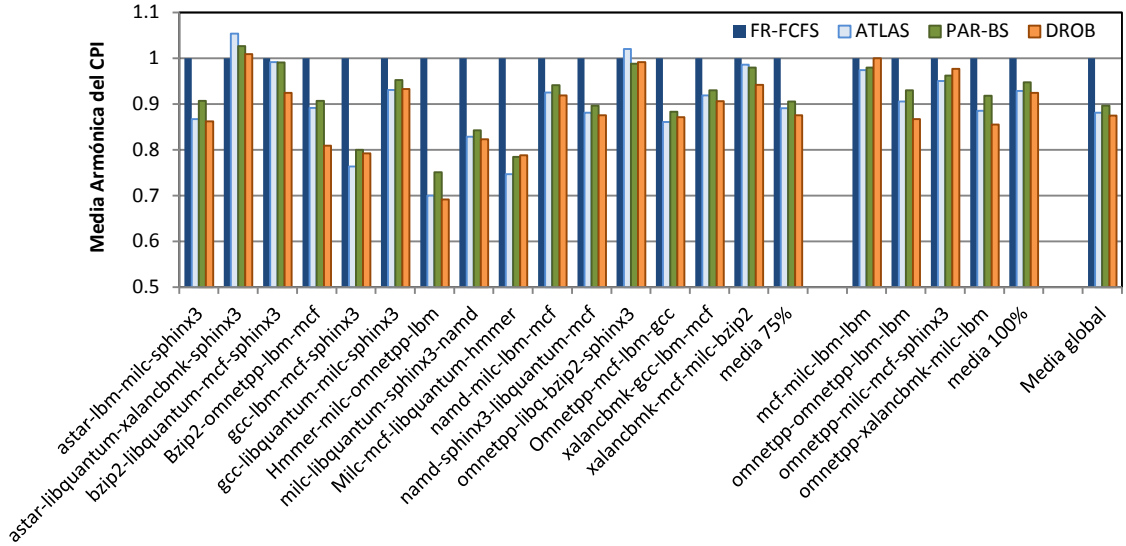

Figura 5-13. Media Armónica del CPI normalizado con FR-FCFS para distintas combinaciones de cargas de trabajo, con un porcentaje de aplicaciones demandantes mayor o igual al 75%.

La Figura 5-12 y la Figura 5-13 muestran el rendimiento global del sistema de nuestra propuesta frente a los algoritmos de arbitraje con los que nos comparamos (FR-FCFS, PAR-BS y ATLAS) para cada una de las 40 cargas de trabajo simuladas agrupadas según el tipo de mezcla visto en la Tabla 5-3. Nuestra propuesta obtiene ligeramente mejor rendimiento que la alternativa que mejores resultados ofrece, ATLAS, superando a FR-FCFS por un 12.5% de media con un máximo del 37% en la combinación *hmmer-gcc-lbm-lbm*, mientras que ATLAS consigue un 11.9% de mejora en media y un máximo del 33%.

Como era de esperar, ninguno de los algoritmos obtiene diferencias notables de rendimiento cuando ninguna de las aplicaciones en ejecución tiene alta demanda en memoria. En estos casos existe ancho de banda suficiente para cubrir las necesidades de las aplicaciones en prácticamente todo momento, de forma que apenas hay interferencias entre ellas y margen de mejora es bajo. A medida que aumenta la presión sobre el controlador de memoria y aparece la contención, las diferencias en rendimiento se hacen más palpables, tal y como se observa en los resultados de mezclas del 50% y 75%. Cuando la presión a memoria es demasiado alta (100%, 4 aplicaciones altamente demandantes), los resultados se amortiguan ligeramente, existiendo casos en los que la ganancia es prácticamente nula. Hay que tener en cuenta que en este caso todavía es posible obtener algo de ganancia debido a que, aunque todas las aplicaciones son demandantes, no todas lo son en igual medida (tienen diferentes MPKI). En cualquier caso, cualquiera de los algoritmos de ordenación mostrados obtiene sus mejores resultados cuando existen diferencias notables en el comportamiento en memoria de las aplicaciones concurrentes. Si tenemos en cuenta únicamente aquellas cargas de trabajo con mayores diferencias, ATLAS consigue un 16.4% de mejora en rendimiento frente a FR-FCFS, mientras que PAR-BS supera a FR-FCFS por un 14.3%, y nuestra propuesta lo hace por aproximadamente un 17% de media.

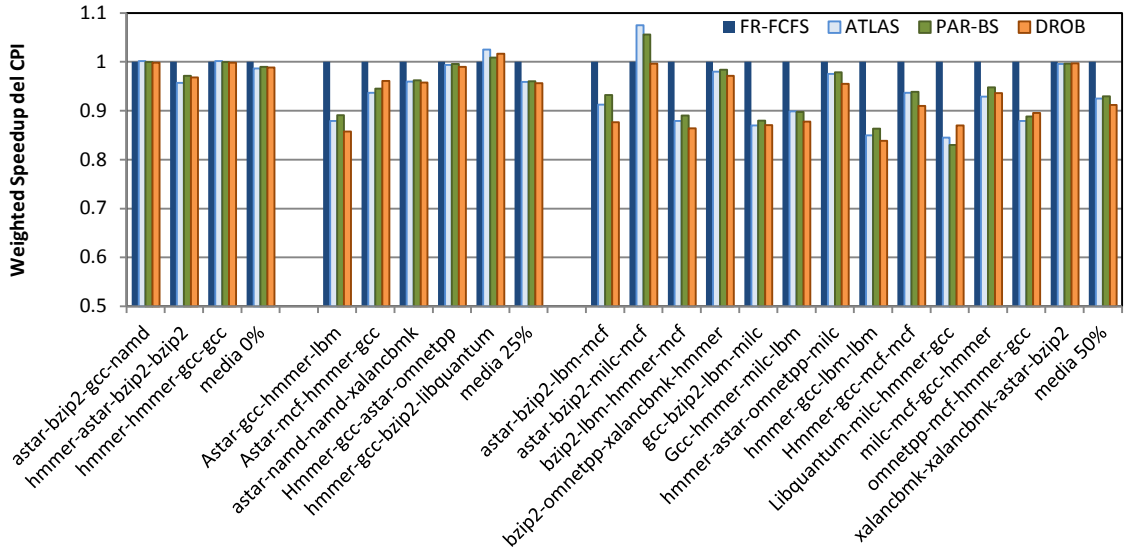

Figura 5-14. Weighted SpeedUp del CPI normalizado respecto a FR-FCFS para distintas combinaciones de cargas de trabajo, con un porcentaje de aplicaciones demandantes menor o igual al 50%.

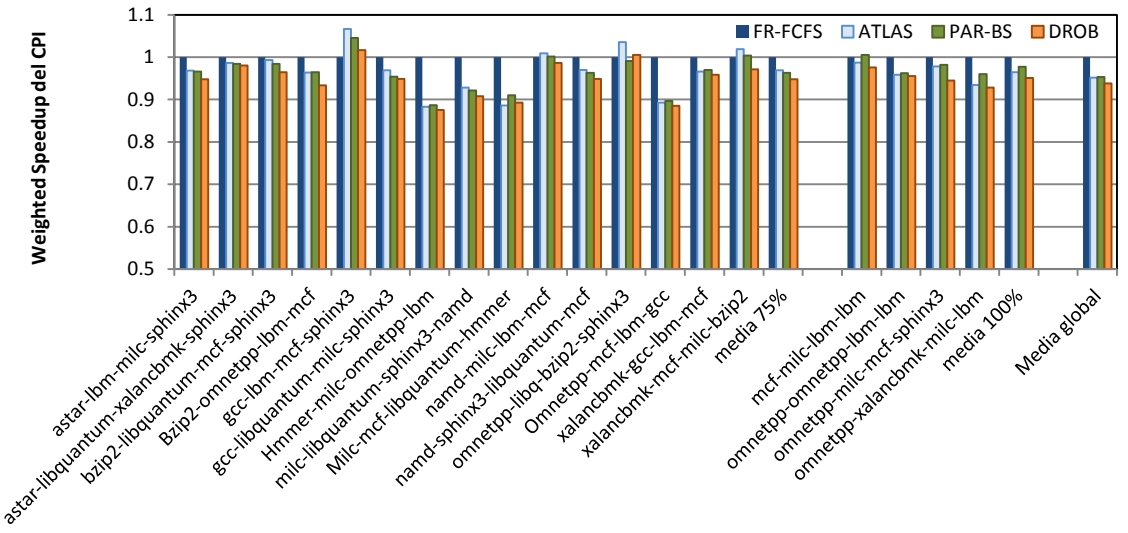

Figura 5-15. Weighted SpeedUp del CPI normalizado respecto a FR-FCFS para distintas combinaciones de cargas de trabajo, con un porcentaje de aplicaciones demandantes mayor o igual al 75%.

Sin embargo, si consideramos de alguna forma el *fairness* en los cáculos, podemos observar como en la Figura 5-14 y en la Figura 5-15, los resultados de *Weighted SpeedUp del CPI* de ATLAS se igualan a los de PAR-BS, mejorando a FR-FCFS por un 5%, mientras que nuestra propuesta supera a ambos en un 2% de media. Podemos así concluir que nuestra propuesta mejora los resultados de otros algoritmos de arbitraje orientados al rendimiento, teniendo en cuenta la criticidad de la instrucción y no solo el

comportamiento del procesador, y obteniendo buenos resultados en el *Weighted SpeedUp del CPI*.

5.6.3 Fairness

Como dijimos anteriormente es importante tener en cuenta la medida de *fairness* a la hora de diseñar un sistema de arbitraje en la compartición de recursos, donde lo que se busca es que ninguna aplicación abuse del ancho de banda a memoria disponible, garantizando unos mínimos y evitando la inanición. El parámetro utilizado a este efecto mide el máximo *slowdown* de las aplicaciones que se están ejecutando en el sistema comparado con su IPC de su ejecución en solitario. Muchos de los algoritmos de arbitraje en el controlador de memoria tienden a mejorar el rendimiento de las aplicaciones con alto IPC, que suelen ser las aplicaciones que más sufren cuando existe conflicto en el controlador de memoria frente a aplicaciones más demandantes, lo que redunda en una disminución del máximo *slowdown*. Sin embargo, los algoritmos que solo se fijan en el *throughput* para tomar decisiones, tienden a penalizar en exceso a las aplicaciones más demandantes en memoria, pudiendo invertir la situación, bloqueando a las aplicaciones más demandantes y provocando una disminución del *fairness* (aumento del máximo *slowdown*, pasando de ser la aplicación de mayor IPC a la aplicación de mayor demanda en memoria). Tal y como se observa en la Figura 5-16 y en la Figura 5-17, nuestra propuesta es la que mejores resultados obtiene en fairness al tener en cuenta la criticidad de las instrucciones, superando a FR-FCFS en un 10% en promedio, y en un 31% de máximo. Además, mejora los resultados obtenidos por PAR-BS y ATLAS en un 6% y un 7.5% de media respectivamente, y hasta un 20% en el mejor caso. Es interesante notar que, aun cuando FR-FCFS obtiene los peores resultados en *fairness* de los cuatro algoritmos, existen cargas de trabajo en los que los algoritmos de arbitraje más complejos afectan negativamente y pierden de forma sustancial respecto a FR-FCFS.

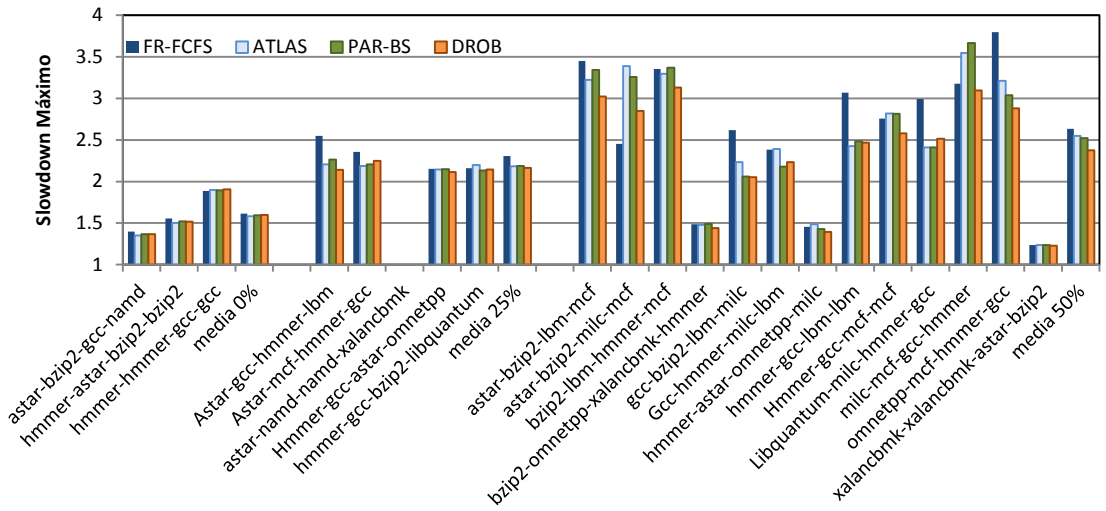

Figura 5-16. Máximo slowdown de los distintos algoritmos de arbitraje para distintas combinaciones de cargas de trabajo, con un porcentaje de aplicaciones demandantes menor o igual al 50%.

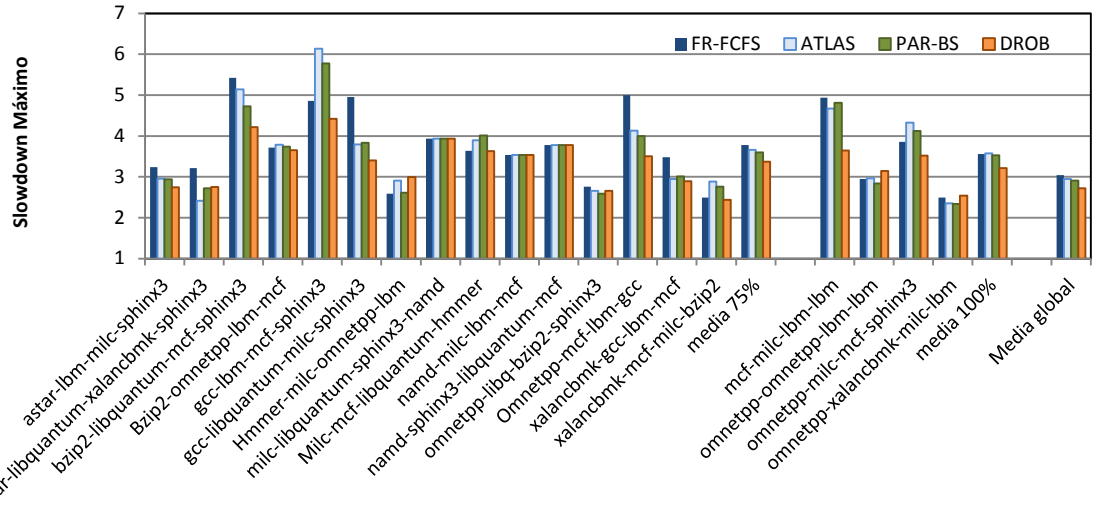

Figura 5-17. Máximo slowdown de los distintos algoritmos de arbitraje para distintas combinaciones de cargas de trabajo, con un porcentaje de aplicaciones demandantes mayor o igual al 75%.

Como se aprecia en la Figura 5-16 y en la Figura 5-17, cuando las aplicaciones en ejecución compiten agresivamente por los recursos en memoria, tanto ATLAS como PAR-BS obtienen de media peores resultados que FR-FCFS (en el peor caso, astar-bzip2-milc-mcf, pierden un 32% y un 40% respectivamente). Hay que tener en cuenta que los datos presentados representan el *slowdown* de las aplicaciones debido a su ejecución conjunta respecto a su ejecución en solitario, por lo que los valores de *slowdown* obtenidos no son sólo debidos a los conflictos en memoria, si no que una parte importante se debe a los conflictos en cache, como se aprecia en los resultados del 0%, donde no existiendo apenas conflicto en el ancho de banda a memoria, el *slowdown* medio de las aplicaciones es de aproximadamente el 50%. Incluso en estas circunstancias, actuando

sobre la forma en la que las peticiones son atendidas en memoria, es posible obtener ganancias en torno al 30%.

5.7 Comportamiento Ajustable Estáticamente

Hasta ahora hemos considerado el parámetro fundamental de nuestro algoritmo, la *Distancia Umbral*, como un valor fijo de 16 instrucciones a la cabeza de un ROB de 128 instrucciones. Bajo estas condiciones, existe un equilibrio razonable entre rendimiento y *fairness*, permitiendo a DROB superar, en ambas métricas, a los algoritmos de arbitraje con los que nos hemos comparado. Sin embargo, dependiendo del entorno de uso, puede ser más interesante favorecer una u otra métrica. Supongamos por ejemplo un CMP en el que se están ejecutando servidores privados virtuales (VPS) de diferentes clientes. En esta situación el fairness es fundamental. No obstante, en un escenario en el que todos los procesadores están ejecutando múltiples aplicaciones del mismo usuario, puede ser más interesante maximizar el rendimiento global del sistema. Nuestra propuesta tiene la capacidad de, modificando la *Distancia Umbral*, maximizar una u otra métrica.

Tal y como se explica en el apartado 5.5.3, la velocidad a la que envejecen las peticiones en el controlador de memoria se puede modificar variando la *Distancia Umbral*. Si tomamos una *Distancia Umbral* menor, la velocidad de envejecimiento se reduce, debido a que las peticiones encoladas deben esperar un mayor número de pasos del algoritmo antes de ser atendidas, lo que implica que pueden adelantarse un mayor número de peticiones que llegan nuevas al controlador. De esta forma, las peticiones de alta prioridad tienen menos competencia a la hora de acceder a los recursos del sistema, lo que implica una mejora en el rendimiento global del sistema. Por el contrario, si aumentamos la *Distancia Umbral*, el proceso de envejecimiento se acelera, las peticiones avanzan más rápidamente y las peticiones críticas que llegan al controlador tienen que competir con un mayor número de peticiones ya encoladas. De esta forma, se reduce el tiempo máximo que una petición permanece encolada, lo que consecuentemente mejora el *fairness*. Intuitivamente, reducir la *Distancia Umbral* favorece a las peticiones más cercanas a la cabeza del ROB, mientras que aumentar la *Distancia Umbral* favorece a las peticiones que llevan más tiempo esperando en el controlador de memoria.

La Figura 5-18 y la Figura 5-19 muestran el rendimiento y *fairness* que se obtienen cuando se varía la *Distancia Umbral*. Como se puede observar en la Figura 5-18, en el valor más bajo de la *Distancia Umbral* (1) obtenemos los mejores resultados de

rendimiento, mejorando a FR-FCFS en un 16% de media, siendo una ganancia del 22% si solo tenemos en cuenta las cargas de trabajo donde existe competición por los recursos de memoria. Por otro lado, en la Figura 5-19 vemos como se puede mejorar los resultados de *fairness* aumentando la *Distancia Umbral*, superando a FR-FCFS en un 12% de media. Se observa además que este comportamiento en rendimiento y *fairness* es consistente para todas las cargas de trabajo evaluadas.

Los otros algoritmos de arbitraje en memoria estudiados poseen también parámetros fijos que pueden variarse. Por ejemplo, PAR-BS limita el número de peticiones por procesador que pueden ir en cada ráfaga (*BatchCap*). Como se puede observar en la Figura 5-18 y la Figura 5-19, variar este parámetro no produce grandes diferencias en rendimiento o *fairness*, e incluso ofrece resultados inconsistentes en algunas de las cargas de trabajo evaluadas. Así, en unos casos la decisión de incrementar el parámetro mejora el rendimiento, mientras que entro caso, esa misma decisión mejora el *fairness*. Así pues, en contraste con DROB, la parametrización de PAR-BS obtiene resultados poco consistentes y difícilmente apreciables, lo que lo hace menos práctico para este propósito. Algo parecido se puede apreciar en ATLAS, donde los cambios son incluso menores.

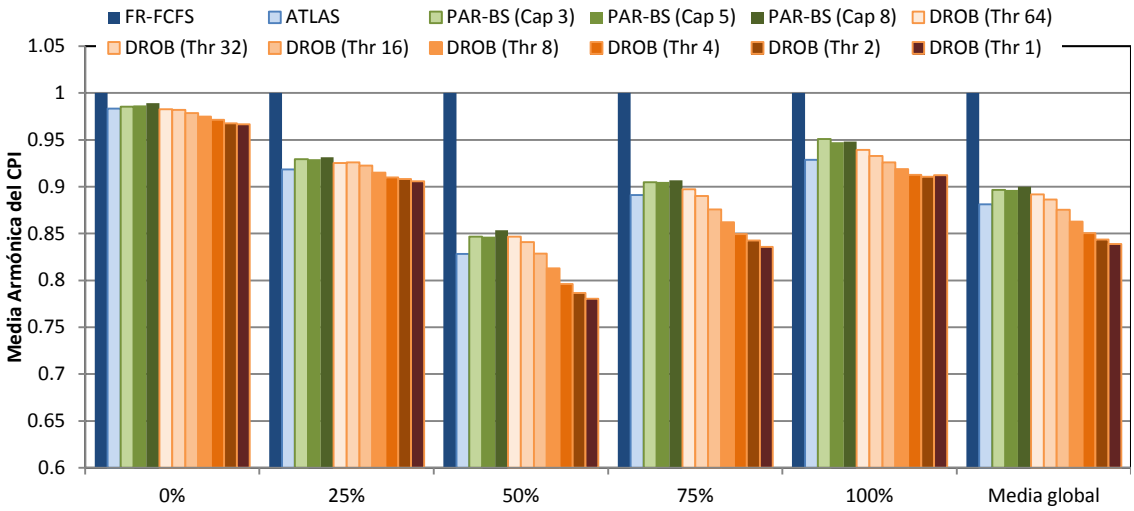

Figura 5-18. Media armónica del CPI normalizada frente a FR-FCFS de DROB para distintos valores de *Distancia Umbral* (Thr) variando de 1 a 64.

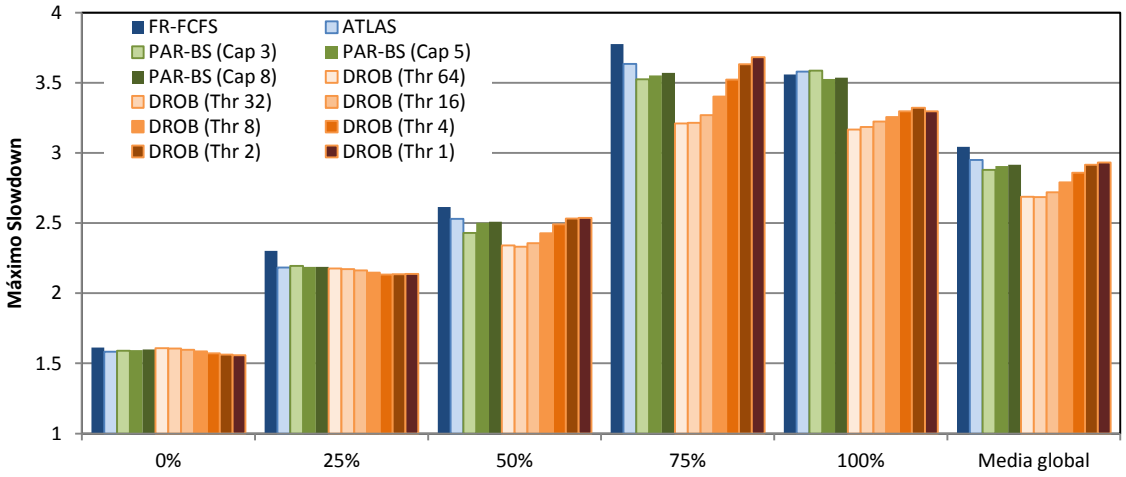

Figura 5-19. Máximo slowdown de DROB para distintos valores de *Distancia Umbral* (Thr) variando de 1 a 64 en comparación con otros algoritmos de arbitraje en memoria.

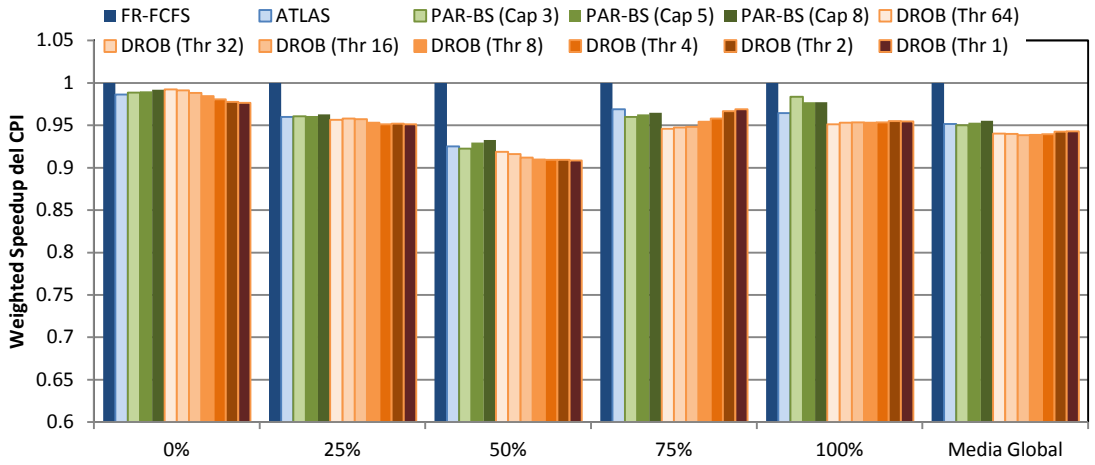

Figura 5-20. Weighted SpeedUp del CPI normalizado frente a FR-FCFS de DROB para distintos valores de *Distancia Umbral* (Thr) variando de 1 a 64.

5.8 Comportamiento Adaptativo

Como se puede comprobar en los resultados mostrados en el apartado anterior, los resultados más equilibrados entre rendimiento y *fairness* se obtienen con una *Distancia Umbral* de 16. Sin embargo, centrándonos en las cargas de trabajo una a una, vemos cómo se consiguen mejores resultados con valores distintos. Si además tenemos en cuenta que el comportamiento de las aplicaciones cambia a lo largo de la ejecución, podría ser interesante que el controlador de memoria fuese capaz de adaptar la *Distancia Umbral* al estado del sistema con el fin de mejorar el equilibrio entre rendimiento y *fairness* incluso más. Para poder evaluar la eficacia de nuestros cambios en la *Distancia Umbral*, implementamos un mecanismo sencillo que evalúa la tendencia en la tasa de fallos de cada aplicación en periodos de un millón de ciclos. Un incremento en esta medida podría

implicar que la aplicación entra en una fase de alto MPKI, o bien que mantiene su MPKI y ha aumentado su IPC (debido al cambio de la *Distancia Umbral*). Para poder distinguir ambos casos, usaremos la distancia media a la cabeza del ROB de las peticiones de ese procesador. Si no hay una alteración significativa en este parámetro, supondremos que el MPKI no ha sufrido variaciones y por tanto los cambios en su comportamiento son debidos a nuestras decisiones sobre la *Distancia Umbral*. Si por el contrario el parámetro sufre grandes variaciones descartaremos ese procesador para evaluar la eficacia de nuestras decisiones en ese intervalo. Determinamos pues con los datos de todos los procesadores que no han variado su MPKI si nuestra decisión sobre la *Distancia Umbral* ha sido positiva (es decir, hay una desviación negativa en el CPI). En ese caso, la *Distancia Umbral* cambia de nuevo en la misma dirección, y en caso contrario, tomamos la decisión opuesta. Los cambios en la *Distancia Umbral* se realizan gradualmente en pasos de a uno, y para tomar una decisión de variarla, el cambio apreciado en el CPI global debe ser mayor del 5%, o en caso contrario se mantiene. Los resultados de rendimiento y *fairness* pueden observarse en la Figura 5-21 y la Figura 5-23 respectivamente. Como se puede apreciar, ambos resultados son parejos o superan ligeramente a los de la configuración estática de mejor caso.

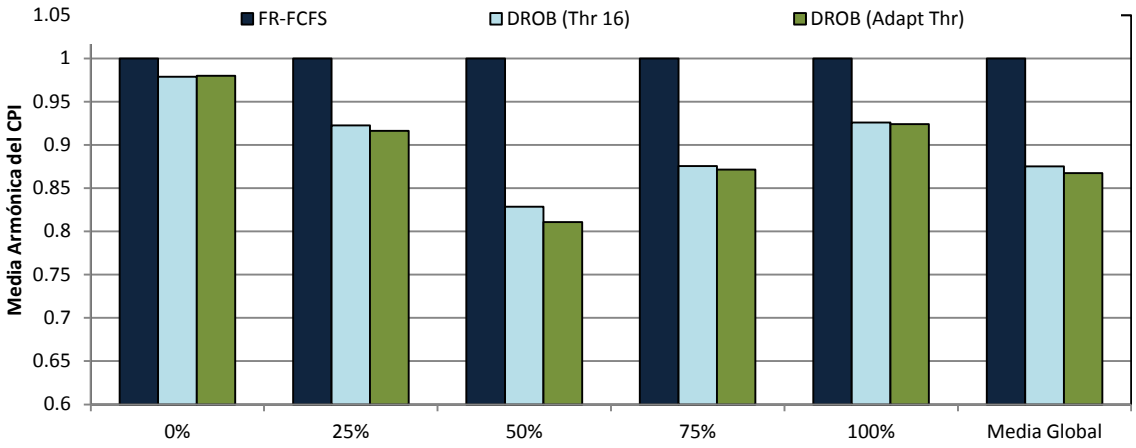

Figura 5-21. Media armónica del CPI para DROB con Distancia Umbral (Thr) adaptativa normalizado respecto a FR-FCFS.

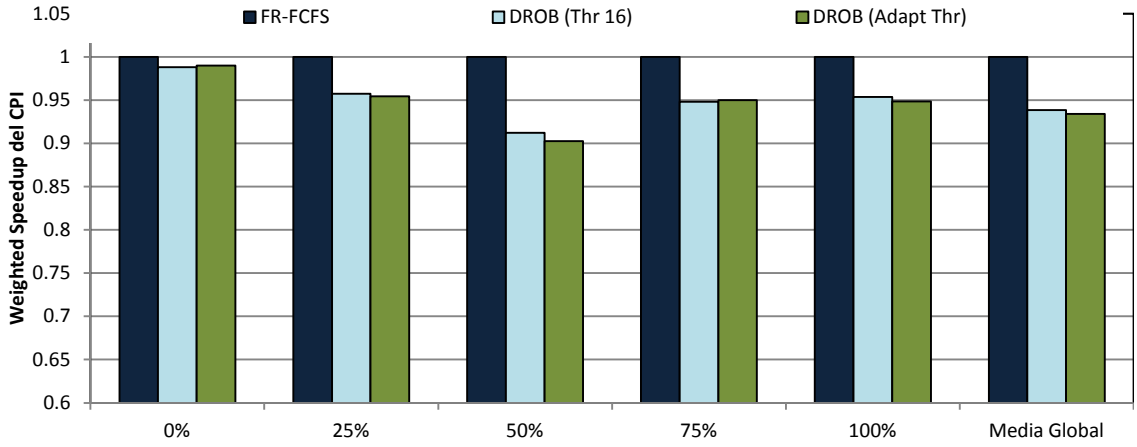

Figura 5-22. Weighted Speedup del CPI para DROB con Distancia Ubral (Thr) adaptativa normalizado respecto a FR-FCFS.

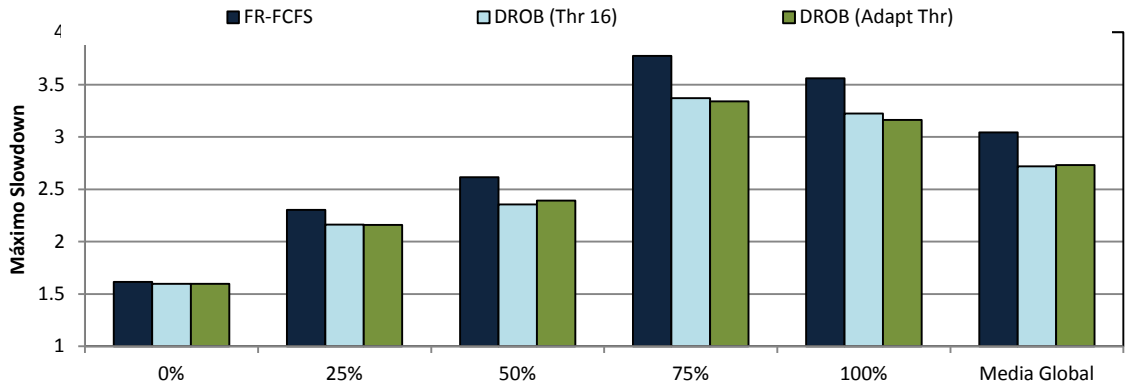

Figura 5-23. Máximo slowdown del CPI para DROB con Distancia Umbral (Thr) adaptativa normalizado respecto a FR-FCFS.

En la Figura 5-24 podemos observar el comportamiento de la *Distancia Umbral* a lo largo de la ejecución de una carga de trabajo concreta. Cada punto representa diez millones de ciclos, de forma que se han tomado diez decisiones sobre la *Distancia Umbral* entre uno y otro. Junto a las distintas gráficas se presentan los resultados obtenidos con cada una de las configuraciones.

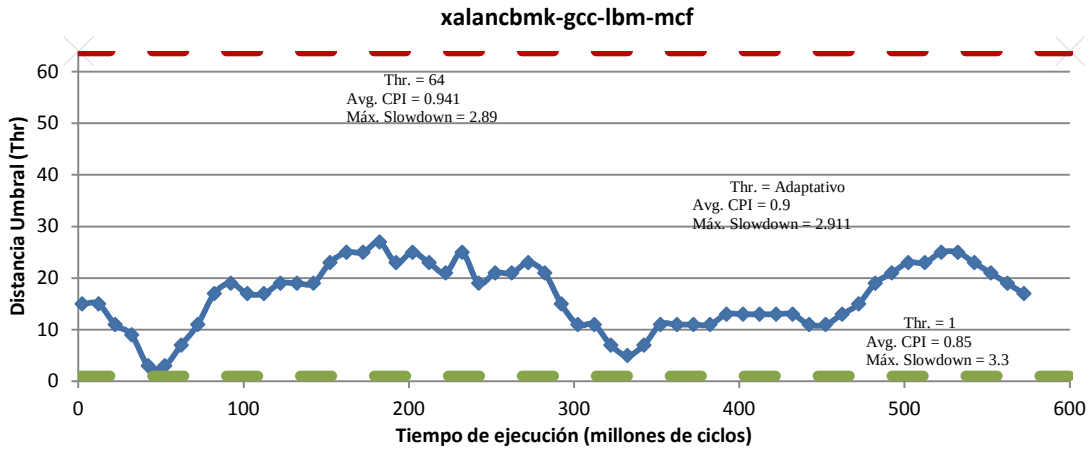

Figura 5-24. Evolución de la Distancia Umbral en la ejecución de una carga de trabajo y los resultados obtenidos frente a los extremos estáticos.

5.9 Implementación

En contraste con FR-FCFS en el que cada cola de un banco puede ser manejada como una cola FIFO, en éste algoritmo es necesario cambiar dinámicamente la prioridad de las peticiones, e igualmente ocurre con todos aquellos algoritmos que requieren de una cola prioritaria en los bancos. Una forma de implementar este tipo de colas prioritarias es sustituyendo la cola FIFO única de cada banco por un conjunto de colas FIFO más pequeñas [98], [99], sin embargo este tipo de solución no escala bien cuando el número de niveles de prioridad es elevado. También es posible el uso de árboles binarios de comparación para establecer la petición más prioritaria [100], [101], aunque esto supone perder el orden FIFO en las peticiones del mismo nivel. Existen soluciones que consiguen ambas ventajas y escalan mejor [102], a costa de una mayor complejidad de hardware y que deben ser evaluadas en cada ciclo de DRAM con el fin de ajustarse dinámicamente. En cualquier caso, con el fin de simplificar el diseño, conviene suponer el menor número de niveles de priorización posible, siendo que el número de bancos y el tamaño de dichas colas no forman parte de la solución.

A la hora de discutir el número de niveles de prioridad necesarios para nuestro algoritmo, hay que tener en cuenta las características del mismo. El nivel más prioritario debe estar reservado para las peticiones marcadas como “alta prioridad” , tales como fetches y peticiones que se consideran en la cabeza del RoB. Los siguientes niveles de prioridad irán en orden creciente, agrupando valores cercanos. Finalmente, el nivel menos prioritario estará reservado a las escrituras en memoria, *writebacks* del último nivel de cache. Por otra parte, el proceso de envejecimiento y reducción de la prioridad de las

peticiones en cada paso del algoritmo, se corresponde con el movimiento de las peticiones adecuadas de un nivel de prioridad al siguiente por pasos. Este tipo de aproximación reduce la precisión del algoritmo, pero permite una implementación sencilla del diseño presentado. En cualquier caso, existen modificaciones en el controlador de memoria igual o más complejas que las propuestas aquí [86], [90], [91]. Y otras propuestas de ordenación por prioridades [103], parecen más complejas de implementar que la descrita.

Vamos a restringir nuestro análisis a 8 niveles de prioridad, siendo el mínimo razonable para obtener un cierto grado de precisión en la toma de decisiones, y en primera instancia consideramos una distribución lineal de la prioridad. En el ejemplo concreto del sistema simulado, siendo el valor máximo esperado igual al tamaño del RoB, 128 entradas, cada nivel de prioridad nuevo equivaldrá a 16 de los no-reducidos.

Es razonable pensar que al disminuir el número de niveles de prioridad disponibles perderemos precisión a la hora de discriminar instrucciones. Afortunadamente, por la experimentación sabemos que los valores más habituales de distancia a la cabeza del ROB una vez hecho el ajuste en el controlador de memoria tienden a estar por debajo de la mitad, siendo más frecuentes los valores más bajos, y donde hay que aplicar un mayor grado de discriminación. Por eso, se plantea una alternativa de división de niveles graduada en función del grado de discriminación necesario, tal y como se muestra en la Figura 5-25.

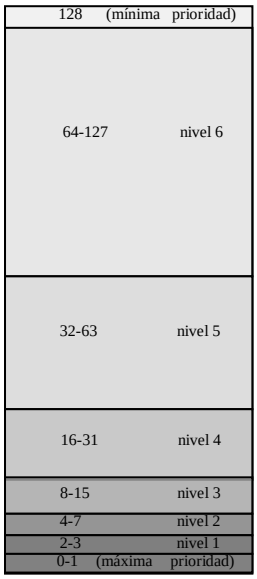

Figura 5-25. Distribución gradual de los niveles de prioridad en función de la distancia a la cabeza del ROB para un procesador con 128 entradas en el Reorder Buffer.

Como se observa en la Figura 5-26 y Figura 5-28, al reducir el número de niveles de prioridad las diferencias no son sustanciales. Se aprecia una ligera disminución del rendimiento frente a una suave mejora del *fairness*, debido probablemente a que al reducir la precisión de los niveles de prioridad, se favorece a las aplicaciones más lentas con valores próximos a otras más rápidas. Si acaso esto es más notable al hacer la distribución gradual, pues la velocidad a la que envejecen las peticiones más alejadas de la cabeza de ROB es mayor, pero compensando ligeramente la perdida de rendimiento por la granularidad obtenida en las distancias más próximas a la cabeza.

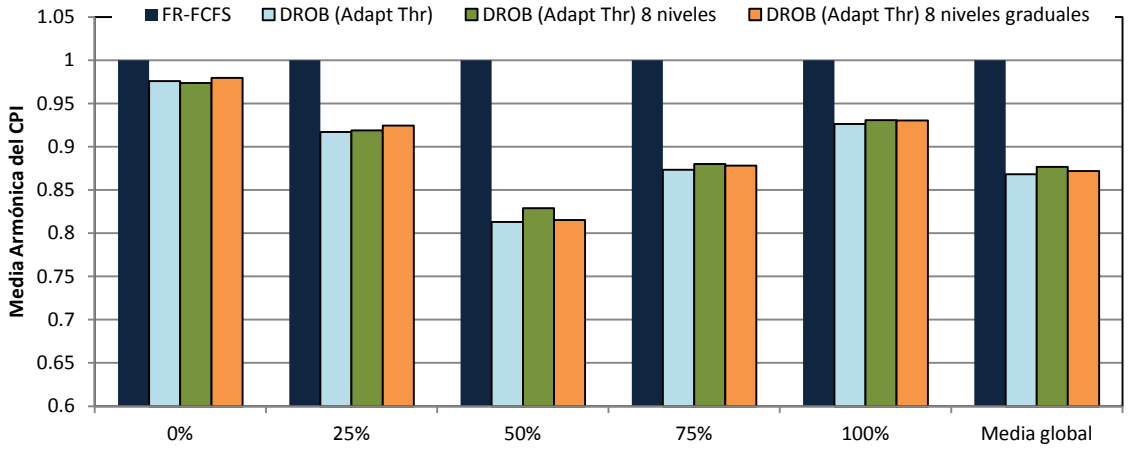

Figura 5-26. Media armónica del CPI para DROB con Distancia Umbral (Thr) adaptativa con 8 niveles de prioridad lineales y graduales.

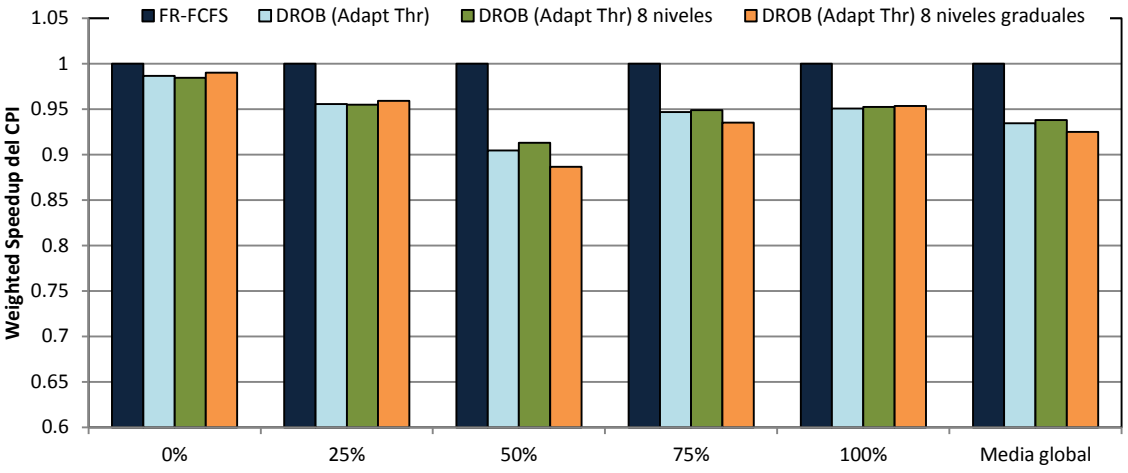

Figura 5-27. Weighted SpeedUp del CPI para DROB con Distancia Umbral (Thr) adaptativa con 8 niveles de prioridad lineales y graduales.

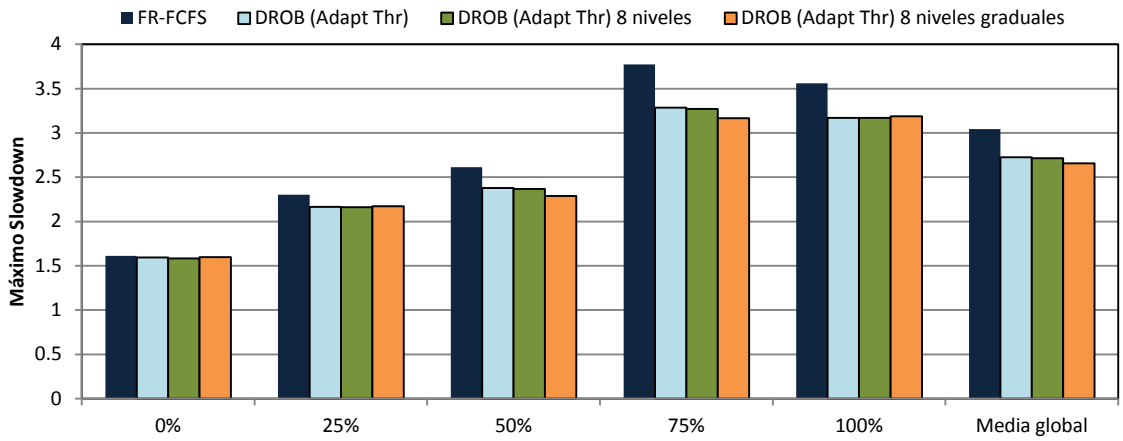

Figura 5-28. Máximo Slowdown del CPI para DROB con Distancia Umbral (Thr) adaptativa con 8 niveles de prioridad lineales y graduales.

5.10 Conclusiones

En este capítulo hemos presentado un algoritmo de arbitraje en el controlador de memoria para sistemas multiprocesadores en chip. Esta solución novedosa se centra en un criterio completamente distinto al de otras soluciones semejantes. Así, mientras otros algoritmos de arbitraje son muy complejos o se centran en inferir el comportamiento del procesador desde la información existente en memoria, DROB usa información proveniente del procesador y asociada a cada petición individualmente.

Las ventajas de DROB son evidentes. Al tratar a cada petición a memoria individualmente, consigue una mayor granularidad en la toma de decisiones. Además, al no depender de un sistema que requiera actualizarse cada cierto número de ciclos y trabajar con información instantánea, su tiempo de reacción es menor que el de otros algoritmos semejantes. Por otra parte, mediante el ajuste de la distancia a la cabeza del ROB, se establece un criterio de aceleración de los procesos más rápidos, que permite un uso más eficiente del ancho de banda a memoria, incrementando el rendimiento global del sistema. Finalmente, el proceso de envejecimiento, sencillo y adaptativo, permite un control sobre el comportamiento del sistema, evitando la inanición y otorgando un equilibrio entre rendimiento y *fairness*.

A lo largo de varios experimentos, se han determinado los parámetros del procesador que mayor impacto tienen a la hora de discriminar las peticiones en memoria. Se ha buscado una forma realista de implementar esta solución y se ha evaluado su

comportamiento mediante una gran variedad de mezclas de aplicaciones, dando como resultado que la propuesta supera en rendimiento y *fairness* a otras propuestas anteriores.

6 Conclusiones y Trabajo Futuro

En esta sección se exponen las principales conclusiones extraídas durante el desarrollo de esta tesis, así como sus contribuciones y las líneas que se van a seguir en el futuro:

6.1 Conclusiones

Se ha hecho un esfuerzo importante por comprender el impacto que tienen los distintos componentes de la jerarquía de memoria en la ejecución de aplicaciones reales dentro de un sistema multiprocesador y cómo pueden influir en su futuro desarrollo. Este proceso ha pasado por un conocimiento profundo de las herramientas de simulación utilizadas, de gran complejidad, obteniendo un dominio sobre las mismas que permite la implementación de nuevas ideas con confianza. El haber trabajado sobre distintos componentes de la jerarquía de memoria ha supuesto un gran esfuerzo en términos de aprendizaje y dominio, pero ha permitido una visión más amplia del problema. La jerarquía de memoria en los CMP es un subsistema complejo y fuertemente entrelazado, de forma que pequeñas modificaciones en alguno de sus componentes pueden encadenar reacciones que hacen difícil evaluar las consecuencias que acarrea una determinada decisión. Del trabajo realizado durante la tesis se han extraído algunas conclusiones como las que se muestran a continuación:

- *En relación a la red de interconexión*: se trata de una parte importante en el diseño de futuros sistemas CMP, tanto más cuanto mayor sea el número de componentes a conectar. No solo es importante que ofrezca buenos resultados en ancho de banda y latencia, sino la forma en la que se interconectan los distintos componentes dentro de la topología. Las simulaciones con aplicaciones reales muestran una disparidad de comportamientos, aunque parece obvio que la importancia de la red de interconexión viene influenciada por el grado de compartición de recursos de la aplicación. Siendo la paralelización de aplicaciones una consecuencia razonable dado el camino que toma la arquitectura de computadores, la red de interconexión tomará un papel más relevante.

- *En relación a la jerarquía de cache*: Ampliar la capacidad de la cache en el interior del chip es una solución directa dado el incremento del nivel de integración que permiten las nuevas tecnologías. Sin embargo, esto lleva asociado problemas como el aumento de la latencia dentro del chip y una necesidad de distribuir la cache de forma eficiente entre los distintos procesadores. Las pruebas y el modelo analítico presentado en la tesis, muestran que es necesario fraccionar la cache en nuevos niveles cuando se llega a una determinada capacidad crítica, de forma que la latencia tenga un crecimiento escalonado. Esta solución se ha comprobado superior a alternativas que aumentan la complejidad del último nivel de cache cuando la distancia entre niveles es alta. Sin embargo, ambas soluciones son compatibles, y una óptima distribución de los recursos entre los distintos procesadores proporciona un incremento sustancial en el rendimiento del sistema, tal y como se demuestra en la propuesta presentada en la tesis.
- *En relación al ancho de banda fuera del chip*: se trata de un recurso escaso en los sistemas multiprocesador, e incluso las soluciones que tratan de paliarlo (como el aumento de la capacidad dentro del chip o el incremento del ancho de banda efectivo) solo retrasan el problema, que debe ser abordado. En situaciones de escasez de ancho de banda, es necesario garantizar un uso equitativo que no dañe el rendimiento y garantice el servicio a todas las aplicaciones que se ejecutan en el chip. En la tesis se plantea una alternativa novedosa que aborda el problema desde un punto de vista distinto a lo presentado en artículos semejantes. Como se puede observar, actuando en un elemento del sistema tan lejano del procesador como es el controlador de memoria, es posible obtener grandes ventajas en rendimiento y *fairness*, en situaciones de escasez de ancho de banda.

6.2 Trabajo Futuro

La jerarquía de memoria es una parte fundamental en los sistemas CMP, y la gran cantidad de componentes y parámetros que lo integran dejan lugar a diferentes líneas de investigación a futuro. Pero además de la resolución de los problemas existentes, como la escasez de ancho de banda a memoria, la incursión de nuevas tecnologías sugiere las siguientes líneas de investigación futuras.

6.2.1 Jerarquía de Cache, Nuevas Tecnologías

Las técnicas de apilamiento vertical y los problemas que llevan asociados abren un campo de exploración interesante dentro de lo desarrollado en la tesis, tanto en la forma en la que se distribuyen los recursos como en el apartado de conexionado. De igual forma, las nuevas tecnologías de integración no volátiles para cache *on-chip* ([8], [9], [104]), suponen un reto desde el punto de vista de viabilidad, bien por efecto de la latencia, tolerancia a fallos o vida útil. En el momento de escribir esta tesis, existen trabajos en proceso de publicación en relación a esta materia.

6.2.2 Arbitraje del Ancho de Banda Fuera del Chip

La solución presentada en esta tesis abre las puertas a futuras líneas de investigación que traten de explotar las características de los procesadores actuales para discriminar las peticiones fuera del chip. Aún cuando en el capítulo correspondiente de la tesis se ha hecho un primer análisis bastante amplio, queda mucho trabajo dado el amplio campo de estudio que supone un procesador de vanguardia. Así, está pendiente de publicación un trabajo relacionado con los *prefetches hardware*, así como un estudio más amplio de las dependencias entre instrucciones y la especulación y su impacto en la criticidad.

6.3 Contribuciones de la Tesis y Publicaciones

A continuación se enumeran las principales contribuciones llevadas a cabo durante la tesis:

- Un sencillo modelo analítico que permite obtener la relación óptima entre niveles de cache en sistemas multiprocesador de memoria compartida.
- Una arquitectura novedosa y eficiente para el último nivel de cache, de fácil implementación y alto grado de escalabilidad.
- Un sistema de directorio incompleto mediante *tags* parciales y control de saturación, que permite almacenar grandes cantidades de bloques en poco espacio a costa de una ligera reducción en su rendimiento.
- Un algoritmo de arbitraje para el controlador de memoria principal, que ordena las peticiones fuera del chip de acuerdo a su criticidad, atendiendo a parámetros propios de un procesador fuera de orden.

Las distintas contribuciones de la tesis han sido publicadas en conferencias y revistas de arquitectura de computadores con revisión por pares, siendo las más relevantes [12], [13], [14], [15], [16], [17] y [18].

7 Bibliografía

[1] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid, "Server Engineering Insights for Large-Scale Online Services," *IEEE Micro*, vol. 30, no. 4, pp. 8–19, Jul. 2010.

[2] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE J. Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct. 1974.

[3] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz, "CPU DB: Recording Microprocessor History," *ACM Queue - Process.*, vol. 10, no. 4, Apr. 2012.

[4] J. Hennessy and D. Patterson, *Computer architecture: a quantitative approach*, 5th ed. 2012.

[5] Tilera, "TILE-Gx8072 ™ Processor," *Specif. Br.*, 2013.

[6] Adapteva, "Epiphany IV - 64-Core Multiprocessor," *Datasheet*, 2013.

[7] Nvidia, "NVIDIA ® Tesla ™," *GPU Comput. Tech. Br.*, 2007.

[8] M. Kund, G. Beitel, C. Pinnow, T. Rohr, J. Schumann, R. Symanczyk, K. Ufert, and G. Muller, "Conductive bridging RAM (CBRAM): an emerging non-volatile memory technology scalable to sub 20nm," in *IEEE InternationalElectron Devices Meeting. IEDM Technical Digest.*, 2005, pp. 754–757.

[9] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano, "A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram," in *IEEE InternationalElectron Devices Meeting. IEDM Technical Digest.*, 2005, pp. 459–462.

[10] ITRS, "2012 Roadmap." [Online]. Available: http://www.itrs.net/Links/2012ITRS/Home2012.htm.

[11] S. Beamer, C. Sun, Y.-J. Kwon, A. Joshi, C. Batten, V. Stojanović, and K. Asanović, "Re-architecting DRAM memory systems with monolithically integrated silicon photonics," in *Proceedings of the 37th annual international symposium on Computer architecture - ISCA*, 2010, pp. 129–140.

[12] J. Merino, V. Puente, P. Prieto, and J. Á. Gregorio, "SP-NUCA: a cost effective dynamic non-uniform cache architecture," *ACM SIGARCH Comput. Archit. News*, vol. 36, no. 2, pp. 64–71, May 2008.

[13] P. Prieto, V. Puente, and J.-A. Gregorio, “Multilevel Cache Modeling for Chip-Multiprocessor Systems,” *IEEE Comput. Archit. Lett.*, vol. 10, no. 2, pp. 49–52, Feb. 2011.

[14] P. Prieto, V. Puente, and J. Gregorio, “Topology-aware CMP design,” in *Interconnection Network Architectures: On-Chip, Multi-Chip (INA-OCMC)*, 2009.

[15] P. Abad, V. Puente, J. A. Gregorio, and P. Prieto, “Rotary router: an efficient architecture for CMP interconnection networks,” in *Proceedings of the 34th annual international symposium on Computer architecture - ISCA*, 2007, vol. 35, no. 2, pp. 116–125.

[16] P. Abad, P. Prieto, V. Puente, and J.-A. Gregorio, “BIXBAR: A low cost solution to support dynamic link reconfiguration in networks on chip,” in *IEEE 30th International Conference on Computer Design (ICCD)*, 2012, no. c, pp. 55–60.

[17] P. Prieto, V. Puente, and J. A. Gregorio, “CMP off-chip bandwidth scheduling guided by instruction criticality,” in *Proceedings of the 27th international ACM conference on International conference on supercomputing - ICS*, 2013, pp. 379–388.

[18] P. Abad, P. Prieto, L. G. Menezo, A. Colaso, V. Puente, and J.-Á. Gregorio, “TOPAZ: An Open-Source Interconnection Network Simulator for Chip Multiprocessors and Supercomputers,” in *IEEE/ACM Sixth International Symposium on Networks-on-Chip*, 2012, pp. 99–106.

[19] M. M. K. Martin, M. D. Hill, and D. J. Sorin, “Why on-chip cache coherence is here to stay,” *Commun. ACM*, vol. 55, no. 7, pp. 78–89, Jul. 2012.

[20] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, “The directory-based cache coherence protocol for the DASH multiprocessor,” in *Proceedings of the 17th annual international symposium on Computer Architecture - ISCA*, 1990, pp. 148–159.

[21] C. Kim, D. Burger, and S. W. Keckler, “An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches,” in *Tenth international conference on architectural support for programming languages and operating systems (ASPLOS-X)*, 2002, vol. 30, no. 5, pp. 211–222.

[22] T. Moscibroda and O. Mutlu, “A case for bufferless routing in on-chip networks,” *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 3, p. 196, Jun. 2009.

[23] P. Abad, V. Puente, and J.-A. Gregorio, “Ligero: A Light but Efficient Router Conceived for Cache-Coherent Chip Multiprocessors,” *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 1–21, Jan. 2013.

[24] H. Dybdahl and P. Stenstrom, “An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors,” in *IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, no. 7491, pp. 2–12.

[25] J. Merino, V. Puente, and J. a Gregorio, “ESP-NUCA: A low-cost adaptive Non-Uniform Cache Architecture,” in *The Sixteenth International Symposium on High-Performance Computer Architecture - HPCA*, 2010, pp. 188–197.

[26] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, “Distance associativity for high-performance energy-efficient non-uniform cache architectures,” in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003, pp. 55–66.

[27] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr., and J. Emer, “Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies,” in *43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 151–162.

[28] I. Corporation, *Intel ® Core ™ i7 Processor Family for LGA2011 Socket*, vol. 1, no. September. 2013.

[29] AMD, “AMD Opteron ™ 6200 Series Processor,” 2011.

[30] L. Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs,” *Comput. IEEE Trans.*, vol. C, no. 9, pp. 690–691, 1979.

[31] P. S. Sindhu, J.-M. Frailong, and M. Cekleov, “Formal Specification of Memory Models,” Palo Alto, 1991.

[32] M. Dubois, C. Scheurich, and F. Briggs, “Memory access buffering in multiprocessors,” in *Proceedings of the 13th annual international symposium on Computer architecture - ISCA*, 1986, pp. 434–442.

[33] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, “Memory consistency and event ordering in scalable shared-memory multiprocessors,” in *Proceedings of the 17th annual international symposium on Computer Architecture - ISCA*, 1990, pp. 15–26.

[34] P. Sweazey and A. J. Smith, “A class of compatible cache consistency protocols and their support,” in *Proceedings of the 13th annual international symposium on Computer architecture - ISCA*, 1986, pp. 414–423.

[35] M. M. K. Martin, M. D. Hill, and D. a. Wood, “Token Coherence: decoupling performance and correctness,” in *Proceedings 30th Annual International Symposium on Computer Architecture - ISCA.*, 2003, pp. 182–193.

[36] L. a. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, “Piranha: a scalable architecture based on single-chip multiprocessing,” in *Proceedings of 27th International Symposium on Computer Architecture - ISCA*, 2000, pp. 282–293.

[37] “OpenSPARC T2 System-on-Chip (SoC) microarchitecture specification,” 2008.

[38] D. Lenoski, J. Laudon, and K. Gharachorloo, “The stanford dash multiprocessor,” *IEEE Comput.*, vol. 25, no. 3, pp. 63–79, Mar. 1992.

[39] A. Gupta, W. Weber, and T. Mowry, “Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes,” in *International Conference on Parallel Processing*, 1990, pp. 312–321.

[40] C. . Tang, “Cache system design in the tightly coupled multiprocessor system,” in *Proceedings of the AFIPS national computer conference and exposition*, 1976, pp. 749–754.

[41] L. M. Censier and P. Feautrier, “A New Solution to Coherence Problems in Multicache Systems,” *IEEE Trans. Comput.*, vol. C–27, no. 12, pp. 1112–1118, Dec. 1978.

[42] Y. Aidong, L. Jun, L. Jun, and H. Yongqin, “Structure research in a sparse directory co-located with last-level cache,” in *2010 3rd International Conference on Computer Science and Information Technology*, 2010, pp. 56–62.

[43] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, “Memory access scheduling,” in *Proceedings of the 27th annual international symposium on Computer architecture - ISCA*, 2000, pp. 128–138.

[44] O. Mutlu and T. Moscibroda, “Stall-time fair memory access scheduling for chip multiprocessors,” in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007, pp. 146–160.

[45] H. Jin, M. Frumkin, and J. Yan, “The OpenMP implementation of NAS parallel benchmarks and its performance,” NASA Ames Research Center, 1999.

[46] A. Alameldeen, C. Mauer, M. Xu, P. J. Harper, M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood, “Evaluating non-deterministic multi-threaded commercial workloads,” in *Proceedings of the Fifth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, 2002, pp. 30–38.

[47] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC benchmark suite: characterization and architectural implications,” in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques - PACT*, 2008, pp. 72–81.

[48] J. L. Henning, “SPEC CPU2006 benchmark descriptions,” *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[49] P. Barford and M. Crovella, “Generating representative Web workloads for network and server performance evaluation,” *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 1, pp. 151–160, Jun. 1998.

[50] L. Barroso, K. Gharachorloo, and E. Bugnion, “Memory system characterization of commercial workloads,” in *25th annual international symposium on Computer architecture - ISCA*, 1998, pp. 3–14.

[51] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, “Simics: A full system simulation platform,” *IEEE Comput.*, vol. 35, no. 2, pp. 50–58, 2002.

[52] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, “Multifacet’s general execution-driven multiprocessor simulator (GEMS) toolset,” *ACM SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, Nov. 2005.

[53] C. J. Mauer, M. D. Hill, and D. a. Wood, “Full-system timing-first simulation,” in *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2002, pp. 108–116.

[54] D. Wood, G. Gibson, and R. Katz, “Verifying a multiprocessor cache controller using random test generation,” *IEEE Des. Test Comput.*, vol. 7, no. 4, pp. 13–25, Aug. 1990.

[55] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, “Cacti 6.0: A tool to model large caches,” *HP Lab.*, 2009.

[56] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. D. Hill, D. A. Wood, and D. J. Sorin, “Simulating a $2M commercial server on a $2K PC,” *IEEE Comput.*, vol. 36, no. 2, pp. 50–57, Feb. 2003.

[57] A. R. Alameldeen and D. A. Wood, “IPC Considered Harmful for Multiprocessor Workloads,” *IEEE Micro*, vol. 26, no. 4, pp. 8–17, Jul. 2006.

[58] A. Snavely and D. M. Tullsen, “Symbiotic Jobscheduling for a Simultaneous Multithreading Processor,” in *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems - ASPLOS*, 2000, pp. 234–244.

[59] A. D. Chen, D. Freeman, and B. H. Leitao, *IBM Power 770 and 780 Technical Overview and Introduction*. IBM Redpaper, 2013.

[60] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson, and M. H. Lipasti, “Achieving predictable performance through better memory controller placement in many-core CMPs,” in *Proceedings of the 36th annual international symposium on Computer architecture - ISCA*, 2009, pp. 451–461.

[61] S. Borkar, “3D integration for energy efficient system design,” in *Design Automation Conference (DAC), 2011 48th …*, 2011, pp. 214–219.

[62] P. P. Gelsinger, “Intel Architecture Press Briefing Today’s News Intel Technology : Delivering on the Promise,” 2008.

[63] M. Monchiero, R. Canal, and A. Gonzalez, “Power/Performance/Thermal Design-Space Exploration for Multicore Architectures,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 5, pp. 666–681, May 2008.

[64] B. M. Beckmann and D. a. Wood, “TLC: transmission line caches,” in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*, 2003, pp. 43–54.

[65] W. J. Dally, “Performance analysis of k-ary_n-cube Interconnection Networks,” *Trans. Comput.*, vol. 39, no. 6, pp. 775–785, 1990.

[66] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, “Scaling the bandwidth wall: challenges in and avenues for CMP scaling,” in *Proceedings of the 36th annual international symposium on Computer architecture - ISCA*, 2009, pp. 371–382.

[67] C. K. Chow, “On Optimization of Storage Hierarchies,” *IBM J. Res. Dev.*, vol. 18, no. 3, pp. 194–203, May 1974.

[68] C. K. Chow, “Determination of Cache’s Capacity and its Matching Storage Hierarchy,” *IEEE Trans. Comput.*, vol. C–25, no. 2, pp. 157–164, Feb. 1976.

[69] S. Przybylski, M. Horowitz, and J. Hennessy, “Performance tradeoffs in cache design,” in *15th Annual International Symposium on Computer Architecture - ISCA*, 1988, pp. 290–298.

[70] S. Przybylski, M. Horowitz, and J. Hennessy, “Characteristics of performance-optimal multi-level cache hierarchies,” in *Proceedings of the 16th annual international symposium on Computer architecture - ISCA*, 1989, pp. 114–121.

[71] A. Hartstein, V. Srinivasan, T. R. Puzak, and P. G. Emma, “On the Nature of Cache Miss Behavior: Is It sqrt2?,” *J. Instr. Parallelism*, vol. 10, pp. 1–22, 2008.

[72] M. H. MacDougall, “Instruction-Level Program and Processor Modeling,” *IEEE Comput.*, vol. 17, no. 7, pp. 14–24, Jul. 1984.

[73] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, “A NUCA Substrate for Flexible CMP Cache Sharing,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 8, pp. 1028–1040, Aug. 2007.

[74] C. Bienia and K. Li, “Benchmarking modern multiprocessors,” Princeton University, Princeton, NJ, 2011.

[75] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar, “An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS,” *IEEE J. Solid-State Circuits*, vol. 43, no. 1, pp. 29–41, Jan. 2008.

[76] D. Wentzlaff, P. Griffin, and H. Hoffmann, “On-chip interconnection architecture of the tile processor,” *IEEE Micro*, vol. 27, no. 5, pp. 15–31, 2007.

[77] A. N. Eden and T. Mudge, “The YAGS branch prediction scheme,” in *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture - MICRO*, 1998, pp. 69–77.

[78] B. Beckmann and D. Wood, “Managing wire delay in large chip-multiprocessor caches,” in *Proceedings 37th Annual ACM/IEEE International Symposium on Microarchitecture - MICRO*, 2004, pp. 319–330.

[79] Z. Guz, I. Keidar, A. Kolodny, and U. C. Weiser, “Utilizing shared data in chip multiprocessors with the nahalal architecture,” in *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures - SPAA*, 2008, pp. 1–10.

[80] B. Beckmann, M. Marty, and D. Wood, “ASR: Adaptive Selective Replication for CMP Caches,” in *Proceedings 39th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO*, 2006, pp. 443–454.

[81] J. Chang and G. S. Sohi, “Cooperative Caching for Chip Multiprocessors,” in *33rd International Symposium on Computer Architecture - ISCA*, 2006, pp. 264–276.

[82] M. Zhang and K. Asanovic, “Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors,” in *32nd International Symposium on Computer Architecture - ISCA*, 2005, pp. 336–345.

[83] G. E. Suh, L. Rudolph, and S. Devadas, “Dynamic Cache Partitioning for Simultaneous Multithreading Systems,” in *International Conference on Parallel and Distributed Computing and Systems*, 2001, pp. 116–127.

[84] M. Qureshi and Y. Patt, “Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches,” in *Proceedings 39th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO*, 2006, pp. 423–432.

[85] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, “Parallel application memory scheduling,” in *Proceedings 44th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO*, 2011, pp. 362–373.

[86] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, “Self-Optimizing Memory Controllers: A Reinforcement Learning Approach,” in *Proceedings 35th International Symposium on Computer Architecture - ISCA*, 2008, pp. 39–50.

[87] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, “Bottleneck identification and scheduling in multithreaded applications,” in *Proceedings of the 17th international conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, 2012, pp. 223–234.

[88] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, “Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior,” in *Proceedings 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 65–76.

[89] O. Mutlu and T. Moscibroda, “Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems,” in *Proceedings 35th International Symposium on Computer Architecture - ISCA*, 2008, pp. 63–74.

[90] O. Mutlu and M. Harchol-Balter, “ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers,” in *Proceedings 16th International Symposium on High-Performance Computer Architecture - HPCA*, 2010, pp. 37–48.

[91] J. Mukundan and J. F. Martinez, “MORSE: Multi-objective reconfigurable self-optimizing memory scheduler,” in *IEEE 18th International Symposium on High-Performance Computer Architecture - HPCA*, 2012, pp. 65–76.

[92] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, “Reducing memory interference in multicore systems via application-aware memory channel partitioning,” in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO*, 2011, pp. 374–385.

[93] F. Liu, X. Jiang, and Y. Solihin, “Understanding how off-chip memory bandwidth partitioning in Chip Multiprocessors affects system performance,” in *Proceedings 16th International Symposium on High-Performance Computer Architecture - HPCA*, 2010, pp. 49–60.

[94] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, vol. 9, no. 5. Cambridge: MIT Press, 1998.

[95] S. Ghose, H. Lee, and J. F. Martínez, “Improving memory scheduling via processor-side load criticality information,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture - ISCA*, 2013, pp. 84–95.

[96] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John, “The virtual write queue: Coordinating DRAM and Last-level Cache Policies,” in *Proceedings of the 37th annual international symposium on Computer architecture - ISCA ’10*, 2010, p. 72.

[97] R. Kalla, B. Sinharoy, and J. M. Tendler, “IBM power5 chip: a dual-core multithreaded processor,” *IEEE Micro*, vol. 24, no. 2, pp. 40–47, Mar. 2004.

[98] R. Brown, “Calendar queues: a fast 0(1) priority queue implementation for the simulation event set problem,” *Commun. ACM*, vol. 31, no. 10, pp. 1220–1227, Oct. 1988.

[99] H. J. Chao, “A novel architecture for queue management in ATM networks,” in *IEEE Global Telecommunications Conference - GLOBECOM*, 1991, pp. 1611–1618.

[100] D. Picker, M. B. Bendak, and R. D. Fellman, “A VLSI priority packet queue with overwrite and inheritance,” in *Proceedings IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1994, pp. 551–555.

[101] J. Rexford, J. Hall, and K. G. Shin, “A router architecture for real-time point-to-point networks,” *ACM SIGARCH Comput. Archit. News*, vol. 24, no. 2, pp. 237–246, May 1996.

[102] S. Moon, K. Shin, and J. Rexford, “Scalable hardware priority queue architectures for high-speed packet switches,” in *Proceedings 3rd IEEE Real-Time Technology and Applications Symposium*, 2000, pp. 203–212.

[103] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, “Prefetch-Aware DRAM Controllers,” in *2008 41st IEEE/ACM International Symposium on Microarchitecture*, 2008, pp. 200–209.

[104] B. B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, O. Mutlu, and D. Burger, “Phase-Change Technology and the Future of Main Memory,” *IEEE Micro*, vol. 30, no. 1, pp. 131–141, Jan. 2010.