Universidad Zaragoza
1542

Trabajo Fin de Máster

Selección de contenidos basada en reuso para caches compartidas en exclusión

Autor

Javier Díaz Maag

Director

Pablo Ibáñez Marín

Escuela de Ingeniería y Arquitectura

2014

Selección de contenidos basada en reuso para caches compartidas en exclusión

RESUMEN

En los últimos años, los sistemas multiprocesador con memoria compartida que incluyen varios procesadores en un mismo chip o circuito integrado están muy extendidos. Estos sistemas suelen incluir una cache compartida de último nivel (SLLC). Recientes estudios revelan que el flujo de referencias que llega a la SLLC muestra poca localidad temporal. Sin embargo, muestra localidad de reuso, es decir, los bloques reusados (referenciados varias veces) tienen más probabilidad de ser referenciados en un futuro. Esto provoca que, si se realiza una gestión convencional, basada en la localidad temporal, el uso de la cache es ineficiente, desaprovechándose la mayoría de su contenido. Existe un número importante de propuestas que tratan este problema para caches inclusivas, pero pocas que se centran en caches exclusivas. Dichas caches se encuentran ya en el mercado, y es previsible que se utilicen más en el futuro.

En este trabajo se propone un nuevo mecanismo de selección de contenidos para caches exclusivas que aprovecha la localidad de reuso que presentan los accesos a la SLLC. Consiste en incluir un elemento denominado Detector de Reuso entre cada cache L2 y la SLLC, al que se dirigen todos los bloques expulsados de la cache L2. Su misión es detectar bloques sin reuso para evitar que sean insertados en la SLLC, realizando *bypass* de los mismos.

Esta propuesta, junto con otras publicadas recientemente, se implementa en un simulador de sistemas completos que modela de forma detallada un sistema con 8 procesadores en chip y su jerarquía de memoria. Para evaluar la propuesta y compararla con otras similares, se simulan ciclo a ciclo un conjunto de cargas multiprogramadas formadas por programas de prueba reconocidos en la comunidad científica.

Configurado adecuadamente, el Detector de Reuso evita la inserción de bloques poco útiles en la SLLC, facilitando que se mantengan los más reusados. Los resultados muestran que ello permite incrementar el rendimiento, por encima de otras propuestas recientes como CHAR o la Reuse Cache. Por ejemplo, para una configuración del Detector de Reuso balanceada entre coste y prestaciones, se obtiene un 8,5% de reducción de la tasa de fallos de la SLLC y un incremento del IPC de un 2,5%, frente a un sistema base con política de reemplazo TC-AGE. Este incremento de prestaciones se distribuye de forma equitativa, ya que aparece en un 90% de las mezclas de programas simuladas y sin perjudicar de forma injusta a ningún programa en particular.

Tabla de contenidos

1 Introducción

En los últimos años, los sistemas multiprocesador con memoria compartida que incluyen varios procesadores en un mismo chip o circuito integrado están muy extendidos. Su presencia es mayoritaria en el mercado tanto en servidores de alto rendimiento como en sistemas de sobremesa, dispositivos móviles y sistemas embebidos. En ellos, el diseño habitual (ver Figura 1) es el de una jerarquía de menoría multinivel, que incluye una cache compartida de último nivel o SLLC (acrónimo de Shared Last Level Cache). Ésta es crítica en términos de coste, prestaciones y consumo. En coste, porque suele ocupar dentro del chip una superficie comparable a la de varios procesadores. En prestaciones y consumo, porque es el último recurso existente antes acceder a la memoria DRAM que, situada fuera del chip, es inferior en prestaciones y consume más energía.

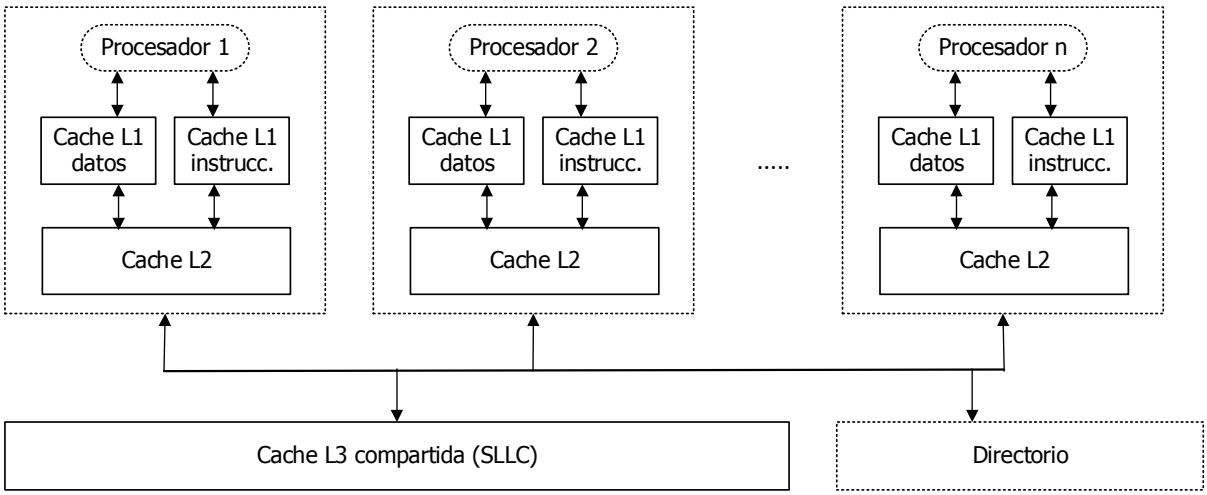


Figura 1: Esquema de jerarquía de memoria en sistemas multiprocesador on-chip.

Por desgracia, varios estudios ponen de manifiesto que los diseños tradicionales no resultan eficientes en su implementación en SLLCs, ya que desaprovechan una fracción mayoritaria del espacio de almacenamiento. Esto es así porque almacenan bloques muertos, es decir, bloques que no van a ser accedidos nunca antes de su expulsión. Frecuentemente, los bloques están muertos en cuanto llegan a la SLLC [1] [2] [3]. La razón de que esto ocurra es que las caches L1 y L2 aprovechan la mayor parte de la localidad temporal, por lo que resulta *filtrada* antes de llegar a la SLLC [4] [5]. Con el objetivo de evitar este efecto, e incrementar la tasa de acierto de la SLLC, se han publicado en los últimos años varias propuestas de modificaciones en la política de inserción y reemplazo en la SLLC (ver sección 4). La mayor parte de los trabajos se refieren a caches inclusivas o no inclusivas, y sólo un grupo reducido [3] [6] se enfoca en una SLLC exclusiva [7].

Al contrario que una cache inclusiva tradicional, una SLLCs exclusiva no almacena los datos que ya están presentes en los niveles interiores de cache. Ello permite mejorar el rendimiento de la SLLC, ya que supone un espacio adicional disponible. Hoy en día, hay ya fabricantes comerciales que incluyen una SSLC exclusiva, o parcialmente exclusiva, en su diseño de microprocesador [8]. A medida que el

número de procesadores dentro del chip crece, también lo hace la cantidad de cache en los niveles interiores y, por lo tanto, la diferencia en rendimiento entre una SLLC exclusiva y una inclusiva. En un futuro que se prevé de muchos procesadores (many-core) dentro del chip, cientos o miles, y SLLCs no mucho mayores que las existentes hoy en día [9], utilizar una cache inclusiva será más ineficiente aún. Por lo tanto, es de esperar que la utilidad de las SLLC exclusivas crezca en un futuro, salvo cambios en el diseño básico de la jerarquía de memoria.

Este trabajo se centra en mejorar la eficiencia y el rendimiento de una SLLC exclusiva en un entorno multiprocesador on-chip. En concreto, se presenta un nuevo mecanismo de selección de contenidos para caches exclusivas que aprovecha la *localidad de reuso* que presentan los accesos a la SLLC [10]. Dicha localidad consiste en que, cuando se referencia un bloque dos veces (se reusa), es probable que dicha dirección se referencie en un futuro cercano. El mecanismo propuesto persigue que sólo se encuentren en la SLLC aquellos bloques que tienen reuso en dicho nivel de cache, es decir, aquellos bloques que son solicitados más de una vez desde las L2 privadas a la SLLC. Para ello, un elemento denominado Detector de Reuso detecta qué bloques expulsados de las L2 no presentan reuso, y evita que sean insertados en la SLLC, realizando *bypass* de los mismos.

Se evalúa esta propuesta simulando un sistema con 8 procesadores en un chip que ejecuta una serie de cargas multiprogramadas. El Detector de Reuso evita la inserción de bloques poco útiles en la SLLC, facilitando que se mantengan los más reusados. Ello permite incrementar el rendimiento, por encima de otras propuestas recientes.

El trabajo está estructurado como sigue. La sección 2 muestra evidencia experimental de la presencia mayoritaria de bloques muertos en una SLLC exclusiva, y de que la localidad de reuso es una propiedad presente en los accesos a la misma. La sección 3 explica en detalle la propuesta del Detector de Reuso. La sección 4 analiza los trabajos relacionados, y los compara con esta nueva propuesta. La sección 5 detalla la metodología empleada, incluyendo el entorno de experimentación y la configuración de los sistemas simulados. La sección 6 presenta los experimentos y analiza los resultados, comparándolos con dos propuestas actuales, que son CHAR [6] y una versión en exclusión de la Reuse Cache [11]. Por último, en la sección 7 se extraen conclusiones.

Adicionalmente, tras las referencias se han incluido dos anexos. El anexo A es breve repaso del contexto en que se realiza este trabajo, y el anexo B detalla el plan de trabajo que se ha seguido, con una descripción de sus fases.

2 Motivación

En esta sección, se analiza el comportamiento de un conjunto de 100 mezclas multiprogramadas creadas combinando de forma aleatoria los 29 de programas de la suite SPEC CPU 2006, ejecutando sobre un sistema con 8 procesadores, caches privadas y una SLLC exclusiva de 8 MB (ver detalles del entorno en la sección 4). El objetivo es comprobar que el la SLLC es poco eficiente en el

aprovechamiento de su espacio, y que una selección de los contenidos basada en el reuso de cada bloque puede resultar de utilidad para incrementar su eficiencia.

La Figura 2 muestra, para las 100 mezclas mencionadas, la fracción de bloques contenidos en la SLLC que de media están vivos a lo largo de la ejecución de la mezcla. Se entiende que un bloque presente en la SLLC está vivo en un determinado momento si experimentará al menos un acceso antes de ser expulsado, por lo cual resulta útil mantenerlo en la cache. Por el contrario, se entiende que un bloque está muerto en un determinado momento cuando va a ser expulsado en un futuro sin haber recibido ningún acceso, es decir, sin resultar de utilidad adicional. En cada mezcla, se toma información de los bloques vivos cada millón de ciclos, y se muestra en el gráfico la media de los valores.

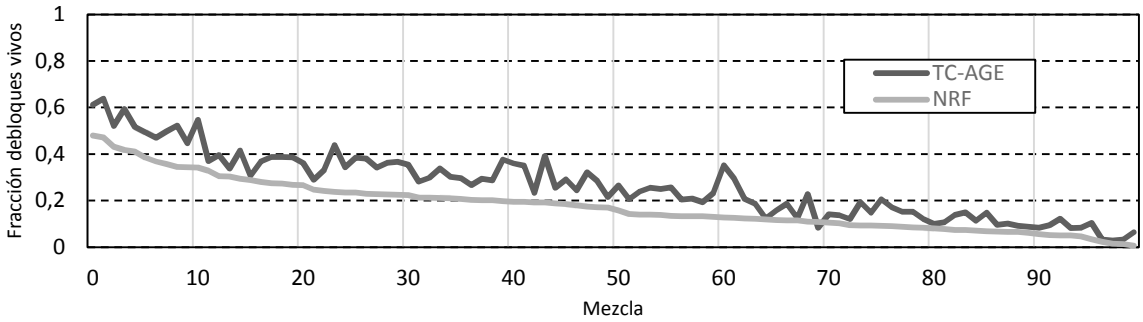


Figura 2: Fracción media de bloques de una SLLC exclusiva de 8 MB que están vivos durante la ejecución, para 100 mezclas multiprogramadas de SPEC CPU2006 (ver sección 4). Se muestran valores para las políticas de reemplazo NRF y TC-AGE. Las mezclas están ordenadas de mayor a menor fracción de bloques vivos con NRF.

La información se representa para dos políticas de reemplazo diferentes, NRF y TC-AGE. La política NRF (acrónimo de Not Recently Filled) es análoga a la NRU (Not Recently Used) en caches inclusivas. NRF utiliza un único bit de reemplazo por bloque, y selecciona como víctima a un bloque aleatorio que no haya sido recientemente insertado, es decir, no tenga el bit de reemplazo a uno. El bit de reemplazo se pone a uno cuando el bloque se inserta en la SLLC, proveniente de una expulsión de una cache L2 privada. Si todos los bits del conjunto están a 1, se cambian todos a 0 salvo el del bloque recién insertado. La política TC-AGE [3] para caches exclusivas es análoga a SRRIP [5] para caches inclusivas, y utiliza 2 bits por línea de cache para almacenar la edad asignada al bloque. TC-AGE selecciona como víctima un bloque aleatorio de entre aquellos del conjunto que tengan la menor edad. La edad se asigna cuando se inserta el bloque en la SLLC. Si el bloque ya ha recibido anteriormente algún acierto en la SLLC, se le asigna la edad 3, y si no se le asigna la edad 1. La información de acierto en la SLLC se guarda en un bit adicional en la cache L2 privada, y se envía a la SLLC junto con el bloque. Cuando, tras un reemplazo, no queda ningún bloque con edad 0 en el conjunto, se resta 1 a la edad de todos los bloques. Es decir, TC-AGE asigna mayor edad, y por lo tanto menor probabilidad de reemplazo, a los bloques que hayan demostrado ser útiles en la SLLC, ya que han sido reusados.

Como puede apreciarse en la figura, la fracción de bloques vivos con NRU varía entre un 1% y un 48%, en función de la mezcla, siendo la media de un 18%. La política TC-AGE demuestra su efectividad

incrementando los bloques vivos en 99 de las 100 mezclas, alcanzando una media del 26%. Estos valores demuestran que la SLLC no utiliza eficientemente su espacio de almacenamiento, ya que la mayoría de los bloques están muertos incluso usando las mejoras políticas de reemplazo propuestas en la literatura.

La Figura 3a muestra, para cada programa que participa en la carga de trabajo mencionada, la media de la distribución de bloques reemplazados de la SLLC, en función del número de accesos que cada bloque ha registrado durante su estancia en la cache. Cada bloque se clasifica según haya recibido un solo acceso (U), dos accesos (R, reuso), o más de dos accesos (M, múltiple reuso). Como política de reemplazo de la SLLC se ha usado TC-AGE.

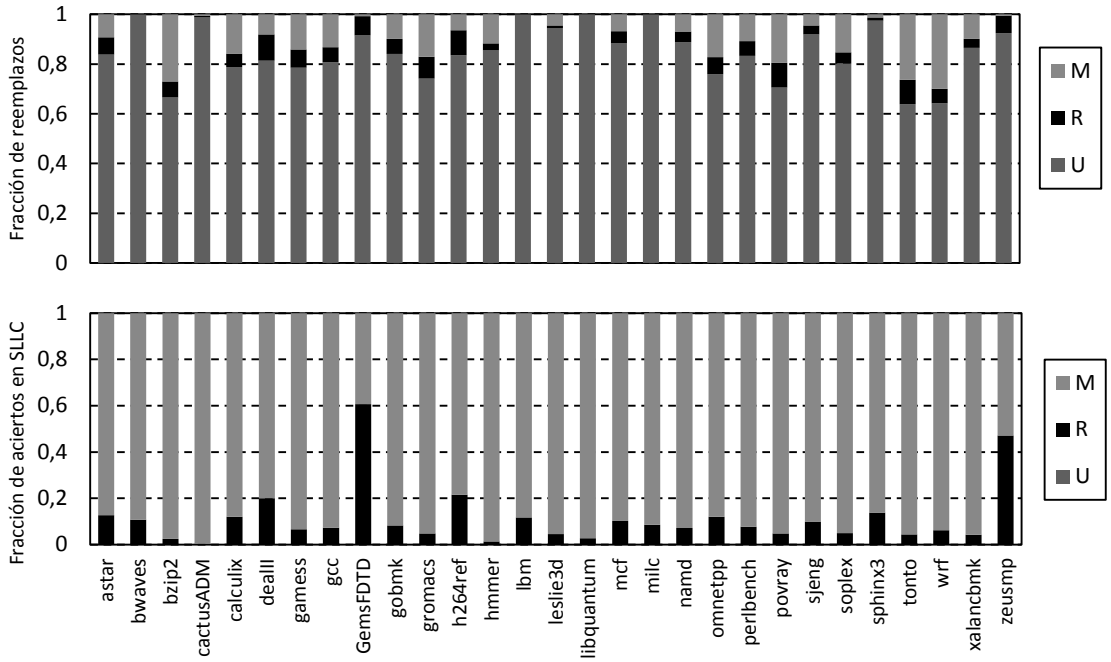


Figura 3: Arriba (a), distribución media de reemplazos de bloques de una SLLC exclusiva de 8 MB con TC-AGE, para 100 mezclas multiprogramadas de SPEC CPU2006 (ver sección 4), categorizadas según el número de accesos que recibe el bloque en la SLLC antes de su reemplazo: (U) un uso, (R) reuso – dos usos, (M) múltiple reuso – tres o más usos. Abajo (b), distribución media de aciertos en la SLLC según las mismas categorías.

Incluso con la política TC-AGE, la gráfica muestra que, en función del programa, entre un 64% y un 99% de los bloques reemplazados de la SLLC tienen un único uso antes de su reemplazo, con una media del 85%. Este uso es el que les insertó en la cache, por lo que su estancia ha sido inútil. Esto es debido a que, en este nivel de la jerarquía de memoria, la localidad temporal de los programas es escasa, ya que ha sido *filtrada* por las caches privadas. Todos estos bloques son buenos candidatos para no ser siquiera almacenados en la SLLC, es decir, para hacer bypass cuando son expulsados de la cache L2.

La Figura 3b muestra, para cada programa, la media de la distribución de los aciertos en SLLC, en función del tipo de bloque sobre el que se producen (U, R, M). La mayor parte de los aciertos de la

SLLC se producen en bloques con múltiple reuso (M), es decir, a nivel de SLLC los programas muestran *localidad de reuso*. En 27 de los 29 programas, entre un 78% y un 99% de los aciertos son en dichos bloques, mientras que en *zeusmp* y *GemsFDTD* son del 59% y del 33% respectivamente.

Estos resultados nos indican que una política de selección de contenidos de la SLLC que sólo almacene los bloques que han demostrado reuso (al menos dos accesos) conseguiría guardar la pequeña porción de bloques con múltiples reusos (M en Figura 3a) que producen la mayoría de los aciertos (M en Figura 3b). Además, esta política impediría la entrada en SLLC de la gran porción de bloques que no se llegan a reusar (U en Figura 3a), lo que disminuiría la probabilidad de que los bloques M fuesen reemplazados.

3 Diseño e implementación de la propuesta

3.1 Diseño general

El diseño de partida es el de una SLLC cuyos contenidos se encuentran en exclusión con los contenidos de las caches privadas de cada procesador. Para mantener la coherencia en la jerarquía de memoria, existe también un directorio que mantiene, para cada bloque presente en dicha jerarquía, tanto su estado como la información precisa de dónde se encuentra, que puede ser uno o varios procesadores y/o la SLLC.

En dicho diseño, los bloques que llegan de memoria se envían directamente a la cache L2 solicitante. Eventualmente, el bloque es expulsado de ella y se envía para su almacenaje en la SLLC. Desde ahí, o bien el bloque es solicitado de nuevo desde alguna cache L2, siendo entonces enviado y desalojado de la SLLC, o bien el bloque es reemplazado por otro que necesita espacio para su inserción.

Sobre este diseño, la propuesta es incluir un elemento intermedio a la salida de cada cache L2, entre cada una de éstas y la SLLC. A este elemento, que llamaremos Detector de Reuso (DR), se dirigen todos los bloques expulsados de la cache L2. Al estar el DR situado fuera del camino de petición de bloques a la SLLC, el retardo que añade no afecta a los tiempos de acierto o fallo de la SLLC. En cambio, afecta al tiempo que necesita un bloque desde su expulsión de la cache L2 a su eventual llegada a la SLLC. La Figura 4 muestra un esquema de este diseño.

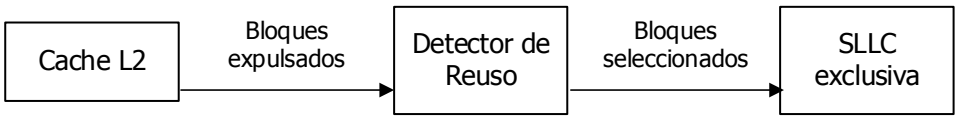


Figura 4: Esquema del diseño general con Detector de Reuso.

El DR decide entre enviar el bloque a la SLLC o no hacerlo, es decir, hacer bypass del mismo. La decisión se basa en las categorías mostradas en la sección anterior: Si un bloque expulsado se clasifica en la categoría U, con un único uso, se hace bypass. Si se clasifica en la categoría R o M, con uno o más reusos, no se hace bypass.

Ni la SLLC ni el directorio requieren modificaciones estructurales para adaptarse al mecanismo del DR. Sí que requieren cambios en su protocolo de coherencia y lógica de control, para tener en cuenta el posible bypass. Si bien en este trabajo se utiliza TC-AGE como política de reemplazo en la SLLC, es posible implementar el DR con cualquier política de reemplazo.

3.2 El Detector de Reuso

El DR está compuesto por un buffer que almacena direcciones de bloques y por su lógica de gestión. La misión del buffer es almacenar las direcciones de los bloques que llegan al detector, con el fin de identificar si es la primera vez que dicho bloque se expulsa de la cache L2 o si ya ha sido expulsado anteriormente. Una primera expulsión implica que no hay reuso por parte de la cache L2, mientras que las siguientes implican que sí lo hay.

El buffer está organizado de forma asociativa por conjuntos, y sus características (tamaño, asociatividad, política de reemplazo) son variables de diseño. De entre ellas, la fundamental es el tamaño. Para que el DR sea efectivo, éste ha de ser lo suficientemente grande como para almacenar una parte significativa de las direcciones de los bloques entre su uso y su reuso. Se define la *cobertura de expulsión* como el espacio de memoria ocupado en conjunto por los bloques expulsados para los que el DR puede hacer seguimiento en un determinado momento. Por ejemplo, si el DR puede hace seguimiento de 1024 bloques de memoria, y cada bloque es de 64B, su cobertura de expulsión es de 64 KB.

El DR utiliza también, para detectar el reuso, el dato de si el bloque fue enviado a la cache L2 desde memoria o proviene de un acierto en la SLLC. La proveniencia de la SLLC indica también reuso. Esta información se almacena en la cache L2, en un bit adicional existente en cada línea. Cuando el bloque es expulsado de la cache L2, este bit se incluye en el mensaje que se dirige al DR. Esta misma información es utilizada por la política de reemplazo TC-AGE, por lo que no comporta necesidades de espacio adicionales si el DR se implementa sobre un sistema que ya esté utilizando TC-AGE.

3.3 Funcionamiento del Detector de Reuso

La Figura 5 muestra un esquema del funcionamiento del DR. Sobre esta figura, se detalla a continuación las operaciones que realiza.

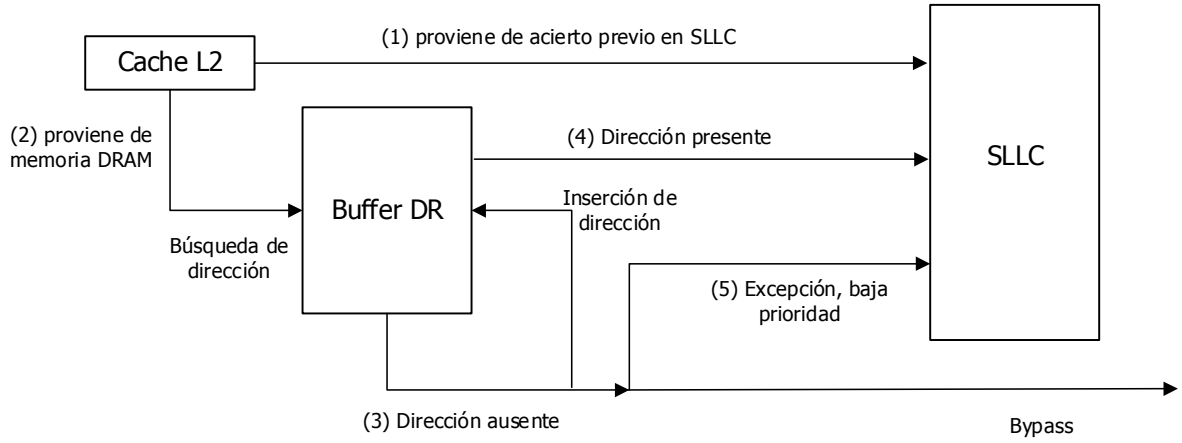


Figura 5: Esquema de funcionamiento del Detector de Reuso.

Cuando llega un bloque expulsado de la cache L2 llega al DR, se inspecciona primero el bit que indica si el bloque provenía originalmente de la SLLC. Si es así, esto indica que el bloque tiene reuso a ese nivel, por lo que se marca para su almacenamiento en la SLLC (1).

Si el bloque provenía de memoria DRAM (2), se busca su dirección en el buffer. Si la dirección del bloque no está presente, esto indica que es la primera vez que se expulsa de la L2, por lo que no muestra reuso. El bloque se marca para bypass, y su dirección se inserta en el buffer (3). Si el bloque sí está ya presente en el buffer, esto indica que es la segunda o sucesivas veces que ha estado en la cache L2, por lo que muestra reuso. Por lo tanto, el bloque se marca para su almacenamiento en la SLLC (4).

La información obtenida de la presencia de la dirección del bloque en el buffer y la obtenida de la proveniencia original del bloque son complementarias. El buffer es el responsable de detectar por primera vez el reuso, almacenando entonces el bloque en la SLLC. Después, tanto la proveniencia desde la SLLC como la presencia en el buffer son indicativos de reuso. En función de la actividad de reemplazo del buffer y de la SLLC, se dan los casos de que en sucesivas apariciones del bloque éste proviene de la SLLC y además está en el buffer, sólo uno de los dos, o ninguno. En este último caso, la detección del reuso está fuera del alcance de la capacidad del buffer y de la SLLC, y el bloque se marca para bypass.

Como excepción, una fracción predeterminada de los bloques marcados para bypass se marca para su almacenamiento con baja prioridad en la SLLC, revirtiendo la decisión de hacer bypass (5). Al recibir un bloque con esta marca, la SLLC lo almacenará sólo si hay una vía libre en el conjunto correspondiente. Se insertará además con la prioridad más baja (probabilidad de reemplazo más alta) de la política de reemplazo. Esto se hace así porque, al ser la SLLC exclusiva, se genera un nuevo hueco

en un conjunto cada vez que se produce un acierto. Si no hay bypassing, este hueco se rellena con relativa rapidez, pero no ocurre así si el nivel de bypass es alto, con lo que se desaprovecha espacio en la SLLC. Al rellenar estos huecos con bloques no reusados, ese espacio se utiliza, si bien no de forma tan eficiente. Las simulaciones realizadas muestran que es suficiente con transformar uno de cada 32 bloques marcados para bypass para obtener la mayor parte del beneficio en rendimiento.

Finalmente, si el bloque está marcado para su almacenamiento en la SLLC, se envía a la misma. Si está marcado para bypass, el funcionamiento depende de si el bloque está sucio o no, es decir, si ha sido modificado o no durante su estancia en la cache L2. Si está sucio, se envía para su escritura directa en memoria y, si está limpio, se envía un mensaje de control para la actualización del directorio. El directorio se actualiza también con el resto de mensajes.

Si se utiliza TC-AGE como política de reemplazo en la SLLC, como es el caso del entorno de simulación utilizado en este trabajo, el bit que indica la proveniencia o no del bloque de la SLLC se envía también en caso de no realizar bypass. Esta información se utilizará en TC-AGE como en el algoritmo original, si bien ahora presenta un significado ligeramente diferente. La primera vez que no se hace bypass está a cero, pero el DR ya ha detectado reuso para el bloque, con al menos dos accesos. Cuando en sucesivas ocasiones está a uno, el bloque ha recibido ya al menos tres accesos. Por lo tanto, TC-AGE otorgará menos probabilidad de reemplazo en la SLLC a bloques con múltiple reuso que a bloques que han sido reusados una única vez.

3.4 Detalles de implementación

En una primera aproximación, el buffer del DR puede implementarse de forma asociativa por conjuntos, conteniendo la etiqueta asociada a la dirección del bloque, un bit de validez, y la información para la política de reemplazo. Cada línea de la cache del DR almacenaría una dirección de bloque.

Aunque esta implementación es sencilla y efectiva, presenta el inconveniente de requerir mucho espacio para almacenar poca información. Por ello proponemos utilizar dos técnicas para reducir el espacio necesario: Almacenar las etiquetas por dirección de sector en vez de por dirección de bloque y comprimir las etiquetas a guardar.

Se entiende por sector a un conjunto de bloques de memoria consecutivos y alineados al tamaño de sector. Almacenar en el DR las etiquetas por dirección de sector permite guardar la información de varias direcciones de bloques consecutivos de memoria en cada línea del buffer del DR. Para cada bloque es necesario mantener un bit de presencia. Por ejemplo, con un tamaño de sector de 4 bloques, una línea se compondrá de una etiqueta (calculada a partir de la dirección del sector), un bit de validez, los bits de reemplazo y 4 bits de presencia.

Cuanto más se incremente el tamaño de sector se podrá hacer seguimiento de más bloques por cada línea, y serán necesarias menos líneas para mantener la cobertura requerida. No obstante, se requiere que exista suficiente localidad espacial en los programas ejecutados. Si no, algunos de los bloques de

cada sector no estarán presentes en el flujo de expulsiones de la cache L2, por lo que ese espacio se desaprovecha.

La compresión de las etiquetas a guardar busca almacenar menos bits para la etiqueta presente en cada línea, manteniendo una buena capacidad de discriminación entre sectores. El proceso que se realiza es como sigue: si llamamos *t* al número de bits de la etiqueta completa, y *c* al número de bits de la etiqueta comprimida, se toman los *t* bits y se separan en varios trozos de tamaño *c*. El último trozo se rellena con bits a "0" hasta ese tamaño. Después, se realiza la operación XOR de todos ellos, obteniéndose un único valor de *c* bits, que es el que se almacena.

La utilización de etiquetas comprimidas puede provocar falsos positivos, ya que son varios los sectores de cache L2 que comparten el mismo valor de etiqueta comprimido. Esto provoca la detección de falsos reusos, y el envío de bloques realmente no reusados a la SLLC. Estos bloques no provocan problemas funcionales, pero sí pueden llegar a degradar del rendimiento. Para limitar esta degradación es necesario mantener un número de bits suficiente. El valor concreto depende del tamaño del buffer y de su asociatividad.

El buffer del DR utiliza FIFO de 1 bit como política de reemplazo. Las simulaciones realizadas indican que utilizar FIFO de 1 bit mantiene un rendimiento del DR muy similar a otras que requieren de más espacio. La política FIFO implica que la información de antigüedad se actualiza sólo durante la inserción de una dirección, y no durante aciertos posteriores. Ello es coherente con la misión prioritaria del buffer, que es detectar el primer reuso de un bloque. Para los siguientes reusos, la información de la proveniencia del bloque de la SLLC es la fundamental, si bien se utiliza también la presencia en el buffer de forma complementaria por si el bloque ha sido expulsado de la SLLC.

La asociatividad utilizada para el buffer del DR es 16. Las simulaciones realizadas indican que mantiene un rendimiento similar a otras asociatividades más altas.

3.5 Coste de hardware

En esta sección, se calcula el número total de bits de almacenamiento que es necesario añadir para implementar el Detector de Reuso sobre el sistema base descrito en la sección 5.2. Aparte de este almacenamiento, el DR requiere también su lógica de gestión.

La configuración del DR considerada tiene una cobertura de expulsión de 2 MB, con 1.024 conjuntos de asociatividad 16, un tamaño de sector de 2 bloques, y utiliza etiquetas de 10 bits. Esta configuración es la utilizada en la sección 6 para realizar la comparativa con otras propuestas, tras el trabajo de selección de una configuración balanceada considerando prestaciones frente a coste.

La Tabla 1 detalla el cálculo del coste. Considerando una SLLC de 8 MB como la descrita en la sección 5.2, el coste total para los 8 procesadores es de un 2,6% del tamaño de la SLLC.

Componente	Coste
Validez	1 bit
Etiqueta de sector	10 bit
Presencia de bloque en sector	2 bits
Reemplazo	1 bit
Total por entrada	**14 bits**
Número de entradas	16.384
Total DR por procesador	**28 KB**
Número de procesadores	8
Total DR multiprocesador	**224 KB**

Tabla 1: Coste de almacenamiento adicional del DR

4 Trabajos relacionados

La localidad de reuso ha sido estudiada en varias propuestas publicadas. Inicialmente se identifica y aprovecha en el ámbito de caches de discos, donde Karedla et al. proponen segmentar la pila de reemplazo LRU para separar, en dos listas, los bloques referenciados una vez de los referenciados varias veces (reusados). Se protege así a los reusados de ser expulsados por un súbito exceso de flujo de los no reusados [12]. Esta misma estrategia se ha aplicado recientemente como política de reemplazo para SLLCs. Albericio et al. identifican que el flujo de accesos a la SLLC presenta localidad de reuso, y priorizan con su política NRR la expulsión de bloques de la lista de no (recientemente) reusados, manteniendo un coste equivalente a NRU [10]. Qureshi et al. limitan con su política LIP el tamaño de la lista no referenciada a un único elemento, y proponen un mecanismo de competición entre conjuntos con distinta política *(set dueling)* para decidir si utilizar LIP o LRU [2]. Gao et al. utilizan la segmentación de la pila LRU para la división de cada conjunto de la SLLC en dos listas de tamaño variable, aplicando otras optimizaciones como la utilización de bypassing si resulta rentable según los resultados de la competición entre conjuntos [13]. Este mismo mecanismo es utilizado por Khan et al. para limitar de forma dinámica el tamaño de las listas, y opcionalmente realizar bypass de bloques, adaptándose al comportamiento reciente de los accesos a la SLLC [14]. Al contrario que en estas propuestas, el DR evita la presencia de bloques no reusados en la SLLC, y utiliza un buffer aparte para que la detección del reuso no esté limitada por la capacidad de almacenamiento de la SLLC o las anteriores decisiones de selección de contenidos realizadas.

Otras propuestas buscan predecir el comportamiento de reuso de los bloques, y utilizan esta predicción para modificar la política de inserción y reemplazo en la SLLC. Jaleel et al. realizan una predicción del intervalo de re-referencia de cada bloque que, en su versión estática (SRRIP), asigna un intervalo intermedio a los bloques recién insertados, y un intervalo mínimo a los que son reusados, priorizando el reemplazo de los bloques con intervalo más largo. En su versión bimodal (BIP), algunos bloques son aleatoriamente insertados con menor intervalo de re-referencia y, en su versión dinámica (DRRIP), se

selecciona entre los dos anteriores utilizando competición entre conjuntos [5]. Wu et al. presentan una evolución de SRRIP donde se busca mejorar su predicción estática, correlándola con otros valores de referencia como el contador de programa (PC), la región de memoria y la reciente secuencia de instrucciones ejecutada [15]. Gaur et al. adaptan SRRIP a su uso en caches exclusivas dándole el nombre de TC-AGE, y proponen nuevos mecanismos de predicción para dicho tipo de caches basándose en el número de veces que un bloque viaja de la SLLC a la cache L2 y en el número de accesos que presenta en la cache L2 [3]. El mecanismo del DR es compatible con TC-AGE, siendo de hecho la política de reemplazo seleccionada para la SLLC a la hora de mostrar resultados en la sección 6.

En la misma línea predictiva, Li et al. realizan un seguimiento de los pares de bloques víctima y entrantes y, al detectar el primer reuso de los dos, aproximan el comportamiento que tendría un algoritmo de bypass óptimo, y predicen que aquellos bloques que se accederán desde el mismo contador de programa se comportarán igual, guiando su decisión de bypass [16]. Seshadri et al. hacen un seguimiento global a través de un filtro Bloom de las direcciones de los últimos bloques expulsados de la SLLC y, si un bloque expulsado es reaccedido pronto, predicen que es un bloque con alta probabilidad de reuso, al que se asigna la menor probabilidad de reemplazo [17]. Chaudhuri et al. proponen un mecanismo (CHAR) que registra el patrón de acceso que han tenido los bloques en todos los niveles de la jerarquía de memoria, y los clasifica en cuatro clases en función del mismo. Para cada clase, se observa de forma dinámica si sus bloques muestran reuso o no, y se predice que el comportamiento futuro de un bloque será el observado para su clase. Esta decisión guía un mecanismo de bypass [6]. Todas estas propuestas son compatibles con el Detector de Reuso, y podrían utilizarse para cambiar la decisión fija que hace el DR de realizar bypass de bloques que no han demostrado previamente reuso.

Argumentando que ninguna de las alternativas existentes resuelve de forma satisfactoria la presencia mayoritaria de bloques muertos en la SLLC, Albericio et al. proponen con la Reuse Cache separar en la SLLC la matriz de etiquetas de la de datos, y reducir el tamaño de esta última sin perder rendimiento, seleccionando como contenido sólo bloques que hayan demostrado reuso en la matriz de etiquetas [11]. El Detector de Reuso utiliza este mismo criterio de selección, pero el resto del diseño es diferente.

En este trabajo se ha seleccionado CHAR y la Reuse Cache como propuestas que representan el estado del arte. Hay dos razones para ello: por un lado, ambas presentan en los artículos publicados rendimientos superiores a otras de las analizadas; por otro, existe una versión de CHAR para caches exclusivas, y es posible la adaptación de la Reuse Cache a su uso en exclusión. El rendimiento del DR se compara con ellas en la sección 6, donde también se dan más detalles de su funcionamiento.

5 Metodología

Esta sección detalla el entorno de experimentación y la configuración del sistema base que se han utilizado para la evaluación de la propuesta y la obtención de los resultados expuestos en la sección 6.

5.1 Entorno de experimentación

Como motor de simulación se utiliza Simics [18], un simulador de ejecución de sistemas completos. También se utilizan los plug-ins Ruby y Opal de Multifacet GEMS [19]. Se usa Ruby para modelar la jerarquía de memoria con un alto grado de detalle: caches, directorio, protocolo de coherencia, red on-chip, buffers, contención, etc., añadiendo además un modelo detallado de una DRAM DDR3. Se utiliza Opal (también conocido como TFSim) para modelar de forma detallada un procesador superescalar con ejecución fuera de orden.

Se ejecuta sobre Solaris 10 para SPARC una carga de trabajo multiprogramada compuesta por aplicaciones de la suite SPEC CPU 2006 [20]. Para localizar el final de la fase de inicialización de cada programa, utilizamos contadores hardware en una máquina real, y ejecutamos todos los binarios SPARC con las entradas de referencia hasta su finalización. Para nuestro sistema con 8 procesadores hemos producido una serie de 100 mezclas, combinaciones aleatorias de 8 programas cada una, tomados de entre los 29 programas que componen SPEC CPU 2006. Cada programa aparece entre 18 y 41 veces, siendo el número medio de apariciones de 27,6 y la desviación típica de 6,1. En cada mezcla, nos aseguramos de que ninguna aplicación está en su fase de inicialización, avanzando la simulación hasta que todas las fases de inicialización están terminadas. Comenzando en este punto, en cada simulación se ejecutan 300 millones de ciclos de calentamiento del sistema de memoria, y luego recolectamos estadísticas para los siguientes 700 millones de ciclos.

No se han hechos esfuerzos por distinguir las aplicaciones por su tipología ni sus patrones o estadísticas de acceso a memoria. Se muestra en la Tabla 2 el número medio de fallos por kilo-instrucción (MPKI) de cada aplicación en todas las mezclas donde aparece, en los tres niveles de la jerarquía de memoria, cuando las ocho aplicaciones de cada mezcla se ejecutan conjuntamente sobre el sistema base.

Aplicación	**L1**	**L2**	**LLC**	**Aplicación**	**L1**	**L2**	**LLC**	**Aplicación**	**L1**	**L2**	**LLC**
astar	7,5	1,1	0,7	gromacs	11,7	3	1,2	perlbench	10,2	1,8	0,8
bwaves	24,5	21,1	20,1	hmmer	3,3	2,4	0,2	povray	11,5	0,2	0,1
bzip2	8,4	3,9	0,9	h264ref	4,2	1,4	0,7	sjeng	6,9	0,8	0,5
cactusADM	20,8	11,4	4,9	lbm	65,4	38,6	36,7	soplex	8,9	7,1	3,1
calculix	8,5	4,3	1,5	leslie3d	40,4	23,2	17,9	sphinx3	18,8	14,3	11,7
dealII	1,6	0,5	0,3	libquantum	45,8	33,2	32,2	tonto	6,7	1,3	0,5
gamess	6,7	1	0,6	mcf	64,9	36	18,9	wrf	14,3	8,9	1,5
gcc	22	6,4	2,1	milc	24,6	23,5	22	xalancbmk	15,1	8,7	2,8
GemsFDTD	42,7	29,7	22,8	namd	1,7	0,2	0,2	zeusmp	32,3	8,7	7,2
gobmk	13,2	1,1	0,3	omnetpp	12,6	9,2	2,2				

Tabla 2: MPKI medio para cada nivel de cache del sistema base (SLLC exclusiva de 8 MB con TC-AGE)

5.2 Configuración del sistema base

Modelamos un sistema base con 8 procesadores superescalares con ejecución especulativa y fuera de orden. Cada procesador consta de 4 vías, una segmentación de 18 etapas y 10 unidades funcionales. El predictor de saltos es de tipo YAGS [21] con un PHT (*Pattern History Table*) de 4K entradas. La Tabla 3 muestra más información del modelo de procesador simulado.

Arquitectura base		SPARC v9
Procesadores		8, superescalares de 4 vías, 2.66 GHz
Segmentación		18 etapas: 4 fetch, 4 decode, 4 dispatch/read, 1 (o más) execute, 3 memory, 2 commit
Buffers		Buffer de reordenación de 128 entradas
Bancos de registros		Enteros: 160 (lógicos) + 128 (renombre) Punto flotante: 64 (lógicos) + 128 (renombre)
Unidades funcionales		4 Enteros, 4 punto flotante, 2 load/store
Predictor de saltos		Tipo YAGS PHT: 4.096 entradas

Tabla 3: Especificaciones del modelo de procesador

Cada procesador tiene 2 niveles de cache privados y todos los procesadores comparten la cache exclusiva de tercer nivel. La SLLC utiliza TC-AGE como política de reemplazo, y tiene 8 MB de capacidad total, con cuatro bancos entrelazados a nivel de línea de cache (64B). Una red de tipo crossbar conecta los procesadores y dichos bancos. Hay dos canales de memoria DDR3 que corren a 667 MHz. La Tabla 4 muestra más información de la jerarquía de memoria simulada. El cálculo de los tiempos de acceso de las caches se ha realizado mediante la herramienta CACTI [22], con un nodo tecnológico de 45 nm.

Cache privada L1 I/D		32 KB, 4 vías, reemplazo LRU, tamaño de bloque 64 B, latencia de acceso de 3 ciclos
Cache privada unificada L2		256 KB inclusiva de L1, 8 vías, reemplazo LRU, tamaño de bloque 64 B, latencia de acceso de 7 ciclos
Interconexión		Red tipo crossbar, ancho de bus de 80 bits, latencia de 5 ciclos
Cache compartida L3 (SLLC)		8 MB exclusiva (4 bancos de 2 MB cada uno), entrelazado por bloques, tamaño de bloque 64 B. Cada banco: 16 vías, reemplazo TC-AGE de 2 bits, latencia de acceso de 10 ciclos, 32 MSHR
DRAM		2 rangos, 8 bancos, 4 KB de tamaño de página, Double Data Rate (DDR3 1333 MHz). 92 ciclos de latencia de acceso bruta
Bus DRAM		2 canales a 667 MHz, cada uno con bus de 8 B, 4 ciclos de DRAM por línea, 16 ciclos de procesador por línea

Tabla 4: Especificaciones de la jerarquía de memoria

6 Resultados

En esta sección se presenta una evaluación de los resultados obtenidos con el mecanismo propuesto, utilizando la metodología expuesta en la sección anterior. Se presentan dos métricas diferentes para resumir el rendimiento obtenido: el número de instrucciones por ciclo normalizado al del sistema base (IPC normalizado) y la reducción en fallos por instrucción frente al sistema base. Los valores mostrados son la media de los resultados obtenidos para cada una de las 100 mezclas. Para cada mezcla, el IPC normalizado obtenido para una propuesta "PROP" se calcula como $\frac{\sum_t IPC_t^{PROP}}{\sum_t IPC_t^{TC-AGE}}$, donde IPC_t es el IPC obtenido para el procesador *t*. La reducción en número de fallos por instrucción se calcula como $1 - \frac{\sum_t F_t^{PROP}}{\sum_t F_t^{TC-AGE}} \cdot \frac{\sum_t I_t^{TC-AGE}}{\sum_t I_t^{PROP}}$, donde F_T es el número de fallos de SLLC medidos a lo largo de la ejecución para el procesador *t*, e I_T es el número de instrucciones ejecutadas para el procesador *t*.

Los primeros tres apartados evalúan la influencia en rendimiento y coste de la variación en el DR del tamaño del buffer, el tamaño de sector y el tamaño de la etiqueta almacenada. Se obtiene así una configuración balanceada entre coste y prestaciones. A continuación, se hace un análisis del funcionamiento del DR, explicando de qué manera reduce la tasa de fallos de la SLLC. En los siguientes dos apartados, se proporcionan datos más detallados a nivel de mezcla y de aplicación. Por último, se realiza una comparativa con otras propuestas publicadas.

6.1 Influencia del tamaño del buffer del Detector de Reuso

En esta sección estudiaremos cómo varía los resultados en función del tamaño del buffer del DR. La Figura 6 muestra el IPC y la tasa de fallos obtenidos de media en las 100 mezclas descritas en la sección 4, normalizados respecto del sistema base con TC-AGE, en función la cobertura de expulsión de cada DR. El tamaño de sector del DR es de 1 bloque, y se almacenan etiquetas completas.

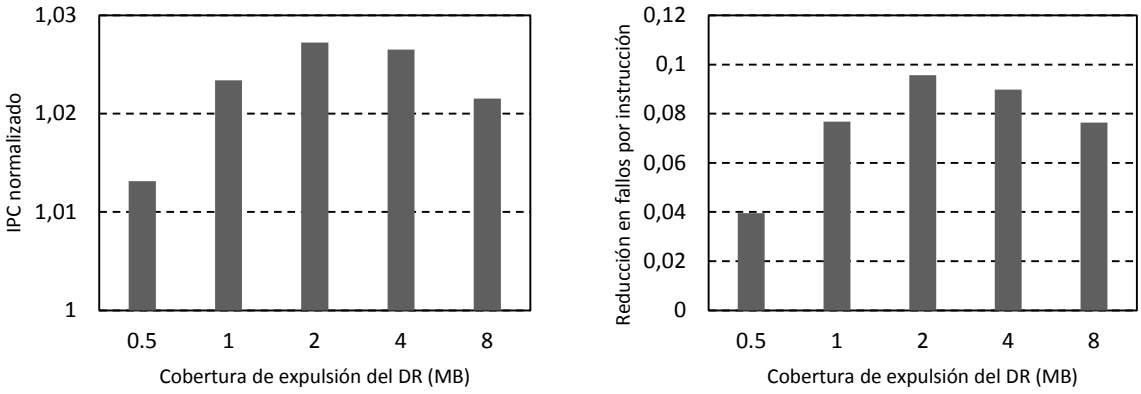


Figura 6: IPC normalizado (izquierda) y reducción en fallos de SLLC por instrucción (derecha) frente al sistema base con TC-AGE, en función de la cobertura de expulsión del DR en cada procesador.

A medida que el buffer del DR incrementa su tamaño, mantiene información de más líneas, ampliando su cobertura de expulsión. Al seguir una política de reemplazo FIFO, esto implica que se puede hacer seguimiento de líneas que han sido expulsadas de la cache L2 hace más tiempo, es decir, detectar reusos más lejanos. La configuración óptima del DR se consigue con una cobertura de expulsión de 2 MB, donde presenta un incremento del IPC de un 2,7%, y una reducción de los fallos por instrucción en la SLLC del 9,6%.

6.2 Influencia del tamaño de sector del Detector de Reuso

La Figura 7 muestra cómo varía el rendimiento cuando se incrementa el tamaño de sector en el buffer del DR. Dentro de una misma cobertura de expulsión del DR, al doblar el tamaño de sector se reduce a la mitad el número de conjuntos.

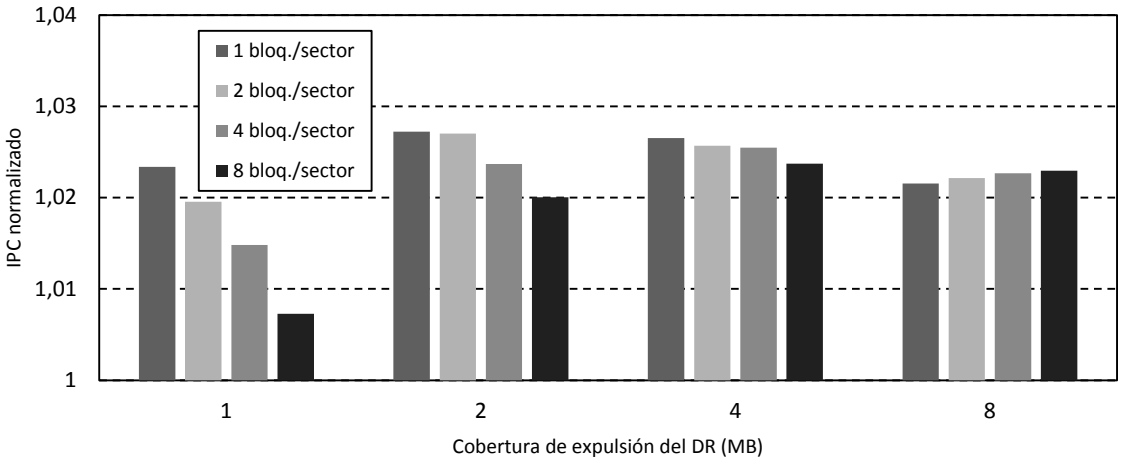


Figura 7: IPC normalizado frente al sistema base con TC-AGE, en función de la cobertura de expulsión del DR en cada procesador, y para diferentes tamaños de sector del DR. El tamaño de sector está expresado en el número de bloques de memoria que sigue cada línea del DR.

Dentro de una misma cobertura, al incrementarse el tamaño de sector la capacidad efectiva es menor, puesto que en algún caso la falta de localidad espacial hace que alguno de los bloques del sector no sean referenciados. Este efecto hace que se detecte menos reuso. Ello produce una degradación del rendimiento respecto del óptimo, salvo que ya se estuviera detectando un exceso de reuso, como puede verse para la cobertura de 8 MB.

A cambio, la cantidad de bits de memoria requeridos es menor. La Figura 8 muestra la cantidad de memoria requerida por cada DR en función la cobertura de expulsión y del tamaño de sector, utilizando una etiqueta de 10 bits. Para una misma cobertura de expulsión, el valor disminuye a medida que incrementamos el tamaño de sector. Esto es debido a que el ahorro de espacio por la reducción del número de conjuntos es mayor que el incremento por añadir bits de presencia de bloque a la línea. La configuración que da el mejor rendimiento, con 2 MB de cobertura, precisa de un buffer de 48 KB en cada DR. Otras configuraciones presentan mejores relaciones entre prestaciones y coste, como por ejemplo aquella con 2 MB de cobertura y tamaño de sector de 2 bloques, que presenta un rendimiento

un 0,02% menor con un buffer de 28 KB, un 42% menor. Esta última es la configuración seleccionada para utilizar en adelante para el resto de resultados.

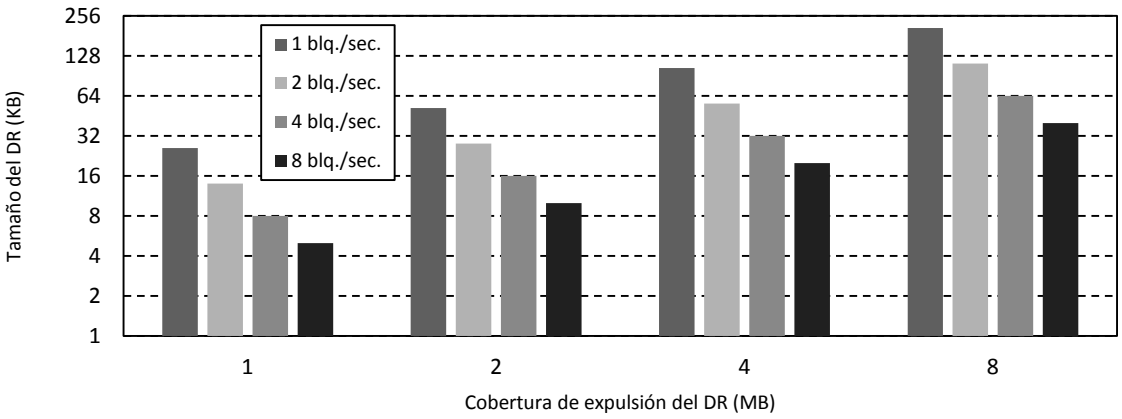


Figura 8: Tamaño del buffer de cada DR en KB, en función de la cobertura de expulsión del mismo, y para diferentes tamaños de sector del DR. El tamaño de sector está expresado en el número de bloques que sigue cada línea. Los valores están calculados considerando una etiqueta de 10 bits.

6.3 Influencia del tamaño de la etiqueta del Detector de Reuso

Para reducir la cantidad de memoria requerida, en el DR pueden almacenarse etiquetas comprimidas, como se ha explicado en el apartado 3.4. La Figura 9 muestra a la izquierda, en función del tamaño de la etiqueta almacenada, la tasa media de errores en el chequeo de la etiqueta debidos a esta compresión. Estos errores son falsos positivos, en los cuales se detecta un falso reuso porque la etiqueta comprimida del sector recibido coincide con la de otro sector diferente, anteriormente registrado.

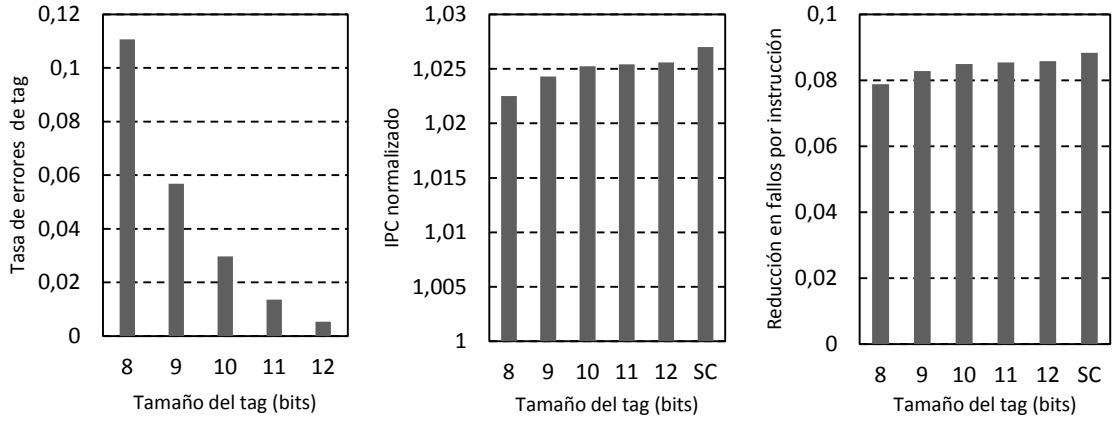


Figura 9: Izquierda: Tasa media de errores en el chequeo de la etiqueta debidos a la compresión de la misma. Centro: IPC normalizado frente al sistema base con TC-AGE. Derecha: Reducción en fallos de SLLC por instrucción frente al sistema base con TC-AGE. Incluye la etiqueta sin compresión ("SC").

El envío a la SLLC de bloques no reusados reduce la efectividad del mecanismo. La Figura 9 muestra, en el centro y a la derecha, el IPC y la tasa de fallos normalizados obtenidos para la configuración de 2 MB de cobertura y un sector de 2 bloques, en función del tamaño de la etiqueta en el DR. La pérdida de rendimiento es muy baja para tamaños de etiqueta como el de 10 bits, donde se empeora un 0,16%

el IPC y un 0,34% los fallos frente a la configuración sin compresión de la etiqueta. Ello justifica la compresión con el fin de reducir la cantidad de memoria requerida.

En adelante, salvo indicación de lo contrario, esta configuración con un DR de 2 MB de cobertura de expulsión, un tamaño de sector de 2 bloques, y 10 bits de etiqueta es la utilizada en los resultados mostrados.

6.4 Análisis del funcionamiento del Detector de Reuso

Resulta de interés detenerse a analizar de qué manera consigue el DR reducir la tasa de fallos de la SLLC. La Figura 10 muestra, para los programas de una mezcla de ejemplo, la distribución de los bloques recibidos desde el punto de vista del DR, en cinco categorías: (U) primer uso, (R) primer reuso, (MD) múltiple reuso detectado sólo por el DR, (MC) múltiple reuso detectado sólo por la proveniencia el bloque de la SLLC, y (MA) múltiple reuso detectado por ambos mecanismos. Una expulsión categorizada como U provoca el bypass del bloque, mientras que el resto envían el bloque para alojarlo en la SLLC. Las expulsiones de tipos U, R o MD indican que originalmente el bloque viene de la memoria, o sea, provienen de un fallo en la SLLC, mientras que los tipos MC y MA indican un acierto previo en la SLLC. El hardware propuesto del DR no puede distinguir entre los casos R y MD, pero se han separado en la gráfica para ejemplificar la complementariedad de los mecanismos de detección de reuso.

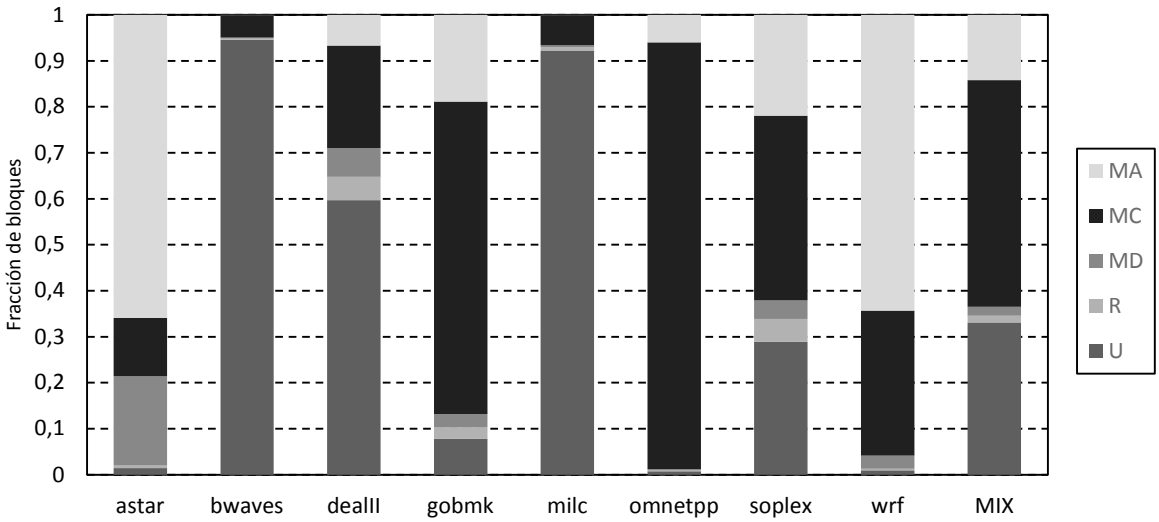


Figura 10: Fracción de expulsiones de bloques de cada programa de una mezcla ejemplo, categorizadas desde el punto de vista del DR según su tipología de reuso en: (U) primer uso, (R) primer reuso, (MD) múltiple reuso detectado sólo por el DR, (MC) múltiple reuso detectado sólo por la proveniencia el bloque de la SLLC, y (MA) múltiple reuso detectado por ambos mecanismos.

El nivel de bypass varía de un programa a otro, ajustándose bloque a bloque al patrón de reuso que éste muestra. En *bwaves* y *milc*, en más del 92% de las expulsiones se ha detectado un solo uso, y no se envían a la SLLC. Ello es coherente con las medidas mostradas en la Tabla 2, que indican que la SLLC apenas reduce el número de fallos por instrucción de estos programas. En el extremo opuesto, en *astar*, *omnetpp* y *wrf* menos del 3% de los bloques expulsados de las caches privadas no muestran

reuso, por lo que hay escaso bypass. El resto de programas, *dealII*, *gobmk* y *soplex*, presentan valores intermedios, con niveles de bypass del 30%, 8% y 24% respectivamente.

La cantidad de bloques que se envían a la SLLC tras detectar el primer reuso (tipo R) varía entre un 0,2% de *omnetpp* y un 5% de *dealII*, con una media del 1,5%. Estos pocos bloques que muestran un primer reuso son accedidos después múltiples veces (tipos MD, MC y MA), siendo la proporción media de 45 detecciones de múltiple reuso por cada primer reuso que se identifica. En ocasiones, el bloque ya ha sido reemplazado de la SLLC, y el DR lo inserta de nuevo (MD). Esto ocurre de media en un 4% de las veces que se detecta múltiple reuso.

La eliminación del envío de bloques de poca utilidad en *bwaves, milc, dealII* y *soplex,* va a permitir a la SLLC conservar mejor los bloques útiles de esos programas, ya que no serán expulsados con tanta frecuencia. Más aún, también va a permitir al resto de los programas de la mezcla conservar mejor sus bloques. La Figura 11 muestra a la izquierda la fracción de la SLLC ocupada por los bloques de cada programa, tanto en el sistema base sin bypass como en el sistema con DR. Tanto *bwaves* como *milc* ocupan en este último mucho menos espacio en la SLLC, el cual se reparte entre el resto de programas, cuyos bloques sufren menos expulsiones. Como puede verse en la Figura 11 a la derecha, la buena selección de bloques permite a *bwaves* y *milc* mantener una tasa de fallos en la SLLC similar (un 0,4% peor en el caso de *bwaves*) pese a la penalización de segundas búsquedas en memoria de bloques que sí muestran reuso. En el resto de programas, a la buena selección de bloques se le une una expulsión menos frecuente de los mismos y una mayor cantidad de espacio disponible, por lo que su número de fallos por instrucción en la SLLC mejora entre un 4,7% de *dealII* y un 77,3% de *omnetpp*. La reducción en fallos para toda la mezcla es del 11,8%, siendo el IPC normalizado de 1,030. La mezcla ocupa la posición 32 dentro de las 100 si se ordenan de mayor a menor incremento de IPC.

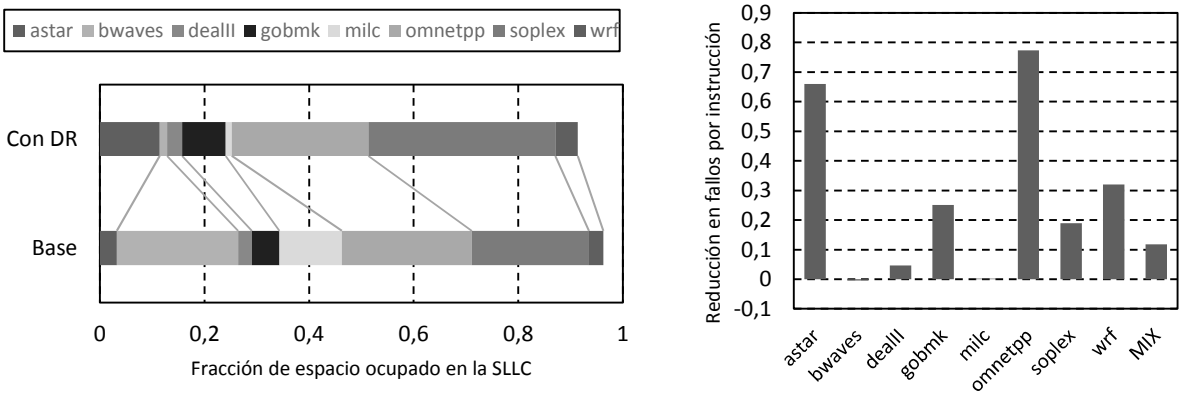


Figura 11: Izquierda: Fracción media del espacio ocupado por los bloques de cada programa de la mezcla de ejemplo, en el sistema base y en el sistema con DR. Se toman datos cada millón de ciclos de ejecución, y se calcula la media. Derecha: Reducción en fallos por instrucción en el sistema con DR, normalizada a la del sistema base con TC-AGE.

6.5 Análisis de rendimiento por mezcla

Con el fin de analizar la variabilidad de los resultados en las distintas mezclas, la Figura 12 muestra arriba el IPC para todas las mezclas multiprogramadas, normalizado al sistema base con TC-AGE sin bypass. Las mezclas se ordenan en el eje horizontal en función de ese IPC normalizado. Abajo muestra el nivel de bypass de cada mezcla.

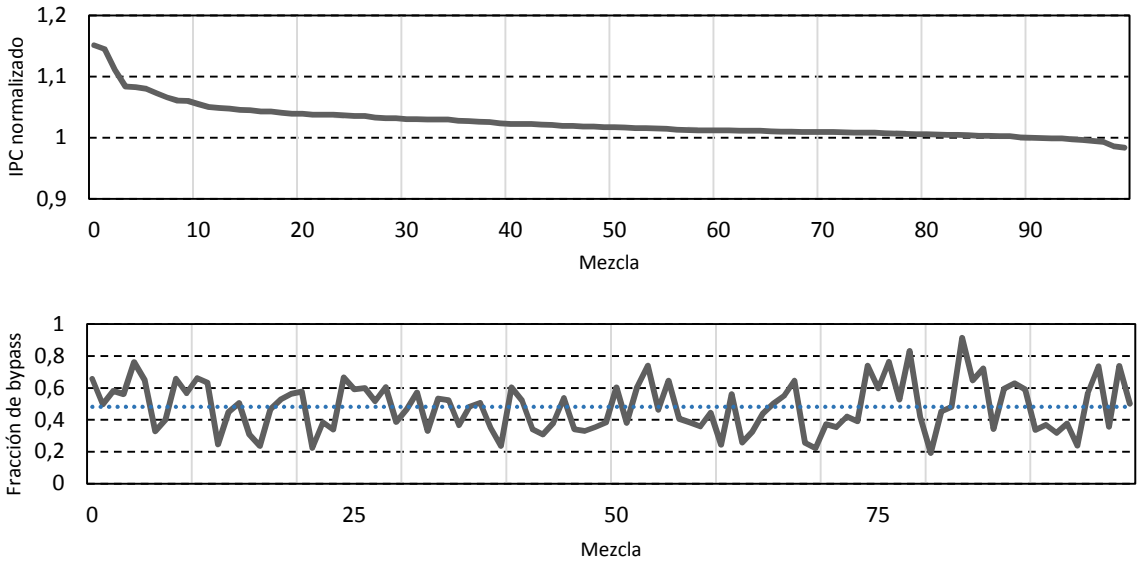


Figura 12: Arriba: IPC para las 100 mezclas, normalizado al sistema base. Abajo: Nivel de bypass de cada mezcla, en el mismo orden que arriba.

Sólo 10 de las 100 mezclas muestran decrementos del IPC frente al sistema base, de las cuales 8 son de menos del 1%, y la mayor es del 1,6%. En el otro extremo, 11 mezclas superan el 5% de incremento, siendo el mayor del 15,1%. La fracción de bypass en cada mezcla varía entre un 19,2% y un 91,4%, siendo la media del 48,2%.

6.6 Análisis de rendimiento por aplicación

Como se ha visto en la sección 6.4, el rendimiento obtenido varía en función tanto de la aplicación en sí como del resto de aplicaciones que se ejecutan en la mezcla. La Figura 13 muestra la distribución de IPC normalizado frente al sistema base con TC-AGE, obtenido para cada aplicación. Se proporcionan cinco valores por aplicación, que son el mínimo, el primer cuartil, la mediana, el tercer cuartil y el máximo.

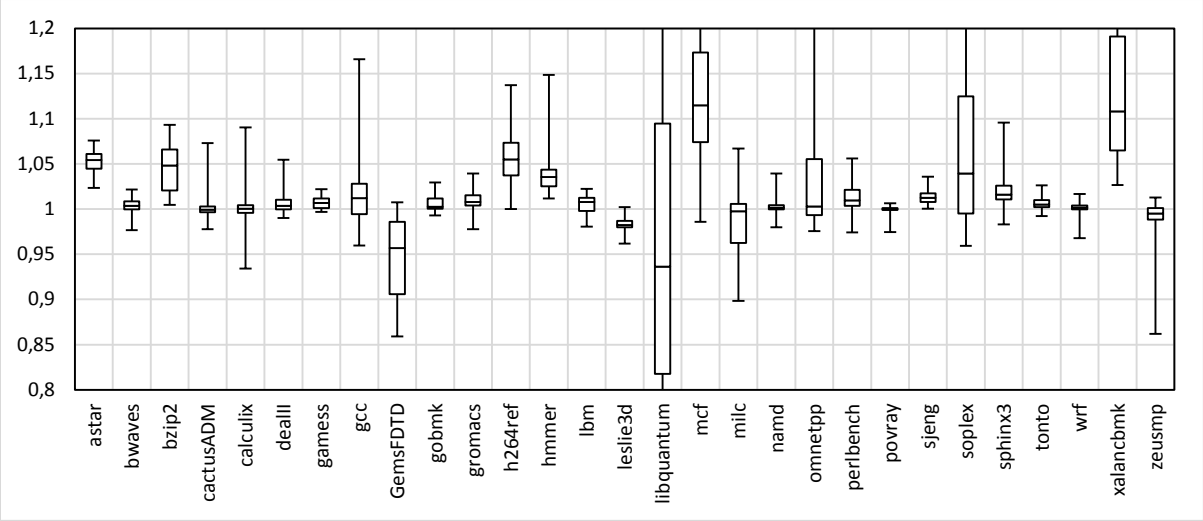


Figura 13: IPC normalizado frente a TC-AGE para todas las aplicaciones.

De las 29 aplicaciones, 6 presentan mejoras en todas mezclas donde participan (*astar*, *bzip2*, *h264ref*, *hmmer*, *sjeng* y *xalancbmk*). Otras 10 presentan mejora a partir del primer cuartil (*bwaves*, *gamess*, *gobmk*, *gromacs*, *mcf*, *namd*, *perlbench*, *povray*, *sphinx3* y *tonto*), aunque en alguna mezcla pierden rendimiento. En 7 de ellas (*calculix*, *dealII*, *gcc*, *lbm*, *omnetpp*, *soplex* y *wrf*), la mediana muestra mejoras, pero el primer cuartil pierde rendimiento. Las 6 aplicaciones restantes (*cactusADM*, *leslie3d*, *libquantum*, *GemsFDTD*, *milc* y *zeusmp*), presentan empeoramientos en la mediana.

De estas últimas, destacan por negativas *libquantum* (0,936 de mediana), *GemsFDTD* (0,957 de mediana) y *leslie3d* (0,982 de mediana), ya que las mediadas del resto están por encima de 0,995. Merece la pena analizar en más detalle estos casos, ya que parecen indicar que el mecanismo del DR es injusto con algunas aplicaciones. La Tabla 5 muestra a la izquierda la fracción media del espacio ocupado por los bloques de cada uno de estos programas en todas las mezclas donde participan, en el sistema base con TC-AGE y en el sistema con DR. En el sistema base, los tres programas ocupan entre 1,8 y 3,0 veces más espacio que el que les correspondería en promedio (12,5%). En cambio, en el sistema con DR se hace bypass de muchos de esos bloques al no mostrar reuso, equilibrándose la ocupación a entre 0,8 y 1,6 veces el espacio promedio. Al realizarse esa reducción de espacio, se pierde rendimiento en la aplicación concreta, ya que la detección del reuso no es perfecta. Sin embargo, como se muestra en la Tabla 5 a la derecha, con el DR las mezclas afectadas mejoran su rendimiento incluso por encima de las demás, puesto que el espacio liberado se utiliza para almacenar bloques con más aciertos de otras aplicaciones. Puede concluirse que TC-AGE asigna de manera injusta un espacio excesivo a estos programas, mientras que con el DR eso no ocurre. Ello lleva a que no alcancen de forma individual el rendimiento anterior, pero logra que las mezclas afectadas incrementen su rendimiento conjunto, lo cual puede entenderse más justo que la situación original.

Programa	Ocupación con TC-AGE	Ocupación con DR	IPC norm. de mezclas donde está presente	IPC norm. de mezclas donde está ausente
libquantum	38,1%	9,6%	1,036	1,022
GemsFDTD	24,1%	15,5%	1,030	1,024
leslie3d	23,2%	19,4%	1,027	1,024

Tabla 5: Izquierda, fracción media del espacio ocupado por los bloques de cada uno de estos programas en todas las mezclas donde está presente. Derecha: Media de IPC normalizado del sistema con DR frente a TC-AGE de aquellas mezclas donde el programa está presente/ausente.

6.7 Comparativa con otras propuestas

En las secciones anteriores, se ha mostrado el rendimiento en comparación con una SLLC sin bypass que utiliza TC-AGE como política de reemplazo. En esta sección se compara también el rendimiento con otras dos propuestas recientes de políticas de inserción y reemplazo basadas en bypass.

Comparación con CHAR: La política CHAR [6] (acrónimo de "cache hierarchy-aware replacement") para SLLC exclusivas es una propuesta de selección de contenidos que basa la decisión de realizar o no bypass en el patrón de acceso que han tenido los bloques en todos los niveles de la jerarquía de memoria. La decisión se toma sobre los bloques expulsados de las caches L2. Los bloques son categorizados en cuatro clases. Para la clase que muestra reuso a nivel de SLLC nunca se realiza bypass. Para las otras tres clases, existe una lógica que decide si es provechoso realizar el bypass para el conjunto de la clase o no, actuando en consecuencia. Para ello, se monitoriza a través de contadores la tasa de acierto de algunos conjuntos de la SLLC para los que se mantiene la política base TC-AGE, y se compara con la de otros conjuntos que implementan CHAR.

La Figura 14 muestra el IPC y la reducción en fallos por instrucción, ambos frente al sistema base con TC-AGE, obtenidos para sistemas con DR y CHAR, con 8 MB de SLLC. Como puede apreciarse, el mecanismo del DR bate en media a CHAR tanto en reducción de fallos (8,5% frente a 5,3%) como en IPC normalizado (2,5% frente a 2,0%). Los resultados en cuanto a reducción de fallos de CHAR son coherentes con la publicación original, pero el IPC normalizado es menor que en dicho artículo. Ello puede deberse a la diferente metodología, ya que los modelos de procesador y jerarquía de memoria utilizados son diferentes en ambos trabajos.

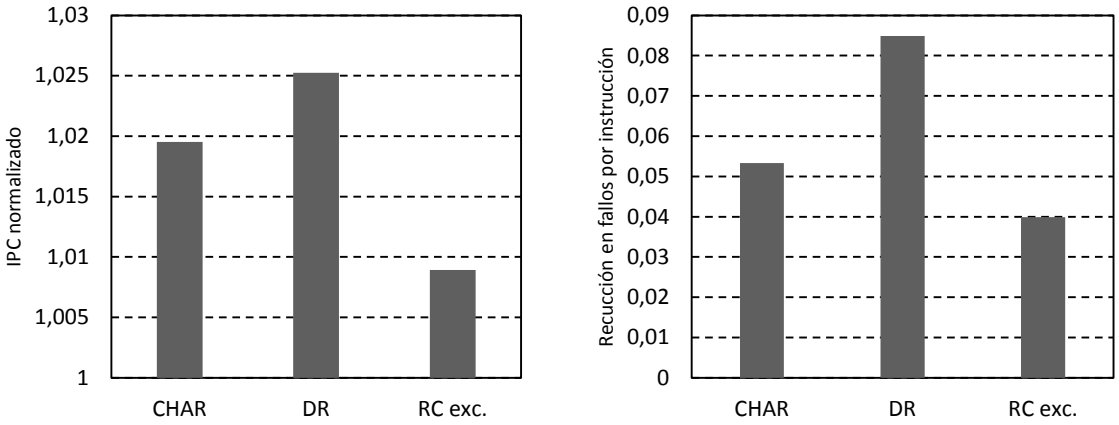


Figura 14: IPC normalizado (izquierda) y reducción en fallos de SLLC por instrucción (derecha) frente al sistema base con TC-AGE, para un sistema con un DR de 2 MB de cobertura de expulsión y un tamaño de sector de 2 bloques, CHAR (en exclusión) y una Reuse Cache exclusiva RC-32/8 con NRR en la matriz de etiquetas y TC-AGE en la matriz de datos. El tamaño de datos de la SLLC es de 8 MB.

Comparación con una Reuse Cache: La Reuse Cache [11] es una SLLC cuyas estructuras de etiquetas y datos están desacopladas, y que almacena sólo los datos de las líneas que han mostrado reuso. Del resto de líneas se hace bypass. Su intención es reducir el espacio necesario en la SLLC eliminando datos de líneas que no están vivas. Para realizar la comparación en igualdad de condiciones, se ha desarrollado una versión de la Reuse Cache donde la matriz de datos funciona en exclusión con las L2 privadas. Esta versión funciona como sigue: Cualquier bloque enviado a una L2 privada incluye un bit adicional que indica si debe ser devuelto a la SLLC al ser expulsado o no (bypass o no bypass). En un primer acceso, se envía el bloque desde memoria a la L2 privada con indicación de bypass, y se inserta la etiqueta en la matriz de etiquetas de la SLLC, al igual que en el diseño original. Esto permite detectar el reuso de la misma forma. En un segundo acceso, con acierto en la matriz de etiquetas de la SLLC y fallo en la de datos, se envía el bloque desde memoria a la L2 privada con indicación de no bypass, sin almacenarlo en la matriz de datos. La etiqueta se mantiene en la matriz de etiquetas. Cuando el bloque se expulsa de la L2 privada, se almacena en la matriz de datos de la SLLC. Posteriores accesos, con acierto tanto en la matriz de datos como en la de etiquetas, envían el bloque a la L2 privada con indicación de no bypass, y lo expulsan de la matriz de datos de la SLLC.

En este trabajo utilizamos para la comparación una Reuse Cache Exclusiva con 32MB equivalentes de etiquetas y 8MB de datos, que puede entenderse como una cache de 32MB donde se ha reducido el espacio en datos hasta los 8MB con la técnica de la Reuse Cache. Esta relación entre etiquetas y datos es la que mejor rendimiento ofrece de entre las que disponen de 8MB de datos, tanto en el artículo original como en simulaciones adicionales que se han realizado. En la matriz de datos se emplea TC-AGE como política de reemplazo, de forma coherente con el resto de propuestas analizadas.

Como puede apreciarse en la Figura 14, el mecanismo del DR también bate en media a la Reuse Cache Exclusiva tanto en reducción de fallos (8,5% frente a 4,0%) como en IPC normalizado (2,5% frente a 0,9%). En la publicación original no se muestran datos acerca de los fallos, y el IPC normalizado que

se obtiene aquí es menor que en dicho artículo, donde se emplea una cache inclusiva. Ello puede deberse a las diferencias entre los resultados con inclusión y con exclusión. También puede deberse a la diferencia en los modelos de procesador, ya que el modelo utilizado en el artículo que presenta la Reuse Cache es el de un procesador con ejecución en orden, mientras que el empleado en este trabajo es superescalar y con ejecución fuera de orden.

7 Conclusiones

En un sistema multiprocesador on-chip, el flujo de referencias que llega a la SLLC muestra poca localidad temporal. Sin embargo, muestra localidad de reuso, es decir, bloques reusados a dicho nivel tienen más probabilidad de ser referenciados en un futuro. Esto provoca que, si se realiza una gestión convencional, basada en la localidad temporal, el uso de la cache es ineficiente, desaprovechándose la mayoría de su contenido. Existe un número importante de propuestas que tratan este problema para caches inclusivas, pero pocas que se centran en caches exclusivas. Dichas caches se encuentran ya en el mercado, y es previsible que se utilicen más en el futuro.

En este trabajo se propone un nuevo mecanismo de selección de contenidos para caches exclusivas que aprovecha la localidad de reuso que presentan los accesos a la SLLC. Consiste en incluir un elemento denominado Detector de Reuso entre cada cache L2 y la SLLC, que detecta qué bloques expulsados de las L2 no han demostrado reuso y evita que sean insertados en la SLLC, realizando *bypass* de los mismos.

Se evalúa esta propuesta simulando un sistema con 8 procesadores en un chip que ejecuta una serie de cargas multiprogramadas. Configurado adecuadamente, el Detector de Reuso evita la inserción de bloques poco útiles en la SLLC, facilitando que se mantengan los más reusados. Los resultados muestran que ello permite incrementar el rendimiento, por encima de otras propuestas recientes como CHAR o la Reuse Cache. Por ejemplo, para una configuración del DR balanceada entre coste y prestaciones, se obtiene un 8,5% de reducción de tasa de fallos y un incremento del IPC de un 2,5%, ambos frente a un sistema base con TC-AGE.

8 Referencias

[1] S. Khan, Y. Tian y D. A. Jiménez, «Sampling Dead Block Prediction for Last-Level Caches,» *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture,* pp. 175-186, 2010.

[2] M. Qureshi, A. Jaleel, Y. Patt, S. Steely y J. Emer, «Adaptive insertion policies for high performance caching,» *Proceedings of the 34th annual International Symposium on Computer Architecture,* pp. 381-391, 2007.

[3] J. Gaur, M. Chaudhuri y S. Subramoney, «Bypass and Insertion Algorithms for Exclusive Last-level Caches,» *Proceedings of the 38th International Symposium on Computer Architecture,* pp. 81-92, Junio 2011.

[4] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr. y J. Emer, «Achieving Non-Inclusive Cache Performance with Inclusive Caches. Temporal Locality Aware (TLA) Cache Management Policies,» *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture,* pp. 151-162.

[5] A. Jaleel, K. B. Theobald, S. C. Steely Jr. y J. Emer, «High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP),» *Proceedings of the 37th International Symposium on Computer Architecture,* pp. 60-71, Junio 2010.

[6] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney y J. Nuzman, «Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches,» *Proceedings of the 21st international conference on Parallel architectures and compilation techniques,* pp. 293-304, 2012.

[7] N. P. Jouppi y S. J. E. Wilton, «Tradeoffs in Two-Level On-Chip Caching,» *Proceedings the 21st Annual International Symposium on Computer Architecture,* pp. 34-45, 1994.

[8] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak y B. Hughes, «Cache hierarchy and memory subsystem of the AMD Opteron processor,» *IEEE micro,* nº 30(2), pp. 16-29, 2010.

[9] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer y B. Falsafi, «Scale-Out Processors,» *ACM SIGARCH Computer Architecture News,* vol. 40, nº 3, pp. 500-511.

[10] J. Albericio, P. Ibáñez, V. Viñals y J. M. Llabería, «Exploiting reuse locality on inclusive shared last-level caches,» *ACM Transactions on Architecture and Code Optimization (TACO),* vol. 9, nº 4, p. 38, 2013.

[11] J. Albericio, P. Ibáñez, V. Viñals y J. Llabería, «The reuse cache: downsizing the shared last-level cache,» *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture,* pp. 310-321, 2013.

[12] R. Karedla, J. Love y B. Wherry, «Caching strategies to improve disk system performance,» *Computer,* nº 27(3), pp. 38-46, 1994.

[13] H. Gao y C. Wilkerson, «A dueling segmented LRU replacement algorithm with adaptive bypassing,» *Proc. of the 1st JILP Workshop on Computer Architecture Competitions,* 2010.

[14] S. Khan, Z. Wang y D. Jimenez, «Decoupled dynamic cache segmentation,» *Proc. IEEE 18th Int. Symp. High Performance Computer Architecture HPCA,* pp. 1-12, 2012.

[15] C. J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr y J. Emer, «SHiP: signature-based hit predictor for high performance caching,» *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture,* pp. 430-441, 2011.

[16] L. Li, D. Tong, Z. Xie, J. Lu y X. Cheng, «Optimal bypass monitor for high performance last-level caches,» *Proceedings of the 21st international conference on parallel architectures and compilation techniques,* pp. 315-324, 2012.

[17] V. Seshadri, O. Mutlu, M. A. Kozuch y T. C. Mowry, «The evicted-address filter: a unified mechanism to address both cache pollution and thrashing,» *Proceedings of the 21st international conference on Parallel architectures and compilation techniques,* pp. 355-366, 2012.

[18] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt y B. Werner, «Simics: A full system simulation platform,» *Computer,* nº 35(2), pp. 50-58, 2002.

[19] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill y D. Wood, «Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset,» *Computer Architecture News,* nº 33(4), pp. 92-99, 2005.

[20] J. L. Henning, «SPEC CPU2006 benchmark descriptions,» *ACM SIGARCH Computer Architecture News,* vol. 34, nº 4, pp. 1-17, 2006.

[21] A. N. Eden y T. Mudge, «The YAGS branch prediction scheme,» *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture,* pp. 69-77, 1998.

[22] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman y N. P. Jouppi, «A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies,» *35th International Symposium on Computer Architecture,* pp. 51-62, 2008.

[23] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu y G. Zyner, «The visual instruction set (VIS) in UltraSPARC,» *Computer Conference, IEEE International,* pp. 462-462, 1995.

9 Anexo A: Contexto del trabajo

Este anexo hace un breve repaso del contexto en que se lleva a cabo este trabajo fin de máster.

9.1 Memorias DRAM y memorias cache

En los sistemas informáticos modernos, existe una disparidad creciente entre el ritmo al que los procesadores necesitan acceder a los datos existentes en memoria y el ritmo al que la memoria puede suministrarlos. Un acceso a un dirección de memoria necesita hoy en día típicamente desde decenas hasta cientos de ciclos de procesador, y dicha cifra tiende a empeorar, dado que las memorias DRAM reducen sus tiempos de acceso a menor ritmo que los procesadores incrementan las peticiones de acceso a sus contenidos. Esto ocasiona que existan tiempos muertos en la ejecución de programas por parte del microprocesador, donde no es posible continuar realizando operaciones porque no se dispone aún de la instrucción siguiente a ejecutar o de algún dato que necesita una operación. El acceso a los contenidos en memoria es, por tanto, uno de los cuellos de botella fundamentales que limitan el rendimiento de los procesadores actuales.

Una de las técnicas utilizadas para reducir el tiempo de acceso a los contenidos consiste en utilizar una memoria más rápida como almacenamiento intermedio entre el procesador y la memoria DRAM. Esta memoria intermedia se denomina memoria cache, o simplemente cache.

Una cache se implementa con una tecnología microelectrónica de mayor rendimiento que la memoria DRAM, por lo general la misma que los microprocesadores, por lo que el acceso a sus contenidos es más rápido. Por contra, esta tecnología tiene una densidad de almacenamiento menor y un gasto energético mayor, por lo que la capacidad de una cache no puede ser la misma que la de la memoria DRAM. En la cache se almacena sólo un subconjunto de los contenidos de la memoria DRAM. Si un contenido demandado está presente en la cache, se suministra desde ahí con un tiempo de acceso reducido; si no, se suministra desde la memoria DRAM, con tiempos más largos. En el primer caso se habla de un acierto en la cache, en el segundo de un fallo en la misma.

9.2 Localidad y gestión de las caches

Cuanto más a menudo está el contenido demandado presente en la cache, es decir, cuanto más tasa de acierto tenga, mayor será la reducción en el tiempo de acceso medio a los contenidos que experimenta el procesador. La gestión de los contenidos de la cache es, por lo tanto, un elemento central en la reducción del tiempo de acceso conseguida gracias al añadido de una cache. Dicha selección de contenidos no parece trivial, ya que debe realizarse antes de que se produzca el acceso por parte del procesador. Es necesario anticipar los contenidos que va a demandarse o referenciarse en un futuro próximo para poder tenerlos en la cache, y que ésta cumpla su función.

Para realizar una gestión de contenidos efectiva se aprovechan dos propiedades observadas en los programas: la localidad temporal y la localidad espacial. La localidad temporal consiste en que, cuando

se referencia una dirección de memoria, es probable que dicha dirección se referencie en un futuro cercano. La localidad espacial consiste en que cuando se referencia una dirección de memoria, es probable que direcciones de memoria cercanas se referencien pronto. La existencia de localidad implica que hay una distribución no uniforme de los accesos a memoria que realiza un programa, y por lo tanto se puede predecir con razonable exactitud qué accesos futuros va a realizar un procesador basándose en sus accesos recientes.

Para aprovechar la localidad temporal, cuando un procesador accede a una dirección de memoria de DRAM, el contenido recuperado se almacena en la cache, con la intención de que esté presente para un próximo acceso, que se prevé cercano en el tiempo. Para aprovechar la localidad espacial, las caches reciben y almacenan no ya bytes o palabras sueltas sino bloques de memoria, que son conjuntos de contenidos de direcciones de memoria adyacentes. Estos contenidos cercanos se prevén que serán necesitados en un futuro cercano.

Si bien lo expuesto en el apartado anterior acerca del momento en que se inserta un contenido en una cache es un caso habitual, no es la única solución. Además, es necesario tener un mecanismo que decida qué bloque se expulsa o reemplaza cuando se inserta un contenido nuevo. Al algoritmo de decisión de cuándo y qué contenidos insertar en una cache, y cuáles reemplazar, se conoce como política de inserción y reemplazo. Al bloque reemplazado se conoce como bloque víctima.

9.3 Jerarquía de memoria

Cuanto mayor es el tamaño de una cache, y por lo tanto más bloques puede almacenar, mejor puede aprovechar la localidad presente en un programa, ya que conserva durante más tiempo los bloques referenciados recientemente. Esto incrementa su tasa de acierto, y disminuye el tiempo medio de acceso a los contenidos que experimenta el procesador.

La búsqueda de una mayor tasa de acierto lleva al diseño de caches lo mayores posible. Sin embargo, debido a la tecnología microelectrónica utilizada, existe una relación inversa entre el tamaño de una cache y su tiempo de acceso, y directa entre en tamaño y su consumo. Cuanto mayor es una cache, más lento es y más consume el acceso a la misma. Una cache grande acierta más, pero responde más lentamente y con mayor gasto energético. En la búsqueda de un compromiso entre tamaño, tiempo de acceso y consumo, se implementan varios niveles de cache entre un procesador y la memoria DRAM, formando una jerarquía de memoria.

La Figura 15 muestra un esquema típico con 3 niveles de cache. El primer nivel (L1) es el más pequeño, rápido y eficiente. Cuando un acceso falla, se realiza la búsqueda en el segundo nivel (L2), que contiene más bloques pero es más lento y consume más. Si también falla, se accede al tercer nivel (L3), aún mayor, más lento y menos eficiente, y si éste falla se accede a la memoria DRAM, el nivel más lento y que más consume.

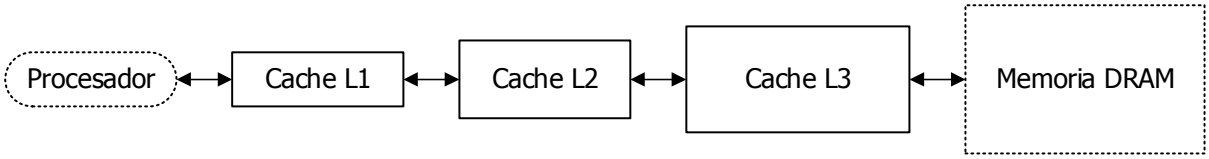
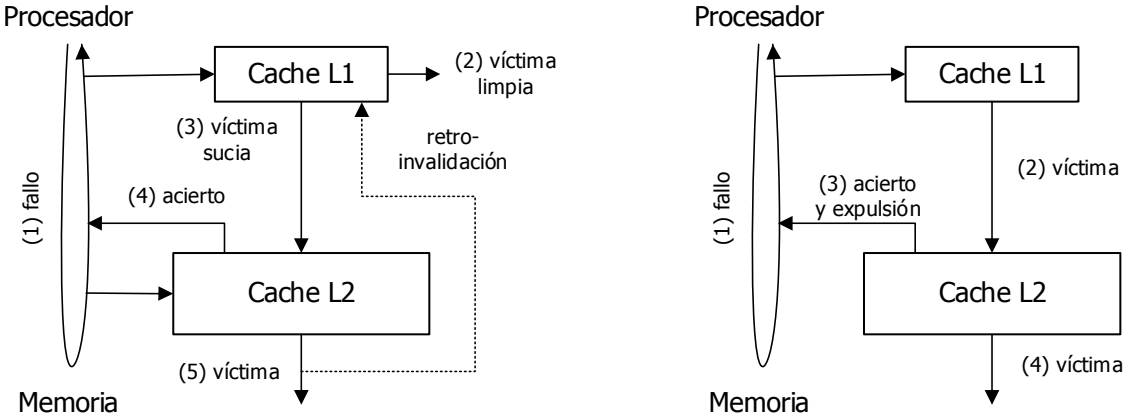


Figura 15: Esquema de cache multinivel.

9.4 Relaciones entre contenidos: inclusión y exclusión

En el esquema original con una única cache, ésta contiene un subconjunto de los contenidos de la memoria principal. Al extender el modelo hacia una jerarquía de memoria con múltiples niveles de cache, la aproximación inicial es mantener esa relación también entre los diferentes niveles de cache. Por ejemplo, en una jerarquía de dos niveles de cache, los contenidos de la cache L1 son un subconjunto de los de la cache L2. Esta relación entre los contenidos se denomina inclusión, y caches inclusivas a las que la emplean.

La Figura 16 muestra a la izquierda un esquema simplificado del funcionamiento de una jerarquía de memoria de 2 niveles de cache donde la cache L2 es inclusiva de L1. Cuando se accede por primera vez a un bloque, éste se recupera de la memoria DRAM, y se envía a los dos niveles de cache (1). Cuando un tiempo después, el bloque se expulsa de L1 porque es reemplazado por otros, pasa a estar presente sólo en la cache L2 (2). Si el bloque ha sido modificado por una escritura del procesador, es necesario enviar el contenido actualizado a la cache L2, para que no se pierda (3). Subsiguientes accesos enviarán el contenido de nuevo a la cache L1 sin expulsarlo de la L2 (4), y así sucesivamente. Si el bloque se expulsa de la cache L2 cuando aún está en la cache L1, es necesario invalidarlo en ésta, lo que se denomina retro-invalidación (5).



Jerarquía en inclusión **Jerarquía en exclusión**

Figura 16: Esquemas de jerarquías de memoria con dos niveles de cache.

Cuando la relación entre dos niveles de cache es de inclusión, los contenidos de la cache de menor tamaño se almacenan también en la otra, es decir, se encuentran duplicados. Dado que todos los

accesos a dichos contenidos van a provocar aciertos en la cache de menor tamaño y nivel, el espacio en la otra se encuentra desaprovechado. Se puede liberar ese espacio y utilizarlo para otros contenidos, con lo que la cache de mayor nivel gana en tasa de acierto. Cuando un bloque de memoria sólo puede estar en uno de los dos niveles de cache, la relación existente se denomina exclusión, y caches exclusivas a las que la emplean.

La Figura 16 muestra a la derecha un esquema similar al anterior, con la diferencia de que la cache L2 es exclusiva de L1. Cuando se accede por primera vez a un bloque desde la memoria DRAM, éste se envía sólo a la cache L1 (1). Cuando un tiempo después, el bloque se expulsa de L1 porque otros necesitan espacio, siempre se envía para su almacenamiento en L2 (2). Si más tarde es accedido de nuevo, se envía de L2 a L1 y el espacio se libera en L2, dejando un hueco que será aprovechado por otros bloques (3). Si el bloque se expulsa de la cache L2 (4), no hay nada que invalidar en L1.

Una cache exclusiva presente la ventaja de una tasa de acierto mayor y ausencia de retro-invalidaciones, pero también el inconveniente de que todos los bloques reemplazados en el nivel inferior han de enviarse al siguiente cuando se expulsan, mientras que en una cache inclusiva sólo es necesario hacerlo si el contenido ha sido modificado. Ello implica una necesidad de mayor ancho de banda en la conexión.

9.5 Jerarquía de memoria en sistemas multiprocesador on-chip

En los últimos años, los sistemas multiprocesador con memoria compartida que incluyen varios procesadores en un mismo chip o circuito integrado están muy extendidos. Su presencia es mayoritaria en el mercado tanto en servidores de alto rendimiento como en sistemas de sobremesa, dispositivos móviles y sistemas embebidos. En el contexto de este tipo de sistemas se desarrolla el presente trabajo.

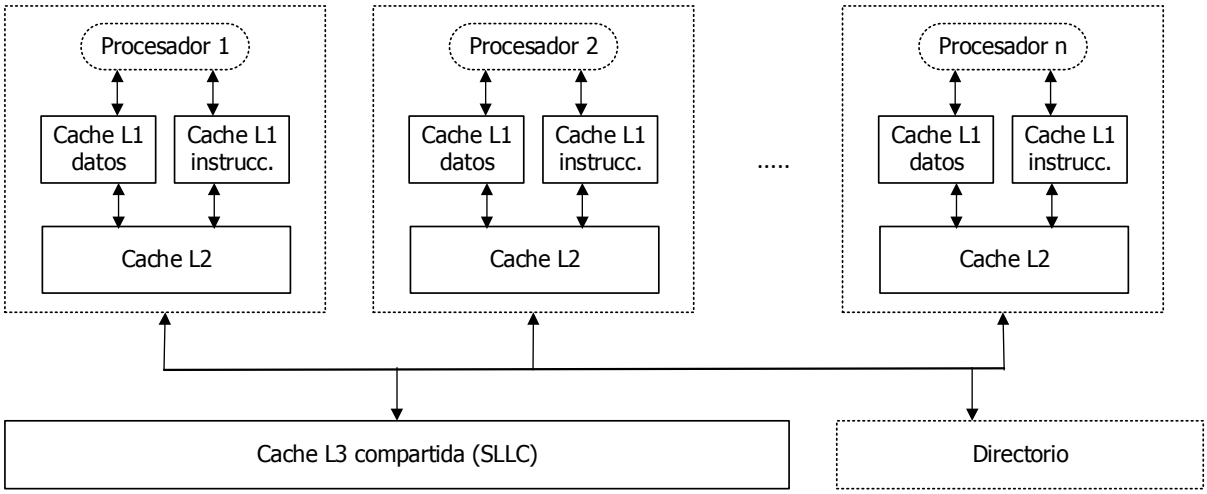


Figura 17: Esquema de jerarquía de memoria en sistemas multiprocesador on-chip.

La Figura 1 muestra una jerarquía de memoria típica de estos sistemas, con tres niveles de cache. Los dos primeros niveles de cache a los que accede un procesador dado son privados y particulares de

dicho procesador, es decir, sólo son utilizados por ese procesador en concreto. Dentro del chip, se encuentran localizados muy cerca o incluso entremezclados con los circuitos del procesador, para evitar retardos de transferencia de datos. El primer nivel además se encuentra dividido entre accesos a instrucciones y accesos a datos. Esto se hace así porque datos e instrucciones son contenidos que se acceden desde diferentes estructuras internas del procesador, y al acceder en paralelo a dos caches se consigue menor latencia y más ancho de banda.

El tercer y último nivel de cache se encuentra en cambio compartido entre todos los procesadores, denominándose cache compartida de último nivel o SLLC (acrónimo de Shared Last Level Cache). Dentro del chip, esta cache compartida se encuentra localizada en su propio espacio dedicado, aparte de los procesadores y conectado a ellos a través de una red interna.

Los tamaños de los diferentes niveles se diseñan intentando minimizar el tiempo y la energía de acceso resultante para la jerarquía completa, y atendiendo al coste en cuanto a espacio. Un tamaño típico hoy en día para cada cache L1 privada son 32 KB, para cada L2 privada son 256 KB, y para una SLLC son 8 MB, si bien las cifras varían en función del diseño.

En un sistema multiprocesador con memoria compartida, aparece la necesidad de gestionar la compartición de los contenidos entre los diferentes procesadores. Cuando dos procesadores acceden a un mismo bloque de forma concurrente, dicho bloque se encuentra replicado en las caches privadas. A partir del momento en que un procesador escribe y modifica el bloque, los demás deben obtener dicho valor modificado cuando vuelven a acceder al contenido, de forma que la visión del mismo sea única y coherente, como si no hubiera caches intermedias y se accediera a la memoria de forma atómica. Al mecanismo que asegura la coherencia de contenidos se denomina protocolo de coherencia. Para ello, otorga y anula permisos de acceso a las diferentes caches, replica los cambios producidos, etc.

Una manera de implementar un protocolo de coherencia consiste en mantener un estado centralizado de cada bloque presente en las caches, con información de qué caches contienen el bloque y con qué permisos concretos. Si la SLLC es inclusiva, esta información puede mantenerse en la propia cache, ya que en ella están presentes todos los bloques de la jerarquía. Si no, esta información centralizada se mantiene en una estructura denominada directorio, que puede verse también en la Figura 1.

10 Anexo B: Plan de trabajo

10.1 Cronograma

Este trabajo fin de master se ha realizado en diferentes fases, a lo largo del curso lectivo 2013-14. La Figura 18 muestra un cronograma resumen de las mismas.

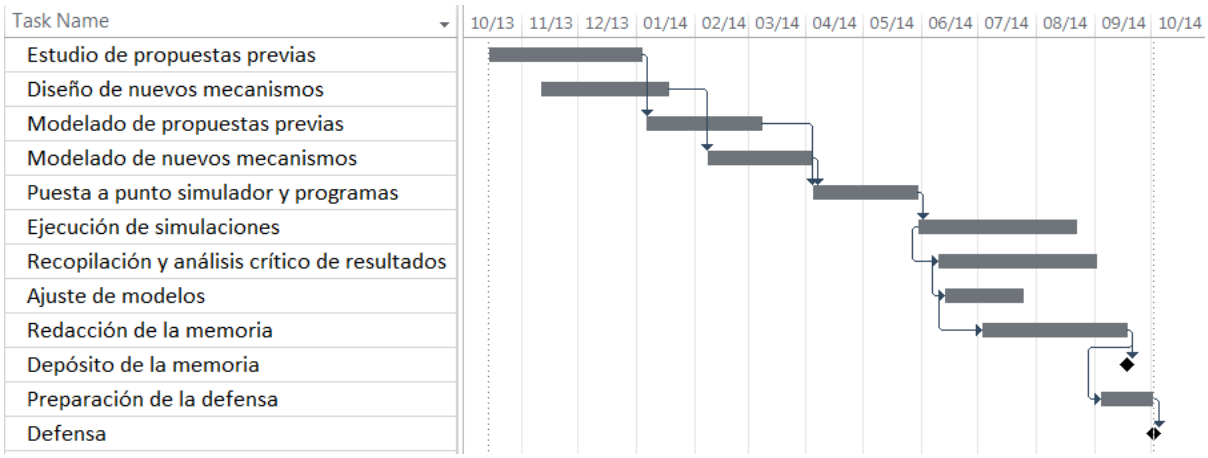


Figura 18: Cronograma del trabajo fin de master.

10.2 Fases del trabajo

A continuación se describen las fases del trabajo, y las actividades realizadas en cada uno de ellos:

- Estudio de propuestas publicadas previamente: Se localizan y estudian los artículos publicados relativos a la misma problemática. Se realiza un análisis crítico sobre los mismos, y de las fortalezas y debilidades de sus propuestas.
- Diseño de nuevos mecanismos de selección de contenidos: Se diseña la Reuse Cache Exclusiva, evolución del diseño de la Reuse Cache inclusiva ya publicada, que se utilizará a la hora de realizar comparativas. Se diseña el nuevo mecanismo de bypass basado en el Detector de Reuso.
- Modelado de nuevos mecanismos de selección de contenidos: Se modela sobre Simics y GEMS la Reuse Cache Exclusiva y el mecanismo con Detector de Reuso.
- Modelado de propuestas publicadas: Se modela sobre Simics y GEMS la SLLC exclusiva que se utiliza como base en las comparaciones, así como la propuesta CHAR ya publicada.
- Puesta a punto de las herramientas de simulación y de los programas de prueba: Se evoluciona el simulador, previamente utilizado en otros trabajos del área de Arquitectura de Computadores de la Universidad de Zaragoza, pasando de un modelo de procesador en orden a un modelo de procesador superescalar fuera de orden. Para ello, se incorpora el módulo Opal de GEMS, y se implementa sobre el mismo la simulación de nuevas instrucciones del juego de instrucciones VIS de SPARC V9, de inicio no soportadas por el módulo y necesarias para los programas de prueba. Se adaptan los programas de prueba de la suite SPARC para su uso con el simulador mejorado.

- Ejecución de programas de simulación sobre un clúster de computación: Se definen los diferentes experimentos a realizar y se ejecutan las simulaciones sobre los clústeres *hermes* y *atps*.
- Recopilación y análisis crítico de resultados: Se toman de la salida del simulador las medidas útiles y se recopilan para su análisis. Para ello, se modela e implementa una base de datos relacional que sirva para almacenar tanto las características de los sistemas modelados como los experimentos, sus parámetros y sus resultados. También se crean los programas de carga de valores en la base de datos a partir de los ficheros de resultados del simulador, y las consultas SQL de extracción de datos. Esta base de datos centraliza toda la información del proyecto, facilita la replicabilidad de los experimentos y permite un análisis más rápido de los resultados. Tras la recopilación de resultados, se realiza un análisis crítico de los mismos.
- Ajustes de modelo: A partir del análisis de los resultados, se realiza el ajuste necesario en los modelos, como corrección de errores, modificación de parámetros, etc.
- Redacción de la memoria: Se redacta y revisa la memoria y sus anexos.
- Depósito de la memoria.
- Preparación de la defensa ante el tribunal: Se prepara y ensaya la presentación a realizar.
- Defensa ante el tribunal.