Incremento en las prestaciones del algoritmo pMRU mediante el uso de una cache víctima

Alejandro Valero, Julio Sahuquillo, Salvador Petit, Pedro López y José Duato [1]

Resumen—
La latencia de memoria se ha convertido en un aspecto fundamental de diseño en los microprocesadores actuales. Para reducir esta latencia, las caches de último nivel forman estructuras de memoria grandes con un gran número de vías. El algoritmo de reemplazo implementado normalmente en caches es el LRU. Sin embargo, en caches de último nivel las prestaciones de LRU se degradan con el número de vías.

Este trabajo define el número de periodos MRU (pMRU) de un bloque como el número de veces que un bloque ocupa la posición MRU mientras permanece en la cache, y propone el algoritmo de reemplazo pMRU, que selecciona la víctima al azar entre aquellos bloques que tienen un solo pMRU. También se proponen variaciones de este algoritmo para explotar la recencia de información y los cambios en el comportamiento de las aplicaciones en tiempo de ejecución, además de incorporar una cache víctima para mitigar el efecto de la componente aleatoria.

Los resultados muestran que, en la media, la mejor versión de algoritmo pMRU obtiene una reducción de MPKI de un 22% respecto a LRU. Comparado con un conjunto de políticas recientes, la reducción de MPKI se encuentra entre un 10% y 11%. Finalmente, añadir una cache víctima pequeña ayuda al algoritmo a reducir el MPKI hasta en un 24% respecto a LRU.

Palabras clave— **Algoritmo de reemplazo en caches, cache víctima, caches de último nivel, periodo MRU.**

I. Introducción

La jerarquía de memoria de los microprocesadores actuales juega un papel fundamental a la hora de reducir las diferencias de velocidad entre el procesador y la memoria principal. La latencia de acceso a memoria se ha convertido en un aspecto de diseño crítico debido al crecimiento de la velocidad del procesador. Es por ello que el diseño y almacenamiento de datos en las memorias cache debe ser eficiente. Las prestaciones del sistema dependen directamente de que los datos que exhiben localidad temporal y vayan a ser utilizados por el procesador se encuentren en las caches, mientras que aquellos datos que ya no se requieren o que no vayan a ser utilizados próximamente se puedan identificar y expulsar de la cache.

En general, las caches explotan la localidad temporal mediante el algoritmo de reemplazo LRU *(Least Recently Used)*. Este algoritmo actúa como una pila que contiene el bloque MRU *(Most Recently Used)* en la primera posición y el bloque LRU en la última, siendo este bloque la víctima cuando se requiere espacio. Este algoritmo resulta eficiente en caches de L1 con poca asociatividad, pero en caches de último nivel con asociatividad mayor como 8 y 16 vías, la implementación del algoritmo LRU tradicional resulta demasiado compleja. Se han propuesto aproximaciones a LRU, pero las prestaciones obtenidas se apartan de las de LRU tradicional [1]. Por otra parte, las prestaciones de LRU se encuentran lejos de las obtenidas por el algoritmo de reemplazo óptimo conocido como algoritmo de Belady [2], que reduce los fallos de cache a un mínimo teórico eligiendo como víctima el bloque que se referenciará más tarde en el futuro.

El algoritmo LRU no obtiene buenas prestaciones en caches de último nivel por varias razones: i) L1 actúa como filtro de L2, con lo que se pierde la localidad temporal en L2, ii) LRU sufre el efecto *cache thrashing* en aquellas aplicaciones donde la cantidad de datos es mayor que la capacidad de cache disponible, resultando en accesos de bloque cíclicos, con lo cual los bloques recorren la pila sin ser utilizados de nuevo y cuando se requieren ya han sido reemplazados y iii) LRU fuerza que un bloque tenga que llegar hasta la última posición para ser la víctima de reemplazo. Sin embargo, como los resultados experimentales mostrarán, muchos bloques almacenados en L2 no vuelven a ser utilizados de nuevo.

La mayoría de bloques no se referencian otra vez cuando dejan la posición MRU. Además, la probabilidad de un bloque de ser referenciado otra vez no depende siempre del orden que ocupa en la pila LRU. Esto significa que, en general, el orden de acceso no es importante para las aplicaciones en caches con un gran número de vías porque la probabilidad no disminuye conforme el bloque desciende por la pila. Sin embargo, la política LRU necesita un número de bits de estado por bloque considerable para mantener el orden de la pila.

En [3] [4] [5], hemos definido el número de periodos MRU (pMRU) de un bloque como el número de veces que un bloque accede a la posición MRU durante su tiempo de vida. Teniendo en cuenta que la mayoría de bloques exhiben un pMRU, se propuso el algoritmo pMRU con el objetivo de explotar este comportamiento para mejorar las prestaciones. Los bloques que muestran un pMRU son considerados en el reemplazo, seleccionando uno de ellos al azar. La complejidad del algoritmo se reduce considerablemente respecto a LRU.

Este artículo extiende los trabajos [3] [4] [5] de tres maneras principales: i) mejoramos el algoritmo pMRU para que se adapte a los cambios en el comportamiento de las aplicaciones, ii) presentamos un análisis de los patrones pMRU que muestran los bloques e introducimos una cache víctima para capturar aquellos que presentan localidad temporal pero que

[1]Grupo de Arquitecturas Paralelas, Universitat Politècnica de València, e-mails: `alvabre@gap.upv.es`, {`jsahuqui`, `spetit`, `plopez`, `jduato`}`@disca.upv.es`.

la política los ha expulsado de L2 y iii) comparamos los resultados de la política con un conjunto de propuestas publicadas recientemente.

El algoritmo pMRU propuesto mejora las prestaciones respecto a LRU y otros algoritmos propuestos recientemente. La mejor versión del algoritmo se refiere como pMRU-b3-adp, y consigue reducir el MPKI en un 22% respecto a LRU, mientras que comparado con otros algoritmos recientes, la reducción se encuentra entre un 10% y 11%. Finalmente, añadir una cache víctima ayuda al algoritmo propuesto a que el MPKI se reduzca en un 24% respecto a LRU sin cache víctima.

El resto del trabajo se organiza como sigue. La Sección II enumera los trabajos relacionados. La Sección III muestra los resultados que motivan la propuesta de este trabajo. La Sección IV presenta el algoritmo pMRU y la propuesta que hace uso de la recencia de información. La Sección V analiza los patrones pMRU de los bloques e introduce la cache víctima. La sección VI muestra las prestaciones frente a otros algoritmos, las mejoras del algoritmo original y los resultados con cache víctima. Finalmente, las conclusiones de este trabajo se presentan en la Sección VII.

II. Trabajos relacionados

Existe en la literatura una gran cantidad de trabajos que se centran en mejorar las prestaciones de la jerarquía de cache mediante algoritmos de reemplazo. En esta sección se hace referencia a un conjunto de estos trabajos que han sido publicados recientemente. Las propuestas de estos trabajos junto con la política pseudo-LRU [6] han sido implementadas y evaluadas en el presente trabajo.

El esquema Bubble, propuesto por Zhang y Xue [7], hace uso de una cola en lugar de una pila. Los bloques entrantes se almacenan en el final de la cola, que es la posición que representa la frecuencia de acceso menor. Al ser referenciados, los bloques promocionan hacia la cabeza de la cola. El bloque víctima se elige del final o la cabeza de la cola según si el acceso previo al conjunto resultó en un fallo o acierto de cache, respectivamente. Este trabajo también presenta una técnica de *divide-y-vencerás*, conocida como DC-Bubble, que divide los conjuntos en grupos de bloques independientes que aplican cada uno de ellos el algoritmo Bubble. Cuando se requiere espacio, se elige un grupo al azar, reduciendo de este modo la complejidad frente a LRU y el efecto de cache thrashing.

En [8] se proponen tres políticas adaptativas basadas en LRU. La primera de ellas, *LRU Insertion Policy* (LIP), inserta todos los bloques que llegan a la cache en la posición LRU, y promocionan a la posición MRU si se vuelven a referenciar. Este comportamiento previene a LIP del efecto de cache thrashing. La segunda política se conoce como *Bimodal Insertion Policy* (BIP) y difiere respecto a BIP en que cada x número de fallos de cache, el bloque entrante se inserta en la posición MRU. Además de protección frente a thrashing, esta política se adapta a los cambios en el conjunto de trabajo. Finalmente, *Dynamic Insertion Policy* (DIP) combina las políticas BIP y LRU mediante una estrategia llamada *Set Dueling*. DIP utiliza una porción pequeña de los conjuntos de cache para obtener las prestaciones de ambas políticas, aplicando así en el resto de conjuntos la política que mejor está funcionando.

Jaleel et al. [9] proponen una familia de algoritmos basados en la predicción del intervalo de re-referencia (RRIP) de los bloques de cache para lidiar con el efecto de cache thrashing y las ráfagas de accesos a datos no-temporales. En su versión más simple, referenciada como política *Static RRIP* (SRRIP), se utiliza un contador con saturación por bloque para predecir si el bloque será re-referenciado pronto o tarde en el futuro. En la inserción de bloque, la *distancia* de predicción se indica como *distante en el futuro* (esto es, el contador se inicia a su valor máximo), mientras que los accesos posteriores a este bloque reducen la distancia a *inmediata en el futuro* (esto es, se reduce en una unidad el valor del contador de bloque cada vez que se accede). El bloque víctima se selecciona de entre aquellos candidatos cuya predicción es distante en el futuro. Si no se encuentran candidatos, todos los contadores aumentan su cuenta en una unidad hasta que uno de ellos se satura. Una mejora de SRRP es la política *Bimodal RRIP* (BRRIP). Difiere respecto a SRRIP en que cada x número de fallos, el contador del bloque entrante se inicia al valor máximo menos uno. Finalmente, las políticas SRRIP y BRRIP se utilizan conjuntamente utilizando la técnica de Set Dueling como en [8], resultando en la política *Dynamic RRIP* (DRRIP).

III. Motivación

Con LRU, un bloque que no se accede desciende por la pila hasta que llega a la posición LRU. Durante este descenso el bloque volverá a la posición MRU sólo si se vuelve a acceder. En caches con un gran número de vías, la probabilidad de retorno a la posición MRU no se distribuye de manera uniforme en las posiciones de la pila. La Figura 1 muestra los resultados para una cache de L2 de 1MB-16vías. El término *pos2* hace referencia a la posición siguiente a la MRU en la pila, *pos3* a la posición siguiente a *pos2*, etcétera. Debido a restricciones de espacio, *pos5-7* hace referencia a las posiciones comprendidas entre la quinta y la séptima, etcétera. *pos16* hace referencia a la posición LRU. Por supuesto, la pro-

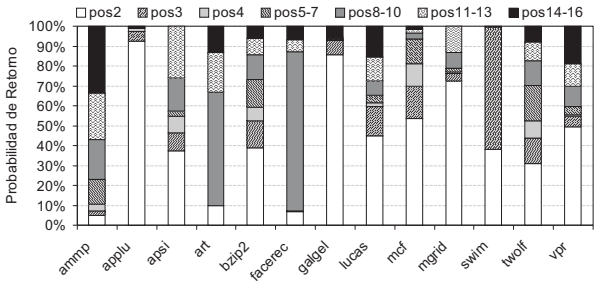

Fig. 1. Probabilidad de retorno a MRU de un bloque en la posición X utilizando el algoritmo LRU.

babilidad de referencia en la posición MRU (*pos1*) no se ha tenido en cuenta.

Aunque podemos ver que en algunas aplicaciones (p.e., *ammp* y *art*) la mayoría de los bloques se referencian en aquellas posiciones cercanas a la LRU, en la mayoría de cargas (p.e., *applu*, *galgel* y *lucas*) se aprecia una probabilidad de retorno elevada en las posiciones cercanas a la MRU. En general, mantener todo el orden de la pila LRU no es necesario en caches con un gran número de vías, y sin embargo, la política LRU requiere un número significante de bits ($log_2(n)$ siendo n la asociatividad de la cache) para mantener todo el orden.

IV. Algoritmo basado en periodos MRU

El concepto de periodo MRU (pMRU) se puede explicar mediante los conceptos de *tiempo de vida* y *tiempo de muerte* de un bloque. Estos conceptos se han utilizado de forma recurrente en la investigación [10]. El tiempo de generación de un bloque se define como el tiempo transcurrido desde que el bloque entra en la cache hasta que se elige como víctima. Este tiempo se puede dividir en los tiempos de vida y muerte de un bloque. El primero se refiere al tiempo desde que el bloque entra en la cache hasta la última referencia al mismo. El segundo alude al tiempo desde la última referencia hasta que se reemplaza. La Figura 2 ilustra el concepto de periodo MRU en el contexto del tiempo de generación del bloque A y asumiendo la política LRU. Inicialmente, el bloque entrante A se almacena en la posición MRU en tiempo *t1*. Este bloque permanece en dicha posición mientras es accedido. En el momento en que otro bloque, en nuestro caso el bloque B, es accedido, A deja esta posición. Cuando esto ocurre decimos que A ha finalizado su primer pMRU. Si A se referencia otra vez, vuelve a la posición MRU e inicia su segundo pMRU. *t2* indica el final de este pMRU que coincide con el último acceso a A. El tiempo de muerte empieza en este punto y acaba en *t3* cuando el bloque A es reemplazado.

Con el objetivo de explorar el potencial del concepto pMRU sobre el algoritmo LRU, se ha obtenido el porcentaje de bloques que presentan un solo pMRU cuando son reemplazados. La Figura 3 muestra los resultados para el conjunto de aplicaciones de SPEC2000 [11] y una cache de L2 de 1MB-16vías. En general, la mayoría de aplicaciones presentan un porcentaje elevado de bloques con un pMRU. Las aplicaciones que no presentan valor son aquellas en las que todos los fallos de L2 contabilizados son forzosos (ver Sección VI).

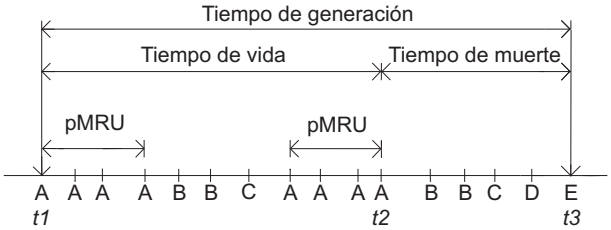

Fig. 2. Tiempo de generación del bloque A.

El algoritmo basado en pMRU hace uso de la información de uno o múltiples pMRUs de cada bloque. Para ello, requiere un bit por bloque (pMRU-bit) que indica si el mismo ha experimentado uno o varios pMRUs. El algoritmo funciona como sigue. Cada vez que un bloque entra en la cache, el pMRU-bit asociado se actualiza a '0' para indicar que su primer pMRU ha empezado. Este bloque pasa a ser candidato a reemplazo cuando deja la posición MRU por vez primera. Si, posteriormente, el bloque se referencia de nuevo, vuelve a ser el MRU y empieza otro pMRU. Esto se indica cambiando su pMRU-bit a '1'. De esta manera el bit de control indica que el bloque ha experimentado múltiples pMRUs.

El algoritmo tratará de seleccionar para reemplazo aquellos bloques que han tenido sólo un pMRU. Si un bloque exhibe buena localidad, acudirá más de una vez a la posición MRU y no será candidato a reemplazo. La víctima se selecciona al azar entre aquellos bloques que tienen un pMRU (excepto el bloque MRU) para simplificar el hardware. Si no hay ningún candidato, la víctima se selecciona al azar entre todos los bloques del conjunto excepto el MRU.

Esta primera versión del algoritmo se puede mejorar fácilmente si tenemos en cuenta la recencia de información. Como se ha visto en la Figura 1, almacenar el orden de los últimos bloques accedidos puede ser importante en términos de prestaciones para la mayoría de aplicaciones. La familia de algoritmos pMRU-bX extiende la propuesta original para explotar también la recencia de información. Estos algoritmos mantienen el orden de los últimos X bloques referenciados y no son candidatos para reemplazo. Por ejemplo, el algoritmo pMRU-b2 no considera como candidatos el bloque MRU y el inmediatamente posterior. Nótese que pMRU-b1 se refiere al algoritmo original. La complejidad se reduce respecto a LRU porque estos algoritmos no necesitan guardar todo el orden de la pila. La política pMRU-bX que mejores prestaciones obtiene es aquella en que $X = 3$ (consultar [3] [4] [5] para más detalles). En las siguientes secciones se utilizará el algoritmo pMRU-b3.

V. Análisis de los patrones pMRU

Para mejorar las prestaciones del algoritmo pMRU, resulta indispensable obtener los patrones pMRU de los bloques durante su tiempo de vida. Con esto, podremos analizar si el esquema es capaz de identificar el comportamiento del patrón.

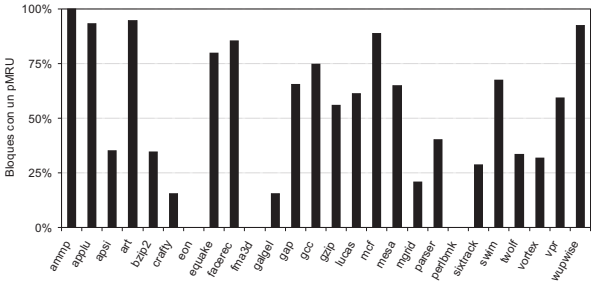

Fig. 3. Porcentaje de bloques reemplazados con sólo un pMRU.

Hemos analizado los patrones de la aplicación *art* por ser una de las cargas que obtiene peores prestaciones independientemente del algoritmo de reemplazo. La longitud de los patrones se corresponde con el número de tiempos de generación que ha experimentado el bloque durante la ejecución de *art*, y han sido obtenidos con la política LRU. Hemos identificado cuatro patrones distintos:

- **Regular-1**. Los bloques que exhiben este patrón cuentan con sólo un pMRU en cada tiempo de vida: {1, 1, 1, 1, 1, 1, 1, 1, 1...}
- **Regular-n**. Patrón con un número constante de pMRUs pero mayores que uno: {3, 3, 3, 3, 3, 3, 3, 3, 3...}
- **Irregular-n**. Los bloques que exhiben este patrón presentan un número diferente de pMRUs (cíclico o no) pero todos ellos mayores que la unidad: {4, 2, 2, 4, 2, 2, 4, 2, 2...}
- **Irregular-1-n**. Este patrón presenta un número variable de pMRUs (cíclico o no), algunos iguales a uno y otros mayores que uno: {1, 1, 1, 7, 1, 1, 1, 7, 1...}

El algoritmo propuesto es capaz de capturar los tres primeros patrones, pero no ocurre lo mismo con el patrón Irregular-1-n. Si algunos o la mayoría de bloques presentan este patrón, que a su vez representan a bloques con un número potencial de pMRUs, éstos serían elegidos como bloque víctima y las prestaciones se verían afectadas.

Sin embargo, el efecto negativo de la componente aleatoria del algoritmo se puede solventar con el uso de una pequeña cache víctima de la cache de último nivel. De esta manera, los bloques reemplazados se almacenan en la víctima y, si la estrategia aleatoria falla, los bloques en la víctima se acceden rápidamente y retornan a la cache de último nivel con el bit pMRU a '1'. La Figura 8 ilustra el esquema trabajando con cache víctima.

VI. Evaluación experimental

Los diferentes algoritmos han sido modelados en una versión extendida del simulador SimpleScalar [12], configurando el simulador para el juego de instrucciones Alpha y lanzando las aplicaciones de SPEC2000, que se evalúan utilizando las entradas *ref*, ejecutando 1000M de instrucciones antes de obtener estadísticas y simulando posteriormente 500M de instrucciones con detalle. La Tabla I resume los parámetros arquitectónicos utilizados en los experimentos.

Las aplicaciones que no estresan la cache de L2 han sido eliminadas del estudio. Para ello, se ha obtenido el porcentaje de fallos forzosos de cada aplicación. La Figura 5 muestra los resultados para una cache de L2

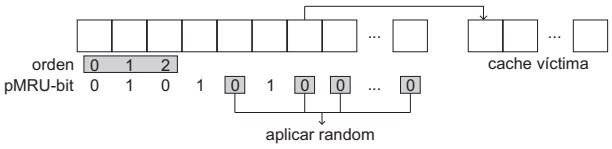

Fig. 4. Diagrama de la política pMRU-b3 con cache víctima.

TABLA I
Parámetros de la máquina.

Núcleo del microprocesador	
Política issue	Fuera de orden
Predictor de saltos	Hybrid gShare/Bimodal: gShare: 14-bits de historia global y 16K contadores de 2-bits. Bimodal: 4K contadores de 2-bits. Selector de predictor con 4K contadores de 2-bits
Penalización predictor	10 ciclos
Ancho fetch, issue y commit	4 instrucciones/ciclo
Tamaño ROB (entradas)	256
# Int/FP ALUs	4/4
Jerarquía de memoria	
Puertos memoria	4
Cache datos/instr. L1	16KB-2vías, 64B-línea
Latencia L1	1 ciclo
Cache unificada L2	1MB-16vías, 128B-línea
Latencia L2	10 ciclos
Latencia memoria principal	300 ciclos

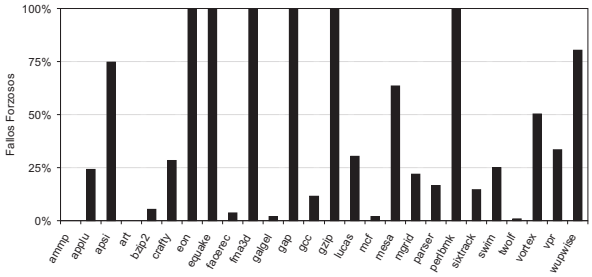

Fig. 5. Porcentaje de fallos de cache forzosos.

de 1MB-16vías. Se ha prescindido de las aplicaciones con un porcentaje de fallos forzosos mayor que un 75% o con un MPKI menor que la unidad[1].

A. Mejorando el algoritmo pMRU-b3

En esta sección abordamos la problemática de que las aplicaciones pueden cambiar dinámicamente su comportamiento en tiempo de ejecución. Por ejemplo, un bloque puede tener múltiples pMRUs en un tiempo dado cuando exhibe buena localidad temporal. En este caso, el bloque no será candidato para reemplazo. Sin embargo, si su localidad cambia o expira en el tiempo, el bloque no será considerado como víctima (a menos que todos los bloques del conjunto se encuentren en las mismas condiciones). En cualquier caso, el bloque está ocupando una línea de cache aunque ya no presenta localidad.

Para solventar el problema, el pMRU-bit de cada bloque puede ser actualizado en tiempo de ejecución de acuerdo con su comportamiento. Con este objetivo, mejoramos el algoritmo pMRU-b3 de la siguiente manera: i) *reseteando* los pMRU-bits en intervalos regulares y ii) eligiendo la mejor longitud del intervalo en tiempo de ejecución para el siguiente periodo.

La primera mejora trata de resetear el pMRU-bit cuando el bloque pierde la localidad temporal. Se han evaluado diferentes intervalos en el rango comprendido entre 32K y 8M de instrucciones que han

[1]Las diferencias en MPKI observadas para todos los algoritmos analizados en este trabajo son menores que 0.4 en las aplicaciones eliminadas.

TABLA II

MPKI DE PMRU-B3 VARIANDO EL INTERVALO DE RESET.

Intervalo	swim	mgrid	applu	vpr	galgel	art	mcf	facerec	ammp	lucas	bzip2	twolf	apsi
32K	11.6	4.9	11.5	1.8	1.2	66.8	16.5	6.0	61.5	8.2	2.0	3.5	2.3
256K	11.6	4.8	11.4	1.8	1.2	59.2	16.0	5.9	59.9	8.2	2.0	3.2	2.4
2M	11.6	5.3	11.2	1.9	1.1	50.5	15.6	6.3	58.9	8.2	2.1	3.1	5.4
8M	11.6	5.5	11.2	2.2	1.1	49.2	15.6	6.8	58.1	8.2	2.2	3.2	6.7
∞	11.6	5.5	11.2	2.3	1.1	48.3	15.6	6.6	51.1	8.3	2.2	3.2	7.3

superado la fase de *commit*. La Tabla II muestra el MPKI que obtiene pMRU-b3 en cada longitud de intervalo considerado. Una longitud infinita significa que no se realiza reset. Se puede observar que 256K instrucciones es el intervalo que, en general, obtiene mejores prestaciones para las aplicaciones con un MPKI menor que 10. Para el resto de cargas, las prestaciones mejoran si no se resetean los bits.

Basándonos en esta información empírica, el algoritmo se ha extendido de la siguiente manera para que sea adaptativo. La política resetea los bits pMRU cada 256K instrucciones en aquellas aplicaciones en que el MPKI es menor que 10, mientras que para el resto de cargas no se realiza el reset. Esta política se refiere como pMRU-b3-adp. Nótese que el número de instrucciones ejecutadas suele estar disponible en el conjunto de contadores de prestaciones de los procesadores. Teniendo esto en cuenta, almacenar los fallos en la cache de L2 durante un intervalo dado es suficiente para distinguir si el MPKI es mayor o menor que 10. La Figura 6 muestra los resultados junto con los algoritmos pMRU-b3 y LRU. El algoritmo pMRU-b3-adp obtiene, en la media, una reducción de MPKI de un 3% y 22% comparado con pMRU-b3 y LRU, respectivamente.

B. Comparativa con otras propuestas recientes

En esta sección evaluamos las prestaciones de pMRU-b3-adp frente a otras propuestas publicadas recientemente como Bubble [7], políticas de inserción adaptativas [8] y políticas RRIP [9]. Los resultados de LRU y pseudo-LRU también se tienen en cuenta. La implementación de pseudo-LRU se ha realizado mediante un árbol binario. Debido a que los algoritmos recientes proponen a su vez variantes adaptativas, hemos elegido en cada caso la versión que mejores resultados obtiene en nuestras pruebas. En particular, el algoritmo Bubble utiliza la técnica DC y grupos de 4 bloques; la mejor política de inserción adaptativa es BIP con $\epsilon = \frac{1}{32}$; y la mejor versión de

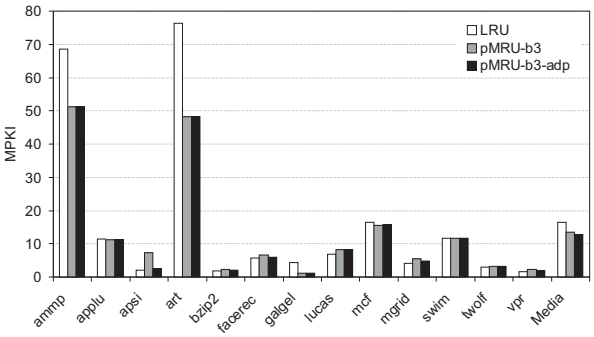

Fig. 6. MPKI de las políticas LRU, pMRU-b3, y pMRU-b3-adp.

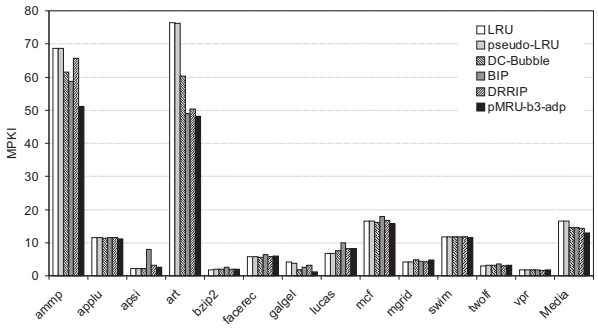

Fig. 7. MPKI de las políticas LRU, pseudo-LRU, DC-Bubble, BIP, DRRIP y pMRU-b3-adp.

RRIP es 3-bit DRRIP HP con 32 entradas de SDM, 10-bit contador PSEL y $\epsilon = \frac{1}{32}$.

La Figura 7 muestra los resultados. Comparado con pseudo-LRU, LRU mejora ligeramente las prestaciones en algunas aplicaciones como *lucas*, mientras que pseudo-LRU reduce el MPKI respecto a LRU en otras como *art*. En la media, ambas políticas obtienen prácticamente el mismo MPKI. Por otro lado, el conjunto de propuestas recientes mejoran el MPKI respecto a LRU y pseudo-LRU, y al igual que en nuestra propuesta, los mayores beneficios se obtienen en aquellas aplicaciones con un gran número de fallos. Sin embargo, la política que más reduce el número de fallos, en la media, es pMRU-b3-adp. En particular, la reducción se encuentra entre un 10% y 11% respecto a DC-Bubble, BIP y DRRIP.

C. Prestaciones con cache víctima

Para tratar que aquellos bloques que presentan el patrón pMRU Irregular-1-n (ver Sección V) no afecten negativamente a las prestaciones, la política pMRU-b3-adp se evalúa junto con una cache víctima donde se aplica LRU. Estos bloques dispondrán de la oportunidad de volver a L2 sin acceder a memoria principal. Hemos asumido un tamaño fijo de cache víctima de 64KB, el cual es mucho menor (16 veces) que el tamaño de la cache de L2 (1MB). Se han testeado caches víctimas con 32 vías y completamente asociativas. A efectos de comparación, también se ha sometido a la política LRU al efecto de la cache víctima.

La Figura 8 muestra los resultados de MPKI. Nótese que, para estos resultados de MPKI, el número correspondiente de aciertos en la cache víctima ha sido sustraído. En caso contrario, el MPKI sería prácticamente el mismo con y sin cache víctima. Se puede observar que el uso de la víctima ayuda a reducir el impacto en las prestaciones que supone reemplazar aquellos bloques que se volverán a referenciar pronto. En particular, pMRU-b3-adp

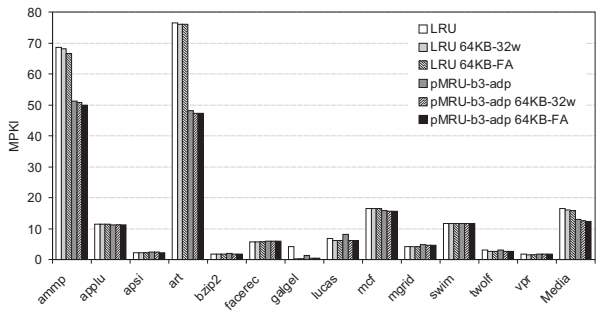

Fig. 8. MPKI de las políticas LRU y pMRU-b3-adp con y sin cache víctima de 64KB-32vías o completamente asociativa (FA).

con cache víctima reduce el MPKI de aquellas cargas donde LRU obtenía mejores resultados. Este es el caso de *lucas*, *twolf* y *apsi*, donde los valores de MPKI obtenidos con la propuesta y cache víctima son mejores que los obtenidos con LRU y cache víctima. En algunos otros casos, aunque el uso de cache víctima no permite superar a LRU, se reducen las diferencias. Este es el caso de *mgrid* y *bzip2*. Por otro lado, como esperábamos, el uso de cache víctima también beneficia a las prestaciones de LRU en algunas aplicaciones (*vpr* y *galgel*). También se puede apreciar que no es necesario una cache víctima completamente asociativa, ya que una cache de 32 vías obtiene prácticamente los mismos resultados con una complejidad menor. El algoritmo pMRU-b3-adp con cache víctima consigue reducir el MPKI hasta en un 24% respecto a LRU sin cache víctima.

Para cuantificar de manera más clara cual es la política que se beneficia más del uso de la cache víctima, se ha obtenido la tasa de aciertos en esta estructura. La Figura 9 muestra los resultados para la víctima de 32-vías. Por supuesto, las aplicaciones donde más se reduce el MPKI son aquellas que obtienen mayor tasa de acierto. Por ejemplo, la tasa de acierto mayor en pMRU es la de *lucas* (más de un 10%) que reduce el MPKI de 8 a 6. Para LRU, la tasa de acierto mayor corresponde a *galgel* y reduce el MPKI de 4 a 0.5. Finalmente, la política pMRU obtiene, en la media, una tasa de acierto de 1.2%, mientras que este porcentaje se reduce a 0.8% en LRU.

VII. Conclusiones

En este trabajo se ha definido el número de periodos MRU (pMRU) de un bloque como el número de veces que un bloque ocupa la posición MRU mientras permanece en la cache, y se ha propuesto para caches de último nivel el algoritmo de reemplazo basado en pMRU, que selecciona la víctima al azar entre aquellos bloques que tienen un solo pMRU. Se ha mejorado el algoritmo original teniendo en cuenta la recencia de información y los cambios en el comportamiento de las aplicaciones durante su ejecución. También se ha hecho uso de una cache víctima para reducir el efecto de la componente aleatoria de selección de bloque víctima del algoritmo pMRU.

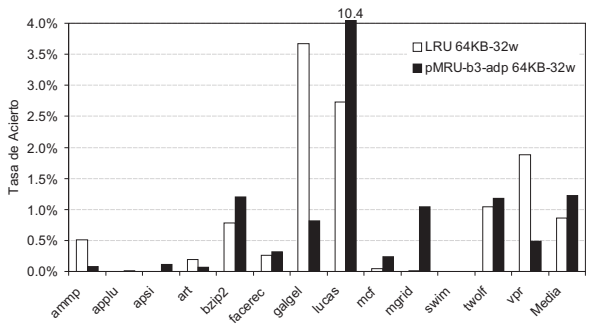

Fig. 9. Tasa de aciertos de la cache víctima de 64KB-32vías con los algoritmos LRU y pMRU-b3-adp en L2.

Los resultados experimentales han mostrado que, en la media, la mejor versión de algoritmo pMRU obtiene una reducción de MPKI de un 22% respecto a LRU. Comparado con un conjunto de políticas recientes, la reducción de MPKI en la media se encuentra entre un 10% y 11%. Finalmente, el hecho de añadir una cache víctima ayuda al algoritmo a reducir el MPKI hasta en un 24% respecto a LRU.

Agradecimientos

Este trabajo ha sido financiado por el Ministerio de Ciencia e Innovación (MICINN), el programa Consolider y los fondos de Plan E, así como por los fondos FEDER de la Comisión Europea, mediante las subvenciones a los proyectos CSD2006-00046 y TIN2009-14475-C04-01.

Referencias

[1] J.-L. Baer, *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*, Cambridge University Press, 2010.

[2] L. A. Belady, "A Study of Replacement Algorithms for Virtual-Storage Computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[3] A. Valero et al., "MRU-Tour-based Replacement Algorithms for Last-Level Caches," in *Proceedings of the 23rd International Symposium on Computer Architecture and High Performance Computing*, October 2011, pp. 112–119.

[4] A. Valero et al., "Improving Last-Level Cache Performance by Exploiting the Concept of MRU-Tour," in *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, October 2011, p. 214.

[5] A. Valero et al., "Algoritmo de reemplazo para cache de último nivel basado en periodos MRU," in *Actas de las XXII Jornadas de Paralelismo*, September 2011, pp. 557–562.

[6] W. A. Wong and J.-L. Baer, "Modified LRU Policies for Improving Second-Level Cache Behavior," in *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, January 2000, pp. 49–60.

[7] C. Zhang and B. Xue, "Divide-and-Conquer: A Bubble Replacement for Low Level Caches," in *Proceedings of the 23rd International Conference on Supercomputing*, June 2009, pp. 80–89.

[8] M. K. Qureshi et al., "Adaptive Insertion Policies for High Performance Caching," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007, pp. 381–391.

[9] A. Jaleel et al., "High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, June 2010, pp. 60–71.

[10] D. Wood et al., "A Model for Estimating Trace-Sample Miss Ratios," in *Proceedings of the 1991 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1991, pp. 79–89.

[11] *Standard Performance Evaluation Corporation. http://www.spec.org/cpu2000.*

[12] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0," *ACM SIGARCH Computer Architecture News*, vol. 25, pp. 13–25, June 1997.