MODIFICACIÓN DEL COMPORTAMIENTO DE LA POLÍTICA DE REEMPLAZO EN EL ÚLTIMO NIVEL DE CACHE EN BASE A DIFERENTES TIPOS DE EVENTOS

JOSÉ FERNANDO NAVACERRADA SANTIAGO

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA, FACULTAD DE INFORMÁTICA,
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Máster en Ingeniería de Computadores

Julio 2014

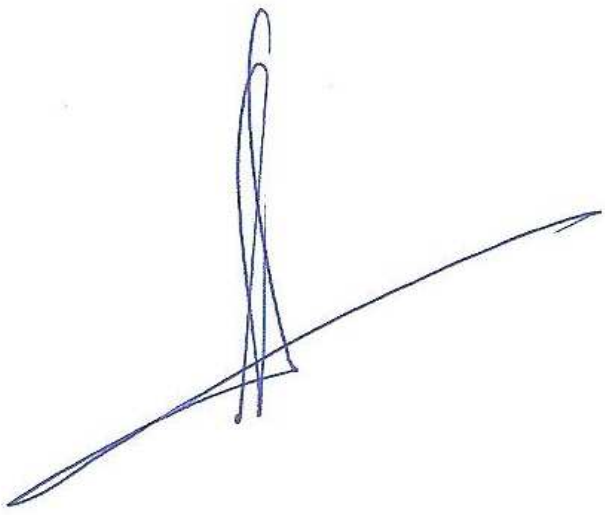
Directores:
Daniel Chaver Martínez
Fernando Castro Rodríguez
Colaborador:
Roberto Alonso Rodríguez Rodríguez
Calificación:
Sobresaliente (9)

Autorización de difusión

JOSÉ FERNANDO NAVACERRADA SANTIAGO

Julio 2014

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "MODIFICACIÓN DEL COMPORTAMIENTO DE LA POLÍTICA DE REEMPLAZO EN EL ÚLTIMO NIVEL DE CACHE EN BASE A DIFERENTES TIPOS DE EVENTOS", realizado durante el curso académico 2013-2014 bajo la dirección de Daniel Chaver Martínez y Fernando Castro Rodríguez, en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.



A mis padres por estar a mi lado siempre y a Laura por su apoyo constante y soportar largas horas de ausencia.

Agradecimientos

A Daniel Chaver Martínez y Fernando Castro Rodríguez por haber tenido tanta paciencia conmigo durante todos estos años. Gracias por haber aceptado dirigir el presente trabajo de investigación después de tantas idas y venidas, orientándome en cada momento y alentándome a seguir adelante cada vez que los resultados no eran los esperados. Mención especial merece también Roberto Alonso Rodríguez Rodríguez por su extraordinario conocimiento sobre el simulador y por haberme brindado su apoyo en todas las dudas que me han surgido en relación con el mismo, sin su ayuda dudo mucho que el trabajo se hubiera podido terminar a tiempo.

Muchas gracias a los tres por permitirme trabajar con vosotros durante estos meses para poder así desarrollar mi trabajo final de máster.

Resumen

Para paliar la diferencia en la evolución tecnológica existente entre el procesador y la memoria de un sistema computacional se hace uso de la jerarquía de memoria. El objetivo de ésta no es otro sino aproximar la velocidad con la que la memoria es capaz de servir los datos a la velocidad a la que estos son demandados por el procesador.

Un rendimiento eficiente de la cache es de vital importancia debido a que es la parte de la jerarquía que se encuentra dentro del chip. Cuando se sobrepasa éste y los datos tienen que ser demandados a memoria principal, la velocidad con la que estos son servidos decrece en varios órdenes de magnitud. Existen multitud de políticas para gestionar distintos aspectos de la cache, como el emplazamiento de los bloques, la gestión de las escrituras, etc. La política que determina qué bloques se deben mantener y cuales descartar cuando existe la necesidad de incorporar nuevos bloques demandados por niveles más cercanos al procesador en la jerarquía de memoria recibe el nombre de política de reemplazo.

El presente trabajo de investigación pretende contribuir a mejorar la eficiencia del último nivel de cache (*Last Level Cache* – LLC) haciendo uso de una política de reemplazo LRU (*Least Recently Used*) modificada. Para ello, basándonos en nuestras observaciones en cuanto a la existencia de distintos tipos de inserción y promoción en las aplicaciones, proponemos modificar las componentes de la política LRU que gestionan dichos eventos, de forma que la decisión se tome en base al tipo de inserción o promoción que tenga lugar. Se proponen tres alternativas:

La primera consistirá en realizar las modificaciones oportunas en la inserción/promoción, para mejorar la eficiencia de una cache de último nivel con la configuración típica de un procesador actual.

Una segunda posibilidad consistirá en tratar de acercar el rendimiento de una cache reducida al conseguido por una cache de tamaño típico, proponiendo de nuevo una gestión específica de cada tipo de inserción/promoción.

Por último, se proponen modificaciones a la política empleada en el nivel compartido de un *chip multiprocessor* una vez más con el objetivo de mejorar su rendimiento.

Palabras clave

Jerarquía de memoria, política de reemplazo, LRU, writeback, inserción, promoción, último nivel de cache, single-core, multi-core, mejora de rendimiento.

Abstract

In order to mitigate the technological difference between the processor and the memory of a computer system, the memory hierarchy was introduced. Its main purpose is to approximate the speed at which the memory is capable of serving data to the rate at which data are demanded by the processor.

Being cache levels usually integrated within the chip in nowadays computers makes performance of these levels a key aspect for a good performance of the whole system. When a miss occurs at the last cache level, data has to be searched from main memory, resulting on an important increase on the effective memory latency. There are many policies to manage various aspects of the cache, such as the location of the blocks, the updating of modified information, etc. The policy that determines which blocks should be kept/discarded from the cache when new blocks are requested by the processor is called the replacement policy.

This research aims to improve the performance of the last level cache (LLC) when it uses a modified LRU (Least Recently Used) replacement policy. For this purpose, based on our observation of the existence of different insertion/promotion types in the applications, we propose to modify the LRU policy so that the decision is taken according to the type of insertion/promotion that takes place. In this work we have addressed three different alternatives:

We first performed changes to the insertion/promotion policies in order to improve the performance of a LLC with the typical configuration of a current processor.

Secondly, making again changes to the management of the insertion/promotion of the blocks, we aimed to approximate the performance of a small cache to the performance achieved by a typical cache.

Finally, we proposed modifications to the insertion/promotion components of the replacement policy used on the Shared LLC of a chip multiprocessor, again with the goal of improving its performance.

Keywords

Memory hierarchy, replacement policy, LRU, writeback, insertion, promotion, last level cache, single-core, multi-core, performance.

Índice de contenidos

Índice de figuras

Índice de tablas

Índice de ecuaciones

Capítulo 1

Introducción

Desde hace tiempo es bien conocida la gran diferencia en cuanto a evolución tecnológica que ha seguido, en un sistema computacional, el procesador con respecto al sistema de memoria. Diferencia que año a año se ha ido acentuando pues mientras que el procesador doblaba su rendimiento aproximadamente cada dieciocho meses, la memoria necesitaba alrededor de diez años para conseguir este hito, recibiendo esta problemática el nombre de *memory Wall*. La figura 1.1 ilustra tal comportamiento.

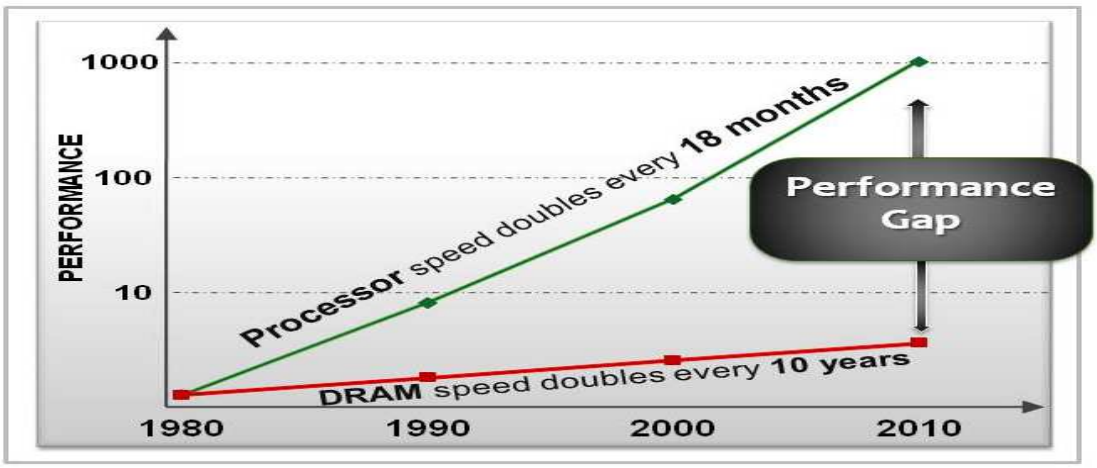


Figura 1.1: Diferencia evolutiva del procesador con respecto a la memoria.

Con el ánimo de paliar estas diferencias y aproximar la velocidad con la que la memoria puede servir los datos a la velocidad con la cuál estos son demandados por el procesador surge la jerarquía de memoria.

La jerarquía de memoria está compuesta por varios niveles, véase la figura 1.2, en los que a medida que nos alejamos de los registros del procesador contamos con memorias mucho más grandes en cuanto a capacidad de datos y en las que el coste por bit es mucho más barato pero por lo contrario son mucho más lentas.

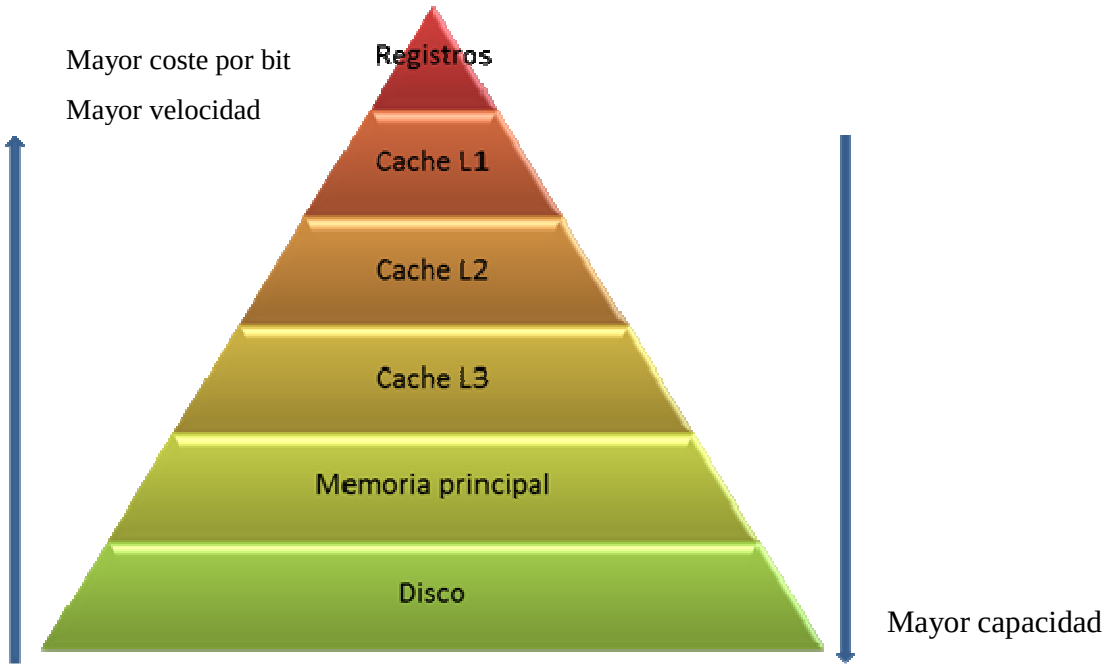


Figura 1.2: Jerarquía de memoria

Si nos encontramos en un nivel determinado de memoria, vamos a referirnos a un nivel superior en la jerarquía a todo aquel que se encuentra más cercano de los registros del procesador. En la misma tesitura entenderemos por un nivel inferior en la jerarquía aquel que se encuentra más cercano a disco. Hacer notar que esta nomenclatura puede ser especificada completamente a la inversa dependiendo de la bibliografía que se consulte. Es por ello que se cree necesaria tal observación para clarificar la forma en la que en adelante nos referiremos a niveles superiores o inferiores de memoria respecto de uno dado.

El propósito fundamental de la jerarquía de memoria es simular un sistema de memoria en el que hacer creer al procesador que se dispone de una memoria tan grande como la mayor

de ellas y tan rápida como la más veloz. Para la consecución de este fin, se deberán tomar una serie de decisiones en cada uno de estos niveles, decisiones encaminadas a facilitar que en todo momento el bloque en el que esté el dato que solicita el procesador se encuentre en el nivel más cercano posible a éste. De esta forma podrá ser servido con mayor rapidez y se contribuirá a disminuir la desventaja tecnológica con la que parte.

1.1 Políticas para la gestión de la cache

En los sistemas computacionales de hoy en día, la jerarquía de memoria juega un papel fundamental a la hora de poder aprovechar las capacidades que ofrecen los procesadores actuales. Con vistas a obtener el comportamiento más eficiente posible y como ya se ha expuesto anteriormente, en la cache se deberán tomar una serie de decisiones entre las que se encuentran: qué lugar elegir para un bloque entrante a cache, qué bloques pueden o no estar en un determinado nivel, cuándo actualizar en niveles inferiores bloques que han sido desechados de un nivel determinado y necesitan ser actualizados, qué bloque elegir en cada momento a la hora de suceder una expulsión, etc.

Este conjunto de decisiones recibe el nombre genérico de política de gestión de la cache. Conforme a ello, podemos hablar de política de emplazamiento, política de inclusión, política de actualización y política de reemplazo. A continuación se describe la funcionalidad de cada una de ellas.

1.1.1 Política de emplazamiento

Cuando se peticiona en un determinado nivel de cache *i* un bloque que éste no tiene, se produce un fallo. Siempre que se da tal circunstancia, la política de emplazamiento determinará el marco que ocupará el nuevo bloque incorporado a la cache. Según esto se establecen tres posibilidades básicas:

Correspondencia directa: En la que cada bloque de memoria sólo puede ser ubicado en un único marco de la cache. Éste es determinado por el *número del bloque de memoria principal* modulo *número de marcos de capacidad de la cache*.

Completamente asociativa: En la que cada bloque de memoria principal puede ir a cualquier marco de la cache.

Asociativa por conjuntos: En la que la memoria cache se divide en distintos conjuntos de marcos. De esta forma, los bloques de memoria principal se asignan mediante correspondencia directa a los conjuntos de la cache. Una vez en éstos, el bloque puede ser alojado en cualquiera de los marcos de los que se compone el mencionado conjunto. Se puede decir que la política de emplazamiento asociativa por conjuntos es una política intermedia entre las dos anteriores que trata de obtener las mejores características de cada una de ellas. Por un lado el coste de hardware es similar al de una política de correspondencia directa y por otro el rendimiento es similar al de una política completamente asociativa.

La figura 1.3 representa gráficamente cada una de las políticas de emplazamiento expuestas.

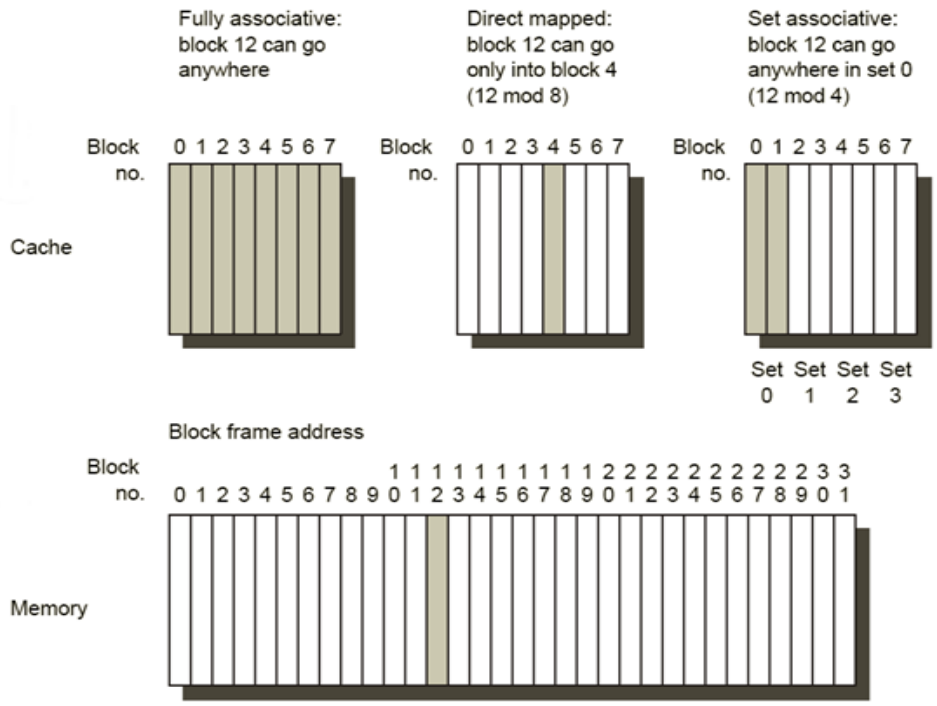


Figura 1.3: Políticas de emplazamiento

1.1.2 Política de inclusión

Si tenemos en cuenta la propiedad de inclusión, una política es *inclusiva* cuando se fuerza a mantener en un nivel inferior de cache los bloques que se encuentran en el nivel inmediatamente superior. Si no se exige esta característica, es decir, un bloque que se encuentra en un nivel determinado de cache puede o no estar en el nivel inmediatamente inferior, se dice que es una política *no inclusiva*. Por otra parte, si se decide que un bloque que se encuentra en un nivel determinado no pueda estar en el nivel inmediatamente inferior, hablamos de una política *exclusiva*.

Considerando las ventajas que la utilización de cada una de ellas nos puede reportar podemos decir que, mientras el uso de una política inclusiva simplifica en gran medida la gestión de la coherencia, la utilización de una política exclusiva maximiza el tamaño efectivo de

la cache debido a la no redundancia de información. Por último, la ventaja si hacemos uso de una política *no inclusiva* es que es mucho más fácil de gestionar, pues no hay que preocuparse de lo que está o no está en otros niveles de cache.

1.1.3 Política de actualización

La política de actualización determinará cuando se actualiza la información, tanto en niveles inferiores de cache como en memoria principal, al haberse producido una escritura en un determinado nivel de cache.

Según esto se pueden considerar dos posibilidades:

- La política de escritura inmediata, o *writethrough*, en la que cada bloque modificado en un nivel determinado de cache, se actualizará en este nivel así como en el nivel inmediatamente inferior dentro de la jerarquía.

- La política de escritura aplazada, o *writeback*, que consistirá en actualizar cada bloque modificado sólo en el nivel en que se encuentre. Dicha modificación será actualizada en el nivel inmediatamente inferior en la jerarquía cuando este bloque abandone el nivel en el que se encuentra.

Si bien es verdad que una política de actualización como *writethrough* evita la existencia de datos desactualizados, también hay que señalar que el tráfico generado es mucho mayor que si utilizamos *writeback*.

1.1.4 Política de reemplazo

Antes de nada, hacer notar que la utilización de una política de reemplazo únicamente tiene sentido cuando a su vez se utiliza una política de emplazamiento asociativa, que por otra parte es lo más habitual. En una política de emplazamiento directo, esta política carece de sentido pues queda determinado claramente en cada caso el bloque que se expulsará de la cache sin hacer ningún tipo de cálculo.

Debido a la incapacidad que tiene una memoria cache de mantener todos los bloques de las aplicaciones que en un momento determinado se estén ejecutando en el sistema, se debe establecer una política que determine qué bloques se deben mantener y cuales descartar cuando existe la necesidad de incorporar nuevos bloques demandados por niveles superiores en la jerarquía de memoria. La encargada de ocuparse de estas tareas es la política de reemplazo. Entre las más extendidas y utilizadas como referencia podemos citar: FIFO (First In First Out) [1], LFU (Least Frequently Used), LIFO (Last In, First Out) [2] y LRU (Least Recently Used) [1].

Siempre que la inserción de un bloque suponga la eliminación de otro en la cache, la política de reemplazo debe decidir el bloque a reemplazar. Las diferentes políticas existentes tratan de identificar cuál será la víctima más adecuada, teniendo en cuenta la información recogida, en diferentes momentos (fases) de su ciclo de vida en la cache. Habitualmente se recoge esta información en la inserción y en la promoción de los bloques, modificando su *estado de reemplazo*.

De acuerdo con las decisiones a tomar en cada fase, esta política puede ser subdividida en tres sub-políticas:

1.1.4.1 Sub-política de inserción

Esta sub-política determinará el *estado de reemplazo* inicial de un bloque cuando éste llega a la cache. Esta decisión variará dependiendo de la política que se utilice. Por ejemplo, la política LRU determinará máxima prioridad de permanencia en cache para los nuevos bloques ocupando la posición *mru* de la *recency stack* (ambos conceptos se definirán en la sección 1.2) mientras que por el contrario en la política FIFO los nuevos bloques insertados ocuparán la primera posición para ser elegidos por la subpolítica de victimización y ser evacuados de la cache. Por poner otros ejemplos que muestren la variedad de la que hacen uso distintas políticas a la hora de determinar el *estado de reemplazo* cuando un bloque entra en la cache, políticas como NRU o RRIP [3] asignan un valor que determinará una predicción lejana de estos bloques en su reúso.

1.1.4.2 Sub-política de promoción

Esta sub-política tendrá la misión de actualizar el *estado de reemplazo* de un bloque cuando éste experimente un acierto.

Si seguimos con el ejemplo de uso por parte de varias políticas, mientras LRU actualiza el *estado de reemplazo* de los bloques que han recibido un acierto llevándolos a la posición *mru* de la *recency stack*, políticas como NRU y RRIP asignan un valor que indique una predicción más cercana en su uso.

1.1.4.3 Sub-política de victimización

Esta sub-política es la encargada de elegir un bloque para reemplazar, comparando los *estados de reemplazo* de los bloques candidatos.

Una vez más, si comparamos cómo cada política gestiona esta decisión, podemos decir que LRU reemplazará el bloque menos recientemente usado, FIFO reemplazará el primer bloque que llegó a la cache, LIFO reemplazará el último bloque que llegó y otro tipo de políticas como NRU o RRIP tomarán esta decisión entre los bloques que tengan un *estado de reemplazo* con un valor de predicción de reúso más lejano.

1.2 Política de reemplazo *Least Recently Used* y patrones de acceso

Se cree conveniente, si no explicar en detalle, al menos sí analizar los conceptos principales que intervienen en una de las políticas de reemplazo más extendidas como es *Least Recently Used (LRU).*

LRU, además de ser la política de reemplazo que sirvió como base a este proyecto, ha sido tomada como referencia para la evaluación de resultados de multitud de trabajos de investigación y es la política de reemplazo empleada en las caches de la mayor parte de

computadores, tanto de propósito general como empotrados, que existen hoy en día en el mercado.

Si se utiliza una política de emplazamiento asociativa por conjuntos, el nivel *i* de cache será dividido en conjuntos capaces de alojar cada uno un número determinado de bloques. Hacer notar que, habitualmente, a un conjunto del nivel *i* le corresponderá albergar de un nivel inferior a éste, un número de bloques mayor que el número de marcos de que se dispone para ese conjunto.

En el momento de producirse un fallo, el bloque es demandado a un nivel inferior en la jerarquía. Cuando el bloque es servido, es insertado en un marco libre si es que existe y si no habrá que expulsar un bloque de los en ese momento ocupan el conjunto para dar cabida al entrante.

En base a determinar en cada caso cuál es el bloque candidato a abandonar el nivel de cache, la política LRU hace uso de una pila llamada *Recency Stack* en la que se ordenan los bloques en función de su orden de referencia, véase la figura 1.4. Cada vez que un bloque recibe un acierto o es insertado, pasará a la posición *Most Recently Used (MRU)* de la *Recency Stack*, desplazando el resto de los bloques una posición hacia la entrada *Least Recently Used (LRU)*. En cambio, cuando tenga que producirse una expulsión para dejar hueco a un bloque entrante, el bloque candidato a evacuar será el que ocupe la posición LRU. De esta forma, la política LRU preserva en un nivel de cache aquellos bloques que ofrecen una mayor localidad temporal en su reúso.

Figura 1.4: Recency Stack para política de reemplazo LRU

Según la clasificación de patrones de acceso mostrada en [3], la política LRU muestra un gran comportamiento cuando, como se dijo anteriormente, la carga de trabajo ofrece una alta localidad temporal en el reúso de los datos. Este comportamiento es el que ocurre en el patrón explicado a continuación.

Patrones de acceso amigables con LRU: Como indica la ecuación 1.1, son patrones que ofrecen una frecuencia de reúso casi inmediata. Para una mejor comprensión de este ejemplo así como de los expuestos a continuación, entiéndase por a_i la dirección de una línea de cache, $(a_1,\ldots,a_k)$ denotará una secuencia de accesos para *k* direcciones y $P\varepsilon(a_1,\ldots,a_k)$ será una secuencia temporal que ocurre con una probabilidad ε. Cuando una secuencia temporal se repita N veces será representada como $(a_1,\ldots,a_k)^N$. Para valores de k menores a la capacidad de la cache este patrón de acceso se ve beneficiado por la aplicación de LRU, cualquier otra alternativa podría degradar su rendimiento.

$$(a_1, a_2, \ldots, a_{k-1}, a_k, a_k, a_{k-1}, \ldots, a_2, a_1)^N$$

Ecuación 1.1: Patrón de acceso amigable con LRU (k< capacidad de cache)

Sin embargo, como se observa en los siguientes patrones, tal circunstancia no siempre se da en la aplicación a ejecutar. Incluso es muy normal que en las diferentes fases de la ejecución de una misma aplicación tengan lugar ambas (baja y alta localidad temporal).

Patrón de acceso *Thrashing*: En relación con la ecuación 1.2, se da este nombre a un patrón de accesos cíclico con una longitud *k* que se repite *N* veces. Cuando *k* es menor o igual que el número de bloques que puede albergar la cache, el conjunto de trabajo cabe dentro de ésta (y por tanto LRU funciona correctamente). Sin embargo, cuando k es mayor que el número de bloques, LRU ofrece un rendimiento muy pobre debido al efecto llamado thrashing. Para dichos patrones, LRU no proporciona ningún acierto a menos que el tamaño de la cache sea

incrementado con el fin de poder albergar *k* entradas del patrón de acceso. Cuando la cache disponible es menor que *k* entradas, una política de reemplazo óptima preserva parte del conjunto de trabajo en la cache. Desgraciadamente, LRU es incapaz de hacer esto.

$$(a_1, a_2, \dots, a_k)^N$$

Ecuación 1.2: Patrón de acceso Thrashing (k> tamaño de cache)

Patrón de acceso *Streaming*: Como se muestra en la ecuación 1.3, un patrón de acceso *streaming* ocurre cuando $k \to \infty$, El patrón de acceso no muestra ningún tipo de localidad temporal en sus referencias. Los patrones de acceso streaming pueden caracterizarse como cargas de trabajo con un intervalo de re-referencia tendente a infinito. Debido a esta circunstancia, estos patrones no reciben aciertos de cache sea cual sea la política de reemplazo que se utilice. Como consecuencia se puede decir que, como la decisión en cuanto al reemplazo de un bloque en estos casos es irrelevante, la aplicación de LRU es una decisión tan adecuada como cualquier otra.

$$(a_1, a_2, a_3, a_4, \dots, a_k)$$

Ecuación 1.3: Patrón de acceso Streaming ($k \to \infty$)

Patrones de acceso mixtos: Los patrones de acceso mixtos se caracterizan por cargas de trabajo donde algunas referencias tienen un intervalo de re-referencia (reúso) casi inmediato mientras que otras referencias tienen un intervalo de re-referencia distante. Las ecuaciones 1.4 y 1.5 muestran dos ejemplos que tienen escaneos sobre bloques de cache (resaltados en gris). Un escaneo sucede cuando se produce una búsqueda o actualización de una lista de bloques. Cuando m+k es menor que el número de bloques disponibles en la cache, el conjunto de trabajo cabe en la cache y LRU funciona correctamente. Sin embargo, cuando m+k es mayor

que el número de bloques disponibles, LRU descarta constantemente bloques que pertenecen al conjunto de trabajo, produciéndose continuos fallos después de un escaneo. En ausencia de escaneos, patrones de acceso mixtos prefieren como política de reemplazo LRU. Sin embargo, en presencia de éstos, una política óptima trata de preservar parte del conjunto de trabajo en cache, habilidad de la que ya se ha comentado carece LRU.

$$[(a_1, \ldots, a_k, a_k, \ldots, a_1)^A \; P\varepsilon(a_1, a_2, \ldots, a_k, a_{k+1}, \ldots a_m)]^N$$

Ecuación 1.4: Patrón de acceso Mixto1 (k<tamaño cache y m>tamaño cache, 0<ε<1)

$$[(a_1, \ldots, a_k)^A \; P\varepsilon(b_1, b_2, \ldots, b_m)]^N$$

Ecuación 1.5: Patrón de acceso Mixto2 (k<tamaño cache y m+k>tamaño cache, 0<ε<1)

Para los patrones *thrashing, streaming* y *mixtos* LRU ofrece un pobre rendimiento. El efecto de este mal comportamiento es especialmente pronunciado en el *Last Level Cache (LLC)*, nivel en el que la localidad temporal ya ha sido previamente filtrada y por tanto este tipo de patrones son más habituales. Es por ello que diversos autores han propuesto algoritmos de reemplazo para este último nivel de cache [2,3,4] con el fin de mejorar el rendimiento conseguido por LRU. Incluso, existe la opción de utilizar una política adaptativa por fases, utilizando mecanismos de monitorización como lo son Set Sampling [5] y Set Dueling [4]. El problema se agrava cuando en el LLC confluyen distintas aplicaciones, circunstancia que se da en un entorno *Chip Multi Processor (CMP)* donde la ejecución de diferentes aplicaciones sobre varios cores comparten este nivel. El nivel referenciado recibe entonces el nombre de *Shared Last Level Cache (SLLC),* existiendo de igual forma multitud de propuestas orientadas a mejorar su rendimiento [6,7,8]. En el próximo capítulo se ampliarán estas propuestas.

La importancia de este nivel de cache reside en que es el último nivel de la jerarquía que, como indica la figura 1.5, se encuentra dentro del chip. Se erige pues en un último bastión en el que los datos son servidos al procesador con una alta velocidad. Una vez sobrepasado este nivel, los datos serán solicitados a memoria principal con el consecuente aumento de varios órdenes de magnitud en cuanto a tiempo de acceso a los datos, por lo cual se produce un descenso muy considerable del rendimiento del sistema.

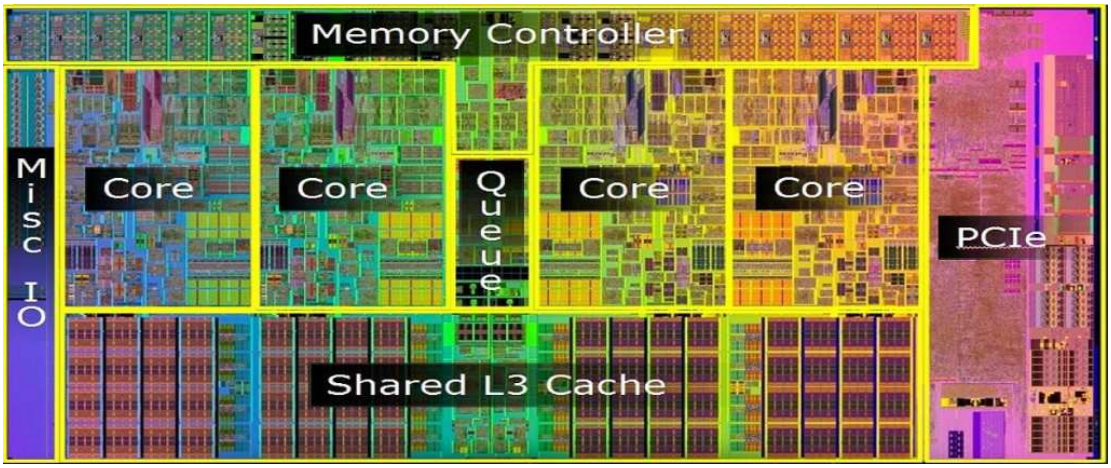


Figura 1.5: Estructura representativa de un Chip Multi Proccesor (CMP).

1.3 Objetivos y organización del trabajo

El presente trabajo de investigación pretende contribuir a obtener una gestión más eficiente del último nivel de cache, tanto en entornos single-core como multi-core, aportando una solución basada en una modificación de la política de reemplazo LRU. Dicha modificación consiste en dar un trato diferenciado por parte de la política a los diferentes eventos de inserción y promoción que se producen a lo largo de la ejecución de una aplicación. Con ello se intenta elevar la tasa de aciertos en este nivel y por tanto evitar en la medida de lo posible que los datos sean demandados a niveles de la jerarquía de memoria que se encuentran fuera del chip, lo que implicaría la ya mencionada pérdida de rendimiento en el sistema.

El resto del documento se organiza como sigue:

- *Capítulo 2*: Recopila la información de los trabajos más recientes en esta área.
- *Capítulo 3*: Se presentan las propuestas realizadas para cumplir con los objetivos propuestos.
- *Capítulo 4*: Se detalla el entorno experimental utilizado.
- *Capítulo 5*: Presentación y análisis de los resultados obtenidos en base a la propuesta hecha en el capítulo 3.
- *Capítulo 6*: Conclusiones y exposición del trabajo futuro que se podría realizar en esta área.

Capítulo 2

Trabajo relacionado

Para la gestión de estas estructuras de memoria que conforman la cache y debido a su reducido tamaño tiene especial relevancia la toma de decisiones respecto a qué bloques almacenar, mantener o llegado el caso desechar. Este tipo de tareas son llevadas a cabo por la política de reemplazo.

Son muy numerosos los trabajos de investigación que se han llevado a cabo hasta el momento sobre la forma de rentabilizar al máximo el rendimiento de la cache como parte fundamental de la arquitectura de memoria. En este capítulo se recogen al respecto varias propuestas representativas tanto en entorno single-core como multi-core, las cuales son expuestas a continuación.

2.1 Mecanismos de evaluación dinámica del rendimiento

Antes de empezar a exponer las políticas a las que se hace alusión en el párrafo anterior se cree conveniente explicar el comportamiento de dos mecanismos recientes, *Dinamyc Set Sampling* y *Set Dueling*, que se utilizan para evaluar y poder comparar de forma dinámica el rendimiento que para una determinada aplicación pueden tener distintas políticas de

reemplazo. Estos mecanismos son frecuentemente utilizados por las políticas propuestas en estos últimos años eligiendo en cada momento el que ofrezca un mejor comportamiento.

2.1.1 Dynamic Set Sampling (DSS)

La alternativa que los autores proponen en [6] para que se puedan tomar el tipo de decisiones a las que hace referencia el párrafo anterior es la de además del *Main Tag Directory* (MTD) de la cache, implementar dos estructuras auxiliares llamadas *Auxiliary Tag Directory* (ATD) cada una de las cuales implementará una política diferente y llevará el control de lo que ocurra con respecto a esa política implementada en un determinado número de conjuntos (no todos los conjuntos de la cache). En el ejemplo de la figura 2.1, se muestra la implementación de una política de reemplazo adaptativa entre dos políticas, LRU y BIP. Esta política utiliza el mecanismo de *Set Sampling* para monitorizar dinámicamente y elegir en cada momento la que ofrezca un mejor comportamiento. En la estructura ATD-LRU se implementará la política LRU (inserta todos los bloques entrantes a la cache en la posición MRU). De la misma forma, en la estructura ATD-BIP se implementará la política BIP (inserta la mayor parte de los bloques entrantes a la cache e la posición LRU y un pequeño porcentaje en la posición MRU). En base al orden normal de petición de los bloques por parte de la cache, un fallo en la estructura ATD-LRU incrementará el contador saturado SCTR. Si por el contrario, el fallo se produce en la estructura ATD-BIP el contador SCTR se decrementará. En base a esto, MTD implementará ambas políticas y consultará el bit más significativo de SCTR para determinar si la cache sigue la política LRU cuando su valor es cero, o bien sigue la política BIP cuando su valor es uno. De esta forma se consigue que la cache siga en todo momento la política que mejor se adapte a la carga de trabajo actual.

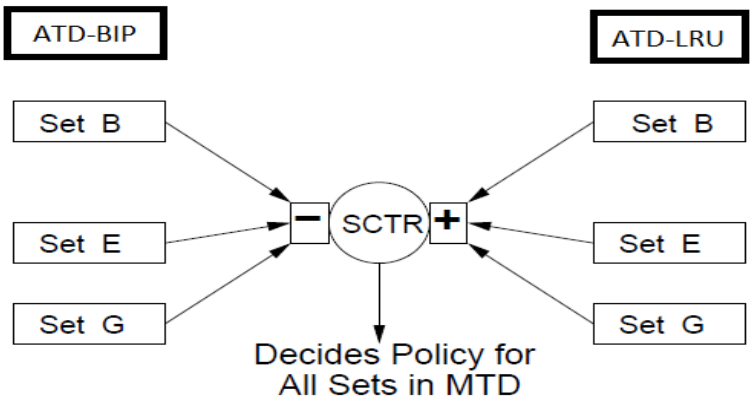


Figura 2.1: Diagrama descriptivo del mecanismo Dynamic Set Sampling.

2.1.2 Set Dueling

Como mejora a DSS, en [4] se propone *Set Dueling*. En lugar de utilizar ATD, *Set Dueling* dedica directamente unos pocos conjuntos de la cache para cada una de las políticas competidoras. En base al comportamiento de estos conjuntos dedicados, la política que incurra en menos fallos será la que establezca la política a seguir por el resto de conjuntos.

La figura 2.2 muestra un ejemplo en el que se propone una política híbrida entre LRU y BIP haciendo uso, para la monitorización del rendimiento de cada una de ellas, de *Set Dueling*. En una cache con dieciséis conjuntos, los conjuntos 0, 5, 10 y 15 están dedicados a la política LRU y los conjuntos 3, 6, 9 y 12 a la política BIP. Cuando ocurre un fallo en alguno de los conjuntos dedicados a LRU el Selector Policy (PSEL), que no es otra cosa sino un contador saturado, incrementará su valor. Si el fallo ocurre en alguno de los conjuntos dedicados a BIP el PSEL será decrementado. Si el bit más significativo de PSEL es cero, el resto de conjuntos (que no están dedicados específicamente a ninguna política) seguirá la política BIP. Por el contrario, si el bit más significativo de PSEL es uno, el resto de conjuntos seguirá la política LRU. Se hace

notar que *Set Dueling* no requiere de ninguna estructura separada de almacenamiento a excepción del contador saturado implementado por PSEL.

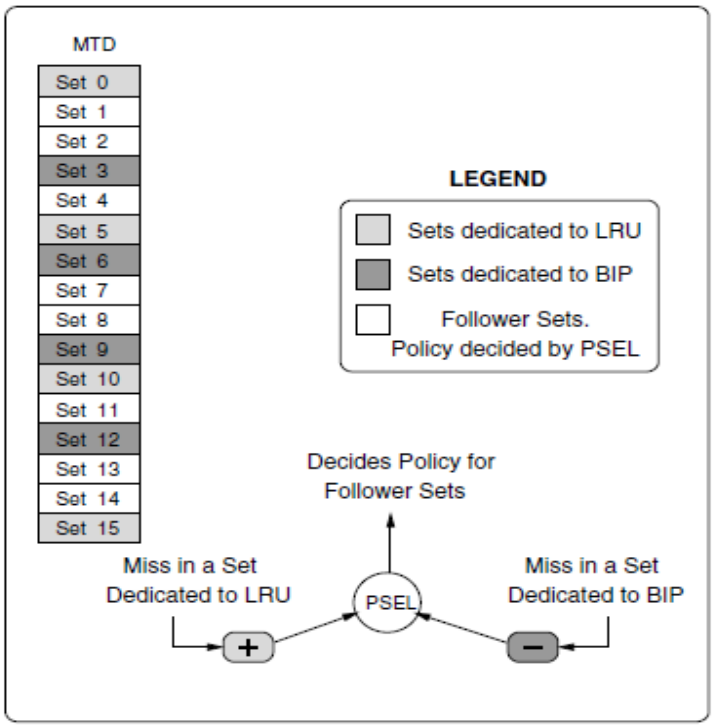


Figura 2.2: Diagrama descriptivo del funcionamiento del mecanismo Set Dueling.

2.2 Políticas de reemplazo

La mejora del comportamiento en la toma de decisiones de la política de reemplazo, es un factor de gran peso a la hora obtener un mejor rendimiento de la cache. La aplicación de este tipo de políticas ha ido evolucionando desde las políticas más clásicas como LRU (Least Recently Used) , LFU (Least Frequently Used), FIFO (First In First Out), Round-Robin o reemplazo aleatorio como se recoge en [10] a otras que se han ido desarrollando a partir de estas clasificándose en diversas familias de políticas orientadas a obtener un mejor rendimiento de la cache.

Este apartado recoge algunas de las propuestas más recientes e innovadoras que hasta el momento se han publicado, tanto orientadas a un solo core como a varios de ellos. Cada una de las cuales está de algún modo relacionada o ha servido como base al propósito de esta investigación.

2.2.1 Políticas singlecore

En este subapartado se tratarán propuestas relacionadas para un solo core.

2.2.1.1 Re-Reference Interval Prediction (RRIP)

Según se expone en [3,11] debido a lo inviable que resulta el desarrollar la política LRU en sistemas con una alta asociatividad de cache, se utiliza la política *Not Recently Used* (NRU) como una aproximación de la misma. La política NRU utiliza un bit extra por cada bloque de cache, este bit se usa para tratar de predecir el reúso del bloque, es decir si se hará referencia a éste en un futuro cercano (0) o distante (1). El algoritmo funciona de la siguiente forma:

1- Al insertar un bloque nuevo en la cache o al darse un acierto en un acceso a un bloque existente, el bit NRU correspondiente a este bloque se asigna a cero para indicar una predicción de acceso cercano.
2- Al tener que desalojar un bloque se busca el primer bloque cuyo bit NRU sea para una predicción de acceso distante.
3- De no haber ningún bloque con su bit NRU indicando una predicción de acceso distante, se cambian los bits NRU de todos los bloques a este estado.

Teniendo únicamente dos estados posibles, la política NRU limita el rendimiento de la memoria cache cuando se encuentra frente a patrones de acceso que son de tipo streaming, en donde los bloques a los que se hace referencia no serán reutilizados. Es decir, NRU podría asignar a una probabilidad de acceso cercano a bloques que no van a recibir un nuevo acceso, permitiendo de esta forma que se contamine la cache con bloques que no van a ser utilizados.

Para solventar el problema de NRU, RRIP [3] asigna más bits de estado a cada bloque de forma que además de poder predecir un reúso cercano y distante se pueda predecir uno lejano, esto es un intervalo de predicción intermedio pero ligeramente desviado al distante. Este array de N bits de predicción se denomina RRPV (del inglés valores de predicción de reúso). En la tabla 2.1 se muestra el valor de RRPV para cada tipo de predicción de acceso.

Predicción de acceso	Valor para N bits	Valor para 2 bits
cercano	0	0
lejano	$2^N - 2$	2
distante	$2^N - 1$	3

Tabla 2.1: Valor de RRPV para cada tipo de predicción de acceso.

El algoritmo RRIP funciona de la siguiente manera:

1- Cada conjunto de cache puede verse como un array de tantas posiciones como asociatividad tenga la cache. En RRIP estas posiciones se recorren de la misma forma, de izquierda a derecha si se visualiza como un array horizontal, siendo la primera posición la de más a la izquierda y la última la de más a la derecha.
2- Al insertar un bloque en la cache se asigna su RRPV a predicción de acceso lejano.
3- Al tener que desalojar un bloque se busca el primer bloque en el conjunto cuyo RRPV sea para una predicción de acceso distante.
4- De no haber ningún bloque con su RRPV indicando una predicción de acceso distante, se incrementa en uno el RRPV de todos los bloques del conjunto hacia predicción de acceso distante y se vuelve al punto 3.
5- Cuando se da un acierto (promoción), el valor RRPV del bloque correspondiente se debe mover hacia predicción de acceso cercano. Los autores proponen dos políticas:
    a. Cambiándolo directamente a predicción de acceso cercano, esto se define como RRIP con prioridad de acierto (RRIP-HP)
    b. Modificar en uno el valor RRPV hacia predicción de acceso cercano (decrementar el RRPV según figura 2.3), esto se define como RRIP con prioridad de frecuencia (RRIP-FP)

Para ilustrar el comportamiento de diferentes políticas vistas hasta ahora, en la parte superior de la figura 2.3 se muestra un patrón de referencias a memoria el cual se desarrolla en la misma figura para cuatro políticas diferentes: *Bélády, LRU, NRU, RRIP-HP*. En dicho cuadro se utiliza una cache de cuatro bloques totalmente asociativa, y en el caso de políticas NRU y RRIP-FP cada bloque tiene un subíndice que indica el valor del bit NRU y de RRPV respectivamente, siendo el RRPV utilizado de dos bits.

$$\{a_1^n, (c_1, c_2, c_3, c_4, c_5), a_1^n\} \qquad \text{con } n = 2$$

Ref	Bélády	LRU	NRU	RRIP-HP
a_1	x_0 x_0 x_0 x_0	*MRU* *LRU* x_0 x_0 x_0 x_0	$x_0{}_1$ $x_0{}_1$ $x_0{}_1$ $x_0{}_1$	$x_0{}_3$ $x_0{}_3$ $x_0{}_3$ $x_0{}_3$
a_1	a_1 x_0 x_0 x_0	*MRU* *LRU* a_1 x_0 x_0 x_0	$a_1{}_0$ $x_0{}_1$ $x_0{}_1$ $x_0{}_1$	$a_1{}_2$ $x_0{}_3$ $x_0{}_3$ $x_0{}_3$
c_1	a_1 x_0 x_0 x_0	*MRU* *LRU* a_1 x_0 x_0 x_0	$a_1{}_0$ $x_0{}_1$ $x_0{}_1$ $x_0{}_1$	$a_1{}_0$ $x_0{}_3$ $x_0{}_3$ $x_0{}_3$
c_2	a_1 c_1 x_0 x_0	*MRU* *LRU* c_1 a_1 x_0 x_0	$a_1{}_0$ $c_1{}_0$ $x_0{}_1$ $x_0{}_1$	$a_1{}_0$ $c_1{}_2$ $x_0{}_3$ $x_0{}_3$
c_3	a_1 c_1 c_2 x_0	*MRU* *LRU* c_2 c_1 a_1 x_0	$a_1{}_0$ $c_1{}_0$ $c_2{}_0$ $x_0{}_1$	$a_1{}_0$ $c_1{}_2$ $c_2{}_2$ $x_0{}_3$
c_4	a_1 c_1 c_2 c_3	*MRU* *LRU* c_3 c_2 c_1 a_1	$a_1{}_0$ $c_1{}_0$ $c_2{}_0$ $c_3{}_0$	$a_1{}_0$ $c_1{}_2$ $c_2{}_2$ $c_3{}_2$
c_5	a_1 c_4 c_2 c_3	*MRU* *LRU* c_4 c_3 c_2 c_1	$c_4{}_0$ $c_1{}_1$ $c_2{}_1$ $c_3{}_1$	$a_1{}_1$ $c_4{}_2$ $c_2{}_3$ $c_3{}_3$
a_1	a_1 c_4 c_5 c_3	*MRU* *LRU* c_5 c_4 c_3 c_2	$c_4{}_0$ $c_5{}_0$ $c_2{}_1$ $c_3{}_1$	$a_1{}_1$ $c_4{}_2$ $c_5{}_2$ $c_3{}_3$
a_1	a_1 c_4 c_5 c_3	*MRU* *LRU* a_1 c_4 c_3 c_2	$c_4{}_0$ $c_5{}_0$ $a_1{}_0$ $c_3{}_1$	$a_1{}_0$ $c_4{}_2$ $c_5{}_2$ $c_3{}_3$

Figura 2.3: Muestra el resultado de utilizar el patrón de acceso que se indica en la parte de arriba de la figura en una cache de 4 vías con las políticas Bélády, LRU, NRU y RRIP-HP.

En la figura 2.3 se muestra como RRIP-HP es la política que se acerca más al comportamiento ideal (Bélády). Según los resultados presentados por el autor es justamente la política RRIP-HP la de menor tasa de fallos. Así mismo, el autor se refiere a esta política como RRIP estática (SRRIP-HP).

SRRIP hace un uso ineficiente de la cache cuando entre cada reúso de los bloques hay más referencias a bloques que ocupan el mismo conjunto que vías disponibles. Para corregir esto, los autores definen la política *Bimodal RRIP* (BRRIP). Esta política asigna el RRPV de un nuevo bloque de dos formas, asignando con una alta probabilidad un RRPV con una predicción de acceso distante y asignando de forma ocasional un RRPV con una predicción de acceso lejano, es decir mayoritariamente funciona como RRIP-HP con inserción como acceso distante y ocasionalmente como RRIP-HP con inserción como acceso lejano.

Debido a que BRRIP no hace un uso eficiente de la memoria para patrones distintos para los que fue creado, se plantea el uso de un RRIP dinámico (DRRIP) el cual usa set-dueling para determinar la mejor política a usar entre RRIP-HP con inserción como acceso lejano y BRRIP.

Los resultados obtenidos por los autores exponen que SRRIP y DRRIP mejoran el rendimiento de LRU en una media de un 4% y un 10% en un sistema single-core cuyo último nivel de cache tiene 2MB y 16 vías. También muestran que, para un chip multi-procesador con 4 cores con un último nivel de cache compartido de 8MB y 16 vías, SRRIP y DRRIP mejoran el rendimiento de LRU en un 7% y un 9% respectivamente.

2.2.1.2 Dynamic Insertion Policy (DIP)

En base al artículo [4] los autores hacen referencia al mal comportamiento de la política de reemplazo en la cache cuando la carga de trabajo no ofrece una alta localidad temporal en orden al reúso de los bloques peticionados. Si la frecuencia de reúso es menor que la asociatividad, esta política sufre de un comportamiento patológico que le obliga a mantener en cache bloques durante más tiempo del necesario. Cuando este problema aparece, existen otro tipo de políticas que sí son capaces de evitar este tipo de patología. Los autores introducen *LRU insertion Policy (LIP)* la cual inserta los bloques entrantes en la cache en la posición LRU de la *recency-stack* en vez de en la posición MRU como hace LRU. Si en un periodo corto de tiempo estos bloques no reciben ningún reúso serán expulsados de la cache. Si ocurre lo contrario serán promocionados a la posición MRU. LIP de esta forma protege a la cache de este tipo de problema pero carece de la habilidad de adaptarse a las modificaciones que en cuanto a frecuencia de reúso puede sufrir una carga de trabajo. Es por eso que los autores exponen *Bimodal Insertion Policy (BIP)*. BIP es muy similar a LIP excepto en que la primera en un bajo porcentaje de los casos inserta directamente el bloque entrante en la posición MRU. De esta forma, BIP se adapta a las características cambiantes del conjunto de trabajo conservando la habilidad para mantener parte de él en cache cuando no se da una alta localidad temporal en el reúso de los bloques entrantes en la cache.

Para una mayor adaptabilidad a los posibles cambios de la carga de trabajo y así poder extraer el máximo rendimiento del funcionamiento de la cache. Los autores exponen *Dinamyc Insertion Policy (DIP)*. Está política consiste en elegir entre LRU y BIP la política que mejor se adapte a las características en cuanto al reúso de los bloques de la aplicación que se esté ejecutando. Esta elección la hará en base al mecanismo Set Dueling ya explicado anteriormente.

El rendimiento que exponen de esta política es que probada en dieciséis benchmarks en los que su ejecución implica un uso intensivo del sistema de memoria reducen el número de fallos por cada mil instrucciones (MPKI) en un 21,3% para una cache con un a L2 de dieciséis vías y 1MB de capacidad.

2.2.1.3 Pseudo-LIFO (PELIFO)

En el artículo [2] los autores intentan abordar desde otro punto de vista el mismo problema del apartado anterior relativo al mal comportamiento que ofrece la política LRU cuando la carga de trabajo no ofrece una alta localidad temporal. La política PELIFO busca ofrecer una solución intermedia, aprendiendo dinámicamente cual es el número de vías que son necesarias para satisfacer los reúsos a corto plazo y reservando el resto para almacenar bloques que produzcan aciertos en reúsos a largo plazo. Este planteamiento parte de una política más básica como es LIFO, en la cual se hace uso de una pila de llenado o *Fill Stack* donde el bloque que entró a la cache en último lugar se convierte en el primer candidato para abandonarla. Se proponen entonces varias optimizaciones para mejorar su rendimiento. La idea principal de la política PELIFO como ya se ha expuesto anteriormente es mantener la parte baja de la pila de llenado para reúsos a largo plazo al igual que hace LIFO pero estableciendo las estrategias para dar oportunidad a los bloques con un plazo de reúso menor y así también aprovechar la localidad temporal.

En la figura 2.4 se ilustra la diferencia de operaciones entre (a) LIFO y (b) PELIFO. Mientras que LIFO realiza todos los reemplazamientos en la cabeza de la pila, PELIFO selecciona dinámicamente una posición intermedia llamada punto de escape o *Escape Point* para garantizar que los reúsos a corto plazo también sean atendidos. PELIFO gira alrededor de dos conceptos principales:

- *Escape Probability*: La probabilidad de escape de una determinada posición *i* de la Fill Stack es definida como la probabilidad de que los bloques de la cache experimenten más aciertos en el resto de posiciones que en *i*. Entre todas las probabilidades de escape obtenidas, sólo las tres más próximas a la parte alta de la pila de llenado son seleccionadas (EP_1,EP_2 y $EP_{3)}$.
- *Escape Point (EP)*: Es la posición de la Fill Stack dónde la Escape probability decrece por debajo de un cierto umbral cuando es comparado con el de posiciones previas. Los bloques situados en esta posición se convierten en candidatos para dejar la

cache cuando un reemplazo suceda, ya que a partir de ese momento, el bloque que ocupa esta posición probablemente no experimentará más reúsos en un corto plazo.

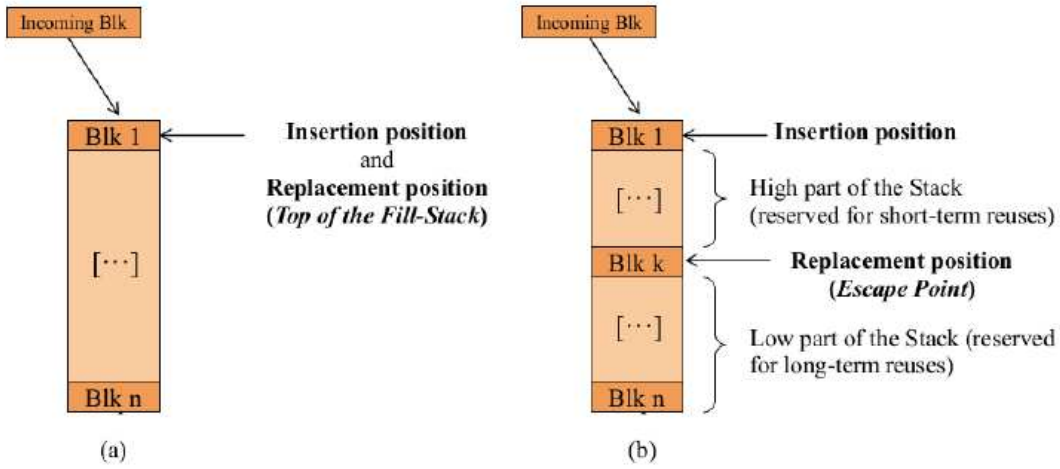


Figura 2.4: Operaciones básicas de reemplazamiento en las políticas LIFO (a) y PELIFO (b)

Los autores exponen unos resultados donde con un gasto extra de un 1% de la capacidad de almacenamiento PELIFO reduce en un 10% de media, con respecto a la política de reemplazo LRU, el tiempo de ejecución para un conjunto de catorce aplicaciones ejecutadas en un mismo core con un último nivel de cache L2 de 2MB de capacidad y asociativa por conjuntos en 16 vías.

2.2.2 Políticas multicore

En este subapartado se tratarán propuestas relacionadas para un entorno multi-core.

2.2.2.1 Adaptative insertion policies for managing shared caches

La extensión de DIP directamente a las caches compartidas mejora el rendimiento de base de la política LRU pero deja pendiente un significativo margen de mejora. DIP es incapaz de distinguir entre aplicaciones que se están beneficiando de la asignación de mayores recursos de la cache de aquellas que no lo están haciendo.

Con el propósito de dar solución a esta carencia de DIP en las caches compartidas en [6] se propone Thread-Aware Dynamic Insertion Policy (TADIP). Para un determinado stream de referencias, la inserción adaptativa trata de dinámicamente elegir entre dos políticas: LRU y BIP. TADIP fija para cada core del Chip Multi Processor (CMP) la política a utilizar tomando una decisión binaria. Si toma el valor 0 se elegirá la política LRU y si el valor es 1 la política a utilizar será BIP. Si se están ejecutando concurrentemente N aplicaciones compitiendo por recursos de cache, el espacio de búsqueda de TADIP intenta encontrar el stream que proporcione un mayor rendimiento entre los 2^N posibles. Cuando N es pequeño se puede utilizar Set Dueling para en tiempo de ejecución comparar todos los posibles strings binarios y elegir aquel que arroje mejor rendimiento. El problema es que a medida que crece el número de aplicaciones que se ejecutan en un mismo CMP el número de posibilidades con respecto al espacio de búsqueda crece exponencialmente lo que hace que esta solución no se escalable. Se proponen dos soluciones escalables. TADIP-Isolated (TADIP-I) y TADIP-Feedback (TADIP-F).

TADIP-I, como muestra la figura 2.5 (b) va aprendiendo la política de inserción de cada aplicación independientemente, no teniendo en cuenta el comportamiento del resto y asumiendo que en ellas se usa la política LRU (señalizado en la figura con el stream <0,0,0,0>). El problema de esta aproximación es que las decisiones de inserción de una aplicación pueden depender de las decisiones de inserción de otras que concurrentemente se están ejecutando.

La ventaja es que al hacer que las decisiones se tomen independientemente por aplicación, se consigue que la solución sea escalable pasando de una complejidad exponencial a lineal.

TADIP-F mejora TAPDIP-I ya que va aprendiendo el mejor rendimiento de la política de inserción de cada core teniendo en cuenta las elecciones hechas en el resto de ellos a cambio de una complejidad 2N.

En TADIP-F, como indica la figura 2.5 (c), cada aplicación hace uso de 2 SDM. Uno de ellos siempre usa la política LRU y el otro siempre BIP. Un fallo en el LRU-SDM incrementará el PSEL de esa aplicación y un fallo en BIP-SDM lo decrementará. El bit más significativo determinará la política a seguir en la aplicación. La cache tiene ocho SDM , un par por cada aplicación. Por ejemplo, la primera aplicación utiliza los strings binarios <0,P1,P2,P3> y <1,P1,P2,P3> donde Px es el bit más representativo de $PSEL_X$. El <0,P1,P2,P3> SDM siempre seguirá la política LRU para la primera aplicación y la política que haya sido seleccionada como de mejor rendimiento en el conjunto de las aplicaciones restantes. De igual modo <1,P1,P2,P3> SDM seguirá la política BIP para la primera aplicación y la política que haya sido seleccionada como de mejor rendimiento para el resto de aplicaciones.

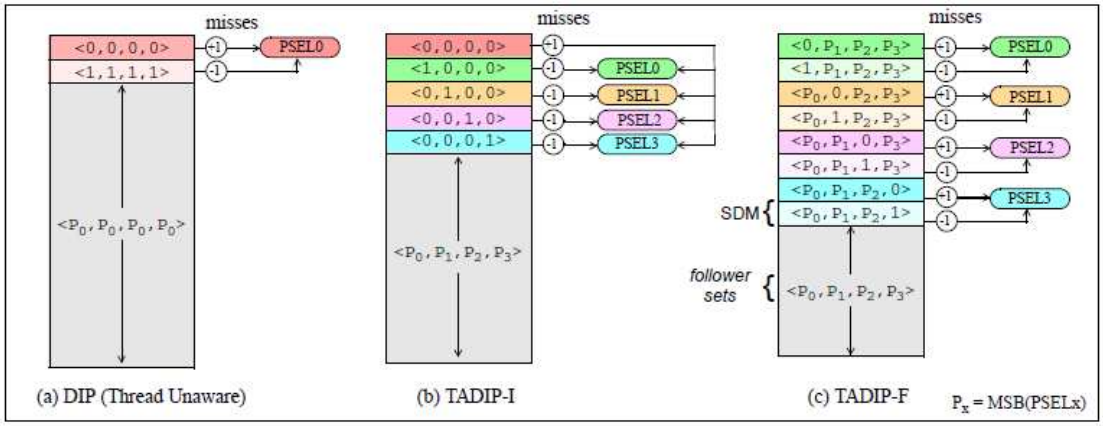


Figura 2.5: Gestión de inserción adaptativa para caches compartidas utilizando como política de inserción (a) DIP, (b) TADIP-I y (c) TADIP-F.

La evaluación de las propuestas sobre sistemas 2-core, 4-core, 8-core y 16-core con memorias en su nivel compartido de 2MB, 4MB, 8MB y 16MB respectivamente, ponen de manifiesto un aumento de media del throughput de las caches compartidas gestionadas por TADIP sobre las caches compartidas gestionadas por LRU de un 14%, 18%, 15% y 17% respectivamente.

2.2.2.2 Utility Based Cache Partitioning (UCP)

En [7] se expone que una de las claves para obtener un alto rendimiento en arquitecturas multi-core es la gestión que se haga del último nivel de cache, con el propósito de acceder lo menos posible a la parte de la jerarquía de memoria que se encuentra fuera del chip.

El objetivo de esta investigación es de como rentabilizar el último nivel compartido de cache intentando conseguir el particionado más óptimo entre las múltiples aplicaciones que en el mismo momento compiten por él.

Tradicionalmente se ha seguido la política de asignar más recursos a las aplicaciones que presentaban una mayor demanda de datos y viceversa. Sin embargo, el beneficio en términos de mejora de rendimiento que una aplicación puede obtener de la porción de cache asignada en el nivel compartido no tiene porqué corresponder con las demandas de cache de ésta. Por ejemplo, una aplicación en la que se de streaming hará una gran petición de bloques con la consecuencia sabida de que estos no se volverán a utilizar antes de ser evacuados de la cache y por lo tanto no se hará un uso eficiente de los recursos asignados.

Para evitar este problema, se propone que la asignación de cache para cada aplicación en el nivel compartido se haga en orden al beneficio en términos de reducción de fallos que es probable obtener. Comportamiento que llevan a cabo con *Utility-Based Cache Partitioning (UCP)*.

Los autores establecen una clasificación de las aplicaciones en tres categorías. La primera de ellas compuesta por benchmarks cuyo rendimiento no se ve mejorado

significativamente a medida que aumentamos el número de vías de la cache y a las que llaman aplicaciones de *baja utilidad*. La segunda clasificación está compuesta por aplicaciones cuyo rendimiento sí que se ve mejorado significativamente a medida que aumentamos el número de vías y al que denominan de *alta utilidad* y el último grupo sería aquel en el que las aplicaciones mejoran su rendimiento significativamente hasta una determinada vía pero después se satura y aunque aumentemos este número, el rendimiento no experimenta mejora alguna debido a que son benchmarks cuyo conjunto de trabajo cabe en caches pequeñas, a este grupo se le denomina de *utilidad saturada*.

Teniendo esto en cuenta, si dos aplicaciones con baja utilidad son ejecutadas conjuntamente, su rendimiento no se va a ver afectado por la cantidad de cache disponible para cada aplicación. De la misma forma, cuando dos aplicaciones con utilidad saturada son ejecutadas conjuntamente, la cache puede soportar el conjunto de trabajo de ambas aplicaciones. Sin embargo, la ejecución conjunta de una aplicación con utilidad saturada con una de baja utilidad puede hacer que el conjunto de trabajo de la primera ahora no se pueda mantener completamente en la cache por lo que su rendimiento se vería afectado. Al igual que si ejecutamos conjuntamente una aplicación de alta utilidad con cualquier otra de los otros tipos. En estos últimos casos se hace imprescindible en términos de mejora del rendimiento de un particionado juicioso de la cache que tenga en cuenta la información de utilidad de la aplicación.

Las evaluaciones llevadas a cabo arrojan una mejora en el rendimiento de un 11% de media mientras que sólo supone una penalización en la capacidad de almacenamiento menor a un 1%.

2.2.2.3 Improving PELIFO cache replacement policy: Hardware reduction and thread-aware extension

En este artículo [8] se proponen diversas técnicas para la mejora de la política PELIFO. La que más relación tiene con este trabajo es el de mejorar su funcionamiento para niveles de cache compartidos, recopilando información dinámicamente sobre la naturaleza de cada aplicación que se ejecuta concurrentemente para mejorar el funcionamiento global de la cache.

En un entorno multi-core, el nivel compartido de cache acomoda bloques de diferentes aplicaciones, cada una con su propio patrón de acceso. La propuesta para el uso de la política PELIFO en este tipo de entornos debe aprovechar la información a cerca de cada aplicación con el objetivo de dedicar más recursos de la cache a aquellas que los están utilizando de forma más eficiente. De esta forma, si mejoramos el ratio global de aciertos en el nivel compartido de la cache, mejoraremos el rendimiento global del sistema.

Los autores proponen dos posibilidades de cambio de la política original. La idea subyacente en ambas soluciones es eliminar de la cache compartida aquellos bloques que pertenecen a aplicaciones que no están aprovechando el mantener los bloques con reúsos a largo plazo.

PELIFO con listas (peLIFO-ls)

Se definen cuatro zonas dentro de la pila de llenado delimitando cada una con los puntos de escape. La primer zona (Z_0) va desde la cima de la pila hasta el primer punto de escape (EP_1), mientras que las zonas segunda (Z_1), tercera (Z_2) y cuarta zonas (Z_3) van desde EP_1, EP_2 y EP_3 respectivamente hasta la parte baja de la pila. Por lo que nos referiremos a Z_0 como la parte alta de la pila de llenado y a Z_1, Z_2 y Z_3 como la parte baja de la pila.

Para cada thread se calcula dinámicamente al final de cada época los ratios de aciertos por bloque en la parte baja de la pila (Z_1, Z_2 o Z_3) durante la época actual y se divide por la media del número de bloques de este thread acomodados en la parte baja de la pila durante

esta época. Después, y a acorde con los resultados obtenidos se generarán tres listas ordenadas (una por cada zona en la parte baja de la pila). De esta forma el valor de ratio más alto y el más bajo estarán situados en la parte alta y la parte baja de las listas respectivamente. Cuando se necesita hacer un reemplazamiento, los bloques correspondientes a threads que están mostrando una suma baja de aciertos por bloque serán prioritarios para su evacuación, debido a que estos bloques hacen un uso de la cache menos eficiente que otros.

Basándose en estas tres listas, PeLIFO-ls opera como sigue dependiendo de la política seleccionada por el mecanismo *Set Dueling* (ente P_1, P_2, P_3, y P_4) a emplear en el reemplazamiento:

(i) Si P_4 es elegida, se emplea LRU.

(ii) Si P_k (con $k \in \{1,2,3\}$ es seleccionada, el bloque que está más cerca de la cima de la pila de llenado y que cumpla los requisitos siguientes es seleccionado como víctima:

a. Que corresponda al thread en la cima de la L_k lista.
b. Su posición actual en la pila de llenado sea mayor o igual a EP_k.
c. No haya recibido ningún acierto en su posición actual de la pila de llenado.

Si no se encuentra ningún bloque que cumpla estos requisitos, se volverá a empezar por la primera condición pero ahora utilizando el siguiente thread en la lista L_k. Si aun así no se encuentra ningún bloque acorde con estos criterios, el bloque seleccionado por la política LRU será eliminado.

Proportional PELIFO (peLIFO-prop)

Esta técnica consiste en reemplazar por thread un conjunto de bloques inversamente proporcional al actual uso de la cache. Para este propósito, 3*N contadores *(blocksToReplace(t_i,Z_j))* dinámicamente almacenan el número de bloques por thread t_i y localizados en la zona Z_j que deben ser reemplazados.

Basándose en los valores calculados por la figura y, PeLIFO-prop opera como sigue dependiendo de la política, seleccionada por el mecanismo Set Dueling (entre P_1, P_2, P_3 y P_4), a emplear en el reemplazamiento:

(i) Si P_4 es seleccionada, se emplea la política LRU.

(ii) Si P_k (con $k \in \{1,2,3\}$) es seleccionada, el bloque más próximo a la cima de la pila de llenado que satisfaga los siguientes criterios será seleccionado para su evacuación:

a. Que corresponda a un thread con el contador *blocksToReplace* correspondiente a Z_k mayor que cero.
b. Su posición actual en la pila de llenado sea mayor o igual que la elegida por EP_k.
c. No haya experimentado ningún acierto en la posición actual de la pila de llenado.

Si se encuentra un bloque que satisfaga estas condiciones, el contador bloksToReplace correspondiente es decrementado. Cuando todos los contadores asociados a una zona en particular alcanzan el valor cero, estos son reseteados con los valores de *blocksToReplace* calculados al comienzo de la presente época. Si ningún bloque con estas características es encontrado, se emplea la política peLIFO original.

Para un conocimiento más exhaustivo de estas técnicas, en el artículo referenciado vienen ejemplos detallados del funcionamiento de las mismas.

Los autores ofrecen unos resultados en los que de media el CPI (*cycles per instruction*) cae en un 7,2% comparándolo con LRU. Teniendo en cuenta la información de uso de cada thread, PELIFO-ls reduce el CPI con respecto a PLIFO y LRU en un 1% y un 8,1% respectivamente. En el caso de PELIFO-prop, esta reducción alcanza un 8,7% y un 15,2% respectivamente. Para la realización de estas pruebas utilizan una cache con un nivel L2 compartido con una capacidad de 8MB y asociativa por 16 vías.

2.2.2.4 The Reuse Cache

Estudios previos han demostrado que convencionalmente el último nivel compartido de la cache (*Share Last Level Cache* - SLLC) es efectivo pero ineficiente debido a que la mayor parte de su contenido es inservible. De hecho, una alta proporción de las líneas almacenadas en este SLLC son líneas muertas. Es decir, no serán vueltas a referenciar antes de que sean desechadas de este nivel de cache hasta el punto que muchas de ellas sólo fueron usadas una vez.

En [11] se propone proveer a este nivel compartido de cache de una política que explote la localidad de reúso seleccionando qué líneas mantener y cuáles desechar basándose en esta característica. De esta forma, sólo las líneas que ya han demostrado un reúso serán mantenidas en este nivel de cache y debido a que el conjunto de líneas que ofrecen esta característica es bastante reducido esto permite drásticas reducciones de tamaño de este último nivel de cache sin que por ello se vea afectado el rendimiento.

Proponen la utilización de dos tipos de arrays, el *data array* que almacenará sólo las líneas que muestran reúso y el *tag array* que deberá contener los identificadores de las líneas que se encuentran en el *data array* así como las que se encuentran en los niveles privados de cache. Además, el *tag array* deberá almacenar información del histórico de las líneas que han sido usadas recientemente, líneas que por otra parte no tienen por qué estar en el *data array* o en los niveles privados de cache.

Debido a que se van a tener más entradas en *tag array* que en el *data array* una solución natural es desacoplarlas rompiendo el mapeo implícito entre tag y datos encontrado en las caches convencionales. El protocolo de coherencia de la cache de reúso también tiene que ser modificado con vistas a implementar nuevos estados de coherencia como por ejemplo reflejar el estado en el que el tag de una línea está presente en el *tag array* pero su correpondiente línea de datos no está presente en el *data array*.

Cuando ocurre un fallo en el tag array, la línea es leída de memoria principal y cargada en el correspondiente nivel privado de cache pero en la SLLC únicamente es cargado el tag y no los datos asociados. Cuando ocurre un acierto en el tag array sobre un tag que no tiene datos asociados en el data array se detecta un reúso. Entonces, la línea es leída de nuevo de memoria principal y cargada en la cache privada y en el SLLC data array al mismo tiempo. Cuando una línea es desechada del data array, su tag permanece en el tag array. Un acceso posterior a esta línea que tenga éxito en el tag array será tomado como un reúso y la línea volverá a ser cargada en el data array.

La propuesta evaluada simulando un sistema 8-cores corriendo sobre él *multiprogrammed and multithreaded workloads* implementando una cache de reúso con un tag array equivalente a una cache de 4 MB pero con sólo 1 MB en el array de datos ofrece el mismo rendimiento de media que una cache convencional con 8 MB. De esta forma la cache de reúso sólo hace uso de aproximadamente el 17% del presupuesto de almacenamiento de una cache convencional.

Capítulo 3

Propuesta de modificación de la política de reemplazo

El propósito de este capítulo es exponer una serie de observaciones relativas a determinados comportamientos que suceden en el último nivel de cache (LLC o SLLC), así como proponer algunas modificaciones, que tienen su fundamento en las observaciones citadas, de la política de reemplazo LRU para intentar obtener un mejor rendimiento de este nivel y por ende de toda la jerarquía de memoria.

3.1 Política de reemplazo: Tipología de inserciones y promociones

Dentro de la terminología aplicada a la gestión de la cache, hablamos de inserción de un bloque en un nivel de memoria cache *i* cuando éste reemplaza a otro bloque existente en un determinado marco u ocupa un marco que se encontrara libre en ese momento. Teniendo esto en cuenta, y considerando que se utiliza una política de inclusión *no inclusiva* y que se hace uso

de una política de actualización *writeback* (que por otra parte es la de uso más habitual), proponemos distinguir entre dos tipos de inserción dependiendo del motivo que la ha ocasionado:

Inserción debida a load-store (i-ls): Se produce cuando el procesador demanda un dato que no está presente en el nivel *i*. Será necesario entonces traerlo de un nivel *j (j>i)* y se insertará en *i* el bloque que lo contiene.

Inserción debida a *writeback* (i-w): Este tipo de inserción viene dado por la expulsión de un bloque en un nivel de cache *i-1*. Si este bloque se encuentra limpio, será desechado sin más, pero si por el contrario está sucio (se modificó en algún momento desde su entrada en el nivel de cache en cuestión) y no se encuentra presente en el nivel *i* deberá ser insertado en este último nivel.

Por otra parte, cuando se encuentra un bloque en un nivel i (acierto de cache), se promociona dicho bloque. Al igual que en el caso de la inserción, de nuevo se propone distinguir entre dos tipos de promoción:

Promoción debida a load-store (p-ls): El procesador demanda un dato que es encontrado en el nivel *i* de la cache y que por lo tanto el bloque que lo contiene es promocionado en dicho nivel.

Promoción debida a *writeback* (p-w): En el nivel *i-1* de la cache es desalojado un bloque. Como ocurre en la inserción debida a *writeback*, si el bloque está limpio se desecha, pero si está sucio y sí que se encuentra presente en el nivel *i*, será promocionado en este último nivel.

Con el objetivo de aclarar mejor estas definiciones, en la figura 3.1 se ilustra un ejemplo en el que tienen lugar los dos tipos de inserciones y los dos tipos de promociones, todos ellos desencadenados a partir de un único acceso de lectura (instrucción de load).

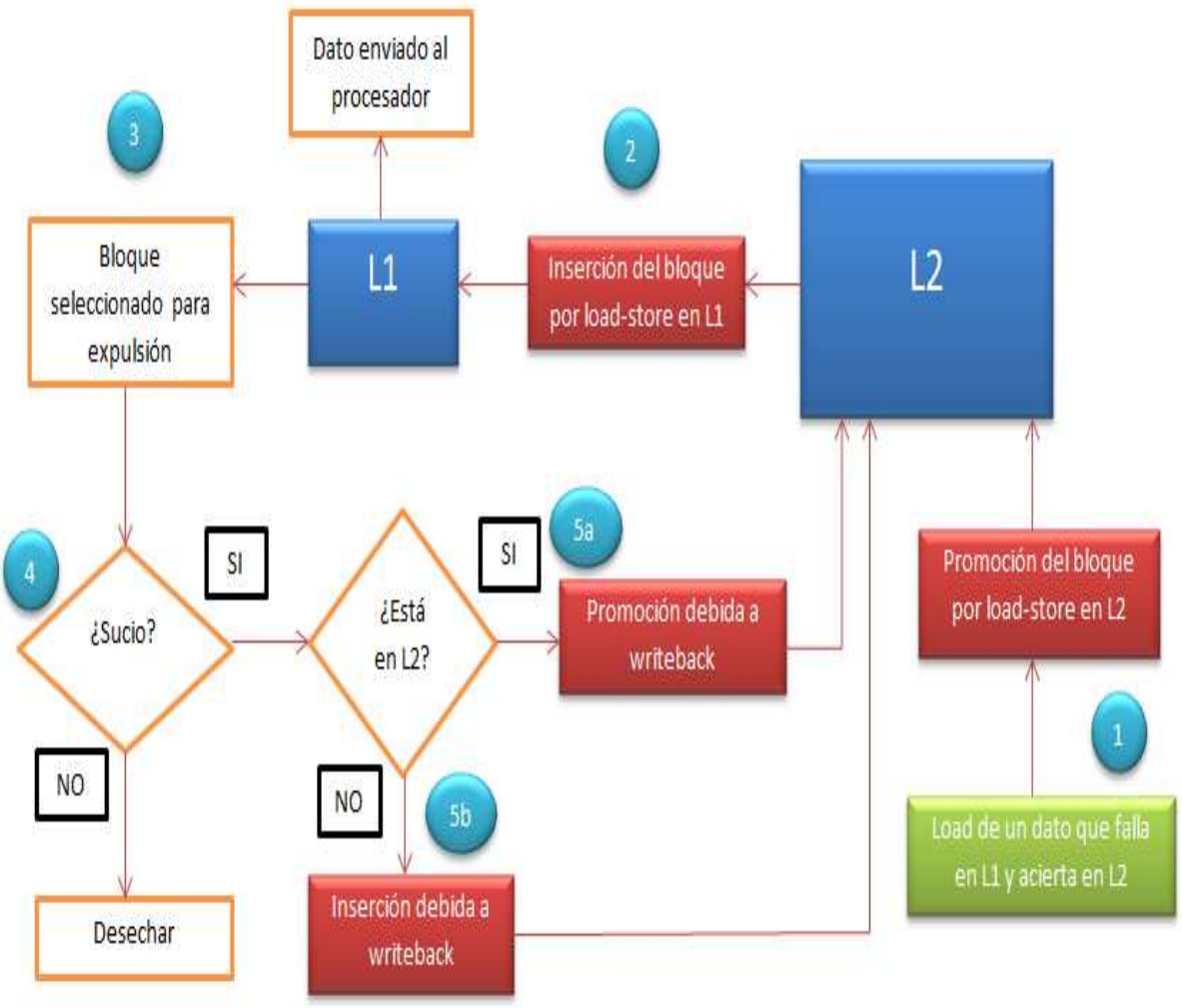


Figura 3.1: Inserciones y promociones en cache.

3.2 Motivación

Dentro de la literatura relacionada, no hemos encontrado ningún trabajo que proponga un trato diferenciado de las inserciones y promociones debidas a *writeback* con respecto a las que no lo son. Sin embargo, en este apartado demostraremos a partir de diversos experimentos, que todo apunta a pensar que una diferencia en el trato por parte de la política de reemplazo de los distintos tipos de inserción/promoción puede contribuir a mejorar el rendimiento de la cache.

3.2.1 Análisis de frecuencia de eventos

No tendría mucho sentido que la política de reemplazo hiciera un trato diferenciado para los distintos tipos de eventos si los porcentajes de ocurrencia estuvieran muy descompensados. Por poner un ejemplo, si una aplicación mostrara los siguientes porcentajes:

	load-store	*writeback*
inserciones	*97%*	*3%*
promociones	*98%*	*2%*

En este hipotético caso, no se podría esperar mucho en cuanto a ganancia de rendimiento de un trato diferenciado de los eventos ocurridos por un load-store de aquellos que han sucedido a consecuencia de un *writeback*. Es por eso que una primera tarea fue la de hacer un estudio de ocurrencia para cada clase que justifique esta distinción de trato entre eventos.

Para llevar a cabo esta tarea, se implementaron una serie de contadores en el simulador gem5, que es el que se utilizó en el trabajo, en los que **i** hace referencia a la suma de inserciones debidas a load-store, **iw** representa la suma de las inserciones debidas a writeback, **p** la suma de las promociones debidas a load-store y **pw** la suma de las promociones debidas a *writeback*. Nótese que sólo se contabilizan las ocurrencias del último nivel de cache, en nuestro caso el nivel tercero de la cache.

En la figura 3.2 se muestra el porcentaje de ocurrencia de inserciones debido a load-store y el porcentaje de ocurrencia de inserciones debidas a *writeback* con respecto al número de total de inserciones por benchmark en el último nivel de cache para aplicaciones SPEC2006. La figura 3.3 muestra la misma relación para promociones debidas a load-store y promociones debidas a *writeback*.

La figura 3.4 por otra parte muestra el promedio de porcentajes de cada tipo de evento considerando todos los benchmarks que se han evaluado en el estudio respecto a la suma de todos los eventos.

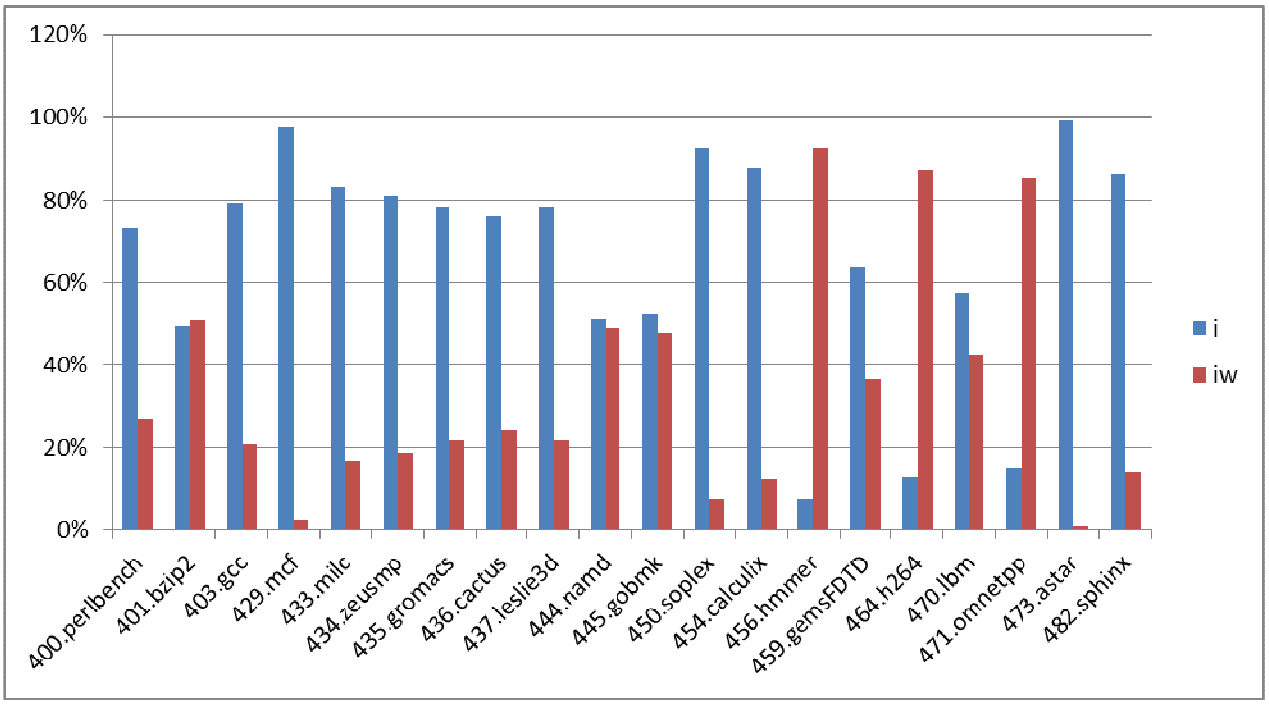


Figura 3.2: Porcentajes de ocurrencia de inserciones debidas a load-store y de inserciones debidas a writeback con respecto al total de inserciones.

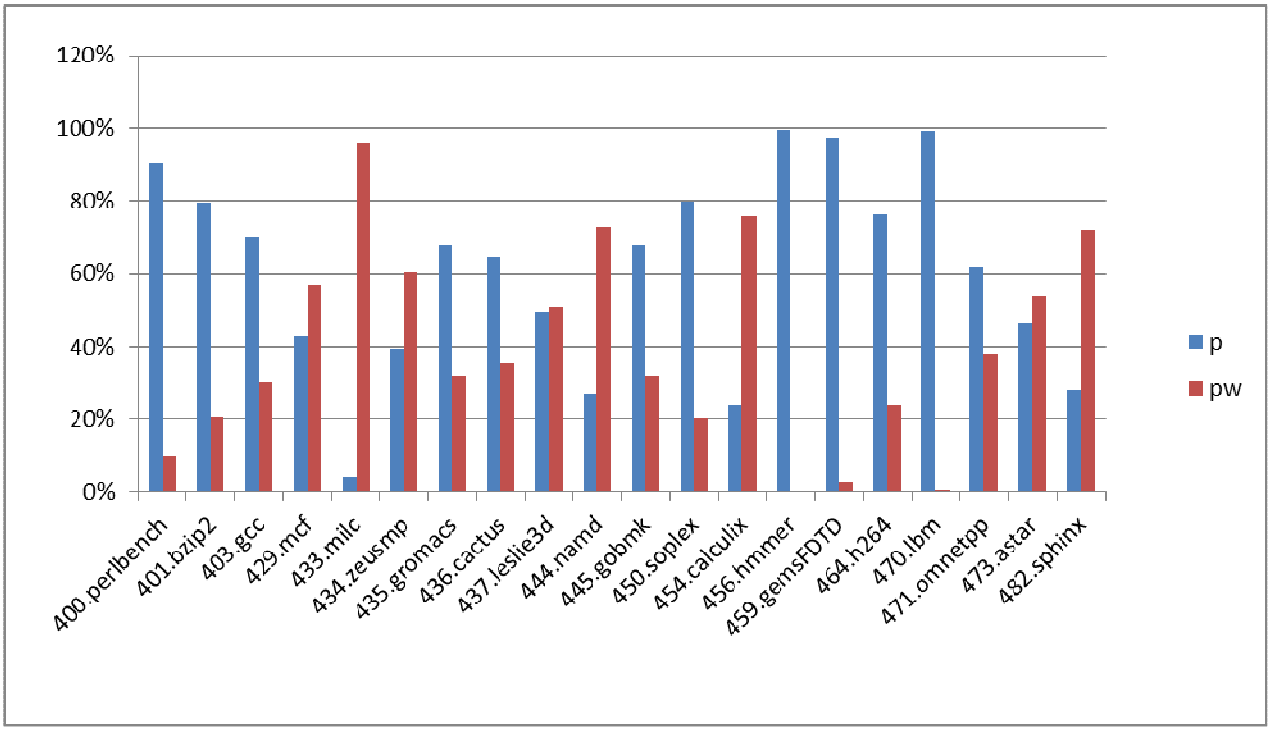


Figura 3.3: Porcentajes de ocurrencia de promociones debidas a load-store y de promociones debidas a writeback con respecto al total de promociones.

Figura 3.4: Porcentajes medios de ocurrencia para los distintos tipos de promociones e inserciones.

A la vista de los resultados, podemos concluir este apartado diciendo que los porcentajes, en cuanto a número de ocurrencias, de los eventos debidos a *writeback* con respecto a los eventos que no lo son, muestran un comportamiento general bastante equilibrado.

3.2.2 Análisis de patrones asociados a eventos

Además de que, como se acaba de demostrar, los distintos tipos de inserción y promoción muestran una ocurrencia equilibrada, debemos comprobar si su comportamiento de cara a la política de reemplazo muestra características diferentes. Para ello, consideraremos dos aspectos cada vez que un bloque es promocionado:

- El último evento que sufrió el bloque antes de la promoción. Para poder tener en cuenta cuál fue este último evento, se hacen las modificaciones necesarias en el simulador para que en cada bloque se registre dicha información y sea actualizada convenientemente, actualización que será llevada a cabo en base a los accesos de diferente naturaleza que un mismo bloque puede tener a lo largo de la ejecución de una aplicación.
- La posición de la *recency stack* que ocupa el bloque cuando es promocionado.

De esta forma podremos calcular, por cada posición de la *recency stack*, el número de bloques que han sido promocionados y cuyo último evento anterior fue uno de entre los cuatro posibles (i-ls,i-w,p-ls ó p-w). Por poner un ejemplo que clarifique esta idea, si nos fijamos en la figura 3.5, la gráfica referente a “i” representaría el número de bloques que en cada posición de la *recency stack* recibieron una promoción siendo el anterior evento una inserción debida a load-store.

Si analizamos estas gráficas para distintos benchmarks, algunas de las cuales están representadas en las figuras de la 3.5 a la 3.9, se pueden observar diferentes patrones dependiendo del cuál fue este último evento entre las cuatro opciones posibles. Por una parte,

la gráfica referente a las posiciones de la *recency stack* en las que el último evento que sufrió el bloque que la ocupa (antes de recibir una promoción) fue una inserción o una promoción debida a writeback muestran generalmente un comportamiento lineal descendente con un alto número de ocurrencias en las primeras posiciones de la *recency stack* y que luego va descendiendo más o menos bruscamente. Por otra parte el patrón encontrado para las posiciones de la *recency stack* en las que el último evento que sufrió el bloque que las ocupa fue una inserción o promoción debida load-store muestra una gráfica que generalmente recuerda a una campana de Gauss y cuyo punto álgido puede variar a lo largo de las posiciones de la *recency stack*.

Las figuras de la 3.5 a la 3.9 representan este tipo de comportamiento para los benchmarks *bzip2*, *gcc*, *mcf*, *gobmk* y *astar* respectivamente.

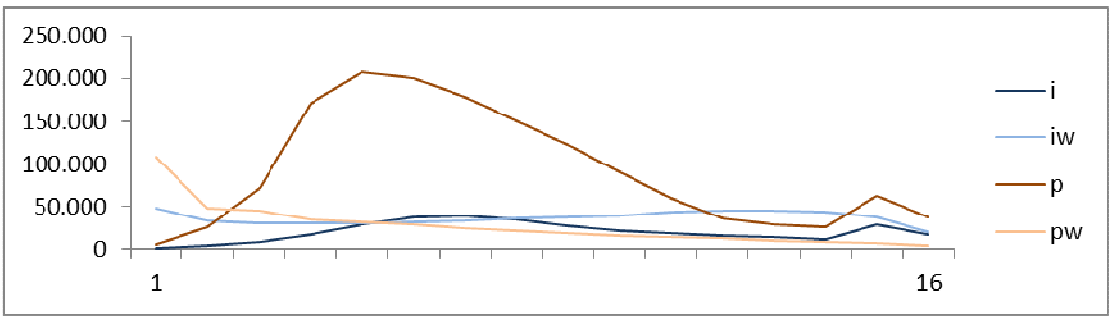


Figura 3.5: Número de ocurrencias para cada tipo de evento en cada posición de la recency stack del benchmark bzip2.

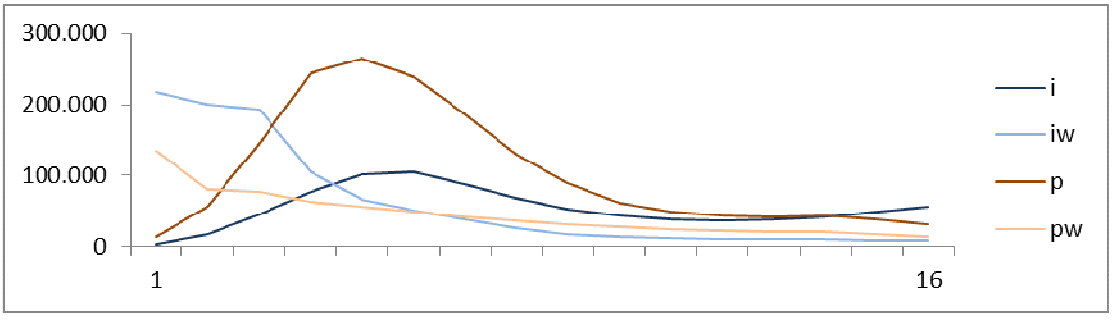


Figura 3.6: Número de ocurrencias para cada tipo de evento en cada posición de la recency stack del benchmark gcc.

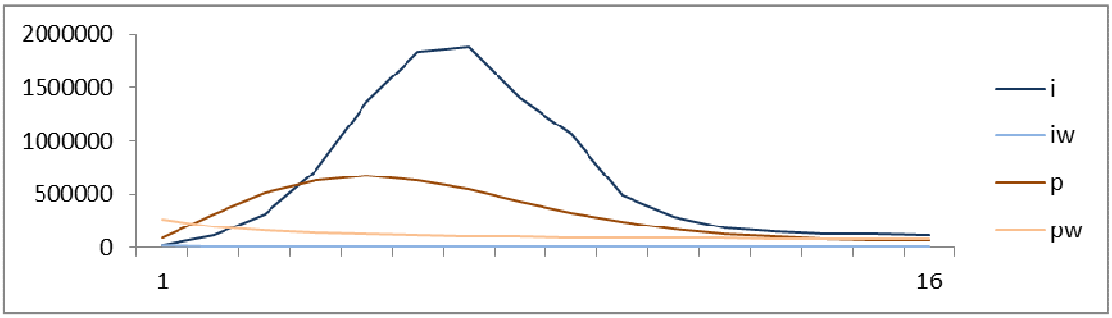


Figura 3.7: Número de ocurrencias para cada tipo de evento en cada posición de la recency stack del benchmark mcf.

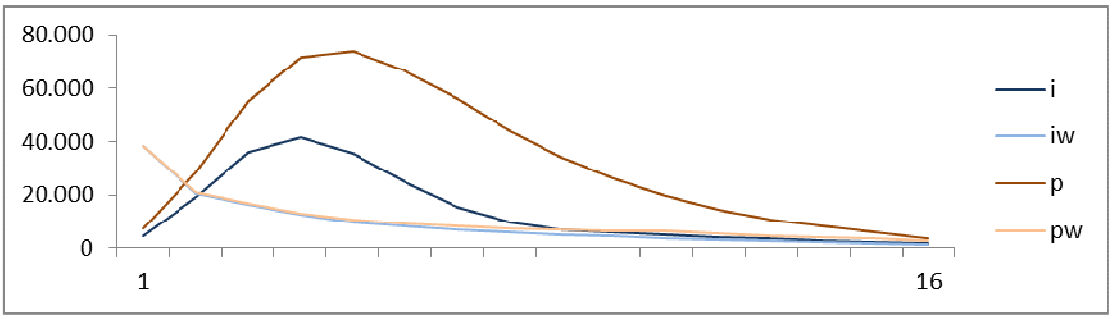


Figura 3.8: Número de ocurrencias para cada tipo de evento en cada posición de la recency stack del benchmark gobmk.

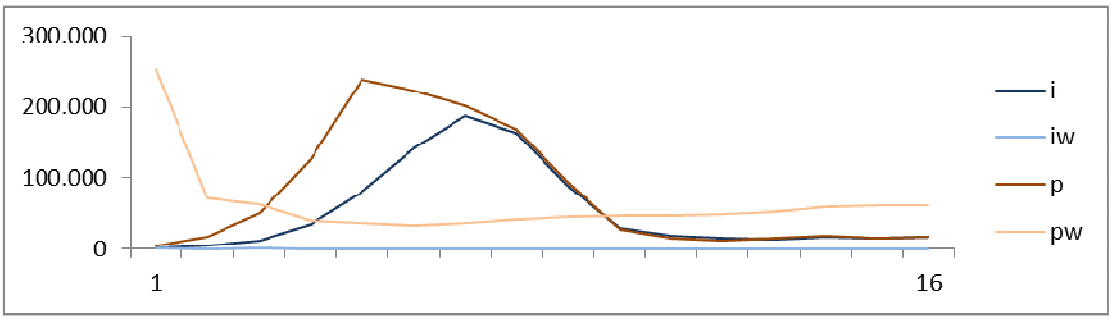


Figura 3.9: Número de ocurrencias para cada tipo de evento en cada posición de la recency stack del benchmark astar.

3.2.3 Análisis del reúso de los distintos tipos de eventos

Como último experimento motivacional, en este apartado se procede a hacer un análisis para comprobar si hay diferencias, en cuanto al éxito en su reúso (número de veces que se va a reutilizar un mismo bloque antes de ser evacuado de la cache), entre los bloques que han sido insertados debido a *writeback* con respecto a los que han sido insertados debido a load-store.

Para llevar a cabo este análisis se implementan los siguientes contadores en el simulador:

i_si: Número de bloques que habiendo sido insertados por un evento de load-store han recibido una promoción.

i_no: Número de bloques en los que el último evento que sufrieron fue una inserción por load-store y que son expulsados de la cache.

iw_si: Número de bloques que habiendo sido insertados por un evento de writeback han recibido una promoción.

iw_no: Número de bloques que el último evento que sufrieron fue una inserción por writeback y son expulsados de la cache.

p_si: Número de bloques en los que su último evento fue una promoción por load-store y reciben una promoción.

p_no: Número de bloques en los que el último evento que sufrieron fue una promoción por load-store y son expulsados de la cache.

pw_si: Número de bloques en los que su último evento fue una promoción por writeback y reciben una promoción.

pw_no: Número de bloques en los que el último evento que sufrieron fue una promoción debida a writeback y son expulsados de la cache.

Nótese que a lo largo de la ejecución de una aplicación, un mismo bloque puede dar lugar a actualización de varios de estos contadores, incluso a la totalidad de ellos. Como muestra de ello, véase la tabla 3.1 en donde se expone un ejemplo de cómo se desarrollaría el ciclo de vida de un bloque en la cache. Éste comenzaría por su inserción (ya sea debida a load-store o a writeback) y continuaría por un número que va de 0 a *n* promociones (debidas a load-store o a writeback) hasta el momento en que es expulsado.

Paso	*1*	*2*	*3*	*4*	*5*	*6*
Evento	*iw*	*p*	*p*	*pw*	*p*	*Expulsión*
Contador a actualizar		*iw_si ++*	*p_si ++*	*p_si ++*	*pw_si ++*	*p_no ++*
Último evento sufrido por el bloque	*iw*	*p*	*p*	*pw*	*p*	*null*

Tabla 3.1: Ciclo de vida de un bloque en cache y actualización de variables.

El estudio que se hizo en este caso fue para, con una configuración del último nivel de cache con 1MB de capacidad y una asociatividad de 16 vías, y haciendo uso de los contadores explicados con anterioridad, obtener los porcentajes de éxito de los bloques insertados o promocionados debido a *writeback* con respecto a los que no lo son, para cada uno de los benchmarks.

Si tomamos como ejemplo el caso concreto del benchmark *gcc*, mostrado en la figura 3.10, se puede ver que gran parte (66%) de los bloques insertados en el último nivel de cache debido a una inserción por *writeback*, consiguen tener éxito en al menos un reúso. Sin embargo, y como muestra la figura 3.11 este porcentaje desciende considerablemente (15%) cuando se trata de inserciones debidas a load-store. Esto supone que un gran porcentaje (85%)

de las inserciones debidas a load-store están siendo desechadas del último nivel de cache sin tan siquiera haber recibido un reúso.

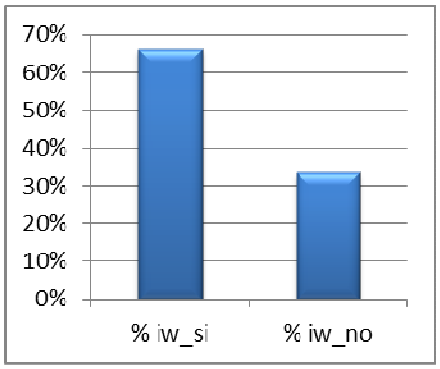


Figura 3.10: Porcentaje de éxito de las inserciones debidas a writeback para gcc.

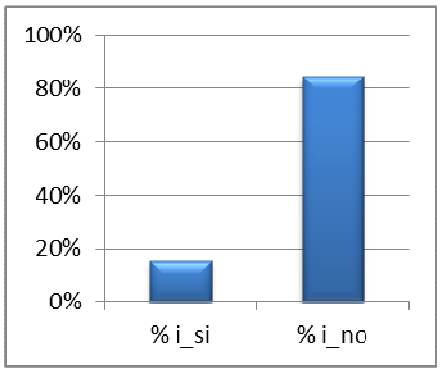


Figura 3.11: Porcentaje de éxito de las inserciones debidas a load-store para gcc.

En las gráficas de la 3.12 a la 3.17 se muestra (para los benchmarks *soplex, calculix* y *h264*) como, si bien esta observación no constituye un patrón a seguir, sí que se trata de un comportamiento más o menos generalizado observado en el resto de los benchmarks. En base a estos datos, podemos concluir este apartado diciendo que el éxito, en cuanto a porcentajes de reúso, de los eventos debidos a *writeback* con respecto a aquellos debidos a load-store se encuentra por lo general descompensado.

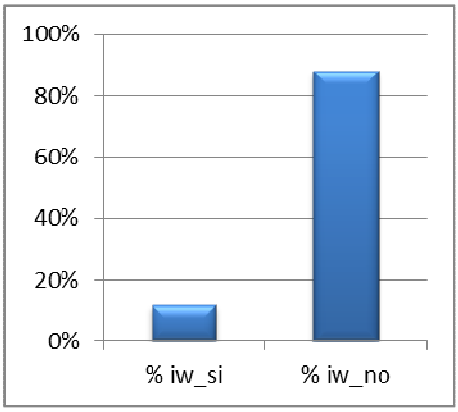


Figura 3.12: Porcentaje de éxito de las inserciones debidas a writeback para soplex.

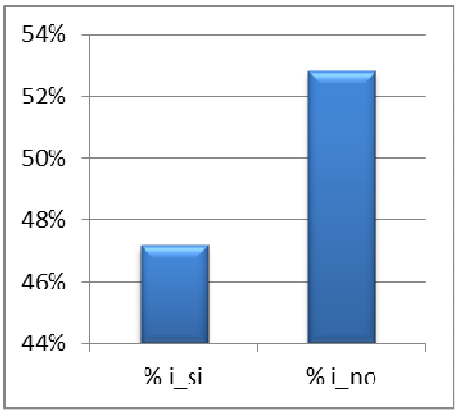


Figura 3.13: Porcentaje de éxito de las inserciones debidas a load-store para soplex.

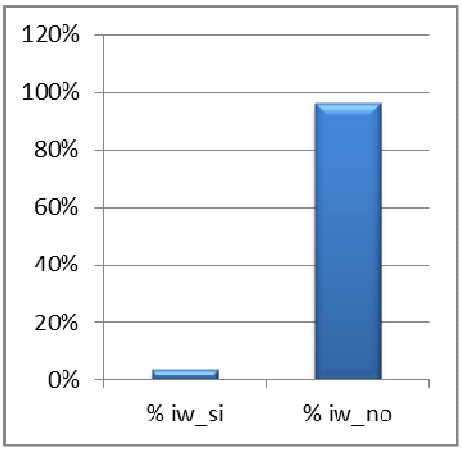


Figura 3.14: Porcentaje de éxito de las inserciones debidas a writeback para calculix.

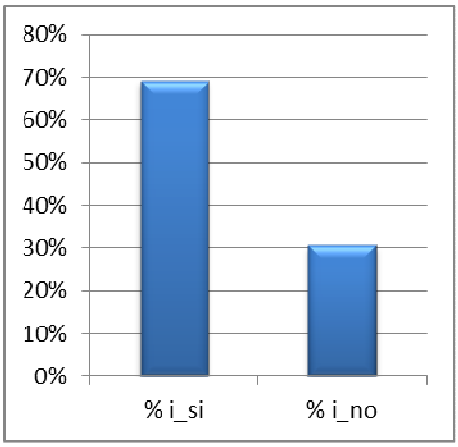


Figura 3.15: Porcentaje de éxito de las inserciones debidas a load-store para calculix.

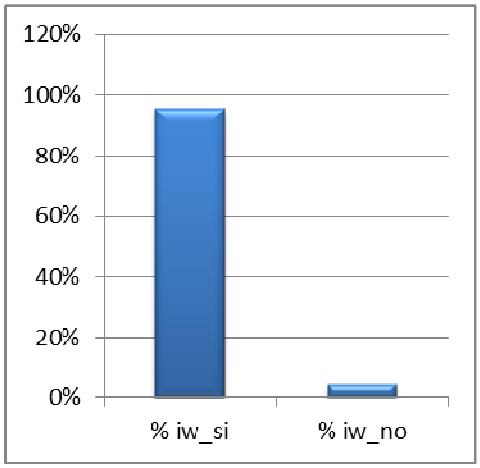


Figura 3.16: Porcentaje de éxito de las inserciones debidas a writeback para h264.

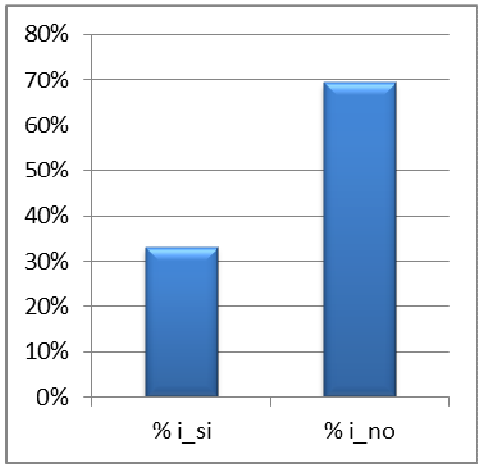


Figura 3.17: Porcentaje de éxito de las inserciones debidas a load-store para h264.

De igual forma, se detectan comportamientos análogos en cuanto a las promociones debidas a load-store con respecto a aquellas debidas a *writeback*.

3.3 Propuestas de mejora de la política de reemplazo

Teniendo en cuenta la uniformidad en el trato por parte de la política de reemplazo que se ha encontrado en todos los trabajos examinados, para todo tipo de evento (inserciones y promociones) que sucede en el LLC y teniendo en cuenta las observaciones hechas en el apartado anterior recopiladas a continuación:

- Compensación de los porcentajes de ocurrencia por parte de los distintos tipos de eventos.
- Descubrimiento de patrones referentes a las distintas partes de la *recency stack* en la que tienen éxito en su promoción bloques cuyo último evento anterior fue uno dado entre los cuatro posibles.
- Descompensación en cuanto al éxito en su reúso entre los eventos debidos a *writeback* con respecto a los que no lo son.

Se procede a explicar la idea que se propone en este trabajo de investigación para incorporar en la política de reemplazo las observaciones previas. Por un lado, en las gráficas de la 3.5 a 3.9 se puede apreciar como en las posiciones de la primera mitad de la *recency stack* se produce el grueso de promociones para cada bloque sea cual sea el último evento que sufrió el mismo. El objetivo sería intentar que no todos los patrones compitieran por las mismas posiciones de la *recency stack* donde están recibiendo la mayor parte de sus promociones desplazando alguno o algunos de ellos a otra parte de la *recency stack* mediante la modificación de los puntos de inserción/promoción. Aplicado a la política de reemplazo LRU, un bloque que es insertado o recibe una promoción en la cache, es inmediatamente promocionado a la posición MRU de la *recency stack*. De lo que se trata es de que no todos los bloques insertados o promocionados en el último nivel de cache sean insertados en la posición MRU sino que dependiendo del patrón asociado unos sí que sean insertados en esta posición y otros en posiciones más alejadas de ésta. De esta forma potenciaríamos aquellos patrones que están recibiendo un mayor número de promociones en unas determinadas posiciones de la *recency stack* sin que ello ocasione un grave perjuicio al resto de ellos.

De igual forma se piensa que es razonable el planteamiento de que un trato diferente, por parte de la política de reemplazo, de las inserciones debidas a *writeback* con respecto a aquellas debidas a load-store, podría mejorar la toma de decisiones de esta política. Se propone entonces favorecer aquellos eventos que están teniendo más éxito en su reúso, insertándolos o promocionándolos a la posición MRU o posiciones cercanas a ésta, en detrimento de aquellos eventos que muestran peores resultados insertándolos o promocionándolos en posiciones más alejadas de MRU. Las modificaciones concretas a la hora de insertar o promocionar los bloques dependiendo del tipo de evento serán detalladas en el capítulo 5.

En base a todo esto se exponen tres propuestas con la intención de mejorar el rendimiento de la política de reemplazo en el último nivel de cache y por extensión el rendimiento global del sistema.

3.3.1 Mejora de la tasa de aciertos en un último nivel de cache de tamaño típico

Para este caso se utiliza una jerarquía de cache de tres niveles. En el último nivel se hace uso de una política de reemplazo LRU, una política de actualización *writeback*, una política de inclusión *no inclusiva* y unas características en cuanto a capacidad (1MB) y asociatividad (16 vías) típicas en un procesador actual. La finalidad de esta propuesta es comprobar si introduciendo las modificaciones oportunas de los puntos de inserción en la *recenty stack*, para cada tipo de inserción y promoción en el último nivel de cache, se consigue mejorar el rendimiento de la jerarquía de memoria.

3.3.2 Aproximación al nivel de acierto de una cache típica de una con tamaño reducido.

El objetivo de esta propuesta es acercar el rendimiento de una cache que hace uso de un tercer nivel con la mitad de posibilidades en cuanto a tamaño (512 KB) y asociatividad (8 vías) al de una cache típica de un procesador actual, basada en un tercer nivel de cache de 1MB y una asociatividad de 16 vías. Para la consecución de este fin y en base a las observaciones expuestas anteriormente se modificaran convenientemente los puntos de inserción y promoción.

3.3.3 Integración de las medidas propuestas en entornos multicore

La finalidad de esta propuesta es comprobar el impacto que tendría la aplicación de las propuestas de mejora de la política de reemplazo sobre el último nivel de cache en un entorno multi-core. Para ello se harán pruebas primero sin escalado de la cache (manteniéndola en 1MB) y posteriormente se harán pruebas escalando la cache. En el caso dual-core se escala la

cache a 2MB y en el caso quad-core primero se harán pruebas con un escalado de 2MB y por último con un escalado de 4MB.

Capítulo 4

Entorno Experimental

En este capítulo se hace una descripción del entorno de simulación empleado en la realización de las pruebas llevadas a cabo para evaluar las técnicas que se han propuesto en este trabajo, así como de los programas de los que se ha hecho uso.

4.1 Simulador arquitectónico

El simulador utilizado en este trabajo de investigación es gem5, que haciendo referencia a [12] nace como resultado de la mezcla de los mejores aspectos de dos simuladores anteriores, M5 y GEMS. M5 le provee de un marco de simulación altamente configurable además de la posibilidad de utilizar múltiples ISAs y diversos modelos de CPU. GEMS complementa estas características con un detallado y flexible sistema de memoria, incluyendo soporte para múltiples protocolos de coherencia cache y modelos de interconexión. Actualmente, gem5 soporta la mayoría de ISAs comerciales (ARM, ALPHA, MIPS, Power, SPARC y x86) además de incluir soporte para Linux para tres de ellas (ARM, ALPHA y x86).

El proyecto es el resultado de la combinación de varias instituciones académicas e industriales entre las que se encuentran AMD, ARM, HP, MIPS, Princeton, MIT y de las universidades de Michigan, Texas y Wisconsin.

El alto nivel de colaboración en el proyecto gem5 combinado con el éxito que tuvieron los simuladores en base a los cuales surgió gem5 así como el uso de una licencia BSD que permite el uso de código fuente en software no libre hacen de gem5 una herramienta muy valiosa para la simulación de sistemas computacionales completos.

4.2 Configuración del simulador

Todas las pruebas se han desarrollado sobre una ISA x86 haciendo uso tanto de un entorno single-core como de un entorno multi-core. La jerarquía de cache simulada es de tres niveles, los dos primero privativos para cada core y un tercer nivel compartido por todos ellos y donde, partiendo de una política LRU convencional, se han programado los cambios que se proponen en este trabajo. La política de inclusión simulada es no inclusiva y la política de reemplazo para el resto de niveles de cache es LRU original. La memoria principal tiene una capacidad de 128 MB. La tabla 4.1 muestra el escenario en cuanto a configuración de la jerarquía cache. En el entorno dual-core además de hacer pruebas manteniendo la capacidad del LLC en 1MB también se realizaron otras escalando su capacidad a 2MB. De igual forma en el entorno quad-core se realizaron además pruebas escalando el LLC primero a 2MB y después a 4MB.

nivel	*tamaño (KB)*	*asociatividad (vías)*	*latencia (ciclos)*	*bloque (bytes)*
L1 d	*32*	*8*	*2*	*64*
L1 i	*32*	*8*	*2*	*64*
L2	*256*	*8*	*20*	*64*
L3	*512/1024*	*8/16*	*60*	*64*

Tabla 4.1: Configuración de la jerarquía cache.

Todas las pruebas se han realizado sobre un total de 1000 millones de instrucciones ejecutadas tras un forwarding de 100 millones. En el caso multi-core, la simulación finaliza cuando la aplicación ejecutada sobre uno de los cores (cualquiera que sea) alcanza esta cifra.

Las figuras 4.1 y 4.2 representan el escenario de trabajo simulado en gem5 para el desarrollo de las diferentes pruebas en los entornos single y multi-core respectivamente.

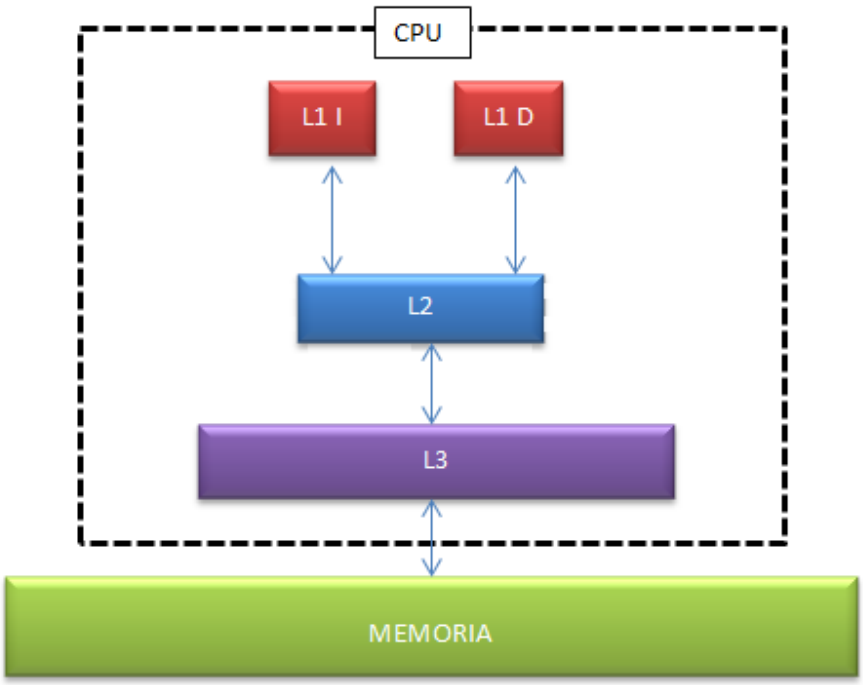


Figura 4.1: Escenario de trabajo simulado en gem5 para el entorno single-core.

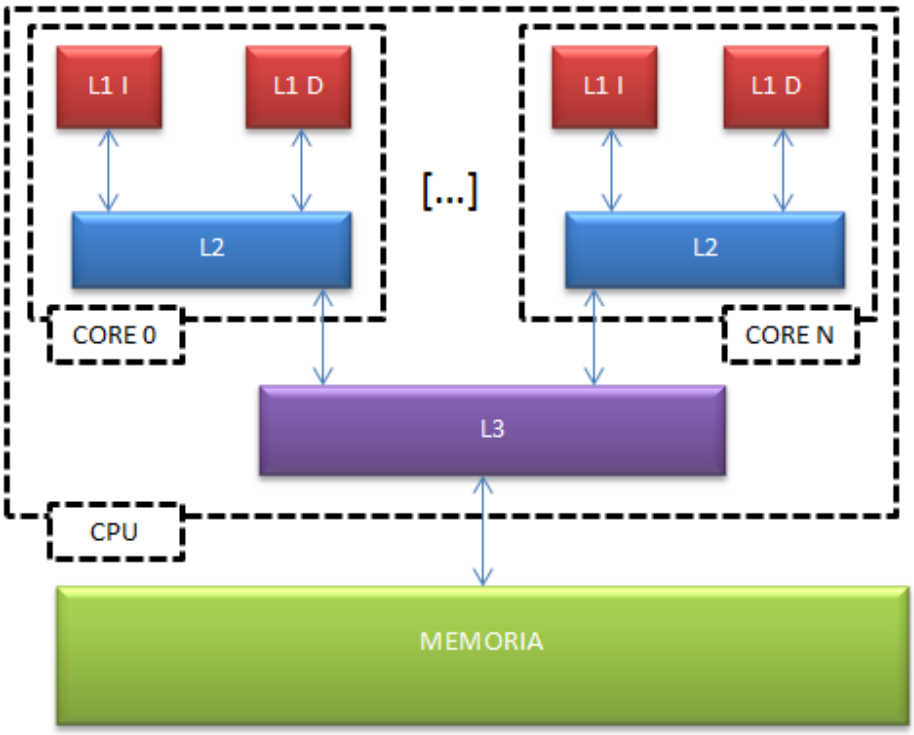


Figura 4.2: Escenario de trabajo para entornos dual-core y quad-core.

4.3 Benchmarks utilizados

La batería de programas utilizados para medir los progresos alcanzados en las pruebas realizadas son los proporcionados por la Standard Performance Evaluation Corporation (SPEC) la cuál es una corporación sin ánimo de lucro formada para establecer, mantener y apoyar un conjunto estandarizado de benchmarks cuya ejecución sobre computadoras de alto rendimiento tiene un comportamiento revelador. SPEC desarrolla suites de benchmarks y también revisa y pública los resultados generados por miembros de su organización.

La batería de benchmarks de los que se compone una suite SPEC están elegidos para tratar de estresar de alguna forma el comportamiento las plataformas o sistemas

computacionales sobre los que son ejecutados con el fin de tener un referente a la hora de poder evaluar las mejoras introducidas.

La tabla 4.2 muestra el conjunto de benchmarks utilizados en las simulaciones llevadas a cabo, todos ellos pertenecientes a la suite SPEC2006.

Benchmarks suite SPEC2006	
400.perlbench	445.gobmk
401.bzip2	450.soplex
403.gcc	454.calculix
429.mcf	456.hmmer
433.milc	459.GemsFDTD
434.zeusmp	464.h264ref
435.gromacs	470.lbm
436.cactusADM	471.omnetpp
437.leslie3d	473.astar
444.namd	482.sphinx3

Tabla 4.2: Relación de benchmarks sobre los que se han realizado las pruebas.

Capítulo 5

Resultados experimentales

En este capítulo, se ponen de manifiesto los resultados obtenidos en el estudio de cada una de las propuestas de mejora a las que se hizo referencia en el apartado 3 del capítulo 3. En relación con esto, podemos dividir el capítulo en los siguientes tres sub-apartados:

5.1 Mejora del rendimiento de una LLC de tamaño convencional en un entorno single-core

A modo de resumen, se puede decir que el objetivo de esta propuesta era el de conseguir mejorar el rendimiento de una cache típica en un procesador actual. Para la consecución de este fin, en el tercer y último nivel de cache y haciendo uso de una política de reemplazo LRU, se modificaron los puntos de inserción/promoción en la Recency Stack (que en la política original es siempre MRU) dependiendo del tipo de inserción/promoción de que se trate (i-ls ó i-w, p-ls ó p-w). Todo ello con el fin de favorecer la estancia en la cache de aquellos bloques que reciben más aciertos y por el contrario evacuar aquellos que menos lo hacen.

La configuración de la cache para este tipo de pruebas fue la que se muestra la tabla 4.1 del capítulo 4 para el entorno single-core con un tamaño de L3 de 1MB.

Teniendo en cuenta las observaciones expuestas en el capítulo 3, se proponen las modificaciones de los puntos de inserción/promoción en la recency stack reflejados en la tabla 5.1, donde cada casilla indica para un determinado tipo de evento el desplazamiento elegido

con respecto a la posición MRU de la recency stack. Así pues, por ejemplo para las inserciones debidas a writeback en el benchmark 400.perlbench se elige la posición MRU+4 de la recency stack. Por otra parte, no se ha realizado ninguna modificación en los benchmarks de los que, una vez estudiado su comportamiento, no se esperaba obtener ningún resultado positivo conforme a la propuesta que se plantea.

Benchmark	*inserciones load-store*	*inserciones writeback*	*promociones load-store*	*promociones writeback*
400.perlbench		*4*		*4*
401.bzip2				*4*
403.gcc				
429.mcf				
433.milc	*6*	*9*		
434.zeusmp				*8*
435,gromacs				
436.cactusADM			*4*	*9*
437.leslie3d				*10*
444.namd		*10*		*lru*
445.gobmk		*4*		*2*
450.soplex		*8*		
454.calculix				
456.hmmer	*5*			
459.GemsFDTD				
464.h264				
470.lbm				
471.omnetpp		*3*		*3*
473.astar				
482.sphinx3				

Tabla 5.1: Modificación de los puntos de inserción/promoción para cache típica.

Los porcentajes de mejora obtenidos, en cuanto a reducción del tiempo de ejecución son las que se exponen en la última columna de la tabla 5.2. Decir tiene que aunque las observaciones realizadas son prometedoras, estos resultados son de momento muy modestos.

Benchmark	*tiempo de ejecución versión original (seg.)*	*tiempo de ejecución versión modificada (seg.)*	*mejora (seg.)*	*mejora (%)*
400.perlbench	*0,6491*	*0,645*	*0,0034*	*0,53%*
401.bzip2	*0,5458*	*0,5456*	*0,0002*	*0,04%*
403.gcc	*1,1766*	*1,1766*	*0*	*0,00%*
429.mcf	*2,8376*	*2,8376*	*0*	*0,00%*
433.milc	*0,9161*	*0,9144*	*0,0016*	*0,18%*
434.zeusmp	*0,495*	*0,4949*	*5,8E-05*	*0,01%*
435.gromacs	*0,4778*	*0,4778*	*0*	*0,00%*
436.cactusADM	*0,5848*	*0,5845*	*0,0003*	*0,05%*
437.leslie3d	*0,6732*	*0,6732*	*2,5E-05*	*0,01%*
444.namd	*0,4252*	*0,425*	*0,0002*	*0,06%*
445.gobmk	*0,8213*	*0,820*	*0,0007*	*0,089%*
450.soplex	*2,9241*	*2,9236*	*0,0004*	*0,017%*
454.calculix	*0,389*	*0,389*	*0*	*0,00%*
456.hmmer	*0,386*	*0,3859*	*0,0001*	*0,03%*
459.GemsFDTD	*0,5527*	*0,5527*	*0*	*0,00%*
464.h264ref	*0,4963*	*0,4963*	*0*	*0,00%*
470.lbm	*3,3851*	*3,3851*	*0*	*0,00%*
471.omnetpp	*0,4998*	*0,4991*	*0,0006*	*0,14%*
473.astar	*0,8041*	*0,8041*	*0*	*0,00%*
482.sphinx3	*0,4327*	*0,4327*	*0*	*0,00%*
TOTALES	*19,4740*	*19,465*	*0,0080*	*0,06%*

Tabla 5.2: Mejoras del tiempo de ejecución para cache típica.

La mejora obtenida sin sólo consideráramos los benchmarks en los que se han realizados cambios en sus puntos de inserción/promoción sería de un 0.1%

5.2 Aproximación del rendimiento de una LLC de tamaño reducido al de una LLC de tamaño convencional en un entorno single-core

En esta segunda propuesta se trata de aproximar todo lo posible el rendimiento de una LLC de tamaño reducido, al de una LLC de tamaño convencional (1MB de capacidad y asociativa por 16 vías), por medio de la modificación de los puntos de inserción/promoción en función del tipo de evento de que se trate.

La configuración de la cache para este tipo de pruebas fue la que se muestra la tabla 4.1 capítulo 4 para el entorno single-core con un tamaño de L3 de 512KB.

En la última columna la tabla 5.3 se muestra el porcentaje extra, en cuanto a tiempo de ejecución, que tarda más la versión con un último nivel de cache de 512KB y 8 vías en la ejecución de los benchmarks con respecto a la versión configurada con 1MB de capacidad y 16 vías en su último nivel de cache. En ambos casos no se ha hecho ninguna modificación en los puntos de inserción/promoción debido a que en este paso previo se está calculando las diferencias existentes entre las dos caches cuyo rendimiento queremos aproximar.

Benchmark	*Tiempo de ejecución 1M 16W*	*Tiempo de ejecución 512KB 8W*	*Tiempo de ejecución diferencia 512 KB - 1M*	*Incremento de tiempo de ejecución (%)*
400.perlbench	*0,649*	*0,651*	*0,002*	*0,32%*
401.bzip2	*0,545*	*0,577*	*0,031*	*5,81%*
403.gcc	*1,176*	*1,202*	*0,026*	*2,23%*
429.mcf	*2,837*	*2,884*	*0,047*	*1,66%*
433.milc	*0,916*	*0,917*	*0,001*	*0,15%*
434.zeusmp	*0,495*	*0,496*	*0,001*	*0,29%*
435.gromacs	*0,477*	*0,484*	*0,006*	*1,30%*
436.cactusADM	*0,584*	*0,5921*	*0,007*	*1,23%*
437.leslie3d	*0,673*	*0,675*	*0,002*	*0,36%*
444.namd	*0,425*	*0,425*	*-1E-06*	*0,00%*
445.gobmk	*0,821*	*0,826*	*0,005*	*0,63%*
450.soplex	*2,924*	*3,025*	*0,101*	*3,46%*
454.calculix	*0,38*	*0,392*	*0,002*	*0,71%*
456.hmmer	*0,38*	*0,436*	*0,050*	*13,01%*
459.GemsFDTD	*0,552*	*0,552*	*0*	*0,00%*
464.h264ref	*0,496*	*0,503*	*0,007*	*1,41%*
470.lbm	*3,385*	*3,387*	*0,002*	*0,06%*
471.omnetpp	*0,499*	*0,499*	*-0,0003*	*-0,06%*
473.astar	*0,804*	*0,823*	*0,018*	*2,35%*
482.sphinx3	*0,432*	*0,433*	*0,000*	*0,13%*
TOTALES	*19,4740*	*19,788*	*0,314*	*1,75%*

Tabla 5.3: Comparación de tiempos de ejecución utilizando un último nivel de cache con 1MB de capacidad y asociatividad de 16 vías con respecto a utilizar uno de 512KB y 8 vías sin modificar puntos de inserción/promoción.

La modificación de los puntos de inserción/promoción en la recency stack en estas pruebas son los que indica la tabla 5.4. Como ya se ha indicado en casos anteriores, la elección de estas modificaciones están basadas en las observaciones expuestas en el capítulo 3.

Benchmark	*inserciones load-store*	*inserciones writeback*	*promociones load-store*	*promociones writeback*
400.perlbench				
401.bzip2		*3*		*3*
403.gcc				
429.mcf				
433.milc		*2*		*7*
434.zeusmp		*1*		*4*
435.gromacs		*3*		
436.cactusADM			*2*	*5*
437.leslie3d		*4*		*6*
444.namd		*6*		*7*
445.gobmk				
450.soplex		*5*		*4*
454.calculix		*3*		
456.hmmer			*3*	*4*
459.GemsFDTD				
464.h264				
470.lbm				
471.omnetpp				
473.astar				
482.sphinx3				*4*

Tabla 5.4: Modificación de los puntos de inserción/promoción en el último nivel de cache con una capacidad de 512KB y asociativa por 8 vías.

En base a estas modificaciones se obtuvieron los tiempos de ejecución que se muestran en la tabla 5.5. En la última columna de esta tabla se pueden observar cual es la nueva diferencia en cuanto al tiempo de ejecución extra que necesita la cache con mitad de posibilidades con respecto a la cache típica, pero en este caso habiendo introducido los

cambios en los puntos de inserción/promoción. De nuevo se obtienen unos resultados modestos.

Benchmark	*Tiempo de ejecución 1M 16W org*	*Tiempo de ejecución 512KB 8W modificada*	*diferencia 512 KB mod - 1M org*	*diferencia en %*
400.perlbench	*0,649*	*0,651*	*0,002*	*0,32%*
401.bzip2	*0,545*	*0,577*	*0,031*	*5,74%*
403.gcc	*1,176*	*1,203*	*0,026*	*2,23%*
429.mcf	*2,837*	*2,885*	*0,047*	*1,66%*
433.milc	*0,916*	*0,915*	*-0,001*	*-0,08%*
434.zeusmp	*0,495*	*0,496*	*0,001*	*0,21%*
435.gromacs	*0,477*	*0,48*	*0,006*	*1,27%*
436.cactusADM	*0,584*	*0,592*	*0,007*	*1,22%*
437.leslie3d	*0,673*	*0,675*	*0,002*	*0,30%*
444.namd	*0,425*	*0,425*	*-0,000*	*-0,05%*
445.gobmk	*0,821*	*0,826*	*0,005*	*0,63%*
450.soplex	*2,924*	*3,024*	*0,101*	*3,44%*
454.calculix	*0,389*	*0,388*	*-0,001*	*-0,27%*
456.hmmer	*0,386*	*0,436*	*0,050*	*12,98%*
459.GemsFDTD	*0,552*	*0,552*	*0*	*0,00%*
464.h264ref	*0,496*	*0,503*	*0,001*	*1,41%*
470.lbm	*3,385*	*3,387*	*0,002*	*0,06%*
471.omnetpp	*0,499*	*0,499*	*-0,0001*	*-0,06%*
473.astar	*0,804*	*0,823*	*0,019*	*2,35%*
482.sphinx3	*0,432*	*0,433*	*0,0001*	*0,13%*
TOTALES	*19,474*	*19,779*	*0,306*	*1,67%*

Tabla 5.5: Comparación de tiempos de ejecución entre utilizar un último nivel de cache con 1MB de capacidad y asociativa por 16 vías en el que no se han modificado los puntos de inserción/promoción con respecto a utilizar uno de 512KB y 8 vías en la que sí se ha hecho tal modificación.

5.3 Mejora del rendimiento de una LLC en un entorno multi-core

En este caso, se intenta evaluar el impacto que puede tener, en cuanto a mejora de rendimiento, la modificación de los puntos de inserción/promoción en un entorno multi-core. Esta propuesta es a su vez dividida en dos, la primera de ellas para un entorno dual-core y la segunda para un entorno quad-core.

5.3.1 Propuesta para entorno dual-core

En este apartado se describen los resultados obtenidos de las pruebas realizadas para un entorno dual-core. La configuración de la cache en este caso fue el que se muestra en la tabla 4.1 del capítulo 4. La primera prueba se hace con una LLC de 1 MB y posteriormente se hace otra prueba escalando ésta a 2MB.

Para la realización de estas pruebas se compusieron una serie de mezclas de dos benchmarks, cada uno de los cuales será ejecutado en un core distinto. Estas mezclas fueron elegidas en base a que los benchmarks que las compusieran tuvieran aproximadamente el mismo número de accesos a L3 y son las que se muestran en la tabla 5.6.

Mecla	*Benchmarks*
mix.0	*454. calculix y 464.h264*
mix.1	*435.gromacs y 400.perlbench*
mix.2	*454. calculix y 400.perlbench*
mix.3	*464.h264 y 400.perlbench*
mix.4	*456.hmmer y 473.astar*
mix.5	*456.hmmer y 401.bzip*
mix.6	*456.hmmer y 403.gcc*
mix.7	*473.astar y 401.bzip*
mix.8	*473.astar y403.gcc*

Tabla 5.6: Benchmarks elegidos para las simulaciones en dual-core.

En cuanto a la modificación de los puntos de inserción/promoción, en este caso se realizaron los cambios que expone la tabla 5.7. Una vez más recordar que las elecciones de los puntos de inserción/promoción están basadas en las apreciaciones que se pusieron de manifiesto en el capítulo 3.

Benchmark	*inserciones load-store*	*inserciones writeback*	*promociones load-store*	*promociones writeback*
400.perlbench	*4*	*10*		*7*
401.bzip2	*8*			
403.gcc	*9*			
435.gromacs	*7*			
454.calculix			*6*	
456.hmmer	*lru*		*lru*	
464.h264		*4*		*4*
473.astar	*3*	*13*	*3*	

Tabla 5.7: Modificación de los puntos de inserción para las pruebas dual-core.

Debido a que en el entorno multi-core la simulación termina cuando una de las aplicaciones de las que componen la simulación llega a la cantidad de instrucciones especificada, no todos los cores ejecutan el mismo número de instrucciones. Es por eso que tomar el tiempo como medida para evaluar los avances obtenidos con la propuesta que se plantea no proporcionaría datos fiables. Se propone pues, tomar la suma del IPC (*Instructions Per Cycle*) de cada uno de los cores como medida para evaluar los resultados en las pruebas tanto de este apartado como del siguiente (dual-core y quad-core). En las tablas 5.8 y 5.9 se pueden ver los porcentajes de las mejoras obtenidas tanto para la versión en la que no se ha escalado el último nivel de cache (1MB) como para el caso en el que si se ha hecho esta modificación (2MB) respectivamente.

MEZCLAS	IPC original	IPC modificada	IPC(mod - org)	mejora %
mix0 (calculix-h264)	2,047	2,047	0	0,00%
mix1 (gromacs-perlbench)	1,916	1,924	0,008	0,42%
mix2 (calculix-perlbench)	1,924	1,924	0	0,00%
mix3 (h264-perlbench)	1,802	1,807	0,005	0,28%
mix4 (hmmer-astar)	1,74	1,758	0,018	1,03%
mix5 (hmmer-bzip)	1,895	1,895	0	0,00%
mix6 (hmmer-gcc)	1,666	1,697	0,031	1,86%
mix7 (astar-bzip)	1,318	1,318	0	0,00%
mix8 (astar-gcc)	1,112	1,113	0,001	0,09%
TOTAL	15,42	15,483	0,063	0,41%

Tabla 5.8: Mejoras obtenidas modificando los puntos de inserción/promoción en un entorno dual-core con un último nivel de cache de 1MB de capacidad.

MEZCLAS	IPC original	IPC modificada	IPC(mod - org)	mejora %
mix0 (calculix-h264)	2,052	2,052	0,000	0,01%
mix1 (gromacs-perlbench)	1,943	1,949	0,006	0,32%
mix2 (calculix-perlbench)	1,933	1,936	0,003	0,18%
mix3 (h264-perlbench)	1,817	1,817	0,000	0,00%
mix4 (hmmer-astar)	1,867	1,872	0,005	0,28%
mix5 (hmmer-bzip)	2,083	2,084	0,001	0,07%
mix6 (hmmer-gcc)	1,82	1,836	0,016	0,91%
mix7 (astar-bzip)	1,393	1,397	0,004	0,26%
mix8 (astar-gcc)	1,134	1,139	0,005	0,45%
TOTAL	16,042	16,084	0,042	0,27%

Tabla 5.9: Mejoras obtenidas modificando los puntos de inserción/promoción en un entorno dual-core con un último nivel de cache de 2MB de capacidad.

Como se puede observar, a medida que se amplía el último nivel de cache una mayor parte del conjunto de trabajo cabe en este nivel y como consecuencia las mejoras obtenidas en general son menores. Este es un comportamiento que se apreciará, incluso acentuará, para las pruebas realizadas en el siguiente subapartado para el entorno quad-core.

5.3.2 Propuesta para entorno quad-core

Este apartado se muestra como una extensión del apartado anterior, en el que se describen los resultados para un entorno con cuatro cores. De la misma forma que para apartados anteriores la configuración de la cache en este caso fue el que se muestra en la figura 4.1 del capítulo 4 con una capacidad de L3 de 1MB. Posteriormente se realizan pruebas escalando este nivel a 2MB y a 4MB.

Las mezclas de benchmarks utilizadas para estas pruebas son las que se muestran en la tabla 5.10. Al igual que en dual-core, se ha tratado de obtener mezclas en las que las aplicaciones realicen un número de accesos a L3 similar.

Mecla	*Benchmarks*
mix_quad.0	*400.perbench, 436.cactus, 445.gobmk y 456.hmmer*
mix_quad.1	*401.bzip, 403.gcc, 456.hmmer y 473.astar*
mix_quad.2	*401.bzip, 434.zeusmp, 436.cactus y 464.h264*
mix_quad.3	*450.soplex, 401.bzip, 434.zeusmp y 403.gcc*
mix_quad.4	*473.astar, 436.cactus, 456.hmmer y 454.caculix*

Tabla 5.10: Bechmarks elegidos para las simulaciones en quad-core.

En la tabla 5.11, se muestran los cambios de los puntos de inserción/promoción seleccionados para estas pruebas. Estos puntos son elegidos en base a las observaciones planteadas en el capítulo 3 y después de haber estudiado su comportamiento.

Benchmark	*inserciones load-store*	*inserciones writeback*	*promociones load-store*	*promociones writeback*
400.perlbench	*6*	*10*		*7*
401.bzip2	*8*			
403.gcc	*9*			
434.zeusmp				*12*
436.cactus			*8*	*lru*
445.gobmk	*3*	*6*		
450.soplex		*lru*		
456.hmmer	*lru*		*lru*	
464.h264	*8*			*5*
473.astar	*3*	*13*	*3*	

Tabla 5.11: Modificación de los puntos de inserción/promoción para pruebas quad-core.

La tabla 5.12 muestra los resultados obtenidos al modificar los puntos de inserción/promoción en el último nivel de cache con respecto a la versión original manteniendo en el mismo una capacidad de 1MB. Por último, las tablas 5.13 y 5.14 muestran los mismos datos pero en estos casos escalando la capacidad de este nivel a 2MB y 4MB respectivamente. Como ya sucedía en dual-core, observamos un empeoramiento de los resultados a medida que se escala el nivel de cache.

MEZCLA	IPC original	IPC modificada	mod - org	mejora %
mix_quad.0	3,155	3,175	0,020	0,64%
mix_quad.1	2,871	2,890	0,019	0,66%
mix_quad.2	3,019	3,017	-0,002	-0,07%
mix_quad.3	2,285	2,287	0,001	0,06%
mix_quad.4	3,306	3,311	0,005	0,15%
TOTAL	14,954	14,983	0,043	0,29%

Tabla 5.12: Mejoras obtenidas modificando los puntos de inserción/promoción en un entorno quad-core con un último nivel de cache de 1MB de capacidad.

MEZCLA	IPC original	IPC modificada	mod - org	mejora %
mix_quad.0	3,224	3,238	0,014	0,44%
mix_quad.1	2,984	3,002	0,018	0,60%
mix_quad.2	3,059	3,057	-0,003	-0,08%
mix_quad.3	2,314	2,314	0,001	0,03%
mix_quad.4	3,373	3,377	0,004	0,11%
TOTAL	14,954	14,983	0,029	0,22%

Tabla 5.13: Mejoras obtenidas modificando los puntos de inserción/promoción en un entorno quad-core con un último nivel de cache de 2MB de capacidad.

MEZCLA	IPC original	IPC modificada	mod - org	mejora %
mix_quad.0	3,340	3,342	0,002	0,06%
mix_quad.1	3,204	3,222	0,018	0,57%
mix_quad.2	3,111	3,111	0,000	0,00%
mix_quad.3	2,429	2,429	0,001	0,02%
mix_quad.4	3,472	3,472	0,000	0,01%
TOTAL	15,556	15,577	0,021	0,13%

Tabla 5.14: Mejoras obtenidas modificando los puntos de inserción/promoción en un entorno quad-core con un último nivel de cache de 4MB de capacidad.

Capítulo 6

Conclusiones y Trabajo futuro

Una vez analizados los resultados de cada una de las propuestas planteadas, se puede concluir que los resultados no son del todo satisfactorios, o no en la medida en la que preveía en un primer momento. Sin embargo, en base a las observaciones apuntadas en el Capítulo 3, se considera que la línea de investigación abierta, con la diferenciación de los distintos tipos de inserciones y promociones en la política de reemplazo, puede tener bastante potencial de cara a mejorar el rendimiento de la política de reemplazo de la cache.

Es por ello que como trabajo futuro se propone continuar con el análisis de la propuesta planteada para intentar obtener un mayor beneficio de su utilización. Por ejemplo, y como posible evolución de la propuesta original, se podrían implementar los mecanismos necesarios para que dinámicamente se puedan modificar los puntos de inserción y promoción. Dotando así a la propuesta de una capacidad de adaptación a las distintas características que la carga de trabajo puede tener a lo largo de la ejecución de una aplicación determinada.

Bibliografía

[1] Asit Dan and Don Towsley – University of Massachusetts - An approximate analysis of the lru and fifo buffer replacement schemes – Amherst. 1990

[2] M. Chaudhuri, Pseudo-LIFO: The Foundation of a New Family of Replacement Policies for Last-level Caches, Proceedings of MICRO-09, New York, 12-16 December,2009, 401–412.

[3] A. Jaleel, K. Theobald, and S. S. Jr, High performance cache replacement using re-reference interval prediction (RRIP), ISCA'10, June 19–23,2010, Saint-Malo, France.

[4] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., Joel Emer,Adaptive Insertion Policies for High Performance Caching, ISCA'07, June 9–13, 2007, San Diego, California, USA.

[5] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, Yale N. Patt, A Case for MLP-Aware Cache Replacement, International Symposium on Computer Architecture (ISCA) 2006. 167-178.

[6] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely Jr., Joel Emer, Adaptive Insertion Policies for Managing Shared Caches, PACT'08, October 25–29, 2008, Toronto, Ontario, Canada.

[7] Moinuddin K. Qureshi, Yale N. Patt, Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches, 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06), 2006, 423-432.

[8] Enrique Sedano, Silvio Sepúlveda, Fernando Castro, Daniel Chaver, Rodrigo González-Alberquilla, Francisco Tirado, Improving peLIFO cache replacement policy: Hardware reduction and thread-aware extension, World Scientific, Journal of Circuits, Systems, and Computers, Volume 23, Issue 04, April 2014.

[9] D. Hennessy, J.L. and Patterson, Computer architecture: a quantitative approach. Morgan Kaufmann Pub, 2011.

[10] Roberto Alonso Rodríguez Rodríguez, Evaluación y optimización de políticas de reemplazo caché en entornos PCM, Máster en Investigación en Informática, Facultad de Informática, Departamento de Arquitectura de Computadores y Automática, curso 2011-2012 .

[11] Jorge Albericio, Pablo Ibáñez, Víctor Viñals, José M. Llabería, The Reuse Cache: Downsizing the Shared Last-Level Cache, MICRO-46: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, December 2013

[12] Nathan Binkert, Bradford Beckmann. The gem5 Simulator. ACM SIGARCH Computer Architecture News. Vol. 39, No. 2, May 2011.